UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Deep Learning

A suite for Non-Archimedean Lexicographic MultiObjective Reinforcement Learning

Project Documentation

TEAM MEMBERS:
Matteo Pierucci
Giacomo Pacini

Academic Year: 2022/2023

Contents

1 Introduction

1.1 Reinforcement Learning

Reinforcement learning is a branch of Machine Learning that focuses on develop autonomous Agents capable of making optimal decisions to achieve certain goals by interacting with the Environment around them.

It is one of the three main Automatic Learning paradigm, but it differs from Supervised and Unsupervised Learning because it deals with sequential decision-making problems, in which the action to be taken by the Agent depends on the current state of the system and determines its future state.

The main components of Reinforcement Learning are:

- **Environment**, it represents the external context or system in which the agent operates. It is responsible for defining the states, actions, and rewards that the agent receives. The Environment transition from one state to another is based on the agent's action, which is the one that maximizes long-term reward.
- **Agent**, it is the learner or decision-making entity that interacts with the environment. It observes the current state of the environment, selects the optimal decision based on its policy and interacts with the Environment executing an action. The Agent learning phase involves updating its knowledge by receiving rewards on taken decisions from the Environment. Its final goal is learning to make optimal decisions that lead to the maximum cumulative reward over time.

As is shown in 6, the Agent takes action based on current Environment state and reward, then the Environment evaluates Next Environment state and reward that are useful to the Agent for taking next action.

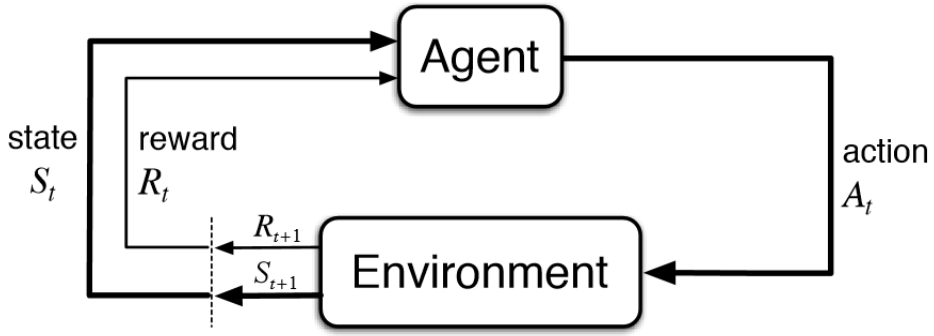


Figure 1: Reinforcement Learning Architecture

1.2 BAN: Bounded Algorithmic Numbers

Algorithmic numbers set is a subset of Non-Archimedean numbers which can be better manipulated by computers.

Definition 1 *$\xi \in E$ is an algorithmic number $\Leftrightarrow \xi = \sum_{k=0}^{l} r_k \alpha^{s_k}$ where $r_k \in R, s_q \in Q, s_k > s_{k+1}$*

In particular Bounded Algorithmic Numbers (BANs) are Algorithmic Numbers after a Truncation operation.

Definition 2 *Let* $P(x) = p_0 x^{z_0} + ... + p_m x^{z_m}, z_{i-1} < z_i, i = 1, ..., m.$
Then $tr_n[P(x)] := \begin{cases} P(x) & n \geq m \\ p_0 x^{z_0} + ... + p_n x^{z_n} & n < m \end{cases}$

Thus a Bounded Algorithmic Number is defined as follow.

Definition 3 $\xi = \alpha^p P(\eta), \xi \in E$ *is a BAN* $\Leftrightarrow \xi$ *is an AN and* $p \in R$

This kind of numbers have high modelling power, substantial informative value and their embedding is light and easy-to-manage.

1.3 Multi-objective Reinforcement Learning

Multi-objective Reinforcement Learning problems aim to find an Agent policy that considers more than one objective. It is a generalization of common optimization tasks where objectives are ordered by priority $f_1 \gg f_2 \gg ... \gg f_n$, this method is called Lexicographic approach.

1.3.1 Standard Scalarization

In order to use single-objective optimization algorithms a weighted scalarization is applied and weights are chosen "in accordance to" lexicographic priority.

Problem reformulation 1 $\min\limits_{\forall x \in D} \sum_{i=1}^{n} w_i f_i(x) \Leftrightarrow \min\limits_{\forall x \in D} w_1 f_1(x) + ... + w_n f_n(x)$

This approach has the following issues:

- no guarantee of problem equivalence
- weights choice is arbitrary and error-prone
- weights are chosen with a trial-and-error approach
- no guarantee to converge to global optimum

1.3.2 Non-Archimedean Scalarization

In this approach weights are chosen as $w_i = \alpha_{1-i}, i = 1, ..., n$

Problem reformulation 2 $\min\limits_{\forall x \in D} \sum_{i=1}^{n} f_i(x)\alpha^{1-i} \Leftrightarrow \min\limits_{\forall x \in D} f_1(x)\alpha^0 + ... + f_n(x)\alpha^{1-n}$

This approach solves issues of the previous one, in praticular:

- guarantee convergence to the optimal solution (if any)
- usage of (non-Archimedean) single-objective algorithms
- problem equivalence

This second approach can support MO Reinforcement Learning tasks.

1.4 Project goal

The goal of our work is designing a Python suite, capable of integrating both BAN and Reinforcement Learning library developed in Julia, which make Environment setup and Agent learning easier to the user.

We started from a Julia Library where mathematical operations between BANs and non-Archimedean multi-objective algorithms were implemented. From this starting point we integrate it in a Python Library that can load MO-Gym environments and call Julia methods to train both standard and non-standard Agents.

It will be explained in the following chapters implementation details and how the suite can be used.

2 Development challenges

2.1 Julia BAN Library

Julia BAN library allows the usage of BANs and supports arithmetic operations between them. It is used by Julia Agents implementations to perform non-standard learning process.

2.1.1 Handling different BAN sizes dynamically

As reward shapes are Environment dependant, our library needs to manage different BAN sizes dynamically. In particular there are three BAN libraries:

- **General implementation**, where data structures fit all BAN sizes
- **Dimension specific implementation**, where data structures are optimized for BANs made up of two and three components

Julia `BAN_SIZE` variable is set in Python Agent class considering `ban_size` parameter passed to constructor:

```
self._main.BAN_SIZE = self.ban_size
```

Then inside `/LMORL/LMORL/BAN/agents/DQN_Gym_BAN_Hybrid.jl` we dynamically manage BAN sizes, loading the specific implementation, if available, or the general one:

```
if BAN_SIZE == 3
    include("../BAN_s3_isbits.jl")
elseif BAN_SIZE == 2
    include("../BAN_s2_isbits.jl")
else
    SIZE = BAN_SIZE
    include("../BAN.jl")
end

```

2.1.2 Julia libraries calls to the method real() on BANs

Julia Libraries need a `real(::Ban)` method, which aims to return the real part of a BAN. But as a BAN is always composed by Real numbers, we assume that such method needs to simply return the BAN itself.

We execute some tests on `real()` applied both on Real, Complex and BAN inside `/LMORL/tests/julia-test-3_method-real-on-Bans/1_method_real_on_Ban.ipynb`.

You can find `real(::Ban)` method call inside `~\.julia\packages\Zygote\HTsWj\src\lib\broadcast.jl` in a function that evaluates derivative of quadratic absolute value:

```
@adjoint broadcasted (::typeof (abs2), x::Numeric) =
    abs2.(x), z -> (nothing, 2 .* real.(z) .* x)
```

We initially tried to overload that method inside the original Library without success. Then we decided to redefine it inside the three BAN Libraries `/LMORL/LMORL/BAN/BAN.jl`, `/LMORL/LMORL/BAN/BAN_s2_isbits.jl` and `/LMORL/LMORL/BAN/BAN_s3_isbits.jl`.

In the general implementation the method returns the real part of all the BAN components.

```
Base.real(a::Ban)  = Ban(a.p, real.(a.num), false)
```

In the dimension dependant implementations the method does the same thing, but as the Class fields are different, it operates on all fields of the object.

```
Base.real(a::Ban)  = Ban(a.p, [real(a.num1), real(a.num2)], false)
```

```
Base.real(a::Ban)  = Ban(a.p, [real(a.num1), real(a.num2), real(a.num3)], false)
```

2.2 Including Julia files from Python

In order to allow the use of the Julia implementation of the agents, the agent class in Python has to include the required Julia files.

We observed that the multiple instantiation of agents in Python lead to errors in the Julia environment namespaces.

In order to overcome this issue we modified the code so that the Julia `include` directive is run only if it was never run before. This is accomplished by
`(@isdefined DQNAgent) ? nothing : include("../../agents/DQN_Gym_BAN_Hybrid.jl")`

2.3 Efficiently call Julia methods from Python

In order to use Julia BAN library we need to efficiently call Julia code from Python. In fact, during Agent's learning, Julia methods are called multiple times, and if these call are inefficient execution time is heavily slowed. Then we tried 2 possible approaches using *PyJulia* library: one using Inline Julia call and another using PyJulia Main object.

2.3.1 Inline Julia Call

In this first approach we pass the parameters to Julia using `Main` object, then we call `_julia_eval()` method to execute the command specified by `cmd_string`.

```
def _add_experience(self, state, action_index, reward, next_state, done: bool):
    self._main.state = state
    self._main.action_index = action_index + 1
    self._main.reward_list = reward
    self._main.next_state = next_state
    self._main.done = done
    self._main.ban_size = self._main.BAN_SIZE
    after_passing = datetime.now()
    cmd_string = """
        reward_ban = parse_ban_from_array(reward_list, ban_size)
        action_index=convert(Int32, action_index)
        add_experience!(agent,state,action_index, reward_ban, next_state, done)
```

```
13      """
14      self._julia_eval(cmd_string)
```

This approach leads to poor performances: each *timestep* of a single *episode* takes about 1 second, and this amount of time increases over successive episodes. We suppose that the cause of this behaviour is related to the amount of allocated objects, required to use Inline Julia call, and the Garbage Collector which does not deallocate objects when they are no longer useful.

2.3.2 Main object Julia Call

As the first approach makes Agent learning phase unfeasible, we tried another approach which uses PyJulia Main object. In this approach we create a Python method `_add_experience()` which calls directly a Julia method called `add_experience_custom_types_b()` (line 4), that passed `state`, `reward`, `next_state` and other parameters.

```
1
2  def _add_experience(self, state, action_index, reward, next_state, done: bool):
3
4      self._main.add_experience_custom_types_b(self._main.agent,
5      state,
6      action_index+1,
7      reward,
8      next_state,
9      done)
```

The method `add_experience_custom_types_!()`, that is called from Python with `add_experience_custom_types_b()`, can call `add_epirience!()` method (line 9) of the Hybrid `DQNAgent` Julia class which update the Agent model.

```
1  function add_experience_custom_types!(
2      agent::DQNAgent,
3      state::StateType,
4      action_index::Int64,
5      reward_list::Array{Float32,1},
6      next_state::StateType,
7      done::Bool)
8
9      add_experience!(
10         agent,
11         state,
12         convert(Int32,
13         action_index),
14         parse_ban_from_array(reward_list, BAN_SIZE),
15         next_state,
16         done)
17 end
```

This second approach outperforms the first one, with a comparable overall execution time with Julia only implementation of the Agent.

2.3.3 Performance comparison

From the image below, it is evident that using the Julia method call to pass parameters with the Main object yields significantly better performance compared to executing `jl_eval()` with a command string parameter. Notably, the time-steps consistently maintain a constant execution time over successive episodes when employing the second approach. Conversely, when using the first method, the execution time of time-steps increases with each subsequent episode. This behavior can be attributed to the allocation of global Julia objects required for parameter passing to `jl_eval()`.

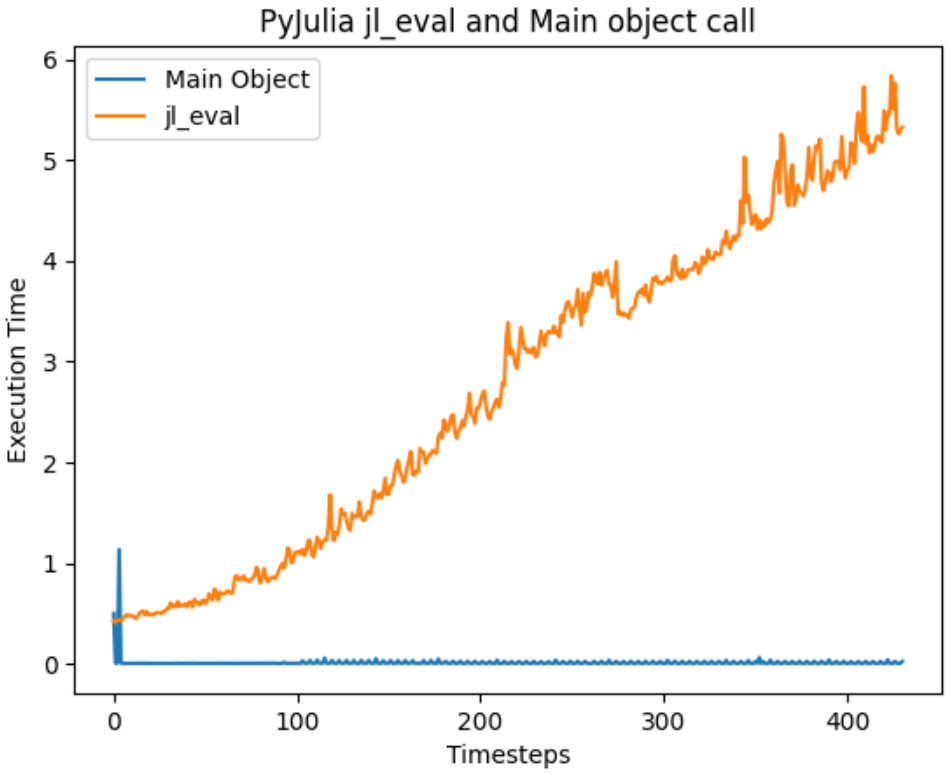
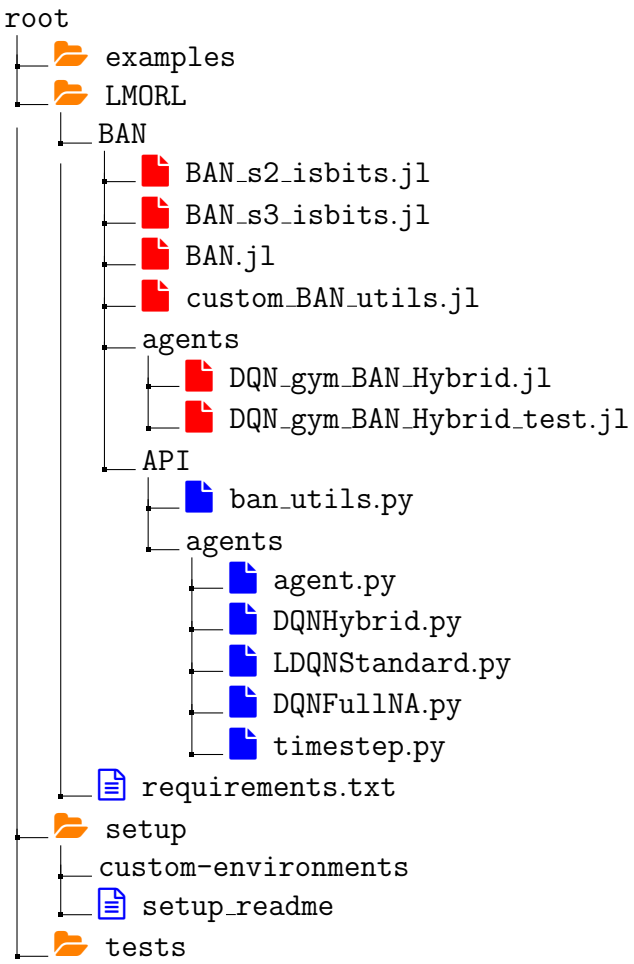


Figure 2: Timesteps execution time comparison

3 Library Structure



The library is made of four directories:

- **examples**: contains some examples of usage of the library.
- **LMORL**: contains the effective source code of the library.
- **setup**: contains some scripts and a read-me file with the instructions for the first installation of the library.
- **tests**: contains some development tests that were performed during the creation of the suite.

3.1 Library interface

The interfaces to the library are located at the path `LMORL/BAN/API`.

There we can find the folder `agents`, which contains the generic parent class `Agent` and various agents implementations.

All the agents implement the common interface inherited from the `Agent` class.

3.1.1 Agent class public methods

The public methods exposed by the agents of the library are:

- **init method**: sets the required parameters that are environment dependent (e.g. the size of the reward (and then of the BAN), the size of the state, the allowed actions) and the hyper-parameters of the agent that are shared for all the agent types.
- **act**: method that, given the current state, returns the action to perform according to the agent policy.
- **run_episode**: method that, given an environment, runs a set of actions according to the agent policy until episode termination. It collects the total reward of the episode, the number of time-steps, the execution time and can return also the rendering of the episode as a GIF image stored in a BytesIO object.
- **learning**:

  Input parameters:

  - **Environment**: the environment object on which the learning has to be performed. It must expose a `step` method.
  - **episodes**: the number of episodes that the learning must last.
  - **Replay frequency**: can be either None or a positive integer; if set to None, the learning is performed only at each episode end, if set to a positive integer, the learning is performed every specified value of time-steps.
  - **Dump period**: by default is set to None, if set to a positive integer, states how many episodes there must be between a dump of the model and the next.
  - **Model name**: string, the path where the model dump has to be stored.
  - **Early Stopping**: a callable that, if defined, must return True if the multi objective value of total reward for the current episode is considered high enough. This check is environment dependent and not mandatory. If the condition is satisfied a consecutive amount of times, the learning is stopped.
  - **THRESHOLD EXCEEDED CONSECUTIVELY**: the amount of consecutive times that an episode is considered a solution for the environment to trigger early stopping.
  - **render**: Boolean parameter that if set to true triggers the rendering in matplotlib of a frame for each time-step of the learning.
  - **verbose**: Boolean parameter that if it is set to true triggers the printing on std out of the statistics of each episode of the learning.

  The learning is performed on the agent for the specified number of episodes, and some statistics on the learning are returned, in particular:

  - **rewards**: it is a python list object, whose length is the same as the number of episodes, each element contains the total reward of each episode.
  - **avg_rewards**: it is a python list object, whose length is the same as the number of episodes, it contains the average of the reward for the last 100 episodes.
  - **timings**: it is a python list object, whose length is the same as the number of time-steps, each element is the time elapsed (in seconds) for the execution of

the time-step.

  - **full_infos_dict**: returns a dictionary made of the custom keys specified by the environment, for each key it returns a list of lists, in which each element is the content of the info for a time-step from the environment. Each list is the set of values returned for an episode.

- **dump_model_to_file**: method that, given the file path where to store the model, store the dump to a file.
- **load_model_from_file**: method that, given the file path from where to load the model, loads the dump from a file.

3.1.2 Implementing custom agent learning and accessing protected methods

Our implementation of the agent class exposes to the user only the method to trigger a learning session for the agent, but does not expose the methods to store a time-step information in the memory of the DQN Agent or to trigger the effective network update process by the replaying of the stored episodes.

This solution was chosen in order to implement an interface the most generic (and flexible) as possible, so that it can work with any type of agent.

By the way, it is possible to access the protected methods of the agent (that are, at most, agent-dependent), by defining a custom class that extends the chosen agent.

For example, if you want to define a custom learning process for a DQNHybrid agent, you can define a custom class MyDQNHybrid which extends DQNHybrid class and which has a public method `my_learning` to perform your custom checks and operations during the learning process.

3.1.3 The BAN (and reward) size

By exploiting the parameter `ban_size` the user can specify the size of the BANs to use inside the agent logic, and how many components of the reward have to be considered for the agent's tasks.

3.1.4 How the reward is managed

Since the library is made to handle multi objective rewards, the reward is considered to be always a list.

Only the first `ban_size` components of the reward are considered.

The reward is handled as a BAN for the effective agents' logics in Julia, while in Python is always handled as a list of floats, since the only operations between rewards that are made in Python are the sum and the assignment.

Every time that a reward is passed to the Julia code, a custom function is called, which is in charge of parsing and converting the list of floats to a BAN of the right size.

3.1.5 Plots & rendering utilities

The library also exposes some ways to easily visualize the results of the learning tasks, in particular there is a method to plot a list of BANs in function of the episode number, so that the user of the library can easily see the trend of the reward during the learning.

3.1.5.1 Plotting BANs

That method is in `LMORL.API.ban_utils.Ban.display_plot()`, and its interface accepts as input parameters:

- **rewards**: rewards must be a list of lists, where each element of the parent list is a MO reward and each element of the child list is a component of a reward (which is considered float). Each MO reward is assumed to have the same number of components
- **num episodes**: is the number of episodes to plot
- **title**: is a custom string label for the plot.

That method invokes the plotting of the generated graph and returns the MatPlotLib Figure object generated.

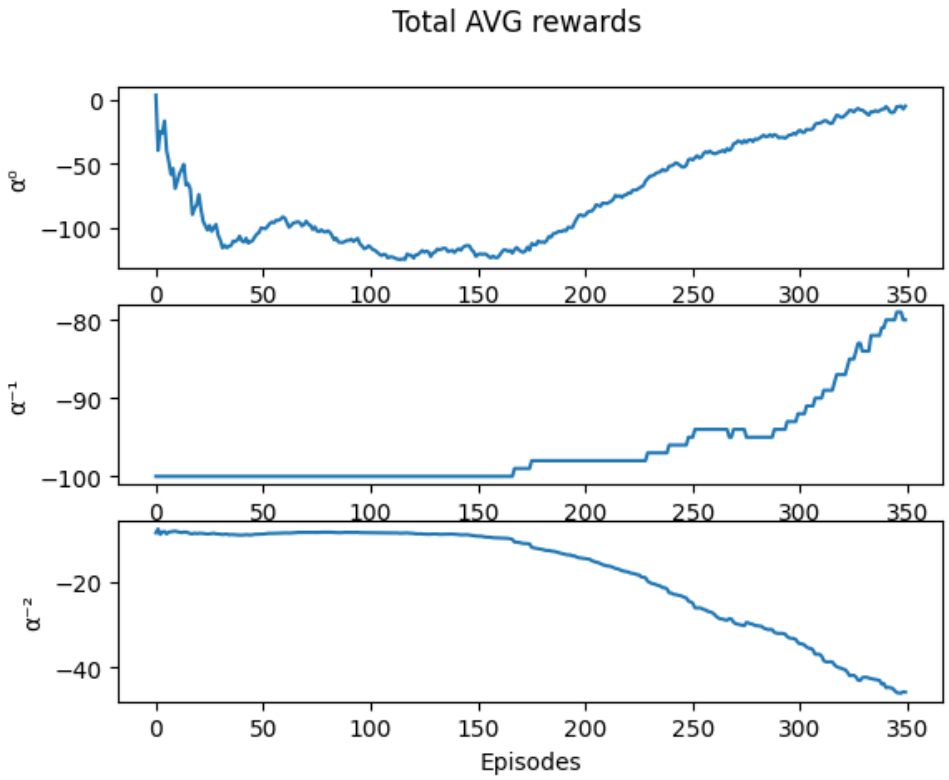


Figure 3: BAN plot example

3.1.5.2 Plotting Epsilon values

There is also a method that shows epsilon parameter behavior during successive episodes. In particular that method is in `LMORL.BAN.API.agents.DQNHybrid.plot_epsilon_values()`, and its interface accepts as input parameter:

- **num episodes**: is the number of episodes to plot.

That method invokes the plotting of the generated graph and returns the MatPlotLib Figure object generated.

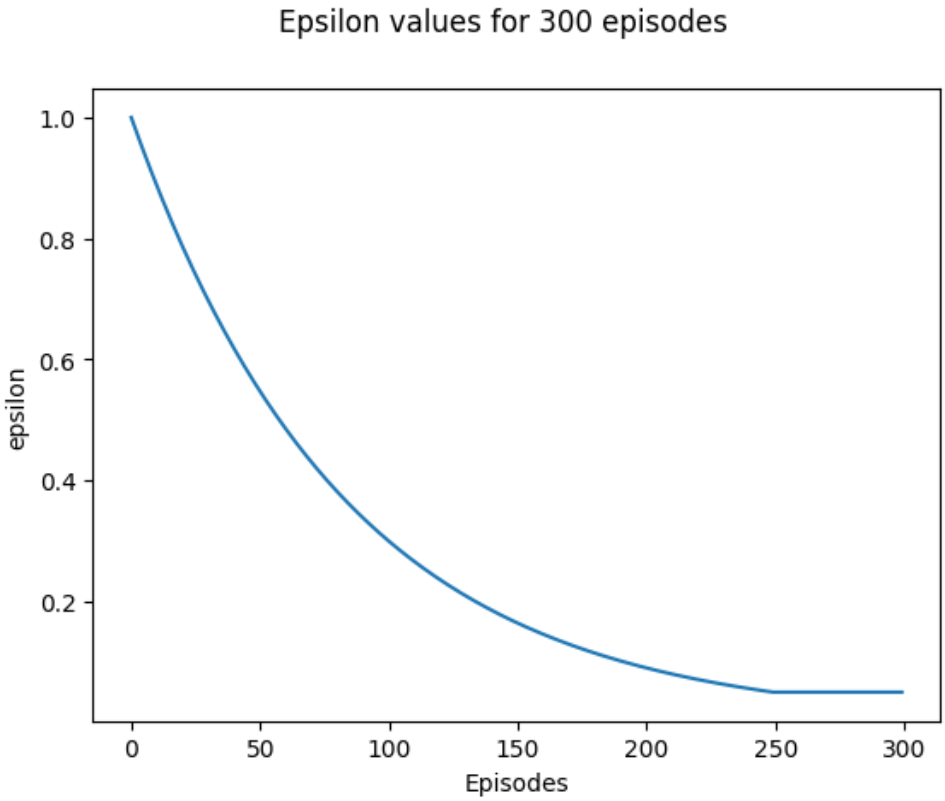


Figure 4: Epsilon plot example

3.1.5.3 Calculating sliding window smoothing average on a sequence of BANs

The method `LMORL.API.ban_utils.Ban.averaged_sequence()` accepts as parameters:

- `sequence`: the sequence of BANs (or in general of vectors) of which it has to be calculated the smoothed sequence. It is required that all the elements of the sequence have the same number of components.
- `window_size`: the width of the sliding window. The sliding window average is calculated on the current element of the sequence and on the previous `window_size` elements at most.

It returns the averaged sequence of vectors.

This method allows to calculate an averaged sequence with a custom window size on a user-specified sequence.

3.2 Library implementation

As already seen, this library exposes different agents that are implemented in different ways; in particular the agents **DQNHybrid** and **DQNFullNA** are based on a Julia implementation of the logic, and their model are stored in Julia too.

The Python class for those models exposes an interface to the effective Julia implementation.

On the other hand the agents can be implemented also completely in Python, how it is done for the **LDQNStandard** agent, whose network and logic is in Python and in particular exploits PyTorch for the NN.

4 Installation and first setup

This library requires multiple dependencies both from other Python packages and both from Julia packages.

4.1 Julia required environment and packages

Since some Python packages will need a working Julia environment in order to be correctly setup, you have to install a Julia interpreter globally. The suggested version is 1.9.
https://julialang.org/downloads/

4.1.1 Julia dependencies installation

To install the required dependencies for Julia, you can run the script *julia-dependencies.jl* located in the `/setup/` folder.

```
julia julia-dependencies.jl
```

4.2 Python dependencies

In `LMORL/BAN/requirements.txt` there are the Python packages dependencies, that can be installed by running `pip install -r requirements.txt`.

4.2.1 Julia Python interface first setup

To install the required dependencies for Julia to be run from Python, you can run the script *julia-api-installation.py* located in the `/setup/` folder.

```
python julia-api-installation.py
```

4.2.2 Mo Gymnasium Box2D dependencies

In order to use the Lunar Lander Gymnasium environment, you have to run `pip install gymnasium[Box2d]`, which on some configurations can lead to missing Visual C++ dependencies error.

You can find the right version of Visual C++ for your system at
visualstudio.microsoft.com/visual-cpp-build-tools/

4.3 Creating custom Gymnasium environments

Gymnasium offers a standard interface for environments definitions, and allows to design and build custom environments, that can eventually be loaded in the same way as the native ones.

4.3.1 Example custom environments

We added to our repository two custom environments based on the Lunar Lander v2 single objective environment and that offer multi-objective rewards.

The custom environments are located in `/setup/custom-environments`.

4.3.2 How to find gym library directory

In order to add a custom environment to the Gymnasium library you have to find the Gymnasium installation folder.

If your gym installation is global, you will find gym envs folder in Python's site-packages folder.

You can print a package folder by using this Python code (in the example it prints the path for gym):

```
import os
import gym
print(os.path.abspath(gym.__file__))
```

4.3.3 Adding the environment to Gym

In order to create a custom environment you need to put .py environment file inside `PATH_TO_PYTHON_GYM_LIBRARY/gym/envs/box2d` for the case of Lunar Lander environments, since are based on the extension Box2D.

4.3.4 Registering an environment

Then edit the `__init__.py` inside `PATH_TO_PYTHON_GYM_LIBRARY/gym/envs` as follow to create a new registration entry:

```
    register(
        id="LunarLander-v2-mo-custom",
        entry_point="gym.envs.box2d.lunar_lander_mo_custom:LunarLander",
        max_episode_steps=1000,
        reward_threshold=200,
    )

    register(
        id="LunarLander-v2-mo",
        entry_point="gym.envs.box2d.lunar_lander_mo:LunarLander",
        max_episode_steps=1000,
        reward_threshold=200,
    )
```

Eventually you can import your custom environment in Python or Julia using the `id` of the registration entry:

```
    Julia   ->  my_env = GymEnv("LunarLander-v2-mo-custom")
    Python  ->  my_env = gym.make("LunarLander-v2-mo-custom")
```

5 Usage example with Lunar Lander Gymnasium environment

In the following it is reported an example of usage of the library from scratch, with the various phases needed, starting from the load of the environment until to the plotting of the results.

5.1 Importing the agent from the library and loading an environment

Since the library is not required to be installed on the system, it is needed to just add to the system path the path to the `LMORL/` folder.

Once added LMORL to path, you can import an agent from the ones exposed at `LMORL/BAN/API/agents/` and load an environment that exposes the standard **Gymnasium** API. In the example it is loaded our custom version of the Multi Objective Lunar Lander environment.

In order to allow the rendering of the environment, some environments requires to specify the `render_mode`.

```
import sys
import os
from pathlib import Path

root_dir = Path(os.getcwd())

if str(root_dir.parents[1]) not in sys.path:
  sys.path.append(str(root_dir.parents[1]))

from LMORL.BAN.API.agents.DQNHybrid import DQNHybrid

import gym

env = gym.make("LunarLander-v2-mo-custom", render_mode="rgb_array")
```

5.2 Instantiating and setting the hyper-parameters of the agent

Each agent can expose different user-definable hyperparameters, depending on its type and requirements.

All the agents of our library let the user specify the number of components of the reward to consider for the reinforcement learning task via the `ban_size` parameter.

In the example below the agent instantiated is DQN Hybrid with gradient clipping enabled.

```
input_size = env.observation_space.shape[0]
num_actions = int(env.action_space.n)
action_space = list(range(env.action_space.n))
```

```
learning_rate = 0.001
epsilon_decay = 0.995
epsilon_min = 0.1
batch_size = 64
hidden_size = 128
BAN_SIZE = 3
max_memory_size=10000
train_start = max_memory_size
use_clipping = True
clipping_tol = 1.0

agent = DQNHybrid(input_size=input_size, num_actions=num_actions,
                  action_space=action_space, learning_rate=learning_rate,
                  epsilon_decay=epsilon_decay, epsilon_min=epsilon_min,
                  batch_size=batch_size, hidden_size=hidden_size,
                  ban_size=3, max_memory_size=max_memory_size,
                  train_start=100, use_clipping=use_clipping,
                  clipping_tol=clipping_tol)
```

5.3 Running the learning of the agent

Via the public method `learning()` exposed by all the agents, it is triggered the agent learning phase.

It is possible to specify how many episodes the learning has to last, and if the learning has to be performed only at episode end or every N time-steps.

The `verbose` parameter set to False in the learning method states that the library will not output the results of each episode of the learning (time elapsed, total reward of the episode, average reward of the last episodes, number of time-steps of that episode).

In addition, it is possible to define a custom callable that will be invoked at every episode end and will check if the total reward obtained in that episode is enough to consider that episode run a solution for the environment. In the code example below, the episode is considered solved if the sum of the components of the total reward of the episode is above 200. This depends on the environment's reward definition and properties.

```
EPISODES = 350
REPLAY_FREQUENCY=None
mname = "dumpDQNHybrid.model"

def early_stopping(reward : list) -> bool:
    if(sum(reward) >= 200):
        return True
    return False

total_rewards = []
total_avg_rewards   = []
total_timings = []

```

```
14
15 rewards, avg_rewards, timings, infos_lists = agent.learning(
16     env=env,episodes=EPISODES,
17     replay_frequency=REPLAY_FREQUENCY,
18     mname=mname, verbose=False,
19     early_stopping=early_stopping)
20
```

5.4 Plotting the rendering of the time-steps of an episode

Via the method `run_episode()` of the agent, it is possible to run a complete episode on the environment using the trained agent with just one line of code.

The method returns the stats about that episode run and the rendered GIF image as a Python `BytesIO` object.

This method allows to set a title for the generated GIF.

The code below is for displaying in a Jupyter Notebook the animated GIF rendered by the episode run.

```
1  from IPython.display import Image as IpyImg
2
3  %matplotlib
4
5  title=f"After {EPISODES} episodes"
6  total_reward, num_timestep, elapsed_episode, gif_file = agent.run_episode(
7      env, title=title,
8      render=False, verbose=False)
9
10
11 display(IpyImg(data=animated_gif_file.getbuffer(),
12     format='png'))
```

5.5 Plotting a series of MO rewards as BANs

Another useful feature exposed by the library is the possibility to plot a series of rewards vectors interpreted as BANs.

Even though, as already stated, internally the library treats the MO rewards as lists of floats by the side of the shared agent interface, those rewards are used as BANs for the agent logic, and then can be plotted as BANs via the method `display_plot()` of the python object `Ban`.

Note: the Jupyter Notebook directive `%matplotlib inline` allows to display the plot in the notebook output instead in a detatched window.

```
1  from LMORL.BAN.API.ban_utils import Ban
2
3  %matplotlib inline
4
5  r = Ban.display_plot(total_rewards, len(total_rewards), "Total rewards")
```

5.5.1 Example of output

In the following image it is reported an example of plot of a rewards series, on *LunarLander-v2-mo* environment using *DQNHybrid* agent with gradient clipping with a memory size of 100.000 time-steps.

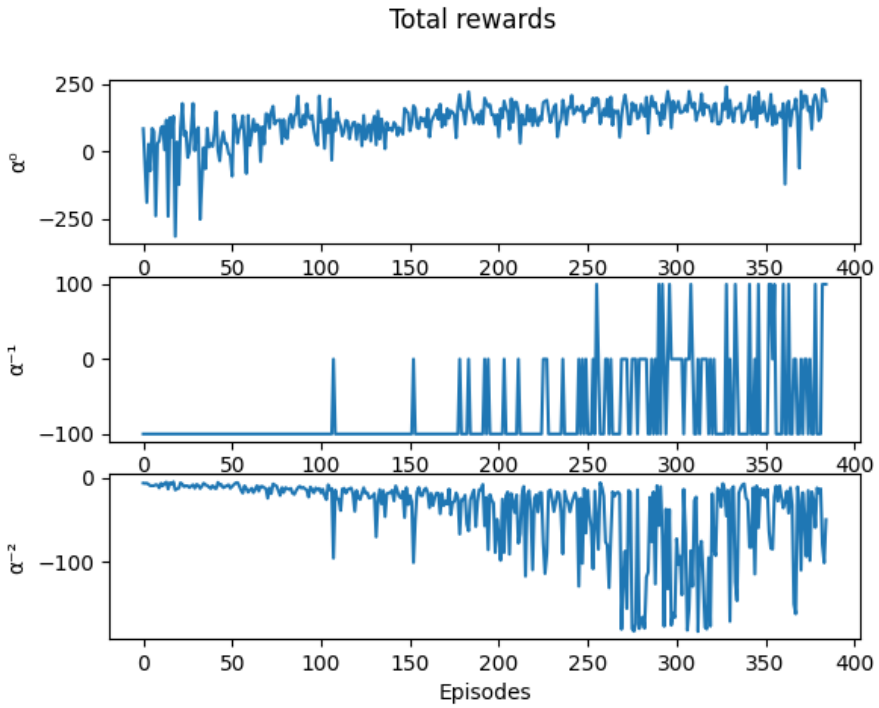


Figure 5: Reward behaviour during episodes

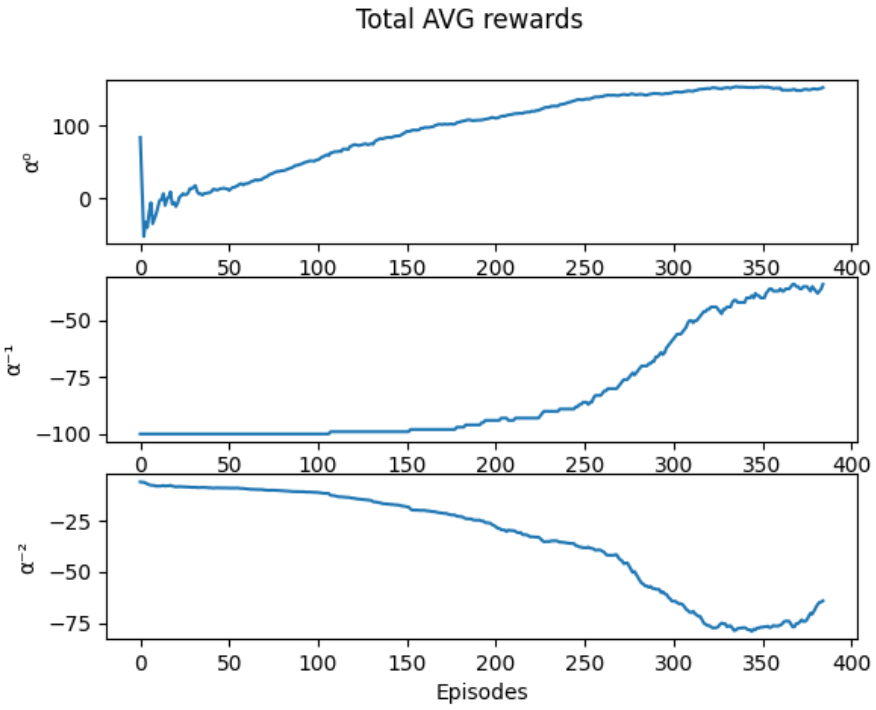


Figure 6: Average reward over last 100 episodes series

6 Appendix: custom environments definitions

For the purpose of testing the performance of the MO agents with a known environment we adopted a couple of modified versions of the original single-objective Lunar Lander environment.

6.1 Lunar Lander general information

6.1.1 Observation Space

The state is an 8-dimensional vector: the coordinates of the lander in ‘x‘ and ‘y‘, its linear velocities in ‘x‘ and ‘y‘, its angle, its angular velocity, and two booleans. that represent whether each leg is in contact with the ground or not.

6.2 LunarLander-v2-mo

This version is located at `setup/custom-environments/lunar_lander_mo`.

In this version the reward is made of three components, and in particular the components are defined in the following way:

- **First Component**: is calculated as

  $-100\sqrt{x^2+y^2} - 100\sqrt{\partial x^2+\partial y^2} - 100\angle + 10l + 10r$

  Where l and r are boolean values stating whether respectively the left and the right legs of the lander are touching the ground, while $\angle$is the angle between the lander and the horizontal axis, ∂x is the x-axis velocity of the lander

- **Second Component**: states if the lander has landed, crashed, or is flying. In case the lander crashed, its value is -100, in case it landed, its value is +100, in case it is flying, its value is 0. The lander is considered flying if its legs are not bot touching the ground or if its propellers are not off.

- **Third Component**: considers the fuel consumption; in the case of the discrete action space, the motors can only be on or off. This component of the reward is -0.3 if the main engine is on, else if one of the side engine is on its value is 0.03. In case none of the engine is on, its value is zero.

It is possible to note that the fact that the lander landed or not does not infere with the value of the first component of the reward, but only with the second one.

This implies the fact that an agent trained on this environment will first learn to fly correctly, and after will learn to land. The fuel consumption is considered only marginally.

6.3 LunarLander-v2-mo-custom

This version is a minor modification to the one previosly explained, in particular the only difference is that the reward of +100 in case of landing and of -100 in case of crashing is added to the first component too, so that it gets more considered by the agent during the learning.

This takes to making the first component of the reward very complete, and so makes the problem near to the single-objective one.

6.4 mo-lunarlander-v2

This is the official implementation of the lunar lander in a multi-objective environment.

In this implementation the first component of the reward is the landed component, the one whose values can be either -100, 0 or +100.

The other components (that reward the ability to fly well of the agent) are demanded to the second field of the reward.

7 Appendix: implemented agents

7.1 DQN Hybrid

This agent is based on the Deep Q Network approach with an hybrid network.

In particular this means that the network of this agent is made of standard components with the exception of the last layer, which is made of BANs.

This allows to handle NA MORL tasks, and impacts on the performances in a lower fashion compared to a Full NA implementation.

7.1.1 Implementation

The implementation of the network and of the methods to update and exploit the network are written in Julia, in order to grant higher performances and compatibility with BANs.

The Python class exposed by our library `DQNHybrid` is in charge of loading the right Julia modules and calling the methods of the Julia implementation.

7.1.2 Source of the code

We started our implementation on the work of Giulio Silvestri for his thesis, that can be found at *https://github.com/LoreFiaschi/DeepLearning*

7.1.3 DQN Memory management

As all the DQN implementations, also the DQN Hybrid needs a memory where to store the tuples of state, action, reward and next_state. In addition, also the Boolean `done` value is stored.

Even though the Python agent class is made so that the previous states can be stored there too, the experience is saved directly in Julia, just where it will be exploited in the phase of replay.

Efficiently handling memory

We implemented a way to store in the memory the previous episodes without the need of allocating every time a new Replay object.

In order to reduce the memory overhead we choose to exploit a fixed memory position array.

```
function add_experience!(agent::DQNAgent,state::StateType,action::Int32,
        reward::Ban,next_state::StateType,done::Bool)

    if agent.occupied_memory < agent.max_memory
        r=Replay(state,action,reward,next_state,done)
        push!(agent.memory, r)
        agent.occupied_memory+=1
        agent.last_inserted_index += 1
    else
```

```
        agent.last_inserted_index=( (agent.last_inserted_index ) % 100 ) + 1
        agent.memory[agent.last_inserted_index].state = state
        agent.memory[agent.last_inserted_index].action = action
        agent.memory[agent.last_inserted_index].reward = reward
        agent.memory[agent.last_inserted_index].next_state = next_state
        agent.memory[agent.last_inserted_index].done = done

    end

end
```

7.1.4 Accepting both discrete or continuous states

As there exist environments using discrete states and continuous states, we implemented ad Julia Union object called `StateType`, which accepts both `Int` and `Float` arrays.

```
StateType=Union{Array{Float32,1},Array{Int32,1}}
```

Then we edit `act!()` and `add_experience!()` methods to accepts `StateType` instead of `Array{Float32,1}` (highlighted lines of code).

```
function act!(agent::DQNAgent,observation::StateType)

function add_experience!(
        agent::DQNAgent,
        state::StateType,
        action::Int32,
        reward::Ban,
        next_state::StateType,
        done::Bool)
```

As a result our Library is capable of working with Environments using both discrete and continuous states.

7.1.5 Enabling gradient clipping

Gradient clipping is a technique used in machine learning and deep learning models to address the problem of exploding gradients during training. When training deep neural networks, gradients can sometimes become extremely large, leading to unstable training and slow convergence.
Gradient clipping imposes a limit or threshold on the magnitude of gradients. If any gradient exceeds the threshold, it is rescaled or clipped to ensure it stays within a predefined range. We implemented a gradient clipping method that uses a clipping tolerance set by the user.

First we developed a `clipvalue_check()` method, which execute the gradient clipping considering tolerance. Then we implemented two overloading functions `clipvalue(a::Ban, tol::Real)` and `clipvalue(A::AbstractArray{Ban}, tol::Real)`, that execute gradient clipping operations for different `BAN_SIZE` values using `clipvalue_check()`.

```
@inline function clipvalue_check(a::Float64, tol::Real)
    return (abs(a) >= tol) ? (a>0 ? Float64(tol) : -Float64(tol)) : a
```

```
end

function clipvalue(a::Ban, tol::Real)
    if BAN_SIZE == 3
        num = (a.num1, clipvalue_check(a.num2, tol), clipvalue_check(a.num3, tol))
    elseif BAN_SIZE == 2
        num = (a.num1, clipvalue_check(a.num2, tol))
    else
        num = []
        num[1] = a.num[1]
        for index in 2:lenght(a.num)
            num[index] = clipvalue_check(a.num[index], tol)
        end
        #num = clipvalue_check.(a.num, tol)
        #num[1] = a.num[1]
    end
        p, num = BAN.to_normal_form(a.p, num)
        return Ban(p, num, false)
end

function clipvalue(A::AbstractArray{Ban}, tol::Real)
        B = similar(A)

        for i in eachindex(B)
                B[i] = clipvalue(A[i], tol);
        end

        return B
end
```

Eventually, `DQN_Hybrid` class constructor has been updated to accept also `use_clipping`, which enable gradient clipping and `clipping_tol`, which sets clipping tolerance.

```
class DQNHybrid(Agent):

    def __init__(self, input_size : int,
                 num_actions : int, action_space,
                 learning_rate : float,
                 epsilon_decay : float,
                 epsilon_min : float,
                 batch_size : int,
                 hidden_size : int,
                 ban_size : int,
                 discount_factor : float = 0.99,
                 max_memory_size: int = 100,
                 train_start: int = 100,
                 use_clipping : bool = False,
                 clipping_tol : float = 1.0,
                 use_legacy : bool = False) -> None:
```

In the end, in the method `experience_replay` of the agent we added a block of code to exploit the clipping strategy that is run only if `use_clipping` was set to true in the agent object.

```
for d in data
    gs = gradient(ps) do
        loss(d...)
    end
    if agent.use_clipping
        a=0
        for i=1:length(ps)
            a+=length(gs[ps[i]])
        end
        f=Vector{Ban}(undef, a)
        copy!(f,gs)
        f=clipvalue(f,agent.clipping_tol)
        copy!(gs,f)
    end
    Flux.Optimise.update!(opt, ps, gs)
end
```

7.2 LDQN Standard

This agent was first proposed by Alessandro Abate in 2020, it is based on the concept of adopting a deep Q-network in which the last layer is made of a matrix, where in each line you have the multi-objective reward expected for a certain action starting from the given state.

This approach enables to adopt a MO reward also without using BANs.

We started to implement this version from the Python implementation of the agent provided us by Ing. Lorenzo Fiaschi.

This version adopts a network in PyTorch and is made for a reward of three components.

7.2.1 Adopting the same interface

Since our aim is to provide a standard interface to use many agents developed also with different languages and technologies, we focused on adapting the code so that the original methods of the agent are called from the same interface methods defined in our parent class `Agent`.

We modified the code so that, even though internally the agent needs `Tensor` objects to work with, the interface methods will receive standard types (e.g. Python `list`) parameters.

In the following is reported one example for the protected method `_add_experience`:

```
def _add_experience(self, state, action, reward, next_state, done: bool):
    state = torch.tensor(state, dtype=torch.float32,
            device=self.device).unsqueeze(0)
```

```
4
5       next_state = None if done else torch.tensor(next_state, dtype=torch.float32,
6               device=self.device).unsqueeze(0)
7
8       reward = torch.tensor(reward[:self.reward_size], device=self.device)
9       # NOTE: in order to obtain the right shape for the reward tensor,
10      # we have to keep only the first reward_size
11      # components of the reward vector
12
13      self.__add_experience(state, action, reward, next_state)
```

As we can observe in the code above, the reward is converted from list to PyTorch Tensor inside the method, in a trasparent fashion to the user.

7.2.2 Allowing user-defined size reward

In order to allow the user of our library to specify the size of the reward also for this agent, we introduced some modifications to the code. In particular:

- **Method** `_add_experience`: in this method (reported above) only the firsts `reward_size` components are kept while converting the reward from list to Tensor.
- **Method** `__get_q_value`: in this method the array of actions is vertically stacked `reward_size` times, in order to match the shape of the output layer of the Q Network.
- **Method** `__experience_replay`: as in the previous point, also here there is the need to match the shape of the output layer of the Q Network, and this is achieved by repeating `reward_size` the array of actions.

7.3 DQNFullNA

This is an agent for which we implemented only the interface, but not the methods.

It is very similar to DQN Hybrid, the only difference relies in the fact that all its network is made of BANs.

8 Conclusions

We developed a comprehensive suite that provides a seamless and user-friendly experience for utilizing both standard and non-standard agents within environments that adhere to the OpenAI Gymnasium API. Our goal was to create a solution that simplifies the process of working with multi-objective agents and environments, making it accessible to a wider range of users.

One of the key advantages of our suite is its ability to optimize the utilization of resources. By carefully optimizing the underlying code and algorithms, we have achieved remarkable performance levels that are comparable to running native Julia code. This means that users can expect the same level of efficiency and execution speed using native Julia code or Julia code wrapped inside our library.

Another notable feature of our library is its flexibility and extensibility. We have designed the suite in such a way that integrating new agents is straightforward and hassle-free. The suite provides a common Agent interface that serves as a foundation for developing and incorporating new agents into the system. This approach eliminates any technological or language constraints, allowing users to leverage their preferred tools and languages while seamlessly integrating new agents into their workflows.

Moreover, our suite is built with a focus on usability and accessibility. We understand the importance of creating a user-friendly environment, particularly when working with complex agents and environments.

In summary, our suite is a powerful and versatile tool that empowers users to work with agents, both standard and non-standard, within OpenAI Gymnasium-compatible environments. Its resource optimization capabilities ensure high-performance execution, rivaling the native execution of Julia code. Additionally, its flexibility and lack of technological or language constraints make it effortless to integrate new agents, all while maintaining a user-friendly and accessible experience.