Documentation :

Energy-Aware Multi-Agent Patrol Environment with Meta-Learning

Made by :
- Hadj Sassi Emen
- Vizi Márk

0/ Environment Setup and Engineering Improvements

Several foundational fixes and extensions were implemented to ensure correctness, stability, reproducibility, and usability of the environment.

a) Containerization : Created Docker Container

The entire project was containerized using Docker.

Purpose:

- Ensure reproducibility across machines

- Eliminate dependency and version conflicts

- Provide a consistent execution environment for training and evaluation

All experiments reported in this document were executed inside the Docker container.

b) Fixed Code to Work on a Single Core

The environment and training loop were adapted to correctly support single-core execution.

Effects : This step simplified debugging and validated core logic before multi-core training.

c) Fixed NaN Distances Due to Agent Overlap

An issue causing NaN distance values during training was identified and fixed.

Cause:

- Overlapping agents or landmarks during initialization or movement

- Division by zero in distance normalization

Fix:

- Added safety checks and minimum distance thresholds

- Ensured stable distance computations in all observation functions

Effects : This fix was critical for preventing silent training failures

d) Fixed Missing Energy Station Spawning

An issue where energy stations were not consistently spawned was fixed.

Impact:

- Energy mechanics were previously ineffective

- Recharging behavior could not emerge

Effects : After the fix, energy stations reliably appear and interact correctly with agents.

e) Fixed Collision Overlap During Movement

A collision handling bug allowing agents to overlap during movement was fixed.

Fix:

-Prevented invalid state updates after collisions
- Enforced proper separation between entities

Effects : This ensures physically plausible interactions and stable dynamics.

f) Added GIF Generation

Support for automatic GIF generation was added.

g) Fixed Multi-Core Training

After stabilizing single-core execution, multi-core training was re-enabled and corrected.

Improvements:

- Proper environment seeding per worker

- Correct vectorized environment construction

- Stable parallel rollout collection

This enabled efficient large-scale training across multiple environments.

h) Added Back Visual Rendering in Docker

Visual rendering was restored inside the Docker container.

Purpose:

- Support debugging of agent behavior

- Allow qualitative inspection of learned policies

Rendering can be enabled using `render_mode="human"` or `rgb_array`.

1/ Map

a) Procedural Map Generation

Maps are generated procedurally at each reset.

Procedural Elements:

- Player spawn locations

- Obstacle placement

- Environment boundaries

This ensures that each episode represents a different task instance.

b) Continuous Action Space

The environment uses a continuous action space.

```
Box(-1.0, 1.0, (2,), float32)
```

Benefits:

- Enables smooth movement and realistic trajectories
- Better suited for PPO-based methods

2/ Energy System

In standard MPE-style environments, agents can move indefinitely without consequence, which often leads to unrealistic policies that rely on constant high-speed motion.
However, we can introduce an energy budget where movement becomes a strategic resource, forcing agents to trade off speed versus endurance, decide when to patrol and when to recharge, and avoid wasting energy on unnecessary motion. Which will result into transforming movement from a free action into a deliberate strategic decision.

a) Agent Energy Attributes

Patrolling agents are initialized with an explicit energy model that constrains their movement and introduces long-term planning requirements.

Location : patrol_env.py → _configure_patrollers

```
a.max_energy = 100.0
a.energy = a.max_energy
a.recharging = False
```

Effects: Agents start with full energy. The“recharging” indicates whether the agent is currently at a station.

b) Energy Drain (Velocity-Based)

Energy drain is proportional to velocity, meaning that faster movement accelerates depletion.

Location : baseEnv.py → _update_energy

```
energy_cost = 0.3 * velocity
agent.energy = max(0.0, agent.energy - energy_cost)
```

Properties:

- Intruders do not consume energy.

- Energy never becomes negative.

Effects: This asymmetry forces patrollers to adapt their strategy depending on their remaining resources.

c) Movement Restriction at Zero Energy

Agents stop moving when energy is depleted.

Location : baseEnv.py → _update_energy

```
if agent.energy <= 0.0:
    agent.state.p_vel[:] = 0.0
    agent.max_speed = 0.0
```

Effects: Agents are physically immobilized at zero energy in a way they introduce irreversible consequences for poor planning.

d) Recharging at Energy Stations

Agents recharge when colliding with an energy station.

Location : baseEnv.py → _handle_recharge

```
if dist <= (agent.size + lm.size):
    agent.recharging = True
    agent.energy = min(agent.max_energy, agent.energy + 2.0)
```

Rules:

-Only landmarks named energy_station_* can recharge agents.

-Recharging occurs **before** energy drain each step.

-Energy is capped at the maximum value.

Effects: In this case energy stations act as strategic landmarks that shape long-term navigation decisions.

e) Energy in Observations

Each agent observes its current energy level as part of the observation vector.

Location : patrol_env.py → observation
patrol_env.py → _calculate_observation_space

```
energy_norm = agent.energy / agent.max_energy
obs_dim += 1
```

-Energy is normalized to [0, 1].

3/ Energy Station Awareness

a) Closest Energy Station Feature

Patrolling agents observe the closest energy station.

Location : patrol_env.py → observation
patrol_env.py → _closest_energy_station

Observation Vector :[distance, direction_x, direction_y]

Rules :

-Only patrollers receive this information.

-Intruders receive zero vectors.

-Observation space is extended: obs_dim += 3

Effect : This enables agents to plan recharging behavior.

4/ Range-Limited Sensing

If agents can sense everything, their behavior becomes unrealistically optimal. Limiting their sensing forces exploration and local decision-making, making tasks more challenging and varied.

a) Sensor Range Definition

Each patroller has a limited sensing radius.

Location: patrol_env.py → _configure_patrollers

```
a.sensor_range = a.patrol_radius * 1.3
```

Effects: This prevents global perception and enforces local decision making.

b) Range-Limited Landmark Perception

Landmarks are visible only within sensing range.

Location : patrol_env.py → _get_landmarks_info

```
if d <= sensing_range:
    ...
else:
    [0, 0, 0]
```

Effects: This introduces uncertainty and prevents agents from exploiting global knowledge.

c) Range-Limited Agent Perception

Other agents are sensed only if they are close enough.

Location : patrol_env.py → _get_other_agents_info

```
if d <= sensing_range:
    ...
else:
    [0, 0, 0]
```

Effects: Agent interactions happen based on nearby encounters, not on full awareness of the entire environment.

d) Range-Limited Goal Perception

Goals are observed only within sensing range.

Location : patrol_env.py → _get_goal_info

```
in_range = d <= agent.patrol_radius * 1.3
```

Effects: This forces agents to actively search rather than directly navigate to known targets.

5/ Line-of-Sight (LOS) Blocking

Even when a target is within sensing range, obstacles can block visibility.
Adding line-of-sight checks prevents agents from “seeing through walls,” introduces uncertainty in perception, and forces agents to move to gain visibility.

a) Obstacle-Based Visibility Test

A geometric line-of-sight check prevents seeing through obstacles.

Location : patrol_env.py → _blocked_by_obstacle

Details:

-Line-segment vs. circle intersection test.

-Only collidable obstacles are considered.

-Goals and energy stations are ignored.

b) Applied to Goal Sensing

Goals are visible only if not blocked.

Location : patrol_env.py → _get_goal_info

```
blocked = self._blocked_by_obstacle(...)
if in_range and not blocked:
    return goal_info
else:
    return zeros
```

6) Sensor Noise

Sensor noise models real-world imperfections and prevents agents from overfitting to exact observations. By adding Gaussian noise, observations become uncertain and agents must rely on overall trends instead of precise values. Also the noise can be turned off for ablation studies to enable controlled comparisons.

a) Configurable Noise Parameter

Gaussian noise simulates imperfect sensors.

Location : patrol_env.py → __init__

```
self.sensor_noise_std = 0.05
```

b) Noise Injection

Location : patrol_env.py → _apply_noise

```
noise = np.random.normal(0, self.sensor_noise_std, size=x.shape)
return x + noise
```

Applied To Landmark observations and other agent observations.

Notes:

-Adds realistic sensing uncertainty.

Can be disabled by setting: sensor_noise_std = 0.0

7/ Rewards & Episode Termination

a) Termination Conditions

An episode ends when:

- The intruder is caught.

- The intruder reaches the goal.

- The maximum number of cycles is reached.

b) Rewards

- Reward structure remains unchanged.

- Energy constraints indirectly shape agent behavior.

c) Outcome

- The environment is suitable for energy-aware and adaptation-focused studies.

8/ Meta-Learning :

The goal of Meta-Learning is to demonstrate that agents adapt quickly to new, unseen maps, rather than overfitting to a single fixed environment.

Approach: RL² with Recurrent Policy

We implement RL²-style meta-learning using a recurrent reinforcement learning policy.
Instead of explicitly learning a separate meta-optimizer, adaptation emerges from the agent's internal memory.

Why RL²?

RL² treats the policy's recurrent hidden state as a form of fast adaptation:

- The agent does not reset its memory between episodes

- The LSTM learns to infer task structure (map layout, obstacles, station placement) from experience

- Adaptation happens within a few steps, not via retraining

a) Recurrent PPO (LSTM Policy)

We use Recurrent PPO with an LSTM-based policy, which allows the agent to:

- Remember past observations and rewards

- Adapt behavior across timesteps within an episode

- Generalize to new environments through memory-based inference

```
from sb3_contrib import RecurrentPPO

model = RecurrentPPO(
    policy="MlpLstmPolicy",
    env=env,
    ...
)
```

b/ Training on Multiple Maps

Map Randomization:

Each environment reset samples a **new random map**, ensuring exposure to a diverse task distribution.

```
def make_sb3_env(seed: int, env_kwargs: dict):
    env = parallel_env(**env_kwargs)
    env.reset(seed=seed)
    ...
```

- Different seeds → different map layouts

- Energy stations, obstacles, and geometry vary per episode

- Forces the policy to learn general adaptation strategies

Parallel Environments:

Multiple environments are trained **in parallel**, each with different random maps.

This accelerates learning and improves robustness across task variations.

```
num_envs = 8
async_vec = MakeCPUAsyncConstructor(num_cpus)(env_fns, obs_space, act_space)
```

c) Memory-Based Adaptation (Core of Meta-Learning)

LSTM Policy Architecture

The policy includes an LSTM layer that maintains a hidden state across timesteps.

```
policy_kwargs=dict(
    lstm_hidden_size=256,
    net_arch=[256, 128],
)
```

What the LSTM learns implicitly:

- - Where obstacles tend to block paths
- - When energy depletion occurs
- - How far energy stations typically are
- - How to modify patrol behavior on unfamiliar maps

No explicit map information is provided, adaptation happened purely from experience.

d) Evaluation on Unseen Maps

During evaluation, the agent is tested on **new maps generated with different seeds**.

```
eval_env = make_sb3_env(seed + 10_000, env_kwargs)
```

- These maps were never seen during training

- Performance reflects **adaptation capability**, not memorization

-Evaluation is deterministic to isolate policy behavior

- Meta-learning is achieved using RL²-style recurrent reinforcement learning

- RecurrentPPO with LSTM enables memory-based adaptation

- Agents learn how to adapt, not just what to do

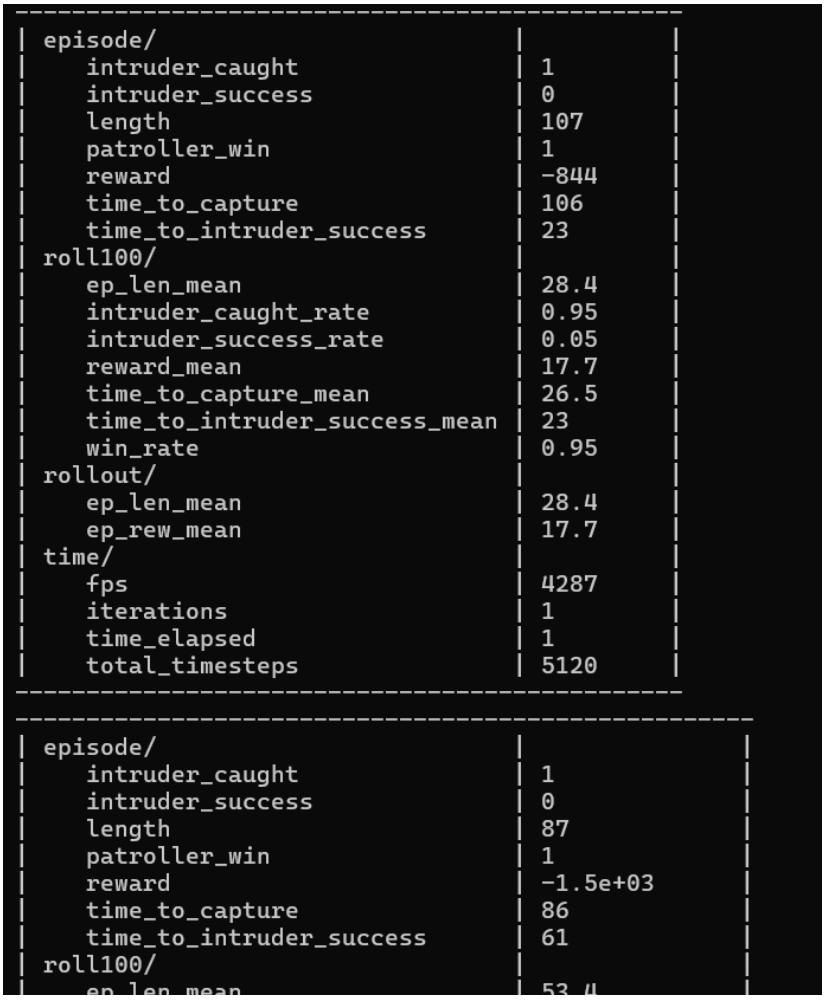
9/ Extended Logging System

The logging system was expanded beyond loss tracking.

Logged Metrics Include:

- Episode reward / Episode length / Intruder caught / success rates / Time to capture
  Win rate per team

Figure 1 – Training Metrics :



episode/	
intruder_caught	1
intruder_success	0
length	107
patroller_win	1
reward	-844
time_to_capture	106
time_to_intruder_success	23
roll100/	
ep_len_mean	28.4
intruder_caught_rate	0.95
intruder_success_rate	0.05
reward_mean	17.7
time_to_capture_mean	26.5
time_to_intruder_success_mean	23
win_rate	0.95
rollout/	
ep_len_mean	28.4
ep_rew_mean	17.7
time/	
fps	4287
iterations	1
time_elapsed	1
total_timesteps	5120

episode/	
intruder_caught	1
intruder_success	0
length	87
patroller_win	1
reward	-1.5e+03
time_to_capture	86
time_to_intruder_success	61
roll100/	
ep_len_mean	53.4

This figure presents the training statistics during early learning iterations.

Figure 2 – Evaluation on Unseen Maps:

```
Model has been saved.
Finished training on patrolling_v1.
Box(-1.0, 1.0, (2,), float32)

Starting evaluation on patrolling_v1 (num_games=15, render_mode=human)
Avg reward: 471.5584726809005
Avg reward per agent, per game:  {'patroller_0': 495.97656054968246, 'patroller_1': 547.6153930393547, 'patroller_2': 376.9046346593987, 'patroller_3': 328.
8346212289722, 'intruder_0': 608.4611539270944}
Full rewards:  {'patroller_0': 7439.6484082245237, 'patroller_1': 8214.23089559032, 'patroller_2': 5653.569519890981, 'patroller_3': 4932.519318434583, 'intr
uder_0': 9126.917308906417}
Patroller team mean score: 437.33280236935195
Intruder score: 608.4611539270944
Patroller win rate: 0.6666666666666666
Intruder win rate: 0.3333333333333333
root@8cb0487c7aa4:/app#
```

This figure shows the evaluation results of the trained model on 15 previously unseen maps.
The patroller team achieves an average reward of 471.56 with a 66.7% win rate, demonstrating effective generalization and adaptive behavior beyond the training environments.

Figure 3– Average Episode Reward :

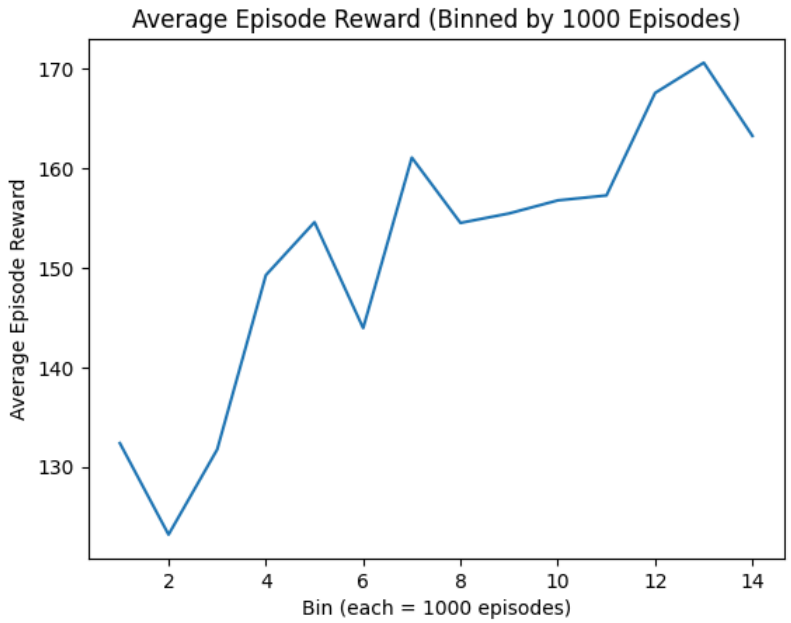


Figure 4– Reward per Episode :

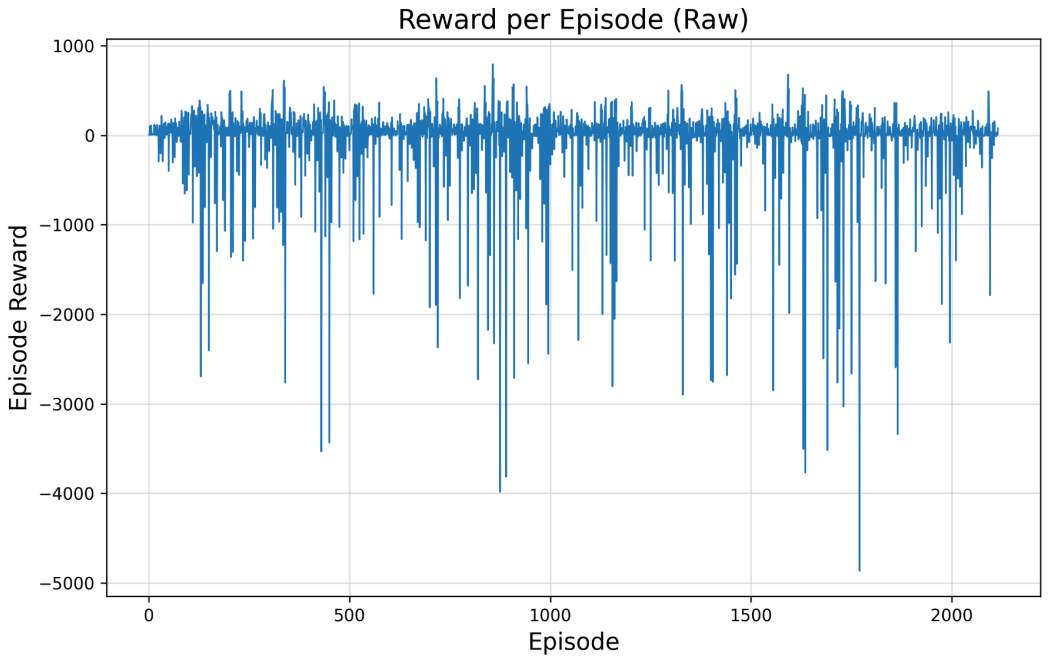
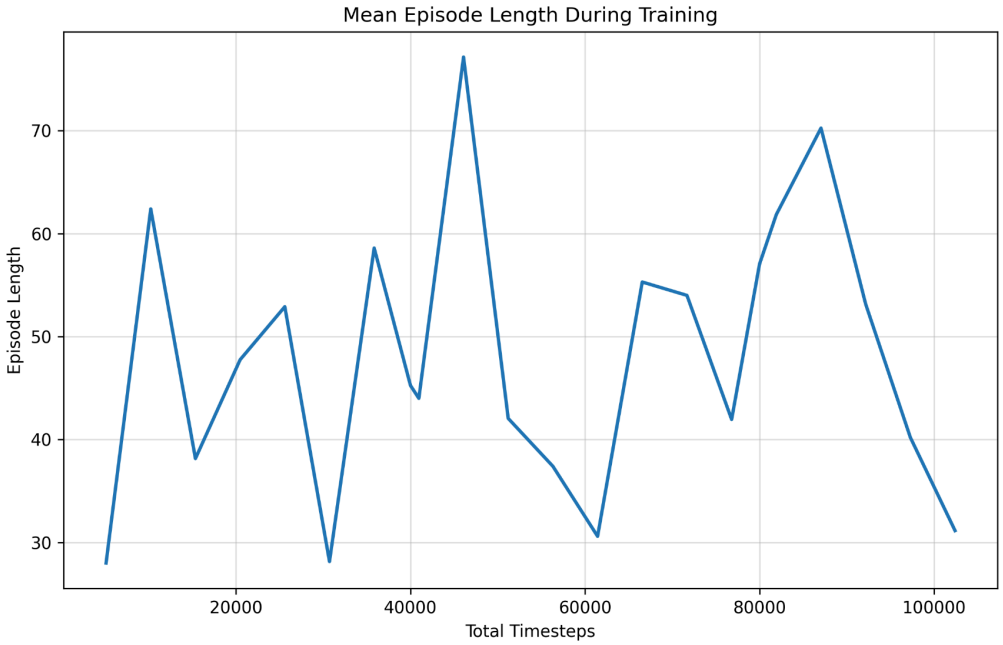


Figure 5– Mean Reward During Training :

Figure 6– Mean episode length during training



Gifs: