Distributed MARL Predator-Prey Final Report

Luca Fabri

luca.fabri@studio.unibo.it

November 19, 2024

Reinforcement Learning (RL) is a sub-area of Machine Learning which focuses on building strategies to make decisions that maximize the future expected reward. The RL finds applications in problems concerning self-driving cars, industrial automation, and finance but also in systems that involve the interaction of multiple agents in a shared environment, where it's more specifically called Multi-Agent RL (MARL). In computational biology, it's useful to study the population of one or more intelligent species that interact with each other to build population models: in this project, a simulation of a predator-prey ecosystem is implemented, using a MARL approach in a mixed environment, i.e. cooperative and competitive, where agents of the same species cooperatively make decisions to maximize their total expected reward. To achieve this goal, the MADDPG algorithm is exploited. The distribution of the system is realized by parallelizing the environments and by introducing a distributed training technique, inspired by the Mava [1] framework.

Contents

1 Goals/requirements

This project aims to implement a distributed application for training a Multi-Agent Reinforcement Learning (MARL) system. Agents' experiences are collected from multiple parallel environments and integrated into a centralized dataset, ultimately used for the learning process.

The goals of the project include:

- Development of a MARL Environment, specifically a Predator-Prey Environment, where predators are trained to catch prey, while prey to run away from predators;
- Development of a Replay Buffer, whose API allows the Environments to store the experiences of each of its agents;
- Development of a Learner Service, whose goal is to train a separate MADDPG model for predators and prey, and update the agents' policy inside each Environment.

Term	Definition
System	The *Distributed MARL Predator-Prey* application
Agent	Decision-making entity interacting with an environment through observations, rewards, and actions
Environment	General term referring to a Multi-Agent Cooperative-Competitive Environment. It is a collection of Agents inside a space, with equal or conflicting reward structure
Learner	Entity capable of training a Multi-Agent Environment through Reinforcement Learning
Replay buffer	Dataset of agent experiences
MADDPG	Multi-Agent Deep Deterministic Policy Gradient algorithm [3]

Table 1: Term's glossary

1.1 Use Cases

A user that interacts with this application is in front of three choices:

- Train the MARL system, by specifying the number of Predator-Prey Environments to run in parallel and setting a configuration file to customize the training process;
- Start a simulation. This option is only possible if a previous training phase has been carried out: each Environment will load the latest trained model and let it be used by its agents;
- Visualize the training results of the last training process, comprising a plot with neural network loss over time;
- Visualize agents of an environment in a scatterplot animation, by specifying the index of the environment.

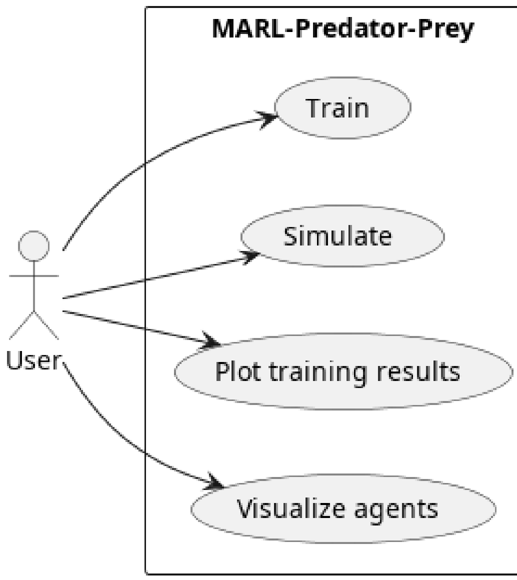

Figure 1: Use cases UML

2 Requirements analysis

In this section, the functional and non-functional requirements of the project are listed.

2.1 Functional requirements

1. **System**
   a) The System must accept a different set of parameters for different runs, in order to configure the training/simulation process;
   b) The System must be able to run in Train mode;
   c) The System must be able to simulate the Environment under study.
2. **Environment**
   a) The Multi-Agent Reinforcement Learning Environment must be a mixed Cooperative-Competitive Environment with continuous state/action space, where agents are free to move in a 2D Euclidian space of w x h dimension;
   b) The Environment must be a Predator-Prey Environment, specifically.
      i. Predators must have a reward function that increases as they approach Preys;
      ii. Preys must have a reward function that decreases as they approach Predators;
3. **Learner**
   a) The Learner must update the policies of the agents inside each Environment, in a distributed way;
   b) The Learner must implement the MADDPG algorithm as the learning algorithm.
4. **Replay Buffer**
   a) The Replay Buffer must collect the agent's experiences of each Environment, in a centralized way;

2.2 Non-functional requirements

1. **Availability**
   a) Fault tolerance: The system should be able to recover from failures and continue to operate.
2. **Deployability**
   a) Portability: The system should be able to run on different platforms.
3. **Modifiability**
   a) Extensibility: The system should be able to add new features easily.
   b) Maintainability: The system should be easy to maintain and update.

3 Architecture Design

The **microservices architecture** has been chosen for the system. This type of architecture consists of decomposing the project into smaller parts so that each one can be deployed independently. This will make it easier to add new features to the system, scale the Environment, and generally maintain the project.

The system is decomposed into three microservices:

1. **Predator-Prey service**. This component is responsible for managing a single Environment. Supports both training mode and simulation mode;
2. **Learner service**. This component is responsible for training each Environment comprising the system;
3. **Replay Buffer service**. This component is responsible for integrating all agent experiences into one single dataset.

For communication, the Replay Buffer service will expose a REST API for batching and storing data. The Environment-Learner link instead uses the Publish/-Subscribe pattern, to send/receive data, made possible through a message broker.

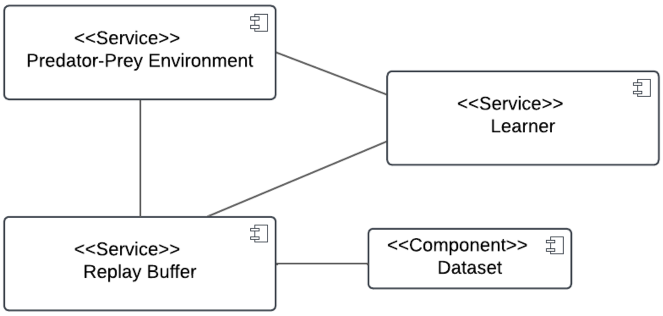

Figure 2: System's Architecture

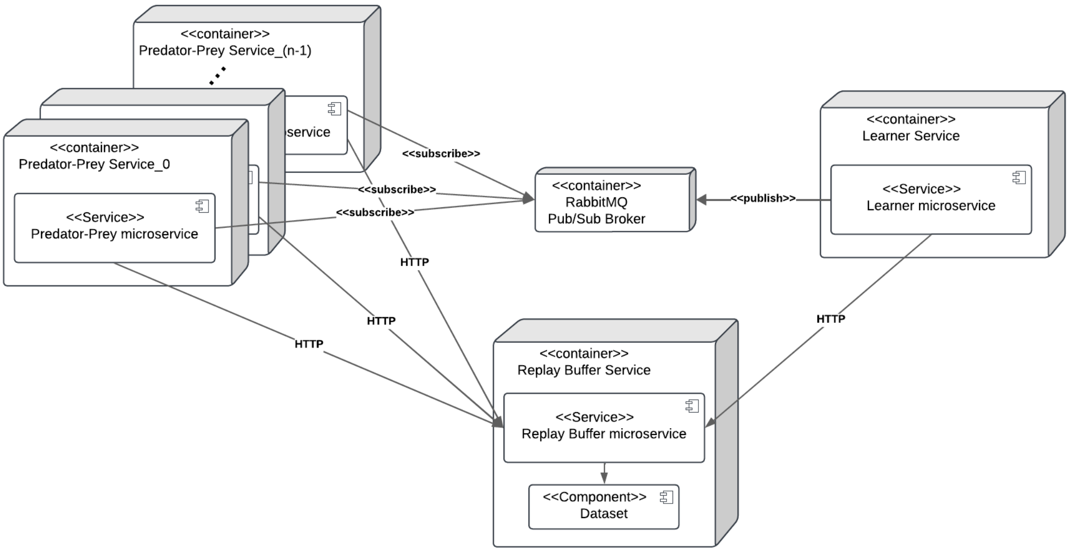

Figure 3: Training mode Deployment View

4 Microservices design and implementation

4.1 Predator-Prey Service

The Predator-Prey Service is responsible for managing the Predator-Prey environment. It allows to start the environment in two modes: Train mode or Simulation mode.

1. In the first mode (Train), it interacts with a distributed Replay Buffer to store agents' experiences and subscribes to policy updates coming from the Learner Service;
2. In Simulation mode, it uses the latest policy (saved from a previous training phase) to start a simulation. In this case, it accepts a random seed as input to control the initial positions of the agents.

4.1.1 Design

The service follows the MVC pattern to better separate the code responsibilities:

- The Model consists of the `Environment` and `Agent` classes;
- The Controller contains the `EnvironmentController` and `AgentController` interfaces and implementations. The first one collects single agents tuples (State, Action, Reward, Next state) and stores the joint tuple inside the Replay Buffer, while the latter is responsible for managing the single `Agent`. It implements the reward and done functions and contains an `AgentPolicyController`.
- The View only contains the service's entry point.

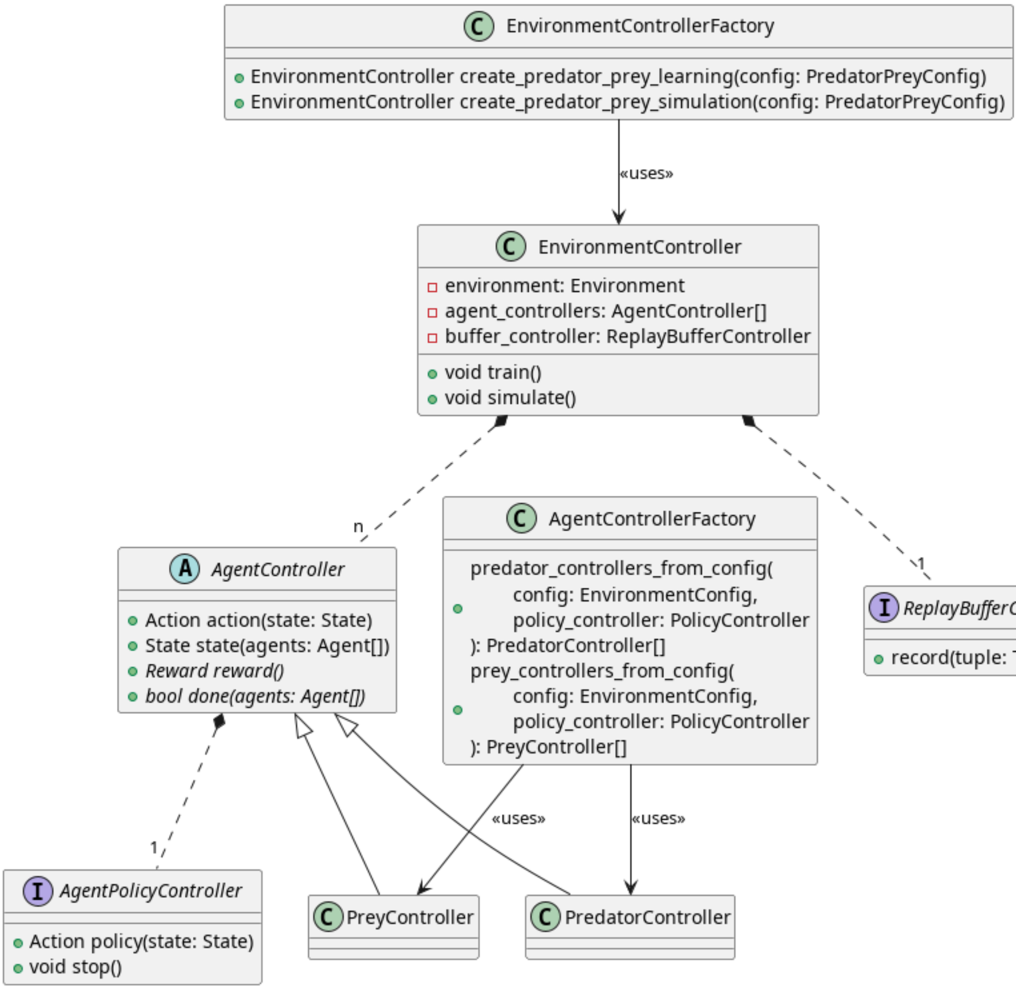

Figure 4: Controllers

4.1.2 Implementation

The controller implementations allow the service to communicate with the others that compose the distributed application.

Specifically:

- The `RemoteReplayBufferController` implements `ReplayBufferController` and allows to record the joint tuple to the distributed Replay Buffer. It leverages the Requests Python library to send

HTTP Post messages to Replay Buffer Service;

- The `AgentPolicyController` instead has two implementations, one for the Train mode and one for Simulation mode. The first one contains an `ActorReceiverController`, that subscribes for policy updates coming from the Learner Service, while the other loads an existing policy from a file, saved from a previous training phase.
  For receiving an agent's policy the implementation leverages Pika, a RabbitMQ client library for Python, by subscribing for both Predators and Prey policy updates.

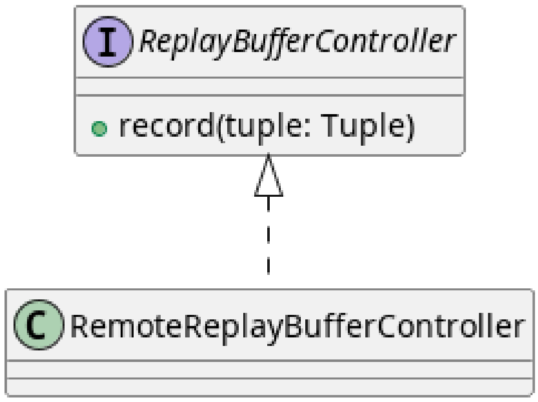

Figure 5: Buffer

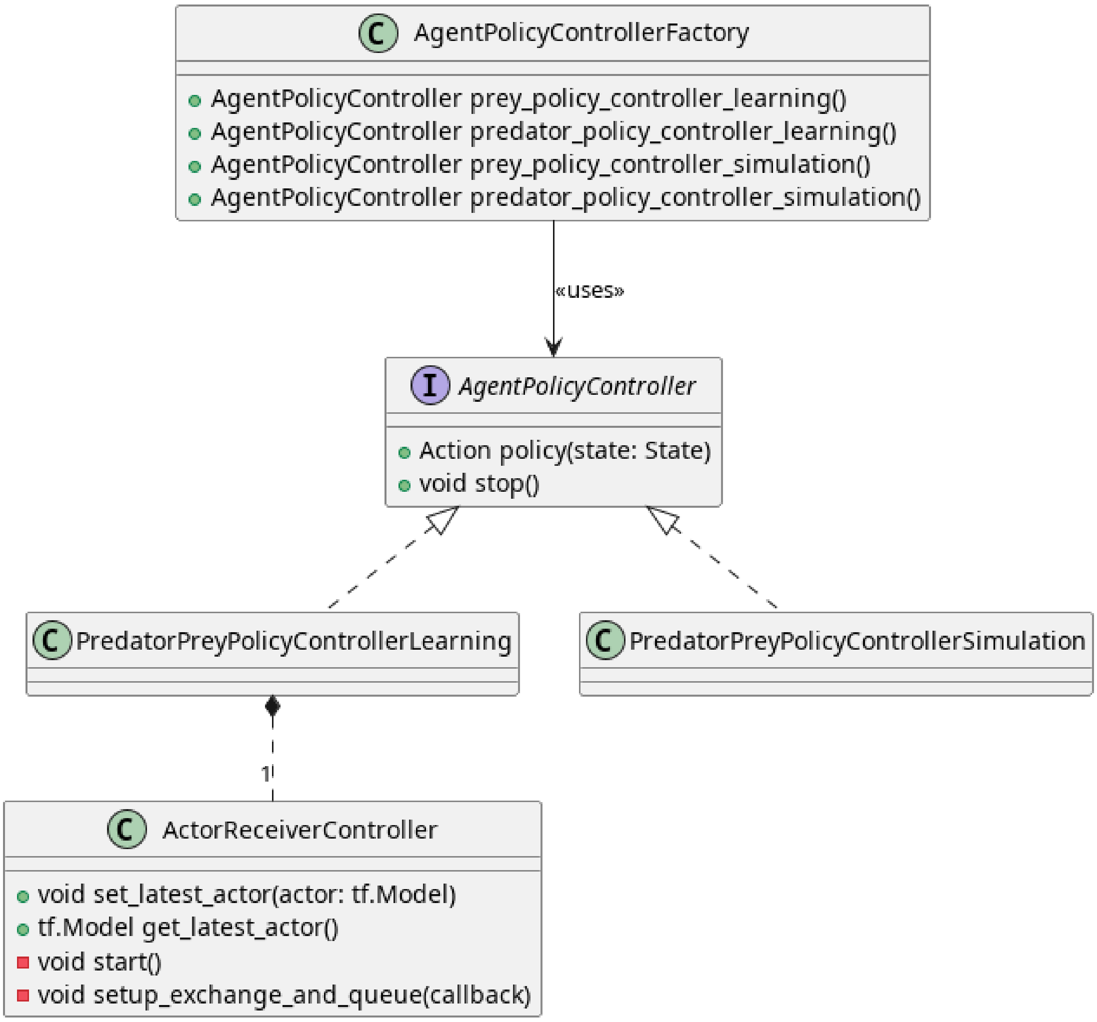

Figure 6: Policy

4.2 Learner Service

The Learner Service is responsible for training the agents belonging to the different MARL environments.

From the Learner's perspective, the environment is unique, as it samples data from the centralized Replay buffer where the agent's experiences are integrated into the same database.

Specifically, the Learner is based on the MADDPG [3] training algorithm, the multi-agent version of DDPG [2], which in turn uses an Actor-Critic model. The Actor-Critic structure decomposes the learning procedure into two parts:

- The Actor, given an agent's observation, decides which action should be taken, so it learns the so-called policy;
- The Critic, also known as the value function, evaluates how good the chosen action is to guide the Actor towards decisions that lead to higher expected rewards.

This algorithm relies on the **centralized learning** and **decentralized execution** framework. During training, all agents have a centralized Critic network that can observe the joint state of the environment and the joint actions of all agents. However, during execution, each agent can make its own decisions based on the learned Actor.

For this reason, the agent's policy update consists of sending the Actor model over the network, from Learner Service to each of the distributed Predator-Prey environments.

Moreover, this decomposition allows each environment to not depend on the Learner in Simulation mode, as each agent can operate with the latest Actor model received during the Training phase.

4.2.1 Design

The design of Learner Service follows the MVC pattern:

- Model contains the `Actor` and `Critic` network models and the configuration parameters;
- Controller comprises three controllers: `ReplayBufferController`, used for batch data from the Replay Buffer, `ActorSenderController` for sending the Actor networks and `LearnerController`, whose implementation is responsible for training the system.
- View only consists of the service's entry point.

4.2.2 Implementation

The controller's implementations are:

- `RemoteReplayBufferController` to batch data from the distributed Replay Buffer Service;
- `PubSubActorSenderController` to update the agents' policies using the Publish/Subscribe pattern. As for the Predator-Prey Service, it leverages the Pika library to publish messages;
- `MADDPGLearnerController` that implements the MADDPG algorithm, using Tensorflow library. The implementation is based on the project MADDPG-Keras.

The `Actor` and `Critic` models inside the Model instead use the Keras API.

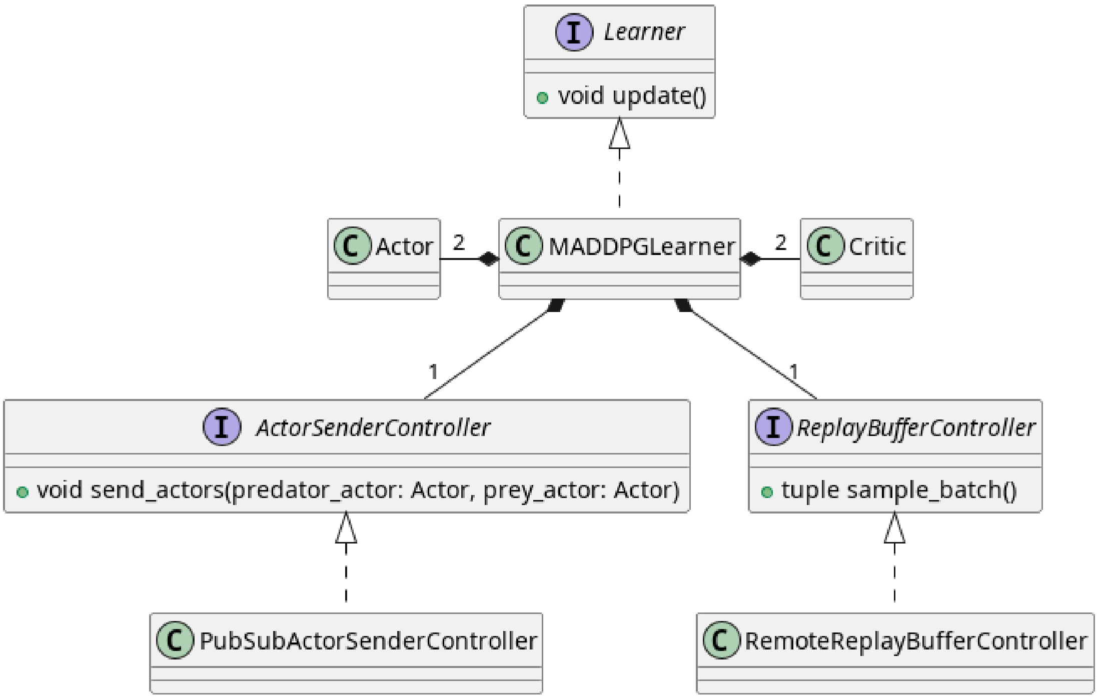

Figure 7: Learner class UML

4.3 Replay Buffer Service

The Replay Buffer Service is responsible for storing the agents' experiences inside a dataset, used for training a MARL environment.

It provides a simple and intuitive API to get and store data.

4.3.1 Design and Implementation

Replay Buffer Service's design is minimal. It comprises a `ReplayBufferService` class that manages the exposed API and a `DataBatchValidator` that verifies the received data batch has the correct structure.

In particular:

- The exposed API is an HTTP ReST API that allows to extract or record a data batch:
  - A HTTP POST request to `/record_data/` containing a data batch as Json records data to the replay buffer;
  - A HTTP GET request to `/batch_data/<size>`, where size is the number of rows to batch is used to extract data.
- `DataBatchValidator` verifies that the received data batch to store has a compliant structure. Each row should contain:
  1. The joint state: the observation of each agent;
  2. The joint action: the action of each agent;
  3. The joint reward: the reward of each agent after executing the respective action;
  4. The joint next state: the observation of each agent after executing the respective action.

Replay Buffer Service is implemented in Python3 using Flask for the HTTP API with Gunicorn for production deployment. The OpenAPI specification is available here.

4.3.2 Self Assessment/Validation

Replay Buffer Service contains tests to verify the HTTP API works as expected. They are implemented using Behave, a Python library for Behavior-Driven Development (BDD).

The scenarios test both `record_data` and `batch_data` APIs and are available at the following link.

5 Deployment

The System is deployable both in Train mode and Simulation mode, through a Docker Compose file that runs all microservices at once.

To do so, navigate to the Bootstrap repository and follow the provided instructions.

In Train mode, all the System's microservices are deployed with the following dependencies.

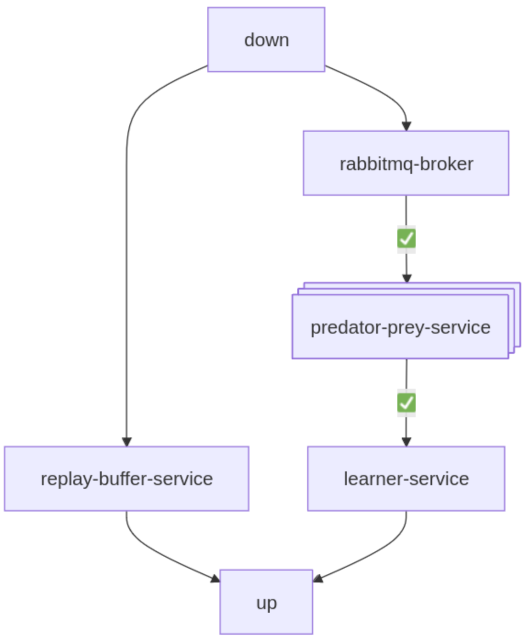

Figure 8: Train mode - Microservices deploy

In the `docker-compose` file, it's guaranteed they are deployed in this sequence by leveraging healthchecks and `depends_on` attributes.

In Simulation mode, only multiple `predator-prey-service`(s) are deployed. They are run in parallel with no dependency on each other.

6 Continuous Integration

In all the microservices repositories a Continuous Integration pipeline is built. The pipeline ensures the dependency installation is always successful and that the code is properly formatted.

The same CI/CD pipeline is adopted by all the microservices, with an exception for `replay-buffer-service` where tests are introduced in the `build` job to ensure they always pass. The CI workflows are the following:

1. `ci.yaml`. It is in turn composed of the following jobs:
    a) `build`: is responsible for checking if the project dependencies are successfully installed. A matrix of OS is provided: Ubuntu, MacOS, Windows;
    b) `format`: checks if the code format is correct leveraging Ruff formatter.
2. `deploy-image.yaml`: builds a Docker image of the microservice and publishes it to GitHub Packages;
3. `gh-pages.yaml`: builds the Code documentation of the microservice using Sphinx and publishes it to GitHub Documentation;
4. `release.yaml`: Produces a release when a Git tag is produces and publishes the code to GitHub Releases.

7 Experiments

The System has been trained for 7h 12m 3s, in Manjaro Linux 24.1 Kernel version 6.6.54-2, using a CPU Intel(R) Core(TM) i7-8550U @ 1.80GHz and 16GB RAM.

The Critic network loss plot has been generated and shown in the following figure and a Predator-Prey-Service-0 animation, comprising the latest 500 steps is available at the following link.

By visually analyzing the Predator-Prey animation and the Critic loss over time, it can be concluded that the algorithm doesn't converge. Both Predators and Preys Critic loss suddenly increases at nearly iteration 3000, and slowly decreases after that point. However, after $7h$ the loss is still nonzero. Moreover, the animation doesn't show any agent behavior: the predators seem to not chase preys, preys seems to not run away from predators.

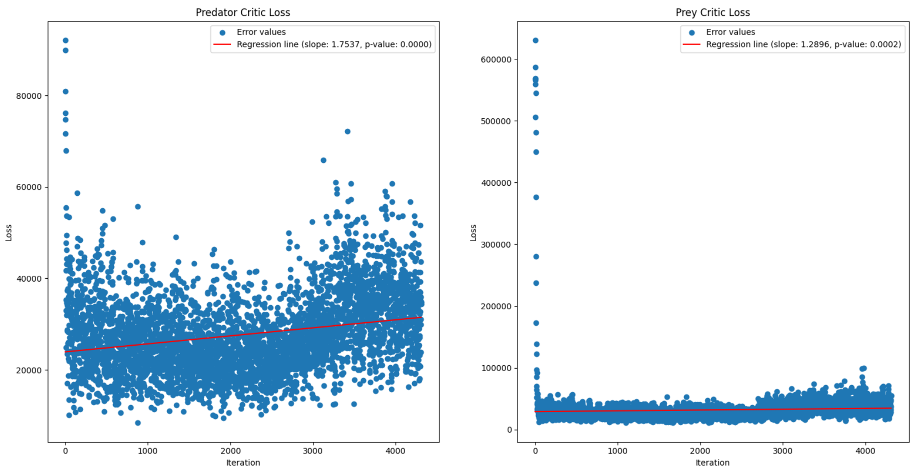

Figure 9: Predator and Prey Critic loss

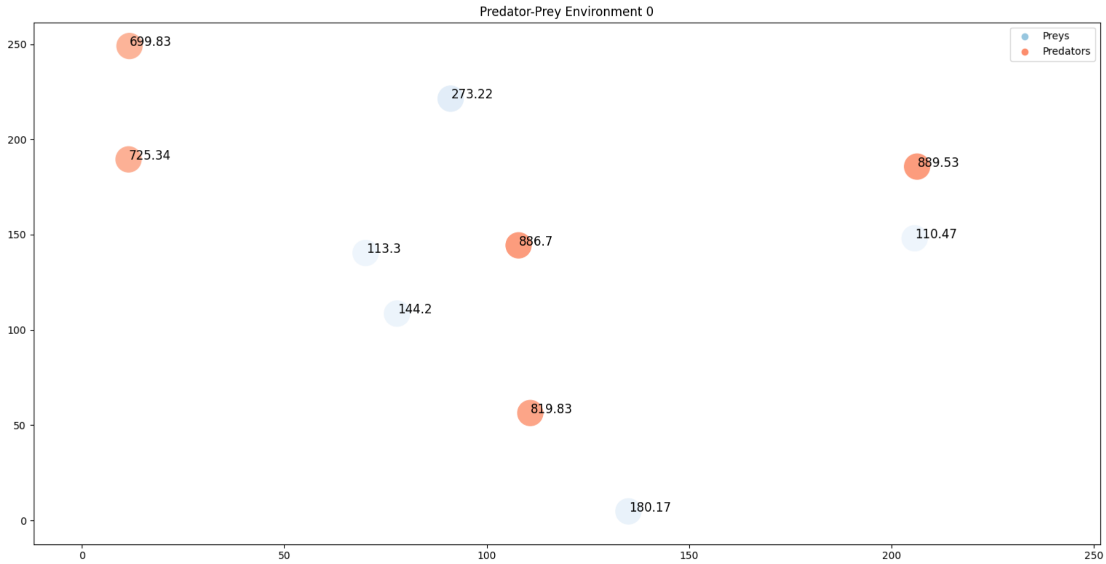

Figure 10: Animation frame

The non-convergence of the system may result from multiple factors:

1. Rate of collected data rows per time unit.
    - The agent's observation function, of both predators and preys is relatively complex: using the hardware setup listed before, each environment step takes nearly $\sim$ 30 seconds. For $\sim$ 7h of training and 5 environments in parallel, only $4, 2$k rows are collected inside the Replay Buffer. This data may be insufficient for the Learner to learn a definitive policy: to collect sufficient data the system should be trained for more time.
    - MADDPG implementation is based on the existing project MADDPG-Keras. The author states: "It takes around 20 hours to train 3 agents in 2 pursuer-1 evader environment for 3000 episodes (100 steps in each episode) on single i5-113G7 processor".
      Due to a lack of time and hardware resources, the student couldn't run the system for the time needed to collect 300k rows.

2. Algorithm implementation and parameters. Modifications had to be applied to the MADDPG project linked before. It cannot be excluded that the final implementation has some implementation errors. Indeed, the algorithm is difficult to test, so human errors may have been introduced. Moreover, parameters like the discount factor and the learning rate have been set to those proposed in the DDPG Pendulum example provided in the Keras website. These parameters should be further verified to be appropriate for this kind of environment.

8 Conclusions

Future experiments should be conducted by training the system in a dedicated machine and for more time. Indeed, as stated in the previous section, I had the impossibility to run the application for further time.

Help from domain experts is also well appreciated to verify the code is compliant with MADDPG algorithm. Implementing MADDPG has been proven challenging due to the lack of available open-source projects for comparison.

References

[1] Ruan de Kock et al. *Mava: a research library for distributed multi-agent reinforcement learning in JAX*. 2023. arXiv: 2107.01460 [cs.LG]. URL: https://arxiv.org/abs/2107.01460.

[2] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG]. URL: https://arxiv.org/abs/1509.02971.

[3] Ryan Lowe et al. *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*. 2020. arXiv: 1706.02275 [cs.LG]. URL: https://arxiv.org/abs/1706.02275.