Universitat Autònoma de Barcelona (UAB)

Bachelor's Degree in Artificial Intelligence

# PRA2

## Practical Activity 2

December 12, 2025

# Contents

**1 Introduction** **3**

**2 Part 1: Prisoner's dilemma** **5**

**3 Part 2: Level-based Foraging** **9**

# 1 Introduction

This practical activity is an individual, evaluable assignment designed to consolidate and apply the concepts covered in the course. Through its completion, students are expected to demonstrate their ability to analyze information, justify decisions, and communicate results in a clear and rigorous manner.

## General Guidelines

- **Individual work**: This activity **must be completed individually**. Collaboration, code sharing, or exchange of partial or complete solutions with other students is strictly prohibited.
- **Use of external sources**: Any external resource used (scientific articles, books, websites, datasets, software libraries, etc.) must be properly cited in the report following a consistent referencing style. Lack of proper citation may result in a penalty.
- **Clarity and structure**: The report should be written in a clear, concise, and well-structured manner. All decisions must be justified, and relevant figures, tables, or code excerpts should be included when appropriate.
- **Reproducibility**: The results presented in the report should be reproducible. If the activity involves coding, all scripts used must be included or made accessible, with instructions for execution.

## Evaluation Criteria

The activity is divided into two parts, each contributing equally to the final grade:

- **Part 1 (5 points)**: Evaluation of the student's ability to understand, analyze, and correctly address the tasks proposed in the first section. This includes conceptual understanding, methodological soundness, and clarity of explanations.

- **Part 2 (5 points)**: Assessment of the application of methods, interpretation of results, and the quality of the critical analysis provided in the second section. Emphasis will be placed on justification of decisions, methodological rigor, and quality of conclusions.

## Delivery format

Students MUST deliver the following documentation (compressed in a ZIP file):

1. A **PDF document** (including your name and NIU) containing detailed responses to all the questions listed above.

   The document should incorporate images, graphs, plots, code snippets, and any other relevant materials necessary to explain and justify your answers clearly.

2. A folder containing the **source code**, or alternatively, a link to a version control system or repository (e.g., GitHub[1] or an equivalent platform).

   The submission may include Jupyter Notebooks (.ipynb) or Python scripts (.py), but it **must** satisfy the following requirements:

   - The code must be well-commented, with clear explanations that facilitate readability and understanding.
   - A file named "requirements.txt" must be included, listing all the necessary libraries and their corresponding versions to ensure correct execution.
   - A "README.md" file must be provided with detailed instructions on how to run each part of the project (e.g., training, testing). Any component that cannot be executed following these instructions will receive a grade of 0.

[1] https://github.com/

# 2 Part 1: Prisoner's dilemma

## Prisoner's dilemma

The Prisoner's dilemma[2] is a game theory thought experiment involving two rational agents, each of whom can either cooperate for mutual benefit or betray their partner ("defect") for individual gain. The dilemma arises from the fact that while defecting is rational for each agent, cooperation yields a higher payoff for each, as depicted in Table 1.

| | C | D |
|---|---|---|
| C | -1,-1 | -5,0 |
| D | 0,-5 | -3,-3 |

Table 1: Prisoner's Dilemma

Alongside this document, we provide the source code for the exercise, which implements the Prisoner's Dilemma environment using the Gymnasium API [2], as well as the code required to implement the IQL algorithm and to train and test the agent.

## Objectives

In this exercise, we will implement the MARL algorithm **Independent Q-learning** (IQL), in which each agent independently learns its Q-function using Q-learning updates, and **Central Q-Learning** (CQL), in which an agent learns a single central policy that receives observations of all agents and selects an action for each agent. We will train agents in the matrix game of Prisoners' Dilemma, regularly evaluate their performance, and visualise their learned value functions.

## Implementation

We will run the Prisoners' Dilemma matrix game as a **non-repeated game**, where agents play a single round of the game and receive rewards based on their actions. This also makes the game **stateless**.

[2] https://en.wikipedia.org/wiki/Prisoner%27s_dilemma

You are provided with source code that may assist you in implementing IQL and CQL, but its use is entirely optional. In other words, you may choose to complete the existing IQL implementation and rely on the provided training, testing, and plotting functions, or you may develop your own implementation from scratch. The structure of the provided source code is as follows:

- `iql.py`: Implements the agent class using the Independent Q-Learning (IQL) algorithm.
- `matrix_game.py`: Defines the matrix game environment, where the state/observation provided to the agents is always the constant value 0.
- `train_iql.py`: Includes utility functions to train the IQL agents and to evaluate the trained model.
- `utils.py`: Contains helper functions to visualize and plot Q-tables, the evaluation process, and the learning dynamics.
- `requirements.txt`: It includes the basic libraries required to run the code.

Install the required dependencies by running the following command:

```
conda create -n pml_pra2 python=3.10.15
conda activate pml_pra2
pip install -r requirements.txt
```

Remarks:

- For these exercises, we will install Python packages and execute them with Python 3. Specifically, all exercises were tested with Python 3.10.15 and Gymnasium 0.29.1.
- We recommend using a virtual environment to avoid conflicts with other Python installations, e.g. using `venv` or `conda`.
- You are **not allowed** to use pre-built implementations or third-party code libraries for this task.

## Questions

Complete the following tasks:

1. Open the `iql.py` file and **implement the missing parts** of the `IQL` class.
    - You will need to **complete** the `act()` and `update()` methods that implement the epsilon-greedy action selection and update the Q-function for the given experience.
    - Also, feel free to modify the `schedule_hyperparameters` function that schedules *epsilon* as a hyperparameter.
2. Once you have implemented these methods, you can run the **training and evaluation** loop by executing the following command:

```
python train_iql.py
```

    - This will train the IQL agents in the Prisoners' Dilemma matrix game and periodically evaluate their performance.
3. Once the training process concludes, a **standard evaluation is performed**, and the results are presented as plots of the learned Q-values for each agent.
    - Include these plots in the report and comment on the training dynamics (e.g., convergence behavior).
    - Determine whether any solution concept can be identified in the resulting policies. Discuss and justify your answer.
4. Implement the **Central Q-Learning (CQL) algorithm** in a file named `cql.py`. You may reuse components from the IQL implementation or develop the algorithm entirely from scratch. As in the previous exercises:
    - Train the CQL algorithm on the Prisoner's Dilemma matrix game.

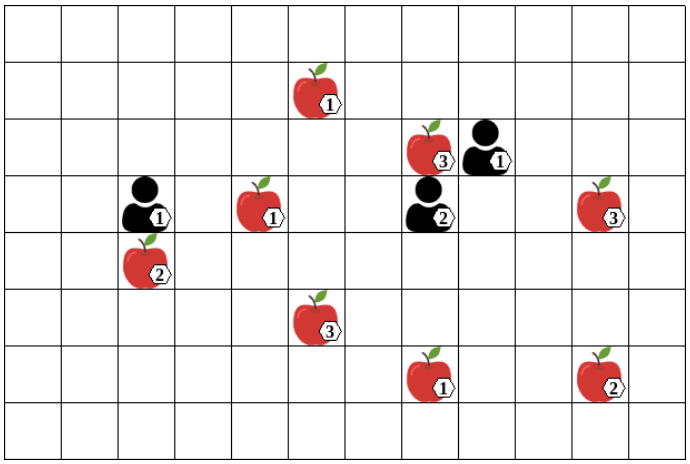
- Perform a **standard evaluation** and include in the report the results, an analysis of the training dynamics (e.g., convergence behavior), and a discussion of whether any solution concept can be identified in the resulting policies.

# 3 Part 2: Level-based Foraging

## Level-based Foraging

The Level-based Foraging (LBF) [1] [3] is a mixed cooperative-competitive game, which focuses on the coordination of the agents involved. Agents navigate a grid world and collect food by cooperating with other agents if needed.

The Level-Based Foraging environment[3] , is a Python-based environment built on top of OpenAI's reinforcement-learning framework and adapted for multi-agent settings. Its efficient implementation enables thousands of simulation steps per second on a single thread, while its rendering tools allow users to visually inspect agent behavior. The simulator supports configurable grid sizes as well as varying numbers of agents and food items. In addition, it includes several game variants—such as fully cooperative mode, where agents must collaborate to collect food, and shared-reward mode, where all agents receive the same reward-making it a valuable testbed for studying credit assignment and cooperative multi-agent learning.



More specifically, agents are placed in the grid world, and each is assigned a level. Food is also randomly scattered, each having a level on its own. Agents can navigate the environment and can attempt to collect food placed next to them. The collection of food is successful only if the sum of the levels of the agents involved in loading is equal to or higher than the level of the food. Finally, agents are awarded points

[3] Available at https://github.com/uoe-agents/lb-foraging

equal to the level of the food they helped collect, divided by their contribution (their level).

While it may appear simple, this is a very challenging environment, requiring the cooperation of multiple agents while being competitive at the same time. In addition, the discount factor also necessitates speed for the maximisation of rewards. Each agent is only awarded points if it participates in the collection of food, and it has to balance between collecting low-levelled food on his own or cooperating in acquiring higher rewards. In situations with three or more agents, highly strategic decisions can be required, involving agents needing to choose with whom to cooperate. Another significant difficulty for RL algorithms is the sparsity of rewards, which causes slower learning.

## Objectives

In this activity, we propose training and testing both the Independent and Centralized Q-learning (IQL and CQL) developed in the previous exercise on the LBF environment. The results, parameters, and other metrics will be compared to evaluate performance.

## Implementation

First, install the `LBF` library using:

```
pip install lbforaging
```

This library provides a wide range of Level-Based Foraging environments following the naming template:

```
Foraging-{GRID_SIZE}x{GRID_SIZE}-{PLAYER COUNT}p-{FOOD LOCATIONS}f
   {-coop IF COOPERATIVE MODE}-v3
```

Additional and highly informative details about the library, its features, and its variants can be found in the official GitHub repository, available at:

- https://github.com/uoe-agents/lb-foraging

Remarks:

- We recommend using Python 3.10 and Gymnasium $\geq 1.0.0$ for this part.
- We recommend using a virtual environment to avoid conflicts with other Python installations, e.g. using `venv` or `conda`.
- You are **allowed** to use pre-built implementations or third-party code libraries for this task.

## Questions

Complete the following tasks:

1. Train both an **IQL** and a **CQL** model to solve the LBF environment under the following two configurations:
   (a) Environment `Foraging-5x5-2p-1f-v3`
      - `field_size` $= 5 \times 5$
      - `players` $= 2$
      - `food_locations` $= 1$
      - `cooperative_mode` $= False$
   (b) Environment `Foraging-5x5-2p-1f-coop-v3`
      - `field_size` $= 5 \times 5$
      - `players` $= 2$
      - `food_locations` $= 1$
      - `cooperative_mode` $= True$
2. Log the training data and plot the **mean episode returns**.
3. For each algorithm, provide a **detailed explanation and justification** of the chosen parameters, describe the training procedure, and discuss the results obtained.
4. Export a **video** demonstrating a trained agent successfully solving an episode.

5. For each algorithm, **describe the agents' behaviour** and relate it to the observed levels of cooperation or competition exhibited by the trained agents.

# References

[1] Christianos, F., Schäfer, L., and Albrecht, S. V. Shared experience actor-critic for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)* (2020).

[2] Kwiatkowski, A., Towers, M., Terry, J., Balis, J. U., Cola, G. D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., and Younis, O. G. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[3] Papoudakis, G., Christianos, F., Schäfer, L., and Albrecht, S. V. Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)* (2021).