MLPro Documentations

Release 1.0.0

FH-SWF AT

Mar 20, 2023

INTRODUCTION

MLPro is a synoptic framework for standardized machine learning tasks in Python!

MLPro was developed in 2021 by Automation Technology team at Fachhochschule Südwestfalen.

MLPro provides complete, standardized, and reusable functionalities to support your scientific research or industrial project in machine learning.

In MLPro, we provide a standarized Python package for conducting research in reinforcement learning (RL) and game theoretical (GT) approaches, including environments, algorithms, multi-agent RL (MARL), and many more. Additionally, we enable our users to reuse the available packages by developing wrapper classes.

Moreover, MLPro focuses not only on Model-Free but also Model-Based RL problem.

Github repository: https://github.com/fhswf/MLPro.git

CHAPTER

ONE

MAIN CONTRIBUTIONS

- Test-driven development
- Clean code
- Ready-to-use functionalities
- Usability in scientific, industrial and educational contexts
- Extensible, maintainable, understandable
- Attractive UI support
- Reuse of available state-of-the-art implementations
- Clear documentations

CHAPTER

TWO

INSTRUCTIONS FOR USE

2.1 What is MLPro?

Add text here!

2.2 Getting Started

Add text here!

2.2.1 Installation

Requirements

- python version 3.0 or higher

Install package from PyPI

MLPro is listed in PyPI, thus you can install MLPro library by executing:

```
pip install mlpro
```

Install package from GitHub

MLPro is also available in GitHub, where you are welcome to access MLPro package as well as contribute for the further improvements.

```
pip install git+https://github.com/fhswf/MLPro.git
```

Alternatively, you can also directly clone MLPro repository into your desired directory.

```
git clone https://github.com/fhswf/MLPro.git
```

2.2.2 Hello World

Add text here!

2.3 Architecture

Add text here!

2.4 Dependencies

Add text here!

2.5 Mathematics

MLPro provides a module that consists of common machine learning functionalities and properties. Additionally, you can also find modules for basic mathematical classes and various classes with elementry functionalties to be reused in higher level classes, for example: logging, load/save, timer, data storing, etc.

We provide *how to files* related to this subject.

This module includes Dimension, Set, Element, ElementList, MSpace, and Espace classes.

```
from mlpro.bf.math import *
```

Additional module includes load/save, logging, set timer, data storing, data loading, and data plotting functionalities.

```
from mlpro.bf.various import *
```

Add text here!

2.6 Machine Learning

This module includes hyperparameter setting, hyperparameter tuning, adaptive, and buffer classes.

```
from mlpro.bf.ml import *
```

Add text here!

2.6.1 Adaptivity

Add text here!

2.6.2 Hyperparameter Tuning

Add text here!

2.7 Overview

Add text here!

2.8 Agents

Add text here!

2.8.1 Policy

Add text here!

2.8.2 Model-Based Agents

Add text here!

2.8.3 Multi-Agents

Add text here!

2.8.4 Custom Algorithms

- **Policy Creation**

  To create a RL policy that satisfies MLPro interface is pretty direct. You just require to assure that the RL policy consists at least these following 3 main functions:

```
from mlpro.rl.models import *

class MyPolicy(Policy):
    """
    Creates a policy that satisfies mlpro interface.
    """
    C_NAME          = 'MyPolicy'

    def __init__(self, p_state_space:MSpace, p_action_space:MSpace, p_
↪ada=True, p_logging=True):
        """
         Parameters:
            p_state_space       State space object
            p_action_space      Action space object
            p_ada               Boolean switch for adaptivity
            p_logging           Boolean switch for logging functionality
```

(continues on next page)

(continued from previous page)

```
        """

        super().__init__(p_ada=p_ada, p_logging=p_logging)
        self._state_space   = p_state_space
        self._action_space  = p_action_space
        self.set_id(0)

    def adapt(self, *p_args) -> bool:
        """
        Adapts the policy based on State-Action-Reward (SAR) data that will␣
↪be expected as a SAR
        buffer object. Please call super-method at the beginning of your␣
↪own implementation and
        adapt only if it returns True.

        Parameters:
            p_arg[0]            SAR Buffer object
        """

        if not super().adapt(*p_args): return False

        ....
        return True

    def clear_buffer(self):
        """
        Intended to clear internal temporary attributes, buffers, ... Can␣
↪be used while training
        to prepare the next episode.
        """
        ....

    def compute_action(self, p_state:State) -> Action:
        """
        Specific action computation method to be redefined.

        Parameters:
            p_state       State of environment

        Returns:
            Action object
        """
        ....
```

This class represents the policy of a single-agent. It is adaptive and can be trained with State-Action-Reward (SAR) data that will be expected as a SAR buffer object.

The three main learning paradigms of machine learning to train a policy are supported:

1. Training by Supervised Learning: The entire SAR data set inside the SAR buffer shall be adapted.
2. Training by Reinforcement Learning: The latest SAR data record inside the SAR buffer shall be adapted.
3. Training by Unsupervised Learning: All state data inside the SAR buffer shall be adapted.

Furthermore a policy class can compute actions from states.

Hyperparameters of the policy should be stored in the internal object **self._hp_list**, so that they can be tuned from outside. Optionally a policy-specific callback method can be called on changes. For more information see class HyperParameterList.

To set up a hyperparameter space, please refer to our *how to File 04* or here.

- **Policy from Third Party Packages**

  In addition, we are planning to reuse Ray RLlib in the near future. For more updates, please click *here*.

- **Algorithm Checker**

  A test script using unittest to check the develop policies will be available soon!

2.9 Environments

Add text here!

2.9.1 Multi-Agent Environments

Add text here!

2.9.2 Custom Environments

- **Environment Creation for Simulation Mode**

  To create an environment that satisfies MLPro interface is immensly simple and straigtforward. Basically a MLPro environment is a class with 5 main functions. Each environment must apply the following mlpro functions:

```
from mlpro.rl.models import *

class MyEnvironment(Environment):
    """
    Custom Environment that satisfies mlpro interface.
    """
    C_NAME          = 'MyEnvironment'
    C_LATENCY       = timedelta(0,1,0)         # Default latency 1s
    C_REWARD_TYPE   = Reward.C_TYPE_OVERALL    # Default reward type

    def __init__(self, p_mode=C_MODE_SIM, p_latency:timedelta=None, p_
↪logging=True):
        """
        Parameters:
            p_mode              Mode of environment (simulation/real)
            p_latency           Optional: latency of environment. If not␣
↪provided
                                internal value C_LATENCY will be used by␣
↪default
            p_logging           Boolean switch for logging
```

(continues on next page)

(continued from previous page)

```
        """

        super().__init__(p_latency=p_latency, p_logging=p_logging)
        self._setup_spaces()
        self.set_mode(p_mode)

    def _setup_spaces(self):
        """
        Implement this method to enrich the state and action space with␣
↪specific
        dimensions.
        """

        # Setup state space example
        # self.state_space.add_dim(Dimension(0, 'Pos', 'Position', '', 'm', 'm',␣
↪[-50,50]))
        # self.state_space.add_dim(Dimension(1, 'Vel', 'Velocity', '', 'm/sec', '\
↪frac{m}{sec}', [-50,50]))

        # Setup action space example
        # self.action_space.add_dim(Dimension(0, 'Rot', 'Rotation', '', '1/sec',
↪'\frac{1}{sec}', [-50,50]))
        ....

    def _simulate_reaction(self, p_action:Action) -> None:
        """
        Simulates a state transition of the environment based on a new␣
↪action.
        Please use method set_state() for internal update.

        Parameters:
            p_action      Action to be processed
        """
        ....

    def reset(self) -> None:
        """
        Resets environment to initial state.
        """
        ....

    def compute_reward(self) -> Reward:
        """
        Computes a reward.

        Returns:
          Reward object
        """
        ....

    def _evaluate_state(self) -> None:
        """
```

(continues on next page)

(continued from previous page)

```
        Updates the goal achievement value in [0,1] and the flags done and␣
↪broken
        based on the current state.
        """

        # state evaluations example
        # if self.done:
        #     self.goal_achievement = 1.0
        # else:
        #     self.goal_achievement = 0.0
        ....
```

One of the benefits for MLPro users is the variety of reward structures, which is useful for Multi-Agent RL and Game Theoretical approach. Three types of reward structures are supported in this framework, such as:

1. **C_TYPE_OVERALL** as the default type and is a scalar overall value
2. **C_TYPE_EVERY_AGENT** is a scalar for every agent
3. **C_TYPE_EVERY_ACTION** is a scalar for every agent and action.

To set up state- and action-spaces using our basic functionalities, please refer to our *how to File 02* or here. Dimension class is currently improved and we will provide the explanation afterwards!

- **Environment Creation for Real Hardware Mode**

  In MLPro, we can choose simulation mode or real hardward mode. For real hardware mode, the creation of an environment is very similar to simulation mode. You do not need to define **_simulate_reaction**, but you need to replace it with **_export_action** and **_import_state** as it is shown in the following:

```
from mlpro.rl.models import *

class MyEnvironment(Environment):
    """
    Custom Environment that satisfies mlpro interface.
    """
    C_NAME          = 'MyEnvironment'
    C_LATENCY       = timedelta(0,1,0)         # Default latency 1s
    C_REWARD_TYPE   = Reward.C_TYPE_OVERALL    # Default reward type

    def __init__(self, p_mode=C_MODE_REAL, p_latency:timedelta=None, p_
↪logging=True):
        """
        Parameters:
            p_mode              Mode of environment (simulation/real)
            p_latency           Optional: latency of environment. If not␣
↪provided
                                internal value C_LATENCY will be used by␣
↪default
            p_logging           Boolean switch for logging
        """

        super().__init__(p_latency=p_latency, p_logging=p_logging)
```

(continues on next page)

(continued from previous page)

```
        self._setup_spaces()
        self.set_mode(p_mode)

    def _setup_spaces(self):
        """
        Implement this method to enrich the state and action space with␣
↪specific
        dimensions.
        """

        # Setup state space example
        # self.state_space.add_dim(Dimension(0, 'Pos', 'Position', '', 'm', 'm',␣
↪[-50,50]))
        # self.state_space.add_dim(Dimension(1, 'Vel', 'Velocity', '', 'm/sec', '\
↪frac{m}{sec}', [-50,50]))

        # Setup action space example
        # self.action_space.add_dim(Dimension(0, 'Rot', 'Rotation', '', '1/sec',
↪'\frac{1}{sec}', [-50,50]))
        ....

    def _export_action(self, p_action:Action) -> bool:
        """
        Exports given action to be processed externally (for instance by a␣
↪real hardware).

        Parameters:
            p_action      Action to be exported

        Returns:
            True, if action export was successful. False otherwise.
        """
        ....

    def _import_state(self) -> bool:
        """
        Imports state from an external system (for instance a real␣
↪hardware).
        Please use method set_state() for internal update.

        Returns:
          True, if state import was successful. False otherwise.
        """
        ....

    def reset(self) -> None:
        """
        Resets environment to initial state.
        """
        ....

    def compute_reward(self) -> Reward:
```

(continues on next page)

(continued from previous page)

```
        """
        Computes a reward.

        Returns:
          Reward object
        """
        ....

    def _evaluate_state(self) -> None:
        """
        Updates the goal achievement value in [0,1] and the flags done and␣
↪broken
        based on the current state.
        """

        # state evaluations example
        # if self.done:
        #     self.goal_achievement = 1.0
        # else:
        #     self.goal_achievement = 0.0
        ....
```

- **Environment from Third Party Packages**

  Alternatively, if your environment follows Gym or PettingZoo interface, you can apply our relevant useful wrappers for the integration between third party packages and MLPro. For more information, please click *here*.

- **Environment Checker**

  To check whether your developed environment is compatible to MLPro interface, we provide a test script using unittest. At the moment, you can find the source code here. We will prepare a built-in testing module in MLPro, show you how to excecute the testing soon and provides an example as well.

2.10 Scenarios

Add text here!

2.11 Pool

Add text here!

2.11.1 Policies

2.11.2 Advantage Actor Critic (A2C)

You can get started as follow:

```
import mlpro.rl.pool.policies.a2c
```

2.11.3 Soft Actor-Critic (SAC)

You can get started as follow:

```
import mlpro.rl.pool.policies.sac
```

2.11.4 Environments

Grid World Problem

You can get started as follow:

```
import mlpro.rl.pool.envs.gridworld
```

Multi-Cartpole

You can get started as follow:

```
import mlpro.rl.pool.envs.multicartpole
```

Bulk Good Laboratory Plant (BGLP)

You can get started as follow:

```
import mlpro.rl.pool.envs.bglp
```

Robot Manipulator on Homogeneous Matrix

You can get started as follow:

```
import mlpro.rl.pool.envs.robotinhtm
```

Universal Robots 5 Joint Control

You can get started as follow:

```
import mlpro.rl.pool.envs.ur5jointcontrol
```

2.11.5 Scenarios

Add text here!

2.12 3rd Party Support

Add text here!

MLPro allows you to reuse widely-used packages and integrate them to MLPro interface by calling wrapper classes.

At the moment, a wrapper class for OpenAI Gym Environments has been tested and is ready-to-use. However, it has not been very stable yet and some minor improvements might be needed.

In the near future, we are going to add wrapper classes for PettingZoo and Ray RLlib.

Soure code of available wrappers: https://github.com/fhswf/MLPro/blob/main/src/mlpro/rl/wrappers.py

2.12.1 OpenAI Gym Environments

Our wrapper class for gym environment is pretty straightforward. You can just simply apply a command to setup a gym-based environment, while creating a scenario.

```
from mlpro.rl.wrappers import WrEnvGym

self._env = WrEnvGym([gym environment object], p_state_space:MSpace=None, p_action_
↪space:MSpace=None, p_logging=True)
```

For more information, please check our how to files here.

2.12.2 PettingZoo Environments

Under construction. The wrapper will be available soon.

```
from mlpro.rl.wrappers import WrEnvPZoo

self._env = WrEnvPZoo([zoo environment object], p_state_space:MSpace=None, p_action_
↪space:MSpace=None, p_logging=True)
```

2.12.3 Ray RLlib

Under construction. The wrapper will be available soon.

```
from mlpro.rl.wrappers import wrPolicyRay

wrPolicyRay(...)
```

2.13 Examples

2.13.1 Basic Functions

We provide some examples of MLPro's basic funcionalities implementation, which is available on our GitHub file.

- File 01: (Various) Log and Timer
- File 02: (Math) Spaces, subspaces and elements
- File 03: (Various) Store, plot, and save variables
- File 04: (ML) Hyperparameters setup

Moreover, you can find the UML diagram of MLPro's basic funcionalities here.

2.13.2 Reinforcement Learning

We provide some examples of MLPro's RL funcionalities implementation, which is available on our GitHub file.

- File 01: (RL) Types of reward
- File 02: (RL) Run agent with own policy with gym environment
- File 03: (RL) Train agent with own policy on gym environment
- File 04: (RL) Run multi-agent with own policy in multicartpole environment
- File 05: (RL) Train multi-agent with own policy on multicartpole environment
- File 06: (RL) A2C Implementation
- File 08: (RL) Run own agents with petting zoo environment

Moreover, you can find the UML diagram of MLPro's RL funcionalities here.

2.13.3 Game Theory

We provide some examples of MLPro's GT implementation, which is available on our GitHub file.

- File 06: (GT) Run multi-player with own policy in multicartpole game board
- File 07: (GT) Train own multi-player with multicartpole game board

Moreover, you can find the UML diagram of MLPro's GT funcionalities here.

2.13.4 User Interface

We provide some examples of MLPro's SciUI funcionalities implementation, which is available on our GitHub file.

- File 01: (SciUI) - Reuse of interactive 2D,3D input space

Moreover, you can find the UML diagram of MLPro's UI funcionalities here.

2.14 Game Theory

Game Theory (GT) is well-known in economic studies as a theoretical approach to model the strategic interaction between multiple individuals or players in a specific situation. Game Theory approach can also be adopted in the science area to optimize decision-making processes in a strategic setting and often use to solve Multi-Agent RL (MARL) problems.

You can easily access the GT module, as follows:

```
from fhswf_at_ml.gt.models import *
```

Some of developed RL frameworks in MLPro can also be reuse in the GT approach. Thus we can just simply inherit some classes from RL frameworks, such as:

1. **GameBoard(rl.Environent)**
   Since you need a unique utility function for each specific player in the GT approach. A local utility function can be defined as below:

```
import fhswf_at_ml.rl.models as rl

class GameBoard(rl.Environment):
    """
    Model class for a game theoretical game board. See super class for more␣
↪information.
    """

    C_TYPE              = 'Game Board'
    C_REWARD_TYPE       = rl.Reward.C_TYPE_EVERY_AGENT

    def compute_reward(self) -> rl.Reward:
        if self._last_action is None: return None

        reward = rl.Reward(self.get_reward_type())

        for player_id in self.last_action.get_agent_ids():
            reward.add_agent_reward(player_id, self._utility_fct(player_id))

        return reward

    def _utility_fct(self, p_player_id):
        """
        Computes utility of given player. To be redefined.
        """

        return 0
```

2. **Player(rl.Agent)**

```
import fhswf_at_ml.rl.models as rl

class Player(rl.Agent):
    """
    This class implements a game theoretical player model. See super class␣
↪for more information.
    """

    C_TYPE      = 'Player'
```

3. **Game(rl.Scenario)**

```
import fhswf_at_ml.rl.models as rl

class Game(rl.Scenario):
    """
    This class implements a game consisting of a game board and a (multi-
↪)player. See super class for
    more information.
    """

    C_TYPE      = 'Game'
```

4. **MultiPlayer(rl.MultiAgent)**

```
import fhswf_at_ml.rl.models as rl

class MultiPlayer(rl.MultiAgent):
    """
    This class implements a game theoretical model for a team of players.␣
↪See super class for more
    information.
    """

    C_TYPE      = 'Multi-Player'

    def add_player(self, p_player:Player, p_weight=1.0) -> None:
        super().add_agent(p_agent=p_player, p_weight=p_weight)
```

5. **Training(rl.Training)**

```
import fhswf_at_ml.rl.models as rl

class Training(rl.Training):
    """
    This class implements a standardized episodical training process. See␣
↪super class for more information.
    """

    C_NAME      = 'GT'
```

You can check out some of the examples on our *how to files* or here.

2.15 UI Framework SciUI

- Platform-independend framework for creation of own UI scenarios
- Ready-to-run application „SciUI“ detects and starts own scenarios
- Focus on real-time visualization and interaction
- Based on Python standards Tkinter and MatPlotLib

If you are looking for an example of SciUI implementation, we provide *how to files* related to this subject.

Add text here!

2.15.1 Interactive Reinforcement Learning

Add text here!

2.16 Release Notes

2.16.1 Version XX.XX.XX

Release Highlights:

New Features:

Issued Fixed:

Documentation Changes:

Others:

2.17 Papers

Add text here!

2.18 Contribution

We look forward to your contributions to MLPro improvements. You can report any bugs, propose further improvement ideas, add pre-builts environments and algorithms, improve our documentations, or any kind of improvements.

You can directly contribute on our public GitHub's repository or reach us per email mlpro@listen.fh-swf.de.

Add text here!

CHAPTER

THREE

CITING MLPRO

To cite this project in publications:

```
@misc{...
}
```

CHAPTER

FOUR

CONTACT DATA

Mail: mlpro@listen.fh-swf.de