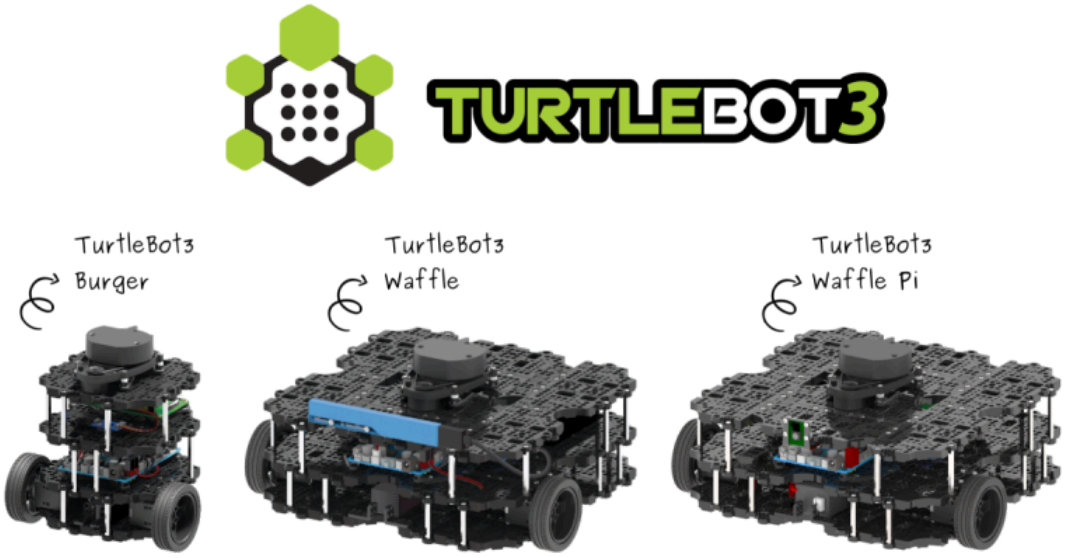
2. Assignment - Cooperative Mingle

Zoltán Barta, PhD student, Department of Artificial Intelligence

10 min read

March 31, 2025

Collective Intelligence



This assignment focuses on implementing a **Multi-Agent Reinforcement Learning (MARL)** system using **TorchRL**. Inspired by real-world cooperative robotics, this challenge requires a team of TurtleBot3 agents to learn **coordinated navigation and obstacle avoidance** in dynamic environments simulated in **ROS2 Humble and Gazebo**. Agents must operate in a **shared arena with static and dynamic obstacles**, reaching individual goals without colliding with one another or the environment.

You will start with an existing MARL project built using **custom Deep Q-Networks (DQN)** integrated with ROS2 nodes. The main task is to **migrate this project to use TorchRL's native ecosystem**.

Project GitHub Link

This task is best addressed using the **Centralized Training with Decentralized Execution (CTDE)** paradigm. During training, agents can access global state information and train with a **centralized critic** to stabilize learning. During evaluation, however, each TurtleBot3 agent must act independently based on **partial, local sensor data** such as LiDAR, odometry, and goal pose. An Introduction to Centralized Training for Decentralized Execution in Cooperative Multi-Agent Reinforcement Learning

While the final goal is to use **TorchRL**'s native environment structure (`EnvBase` , `TensorDict` , etc.), you may initially use **PettingZoo environments** with the official `PettingZooWrapper` provided by TorchRL, if helpful for bootstrapping.

Assignment Directions

- **Adapt the Existing TurtleBot3 Environment Using Native TorchRL and MAPPO**
  Your primary task is to **rebuild the existing ROS2 + Gazebo-based MARL project using TorchRL's native APIs**, replacing the current DQN implementation with a **Multi-Agent Proximal Policy Optimization (MAPPO)** architecture. Instead of using PettingZoo, adapt the custom simulation environment to be compatible with `EnvBase` and `TensorDict`.

  You will design a full training pipeline using TorchRL that supports:
  - **Centralized Training with Decentralized Execution (CTDE)**
  - **MAPPO** implementation using TorchRL's policy and value networks
  - Integration of a **configuration management system** (e.g., Hydra or structured YAML)
    - **Inclusion of logging** via both Weights & Biases (WandB) and TensorBoard
    - Docker for reproducibility and cross-platform compatibility
    - Structured **unit testing** (at least 2 components)
    - **Visualization** outputs (e.g., GIFs, performance plots)

- A clear and well-maintained `README.md` with **setup and usage instructions**

Elements to Preserve:

- **Gazebo + ROS2 Environment**: Preserve the existing ROS2 Humble and Gazebo simulation setup, including TurtleBot3 motion primitives, sensor models, and real-time physics-based interactions.
- The core challenge remains **multi-agent goal-reaching without collisions**, where agents must coordinate spatially in a confined shared arena with limited sensing.
- **Multi-Agent Setting**

Elements to Improve or Redesign:

- Utilize **Proximal Policy Optimization**, specifically the clipped variant (PPO-Clip), as the core learning algorithm. PPO-Clip (Optionally, you could experiment with MADDPG, QMIX, VDN)
- Extend the task dynamics by introducing **tighter corridors, dynamic door states**, or shifting obstacles (e.g., mobile barriers or rotating gates) to enforce collision-aware path planning.
- **Curriculum Learning**: Implement a curriculum that starts with low-density, obstacle-free arenas, gradually introducing more agents, goal proximity overlaps, and cluttered layouts to promote scalable coordination. Curriculum Learning
- Refine the reward function to **penalize overshooting, deadlocks, or collisions** while positively rewarding coordinated timing, goal occupancy success, and spatial distribution. Reward Shaping
- Optionally, **inject LiDAR noise or add occlusion logic** to simulate more realistic sensing and encourage robust policies.
- **Design and track custom metrics** to evaluate the performance of your swarm, such as:
  - **Average time-to-goal per agent
  - **Number of idle or blocked agents per episode
  - **Collision count and type (agent-agent, agent-wall)

A Possible Structured Plan for Implementation Using Native TorchRL

0. Possible Directory Structure

```
marl-task/
├── configs/                         # Hydra or YAML configs for experiment control
│   ├── base.yaml
│   ├── env/
│   │   ├── task.yaml
│   ├── algo/
│   │   ├── ppo.yaml
│   ├── agent/
│   │   ├── default.yaml
│   └── experiment/
│       ├── exp.yaml
│
├── docker/                          # Dockerfile and entrypoints
│   ├── Dockerfile
│   └── entrypoint.sh
│
├── logs/                            # TensorBoard / WandB logs (auto-created)
│
├── outputs/                         # Visualizations (e.g., GIFs, videos)
│   ├── gifs/
│   └── metrics/
│
├── models/                          # Trained model checkpoints
│   ├── ppo/
│   │   ├── seed_1.pt
│   │   ├── seed_2.pt
│
```

```
├── src/                          # Source code
│   ├── envs/
│   │   ├── env.py
│   │   └── metrics.py            # Custom metrics
│   │
│   ├── agents/
│   │   ├── ppo_agent.py          # PPO policy/training logic
│   │   └── utils.py
│   │
│   ├── rollout/
│   │   ├── evaluator.py          # Evaluation logic
│   │   └── visualizer.py
│   │
│   └── main.py                   # Entry point: loads config and runs training
│
├── test/                         # Unit tests
│   ├── test_env.py
│   ├── test_metrics.py
│
├── .gitignore
├── requirements.txt
├── README.md
└── LICENSE
```

1. Environment Setup

Define a Custom TorchRL-Compatible Environment

Create a class `Env(EnvBase)` in `src/envs/env.py` with the following methods:

- `reset(self) -> TensorDict`
- `step(self, actions: TensorDict) -> TensorDict`

Define:

- `observation_spec`
- `action_spec`
- `reward_spec`
- `done_spec`

Ensure all I/O uses `TensorDict`. Observations should be partial and relative, including distance to the shape center and nearest neighbor. Use `torchrl.envs.transforms` for normalization or preprocessing.

Optional: PettingZoo Wrapper

Use `PettingZooWrapper` from `torchrl.envs.libs.pettingzoo` if adapting from existing environments:

```
from torchrl.envs.libs.pettingzoo import PettingZooWrapper
wrapped_env = PettingZooWrapper(pettingzoo_env)
```

2. Agent and Model Definition

Define Policy and Critic Modules

In `src/agents/ppo_agent.py`, implement:

- A shared `TensorDictModule` policy:

```
policy = TensorDictModule(network, in_keys=[...], out_keys=["action"])
```

- A centralized critic using `ValueOperator`:

```
critic = ValueOperator(critic_network, in_keys=[...])
```

This supports the CTDE paradigm: centralized critic with decentralized policy execution.

3. PPO Training Setup

Collector Configuration

Use `SyncDataCollector` or `MultiSyncDataCollector`:

```
collector = SyncDataCollector(
    create_env_fn=env_fn,
    policy=policy,
    frames_per_batch=2048,
    total_frames=...
)
```

Loss Function

Use `ClipPPOLoss`:

```
loss_module = ClipPPOLoss(
    actor=policy,
    critic=critic,
    clip_epsilon=0.2,
    entropy_coef=0.01
)
```

4. Training Loop

Training in `main.py`

Set up the training loop using `collector`, `replay_buffer`, `loss_module`, and `optimizer`:

```
for batch in collector:
    for _ in range(ppo_epochs):
        loss = loss_module(batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5. Evaluation and Logging

Logging

Use TensorBoard or W&B:

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir=...)
writer.add_scalar("reward/mean", mean_reward, step)
```

Evaluation

Run trained policies with local observations only (CTDE) and export GIFs using `pygame`, `matplotlib`, or `imageio`. Store results in `outputs/`.

6. Configuration Management

Hydra Integration

Use Hydra or structured YAML configs in `configs/`:

- `configs/env/task.yaml`

- `configs/algo/ppo.yaml`
- `configs/experiment/sweep.yaml`

Launch with:

```
python src/main.py +experiment=task +algo=ppo
```

7. Testing

Unit Tests

Place tests in `test/`:

```
def test_env_reset():
    env = Env(...)
    td = env.reset()
    assert "observation" in td
```

8. CTDE Framework Details

- The shared policy is trained with access to a centralized value function.
- Execution uses only local observations per agent.
- During inference, policies should operate without access to the global state or other agents' observations.
- Ensure the actor's input keys are restricted to local observations, while the critic receives richer information.

9. Docker for Reproducibility

Add Docker Support

Create a `docker/` folder with the following:

- `Dockerfile`:

```
FROM python:3.12-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "src/main.py"]
```

- `entrypoint.sh` (optional launcher script)

Build and run:

```
docker build -t marl-task .
docker run --rm marl-task
```

PowerPoint Presentation

While presenting your work is not mandatory, **not presenting will limit your maximum grade to 3**. The presentation serves as a concise overview of your project.

Duration

- Aim for **a few well-organized slides** that complement your documentation without repeating it verbatim.

Suggested Structure

1. **Title & Objective**
   - Briefly state the objective.
   - Mention which direction you chose (migration or reimplementation).

2. **System Architecture**
   - Give a high-level overview of your system (environment, agent setup, training loop).
   - Highlight the use of **TorchRL**, and explain your training logic (CTDE, PPO-Clip).
   - Optionally include a block diagram of the pipeline (env → collector → buffer → PPO → evaluation).
3. **Environment & Task Setup**
   - Describe the environment design:
     - Custom vs. PettingZoo-based
     - Agent count and spawn logic
     - Obstacles and dynamics (if any)
   - Explain how agents observe the world and what actions they take.
4. **Key Design Choices**
   - Discuss reward structure, curriculum learning, and logic.
   - Explain any metric(s) you implemented for evaluation
   - Mention logging strategy (WandB, TensorBoard) and how configuration and reproducibility are handled.
5. **Results & Visualizations**
   - Show GIFs or short clips of trained agents forming shapes.
   - Present reward curves, training stability plots, or metric graphs.
   - Provide insights into what worked, what didn't, and what improved after tuning.
6. **Conclusion & Future Work**
   - Summarize key takeaways.

Important Notes

- The **core of your submission is your documentation and code**, which will be the primary basis for grading.
- The presentation is an opportunity to highlight your contributions and insights.

Assignment Submission and General Rules

- All development must be carried out within a **GitHub repository**.
- If working as a **team**:
  - The **collaboration strategy** (e.g., shared or individual branches) can be determined by the team.
  - **Task division must be clearly defined** and documented in the project's `README.md` file (e.g., who worked on the environment, training logic, visualization, etc.).
- If working **individually**, each student must develop their solution on a **separate branch** within the repository.
- Once development is complete, you (or your team) must upload a **single ZIP file** to Canvas containing:
  - The **entire project repository** (excluding large model files or checkpoints to keep the size manageable).
  - The **presentation in PDF format**.
- **Collaboration is highly encouraged**, as this is a larger-scale assignment that benefits from cooperative design and debugging.

👤 Zoltán Barta

📅 March 31, 2025