Cops and Thieves

Project for the Agent Systems course

Tomasz Kawiak, Mateusz Mazur

Faculty of Electrical Engineering, Automation, Computer Science and Biomedical Engineering, AGH
Field of study: Computer Science and Intelligent Systems
Specialization: Artificial Intelligence and Data Analysis

June 3, 2025

1. Introduction
2. 1st progress update
3. 2nd Progress Update
4. 3rd Progress Update
5. Final Progress Update

Introduction

Introduction

Cops And Thieves (*cops and robbers*) is a strategic pursuit-and-evasion game where two opposing agent types operate in a shared environment. Thieves aim to evade capture, while cops patrol, chase, and arrest thieves to maintain order. The game mechanics involve agent coordination, pathfinding, and adaptive decision-making.

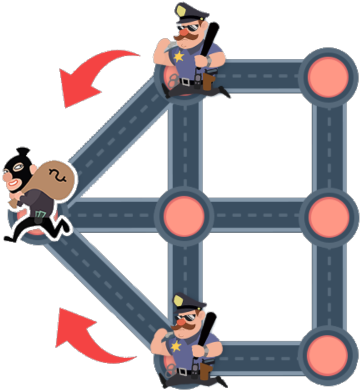


Figure 1: *Cops and Thieves* game depiction.
Source: *Catch The Thief: Help Police* by MicroEra

Motivation

Comparison of other approach for a problem considered previously on engineering studies course *Development Workshop*. Our project – *Chase model* – was also implementation of the cops and thieves game. This project aims to hopefully improve our earlier attempt.

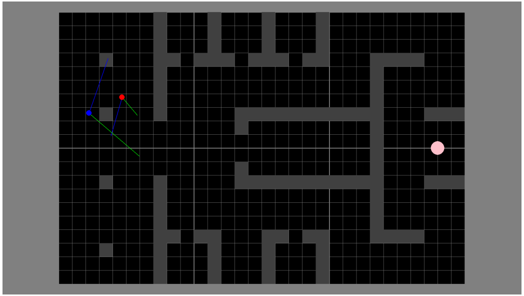


Figure 2: *Chase model* – game area.

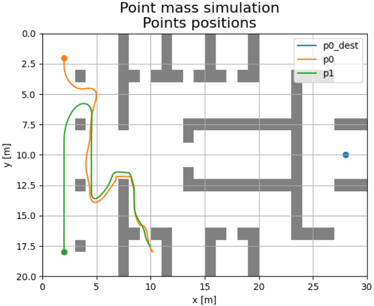


Figure 3: *Chase model* – movement chart.

Figure 4: We are back

As depicted in fig. 4, we are back to face the challenge of the cops and thieves problem, but this time with a different approach.

Goal

The main goal of this project is to train *environment-agnostic* agents:

- *Cops*: search and chase the thief.
- *Thief*: hide and try not to get caught.

A key aspect for both agents will be their ability to analyze their surroundings and act based on past observations. For our project we'll initially focus on only 2 agents (1 cop and 1 thief), with possibility of increasing the number of cop agents to observe cooperative behavior and more sophisticated search patterns.

We expect to achieve the following behaviors from agents:

Agent Type	Expected Description
Cop	Search and chase (if more cops, cooperation)
Thief	Evade capture and hide efficiently

Technological Stack

In our project we intend to use the following frameworks:

- skrl (fig. 5) for MARL implementation.
- PettingZoo (fig. 6) to guarantee MARL environment standards.
- pymunk (fig. 7) as 2D physics engine, complemented by:
  - pygame (fig. 8) for visualization.

Figure 5: skrl logo

Figure 6: PettingZoo logo

Figure 7: pymunk logo

Figure 8: pygame logo

1st progress update

Change log

Agents

As stated in our first presentation, we have 2 agent types, namely, cops and thieves. Because this part is critical for our project to move forward, we have implemented agent functionality, which covers:

- observation space implemented using vision controller (ray casting),
- action space (enabling movement of our agents in 2D space),
- initial reward function.

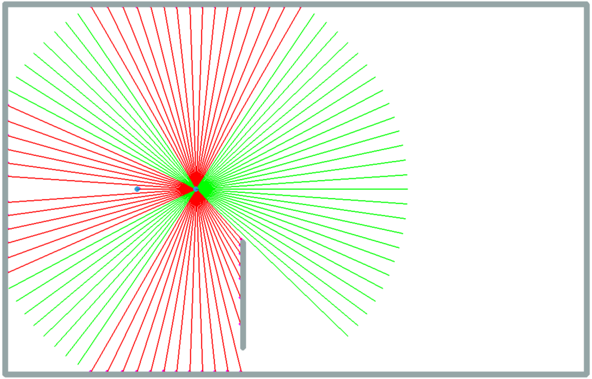


Figure 9: Visualization of vision controller

Environment

Using *PettingZoo* library we have implemented initial environment, compliant with `Parallel API` in order for all agents to simultaneously perform action and observations. As part of this we have implemented:

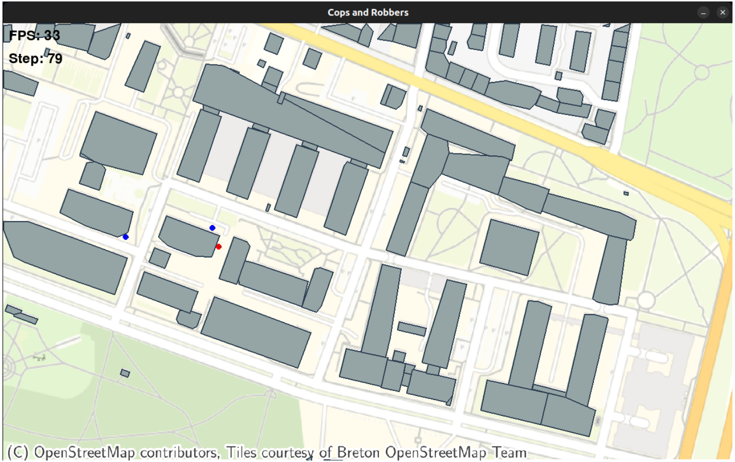
- rendering together with map generation (explained in-depth later on).
- Basic methods needed for *Reinforcement Learning* algorithms (`reset`, `step`, etc.).
- Shared observation space:
  - needed for learning loop, where agents from the same category share a common observation that represents global information about the environment,
  - constructed based on their intrinsic priority (i.e. cops wanting to catch a thief), shared observation holds the most important information about their observations.

GUI & Visualization

Apart from a command line interface (CLI) “visualization” (suited for training the agents), we have also implemented a graphical user interface (GUI) that allows us to visualize the environment and the agents’ actions. The GUI is presented in fig. 10.

Environment map generation

We have implemented a simple map generation tool that allows us to create a *map* file based real-world data from OSM. It generates obstacles for all the buildings in the area and places agents in given locations. Moreover, it generates a png file with a depiction of the area to be used as a background for the GUI. Figure 10 presents a screenshot of the GUI with a map of the AGH University of Science and Technology in Kraków, Poland.



Cops and Robbers
FPS: 33
Step: 79
(C) OpenStreetMap contributors, Tiles courtesy of Breton OpenStreetMap Team

Figure 10: GUI of the application

2nd Progress Update

Change Log

Environment

Since the last report, we have undertaken significant refactoring and bug-fixing efforts to ensure compliance with the skrl framework.

skrl & RL Algorithm

For our Multi-Agent Reinforcement Learning (MARL) task, we have chosen to utilize MAPPO (Multi-Agent Proximal Policy Optimization) due to the cooperative nature of our environment.

This phase involved:

- Introducing new spaces: *shared observation space* and *state space*
- Restructuring the observation and action spaces to fully leverage PyTorch tensors

Agents

On the agent side, we implemented minor fixes and further optimizations in the vision controller.
We are also addressing ongoing challenges with the reward function, where agents are exploiting the reward mechanism, leading to unintended behaviors during episodes.

Models

For the MAPPO implementation, we developed both policy and value networks.
Both networks use a Multi-Layer Perceptron (MLP) architecture with two hidden layers.
The policy network outputs the best possible action for a given state, while the value network serves as a critic, estimating the expected return for the agent's actions.

Policy Network

Utilizes `CategoricalMixin` base skrl class due to its stochastic nature, and because of the categorical action space.

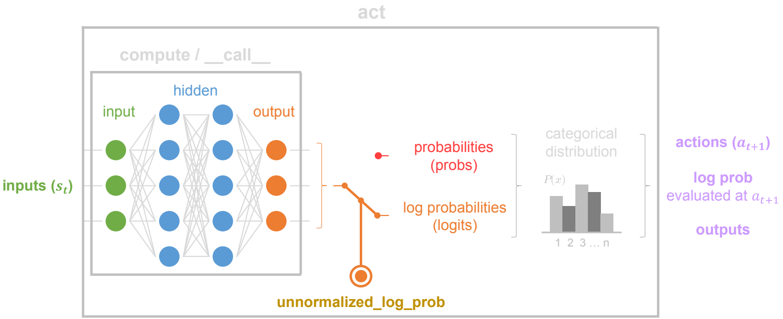


Figure 11: Concept of policy network. Source

Value Network

Utilizes DeterministicMixin base skrl class. Takes the global state of the environment.

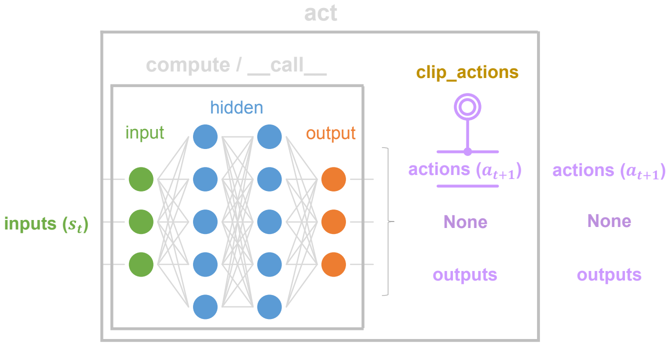


Figure 12: Concept of value network. Source

3rd Progress Update

Policy Network

- Acknowledging the spacial nature of the observation space, we have transitioned to a Convolutional Neural Network (CNN) architecture for the policy network.
- Additionally, we have distinguished 2 different channels for the agent's own observations and the shared observations.
  - This implementation involves 2 1D CNN layers.

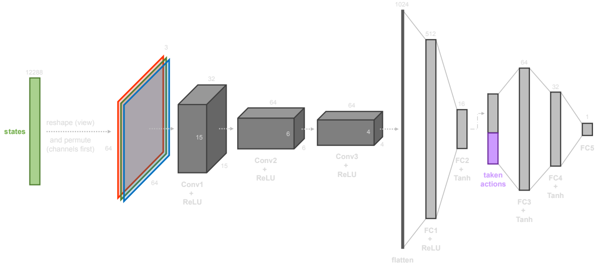


Figure 13: Concept of updated policy network

Spawn Regions

- We have introduced spawn regions to the environment, allowing agents to spawn in random locations specified within a defined area.
- This feature enhances the diversity of agent spawn points, promoting exploration and reducing the risk of overfitting.

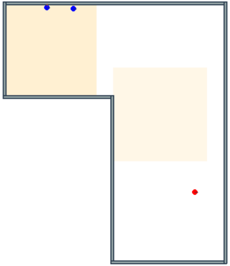


Figure 14: Spawn Regions in *lbirinth* map

Demos of Initial Results

MAPPO Shortcomings and Self-Play

- The MAPPO algorithm has shown promising results in cooperative tasks, but it is not without its challenges.
  The main bottleneck is the sample efficiency of the algorithm, which can lead to slow convergence and suboptimal policies.

- To address this, we are exploring self-play techniques, where agents learn from their own experiences and adapt their strategies based on the evolving dynamics of the environment.

Naive Self-Play

- In our initial experiments, we implemented a naive self-play approach, where agents learn from their own experiences without any external guidance.
- This method intended for the agents to learn both cooperative and competitive strategies (e.g., behaving as a cop or as a thief).
  Due to the complexity of the models, and the overcomplexity of the environment, this approach has not yielded satisfactory results and was ditched.

Sampled Self-Play

- We then implemented a sampled self-play approach, where agents are trained in cycles. Each cycle consists of a training phase and a self-play phase.

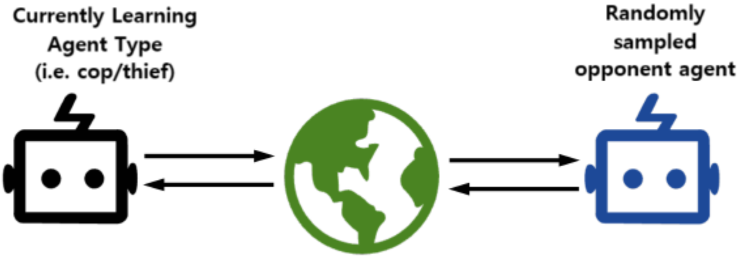


Figure 15: Sampled Self-play

Training Sampled Self-Play

1. **Sample Agents**: Sample newest agent from the trained pool.
    - If the pool is empty, create a new agent.
2. **Sample Opponents**: Sample opponent agents from the trained pool.
3. **Train**: Train the sampled agents against the sampled opponents in the environment.
    - For the training phase, we use the MAPPO algorithm.
    - When training, the policy and value networks are frozen for the opponent agents.
4. **Validate**: Validate the performance of the sampled agents against earlier sampled opponents.
5. **Save & Alternate**: Save the trained agent and proceed to learn the next agent of the opposing type.
6. **Repeat**: Repeat until a predefined number of cycles is reached.

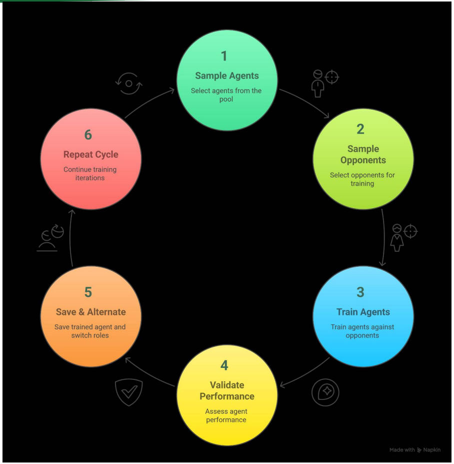

1
Sample Agents
Select agents from the pool
2
Sample Opponents
Select opponents for training
3
Train Agents
Train agents against opponents
4
Validate Performance
Assess agent performance
5
Save & Alternate
Save trained agent and switch roles
6
Repeat Cycle
Continue training iterations
Made with Napkin

Figure 16: SSP Training Loop

Self-Play Shortcomings

- The sampled self-play approach has shown some promise, but the random nature of the sampling process can lead to inconsistent performance and overfitting to specific opponents.
- Also, training agents based on the oldest agents in the pool can lead to forgetting of the latest strategies and behaviors.
- To address this, we are implementing a more sophisticated sampling strategy.

Prioritized Fictitious Self-Play (PFSP) with Population-play validation

- We are currently implementing a Prioritized Fictitious Self-Play (PFSP) approach, which aims to improve the efficiency of self-play by prioritizing the training of agents based on their performance and the diversity of their strategies.
  - As a measure of performance, we are using Win Rate (WR) and the number of wins.
  - Agents tend to sample opponents with the win rate closest to 50%.
    - This ensures that agents are trained against opponents that are neither too weak nor too strong, promoting balanced learning.
- The PFSP approach also incorporates a population-play validation mechanism, where agents are evaluated against a diverse set of opponents to ensure that the statistics of the training pool are representative of the overall population (mitigates the risk of overfitting to specific opponents and promotes generalization across different strategies).

Final Progress Update

Goal reminder

The main goal of this project is to train *environment-agnostic* agents:

- *Cops*: search and chase the thief.
- *Thief*: hide and try not to get caught.

A key aspect for both agents will be their ability to analyze their surroundings and act based on past observations. For our project we'll initially focus on only 2 agents (1 cop and 1 thief), with possibility of increasing the number of cop agents to observe cooperative behavior and more sophisticated search patterns.

We expect to achieve the following behaviors from agents:

Agent Type	Expected Description
Cop	Search and chase (if more cops, cooperation)
Thief	Evade capture and hide efficiently

Process

1. Envronment setup
   - pettingzoo compliant with parallelenv parent class. Utilizing pymunk
2. Map setup
3. Agent setup
   - General entity base class
   - observation space, shared observation space
4. Intial mappo integration with skrl
5. Policy/ value network setup
   - first with MLP, then with CNN

6. Update to map with spawn regions, limiting overfitting
7. Found limitations of plain mappo, switched to self-play aiding mappo in agent training
8. Further training loop enhancements with better opponent sampling, easing the training process for cop agents
   - first with random sampling
9. Implementing Ficticious Self Play (FSP) [2] to improve training efficiency
   - opponent sampling based on agent performance- win rate closest to 50%
10. LSTM implementation for policy/value networks.

Current setup

Reward function update

For RL and MAPPO specifically, the reward function should be scaled between -1 and 1.

Thief Reward Function

The reward r_{thief} is:

$$r_{\text{thief}} = \begin{cases} -1 & \text{if } t_0 \text{ (cop terminated)} \\ 1 & \text{if } t_1 \text{ (thief terminated)} \\ \tanh\left(\frac{d_{\text{min}} - 100}{50}\right) \div 10 & \text{if cop is visible} \\ 0.15 & \text{otherwise} \end{cases}$$

Where:

- o = observation (with keys `"distance"` and `"object_type"`)
- $t = (\text{is\_terminated\_cop}, \text{is\_terminated\_thief})$
- $\text{cop\_mask} = (o[\text{"object\_type"}] == \text{ObjectType.COP.value})$
- $d_{\text{min}} = \min(o[\text{"distance"}][\text{cop\_mask}])$

Cop Reward Function

The reward r_{cop} is:

$$r_{\text{cop}} = \begin{cases} 1 & \text{if } t_0 \text{ (cop terminated)} \\ -1 & \text{if } t_1 \text{ (thief terminated)} \\ \text{time\_penalty} + 1.5 \cdot e^{-d_{\min}/50} & \text{if thief is visible} \\ \text{time\_penalty} - 0.02 & \text{otherwise} \end{cases}$$

Where:

- o = observation (with keys "distance" and "object_type")
- $t = (\text{is\_terminated\_cop}, \text{is\_terminated\_thief})$
- $\text{thief\_mask} = (o[\text{"object\_type"}] == \text{ObjectType.THIEF.value})$
- $d_{\min} = \min(o[\text{"distance"}][\text{thief\_mask}])$
- $\text{time\_penalty} = -0.02$

Network (updated – LSTM)

- To remember the previous observations, we use LSTM networks.
- This allows the agent to learn from past observations and make better decisions based on the history of its actions and observations, e.g. when the thief hides behind a wall, the cop can remember that and search that area again later.

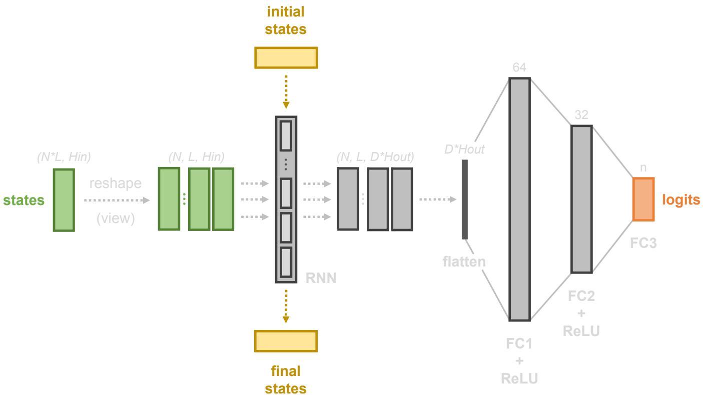


Figure 17: RNN with LSTM

Learning infrastructure

- We use a custom learning infrastructure based on skrl extended with Prioritized Fictitious Self-play (PFSP) ([2]) to balance the learning process of a policeman and a thief.

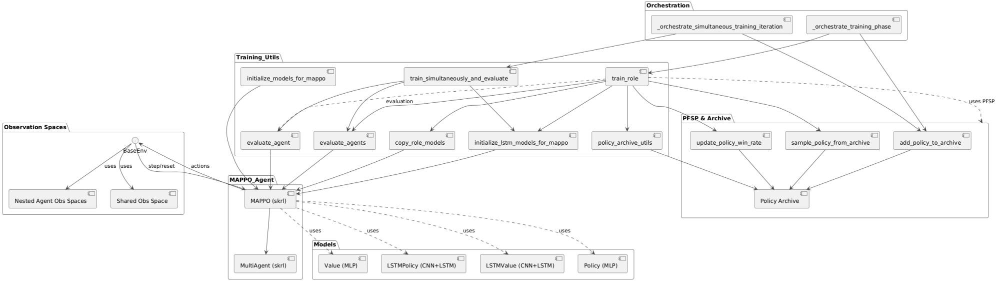


Figure 18: Learning infrastructure

Comparison with OPEN AI

Feature	Our Approach	OpenAI [1]
Environment	2D physics-based (PyMUNK)	3D physics-based (MUJOCO)
Network Architecture	LSTM + CNN	LSTM + self-attention mechanisms
Training Method	MAPPO + PFSP (Prioritized Fictitious Self-Play)	PPO + GAE (Generalized Advantage Estimation)
Cooperative Behavior	Multi-agent, supports cop (or thief) cooperation	Multi-agent, supports cop (or thief) cooperation
Learning time	10^6 episodes	$132 * 10^6$ episodes
Final result	unattained, on the right track	achieved
Opponent Sampling	Performance-based (win rate $\sim$50%)	Random or fixed

Conclusions

- Due to the high computational cost of MAPPO and the sample inefficiency, the training required for observing significant behaviors is quite long. OpenAI [1] required $3 - 4 \cdot 10^8$ episodes to achieve significant chase and hide behaviors.
  - An important thing to note here is that their training was based on a more sample efficient attention-based algorithm, whereas our approach is based on a more sample inefficient PPO-based algorithm.
- agent training is ongoing, but we have already observed some interesting behaviors:
  - the cop is able to chase the thief and catch it in some cases
  - the thief is able to hide and evade capture in some cases

Live demo

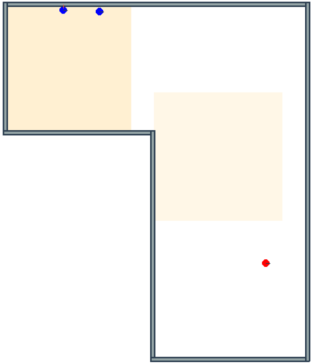


Figure 19: Live demo

Future work

- Train the agents further to achieve more complex behaviors
- Utilize maps as a curriculum learning platform
  - Create more complex maps with different layouts and obstacles
  - Use the maps to train the agents in a curriculum learning fashion, e.g. start with simple maps and gradually increase the complexity
- Improve the MAPPO training process
  - Improve the reward function to encourage more complex behaviors
- Improve the agent architecture
  - Add more complex features to the agent architecture, e.g. attention mechanisms

Thank you for your attention

Questions?

Bibliography

Bibliography

[1] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B. and Mordatch, I. 2020. Emergent tool use from multi-agent autocurricula.

[2] Heinrich, J., Lanctot, M. and Silver, D. 2015. Fictitious self-play in extensive-form games. *Proceedings of the 32nd international conference on international conference on machine learning - volume 37* (Lille, France, 2015), 805–813.