Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

MLPro 1.0 - Standardized reinforcement learning and game theory in Python

Detlef Arend [*], Steve Yuwono, Mochammad Rizky Diprasetya, Andreas Schwung

Department of Automation Technology and Learning Systems, South Westphalia University of Applied Sciences, Lübecker Ring 2, Soest, 59494, Germany

A R T I C L E   I N F O

Keywords:
Machine learning
Reinforcement learning
Game theory
Automation
Scientific software development
Python

A B S T R A C T

Nowadays there are numerous powerful software packages available for most areas of machine learning (ML). These can be roughly divided into frameworks that solve detailed aspects of ML and those that pursue holistic approaches for one or two learning paradigms. For the implementation of own ML applications, several packages often have to be involved and integrated through individual coding. The latter aspect in particular makes it difficult for newcomers to get started. It also makes a comparison with other works difficult, if not impossible. Especially in the area of reinforcement learning (RL), there is a lack of frameworks that fully implement the current concepts up to multi-agents (MARL) and model-based agents (MBRL). For the related field of game theory (GT), there are hardly any packages available that aim to solve real-world applications. Here we would like to make a contribution and propose the new framework MLPro, which is designed for the holistic realization of hybrid ML applications across all learning paradigms. This is made possible by an additional base layer in which the fundamentals of ML (interaction, adaptation, training, hyperparameter optimization) are defined on an abstract level. In contrast, concrete learning paradigms are implemented in higher sub-frameworks that build on the conventions of this additional base layer. This ensures a high degree of standardization and functional recombinability. Proven concepts and algorithms of existing frameworks can still be used. The first version of MLPro includes sub-frameworks for RL and cooperative GT.

1. Introduction

In the last decade, numerous professional frameworks have emerged that are quite rightly popular in the ML community. These already cover many sub-areas of this intensively studied research area. However, the large number of available frameworks is both a blessing and a curse. Newcomers and professionals alike are spoiled for choice when it comes to putting together the right software components for their projects. And in very few cases frameworks can be combined without the realization of an own individual integration. The latter makes comparison with other works difficult, if not impossible — quite apart from the effort involved in implementing own standards.

In addition, the scientific perspectives, proceedings and models are constantly evolving. On closer inspection, it turns out that there are often no frameworks at all available for current, advanced approaches. Reusable standards, for example in the field of RL (Sutton & Barto, 2018) for model-based agents (MBRL[1]) in combination with action planning (MPC[2]) as part of multi-agent systems (MARL[3]) or for methods of cooperative GT (Elkind & Rothe, 2016; Owen, 2013), are currently being sought in vain.

According to the authors, another shortcoming is that various established packages are not close to real applications. The precise handling of simulated or real time or the complete definition of physical quantities including units is often not provided at all. Industrial hardware, on top of that, is often controlled in completely different environments and not as an integrated and standardized component of the overall design.

All of these aspects have motivated us to design a new framework called MLPro, which initially follows the approach of embedding the most popular functionalities and algorithms in complete and consistent ML model and process landscapes at a scientific level. Researchers, developers, engineers and students can manage their ML tasks in a standardized way during the entire life cycle, from definition to training

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

[*] Corresponding author.

E-mail addresses: arend.detlef@fh-swf.de (D. Arend), yuwono.steve@fh-swf.de (S. Yuwono), diprasethya.mochammadrizky@fh-swf.de (M.R. Diprasetya), schwung.andreas@fh-swf.de (A. Schwung).

[1] Model-based reinforcement learning (Kaiser et al., 2019; Pal & Leon, 2020).

[2] Model predictive control (Williams et al., 2017).

[3] Multi-Agent reinforcement learning (Busoniu, Babuska, & De Schutter, 2008).

https://doi.org/10.1016/j.mlwa.2022.100341
Received 25 March 2022; Received in revised form 21 May 2022; Accepted 21 May 2022
Available online 28 May 2022

to real application, which foreseeably reduces the effort for education and realization. Furthermore, results become comparable.

A further outstanding feature of MLPro is that the basics of machine learning have been implemented at a deep and correspondingly universal level in the architecture. The core component is the abstract adaptive ML model, which is trained in the associated ML scenario in simulation mode and executed in the same scenario in real mode. The mathematically/physically well-founded definitions of dimensions, sets and spaces are anchored here, as is the optional tuning of the hyperparameters contained in the ML model. All higher sub-frameworks – in the first version for RL and GT – are based on these fundamental concepts and extend them in such a way that maximum standardization and recombinability are guaranteed even for hybrid applications. A constantly growing number of public third-party packages are included at various levels of the architecture using wrapper technology in such a way that process integrity is maintained at all times.

The contributions we aim to make to ML research can be summarized as follows:

- We evaluate and classify 13 relevant ML frameworks using a system of objectively comprehensible criteria.
- We propose a new ML framework, namely MLPro, where we focus on RL and GT in the first version. It is intended as the one development environment in which researchers, engineers and students can implement, validate, compare and exchange their ML projects.
- MLPro 1.0 provides state-of-the-art ML models and end-to-end processes for MARL/MBRL as well as for cooperative GT. The widely-used 3rd party functionalities are integrated to the proposed framework.
- We demonstrate the usability of MLPro in two simulated industrial applications, such as a bulk goods laboratory plant and a UR5 robot arm.

The paper is structured as follows. In Section 2, we first address the question of which software packages exist for ML, how they can be classified and what gaps they leave. The motivation for the creation of the new and comprehensive framework MLPro is derived directly from this. Section 3 then describes the fundamentals and overarching concepts of MLPro. The focus here is on design and architecture as well as the description of selected basic concepts. Based on this, the sub-framework MLPro-RL for reinforcement learning is described in detail in Section 4. In particular, those aspects are discussed here that, in our opinion, represent an innovation compared to the currently available frameworks. Section 5 is dedicated to the second sub-framework MLPro-GT for cooperative game theory. Here the high degree of reusability of existing concepts shall be illustrated as well as the implementation of the GT specifics through to potential-based approaches. In Section 6, we demonstrate the implementation steps of MLPro and the practical benefits of MLPro on industry-related sample applications from RL and GT, before Section 7 summarizes the current state and the starting points for further expansion of the framework.

2. Background and related work

To make a statement on the availability of standardized ML functionalities, we have examined and classified a representative selection of common open-source software packages with Python API. The documented range of functions was considered here, but not the additional community developments. Although the latter often achieve a high degree of quality and professionalism, they are usually not subject to the same criteria of standardization, maintenance and further development as the underlying software package itself. Also, such additions are usually much more personalized and therefore heavily dependent on the current and future commitment of individual developers.

In Section 2.1, the criteria for classifying and grouping the packages are first defined and explained. Section 2.2 documents the results of the classification of all 13 packages considered. Finally, Section 2.3 summarizes and interprets the results.

2.1. Classification and grouping criteria

All examined packages were classified according to the same criteria and divided into ten classes, which illuminate the packages from different perspectives in order to convey as holistic an impression as possible. The criteria themselves were chosen to such a degree that they are objectively comprehensible on the basis of the information that is publicly available. All underlying classes and their criteria are listed below.

Unless a statement to the contrary is made, the criteria were rated with a 1 (fulfilled) or 0 (not fulfilled). This results in a degree of fulfillment in percent for each class and package. The criteria are not weighted for the sake of simplicity.

1	**ML — Abstract/overarching ML concepts**
	Here it was evaluated whether a package has standardized the basic concepts of machine learning at an abstract and therefore universal level. Neither learning paradigms nor concrete forms of ML models such as artificial neural networks are defined at this level. According to the authors, this is an indication of whether the framework as a whole pursues a holistic approach or is only geared towards special learning paradigms or ML model types.
1.1	**Prototype of an adaptive ML Model**
	An abstract template for an adaptive ML model with hyperparameters.
1.2	**Abstract Surroundings Model**
	Corresponds at a higher level to a data set or data stream for Supervised Learning (SL)/Deep Learning(DL)/Unsupervised Learning(UL) or an RL environment, etc.
1.3	**Abstract ML Scenario Model**
	A template for a concrete ML arrangement consisting of an adaptive ML model and its surroundings.
1.4	**General Training Process**
	To train an adaptive ML model in a defined ML scenario.
1.5	**General Hyperparameter Tuning Process**
	Optional optimization of the hyperparameters of an ML model as part of the training process.

2	**SL — Supervised Learning**
	Criteria for standards in context of supervised learning.
2.1	**Specific adaptive ML Models**
	E.g. for feedforward neural networks (FNN) like the perceptron (Rosenblatt, 1958)
2.2	**Specific adaptive ML Models for Deep Learning (DL)**
	E.g. for deep neural networks like the multilayer perceptron (MLP) (Murtagh, 1991) or convolutional neural networks (CNN) (Matsugu, Mori, Mitari, & Kaneda, 2003)
2.3	**Data Set Management**
	Are standardized functionalities provided for import, export and access to SL-specific data sets?
2.4	**State-of-the-Art Algorithms**
	For training the ML models provided.

2.5	**Standardized Training Process**
	Is a training process provided that standardizes how the ML model interacts with its learning surroundings? Are training data and the trained ML model made available persistently for evaluation or use? Are random generators seeded in order to be able to reproduce training runs (benchmarking)?
2.6	**Standardized Hyperparameter Tuning**
	Does the training process support hyperparameter tuning?
2.7	**Sample Data Sets**
	Are specific sample data sets provided for training SL models?

3	**UL — Unsupervised Learning**
	Criteria for standards in context of unsupervised learning.
3.1	**Specific adaptive ML Models**
	E.g. for recurrent neural networks (RNN) as class activation maps (CAM) like Hopfield nets (Hopfield, 1982) etc.
3.2	**State-of-the-Art Algorithms**
	For training the ML models provided.
3.3	**Standardized Training Process**
	See 2.5
3.4	**Standardized Hyperparameter Tuning**
	See 2.6
3.5	**Sample Data Sets**
	Are specific sample data sets provided for training UL models?

4	**RL — Reinforcement Learning**
	Criteria for standards in context of reinforcement learning.
4.1	**Single-Agent Model**
	Is a single-agent model provided with a standard interface to interact with an environment?
4.2	**Multi-Agent Model (MARL)**
	Does the framework support MARL by providing a multi-agent model with a standard interface to interact with an environment?
4.3	**Model-based Agent Model (MBRL)**
	Does the framework support MBRL by providing standardized environment models that can optionally be added to the single-agent model?
4.4	**Action Planning (MPC)**
	Does the framework support model predictive control by providing standardized mechanisms for action planning that can optionally be added to a single-agent (in combination with 4.3)?
4.5	**State-of-the-Art Algorithms**
	RL learning strategies like Deep Q-Network (DQN) (Mnih et al., 2013), Deep Deterministic Policy Gradient (DDPG) (Silver et al., 2017), Proximal Policy Optimization (PPO) (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017), Advantage Actor Critic (A2C) (Mnih et al., 2016), etc.
4.6	**Single-Agent Environment Model**
	Does the framework provide a standard model for single-agent environments?
4.7	**Multi-Agent Environment Model**
	Does the framework provide a standard model for multi-agent environments?
4.8	**Standardized Training Process**
	See 2.5. Additionally: does the training process offer episode management as is usual in RL?
4.9	**Standardized Hyperparameter Tuning**
	See 2.6
4.10	**Sample Single-Agent Environments**
	Does the framework provide sample environments with a single-agent reward structure?
4.11	**Sample Multi-Agent Environments**
	Does the framework provide sample environments with a multi-agent reward structure?

5	**GT — Game Theory**
	Criteria for standards in context of game theory.
5.1	**Cooperative Game Theory**
	Does the framework provide standards for cooperative game theory?
5.2	**Competitive Game Theory**
	Does the framework provide standards for competitive game theory?
5.3	**Single-Player Model**
	Is a single-player model provided with a standard interface to interact with a game board?
5.4	**Multi-Player Model**
	Is a multi-player model provided with a standard interface to interact with a game board?
5.5	**Model-based Player Model**
	See 4.3
5.6	**Action Planning (MPC)**
	See 4.4
5.7	**State-of-the-Art Algorithms**
	See 4.5. Additionally/alternatively learning algorithms in context GT.
5.8	**Multi-Player Game Board Model**
	Does the framework provide a standard model for multi-player game boards?
5.9	**Standardized Training Process**
	See 4.8
5.10	**Standardized Hyperparameter Tuning**
	See 2.6
5.11	**Sample Game Boards**
	Does the framework provide sample game boards with a multi-player reward structure?

6	**RW — Real World Application Support**
	Criteria that qualify the framework for real-time applications.
6.1	**Real Operation Support/Mode**
	Do the ML surroundings have a mode that allows switching between simulation and live operation?
6.2	**Support of units** We evaluate and classify the most relevant ML frameworks
	Can units be defined when specifying input, output or state variables?
6.3	**Precise time management**
	Are there functions for precise measurement of real or simulated time?

6.4	**Detailed logging**
	Are there functions for structured, detailed and standardized logging of events? Are these used by the operating components of the framework so that, for example, training or real operation can be traced ("flight recorders")?
6.5	**Data Preprocessing**
	Are there standards or functionalities for pre-processing data (normalization, feature reduction, filtering, noise reduction, ...)?

7	**Spread in Research and Science**
	Here, the spread of a package in research and science was surveyed based on the number of citations of its associated publications. The publications that were created by the project team to introduce or explain the package were taken into account. These are also listed in the bibliography and referenced in Section 2.2. The package with the largest number of citations was rated 100%, the other packages correspondingly lower. The number of citations was collected using Google Scholar as of March 8, 2022.

8	**Edu — Educational Features**
	Here we have highlighted criteria that make it easier for beginners or students to work with a framework.
8.1	**Comprehensive Documentation**
	Is a comprehensive documentation provided that explains the core concepts and the Application Programming Interface (API) of the framework?
8.2	**Ready to run example codes**
	Are self-learning example program codes provided that are easy to run and understand?
8.3	**Tutorials**
	Are tutorials provided that explain certain aspects of the framework?
8.4	**Training Programs/Certification**
	Are professional training programs provided? Can developers get certified?
8.5	**Descriptive Publications**
	Has the project team published scientific articles on the framework?
8.6	**Interactive User Experience**
	Are functionalities provided for an interactive user experience that can be used for self-studies or presentations?

9	**Quality**
	Various criteria were examined here, which allow a statement to be made about the quality of the framework and the project behind.
9.1	**Professional Source Code Management**
	Is the source code maintained and versioned in a professional environment?
9.2	**Professional Project Management**
	Can a professional project management method be identified (task management, milestones, perhaps agile methods such as Scrum or DevOps)?
9.3	**Test Automationl**
	Is the quality assured by automated unit tests?
9.4	**CI/CD**
	Are continuous integration (CI) and/or continuous delivery (CD) methods used?
9.5	**API Documentation**
	Is there an API documentation for all supported programming languages?
9.6	**Clean Code Paradigm**
	Is a style guide (or corresponding templates) provided for contributors, which specifies the consistent and qualitative design of source code according to the Clean Code Paradigm (Martin, 2011)? Were random samples able to verify compliance with this in the project?
9.7	**Publications**
	See 8.5

10	**Tech — Technical Features**
	Various basic characteristics from the areas of base technology and software development were examined here.
10.1	**GPU/CUDA Support**
	The use of GPUs (Graphic Processing Units) via NVidia's standard API CUDA (Compute Unified Device Architecture) for massively parallel calculation of complex numeric tasks is a key technology in the ML context.
10.2	**Advanced Mathematics**
	Tensor calculations, optimization algorithms, etc.
10.3	**Model Deployment (MLOps)**
	Are there mechanisms to deploy and maintain ML models in production?
10.4	**Data Management**
	Are overarching functionalities for a standardized data management provided?
10.5	**Data Plotting**
	Are overarching functionalities for a standardized data plotting provided?
10.6	**Python API**
	Is an API provided for Python?
10.7	**Java API**
	Is an API provided for Java?
10.8	**C++ API**
	Is an API provided for C++?
10.9	**APIs for further Programming Languages**
	Are APIs for further programming languages provided?
10.10	**REST API**
	Can core functionalities of a framework be accessed via REST services?

Furthermore, the packages under consideration can be roughly divided into three groups — basic packages, sample providers and ML frameworks. The packages in the first group provide rather unspecific help functions such as hyperparameter tuning, model deployment or lifecycle management. On the other hand, we understand sample providers to mean packages that provide test data or test configurations for training ML models. The third group of ML frameworks includes packages that cover essential sub-areas of machine learning.

2.2. *Classification results*

Below we present the results of our evaluation in the form of spider diagrams for each framework. The raw data on which the charts are based are listed in Appendix.

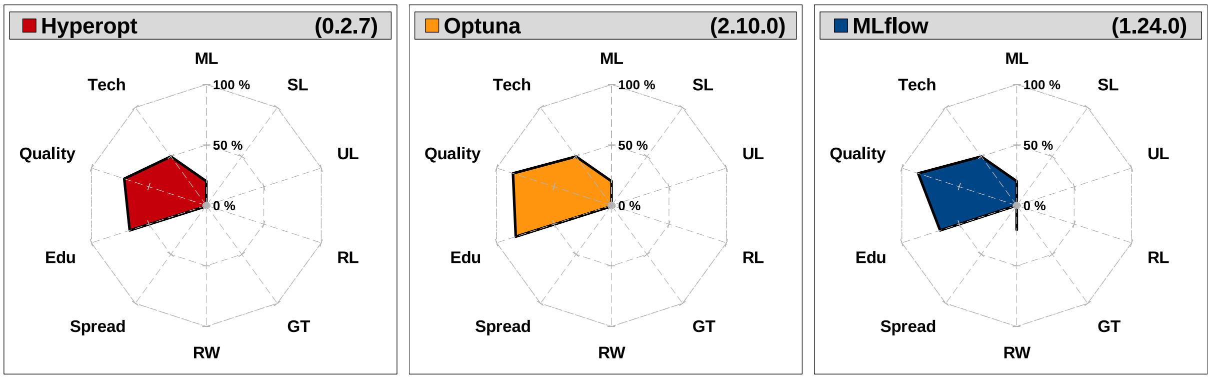


Fig. 1. Classification of basic packages.

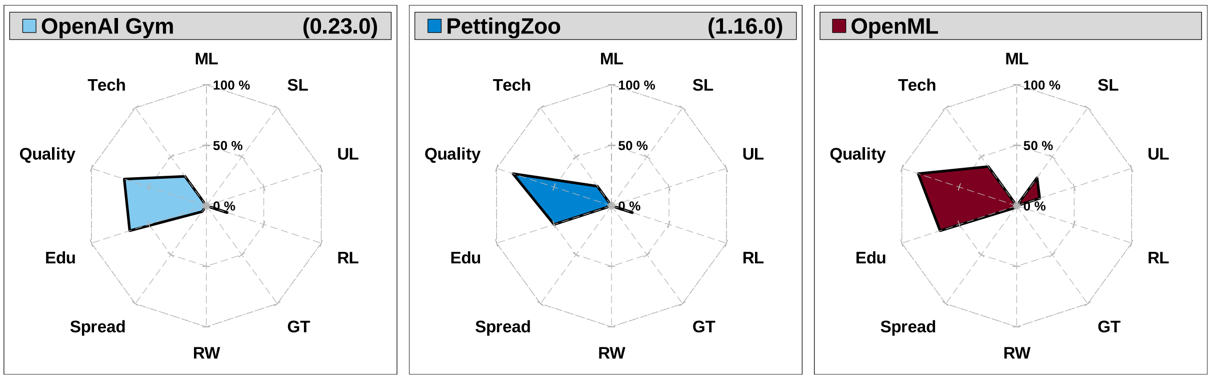


Fig. 2. Classification of sample providers.

In the group of basic packages, the two packages Hyperopt (Bergstra, Komer, Eliasmith, Yamins, & Cox, 2015) and Optuna (Akiba, Sano, Yanase, Ohta, & Koyama, 2019) for hyperparameter tuning and the package MLflow (Chen et al., 2020; Zaharia et al., 2018) for model deployment and lifecycle management were examined. The results are shown in Fig. 1.

In the group of sample providers, OpenAI Gym (Brockman et al., 2016) and PettingZoo (Terry et al., 2020) as well as the online test database OpenML (Casalicchio et al., 2019; Feurer et al., 2019; Vanschoren, van Rijn, Bischl, & Torgo, 2014) were evaluated. The first two packages provide standardized environments for single-agents (OpenAI Gym) and multi-agents (PettingZoo) in the context of RL. OpenML, in turn, provides a wealth of data sets that can be consumed via APIs for various popular programming languages. Algorithms can be compared there in a standardized benchmark process. Fig. 2 illustrates the results of this group.

A total of seven packages were considered in the third group of ML frameworks. The oldest package in this group is scikit-learn (Buitinck et al., 2013; Pedregosa et al., 2011). It provides a wealth of state-of-the-art algorithms and methods for data analysis and prediction. In addition, it provides a large number of example data sets — directly or via data loader functionalities.

The two newer packages TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019) are similar in content and provide standards and algorithms for deep neural networks. Both provide a GPU/CUDA-driven high-performance substructure for complex tensor calculations. TensorFlow is widely used in industry while PyTorch is gaining popularity in science. There are plenty of comparative articles to help in deciding between these two packages. For example, Chirodea et al. (2021) sees, on the one hand, the PyTorch package slightly ahead in terms of performance, but on the other hand, emphasizes the easier entry into TensorFlow due to the included high-level frameworks for ML such as Keras.

Another framework that takes a holistic approach is Ray (Liang et al., 2017; Liaw et al., 2018). Based on a profound foundation for parallel processing (Ray-Core) and hyperparameter tuning (Ray-Tune), standards and state-of-the-art algorithms are provided, especially for the RL sub-area (Ray-RLlib).

Stable Baselines3 (Raffin et al., 2019, 2021) provides state-of-the-art algorithms in the field of RL. The possibility of standardized training in OpenAI Gym Environments makes it an end-to-end framework in the context of model-free single agents.

In the field of GT, two frameworks were examined. The first of the two – namely OpenSpiel (Lanctot et al.) – is dedicated to solving game tasks using methods from RL. It provides a wealth of games in which players can be trained. The second package Nashpy (Knight & Campbell, 2018) focuses on competitive 2-player games. Fig. 3 shows the results of this group of frameworks.

2.3. Summary and conclusions

First of all, it should be mentioned that all the software packages considered meet high-quality standards. Without exception, the public development platform GitHub is used for source code management and release as well as for project and task management. Good online documentation with example programs and tutorials as well as at least one scientific publication are also available for all packages.

In terms of the number of citations, scikit-learn, TensorFlow and PyTorch are by far the most widely used packages in science, with the former coming out on top, not least because of its long history. These three packages have a clear focus on supervised learning, with both TensorFlow and PyTorch showing their strengths, particularly in deep learning. According to the authors, TensorFlow is predestined for the industrial sector with its broad support of programming languages and target platforms as well as its developer certification program. PyTorch, on the other hand, has a slightly narrower focus. However, it tends

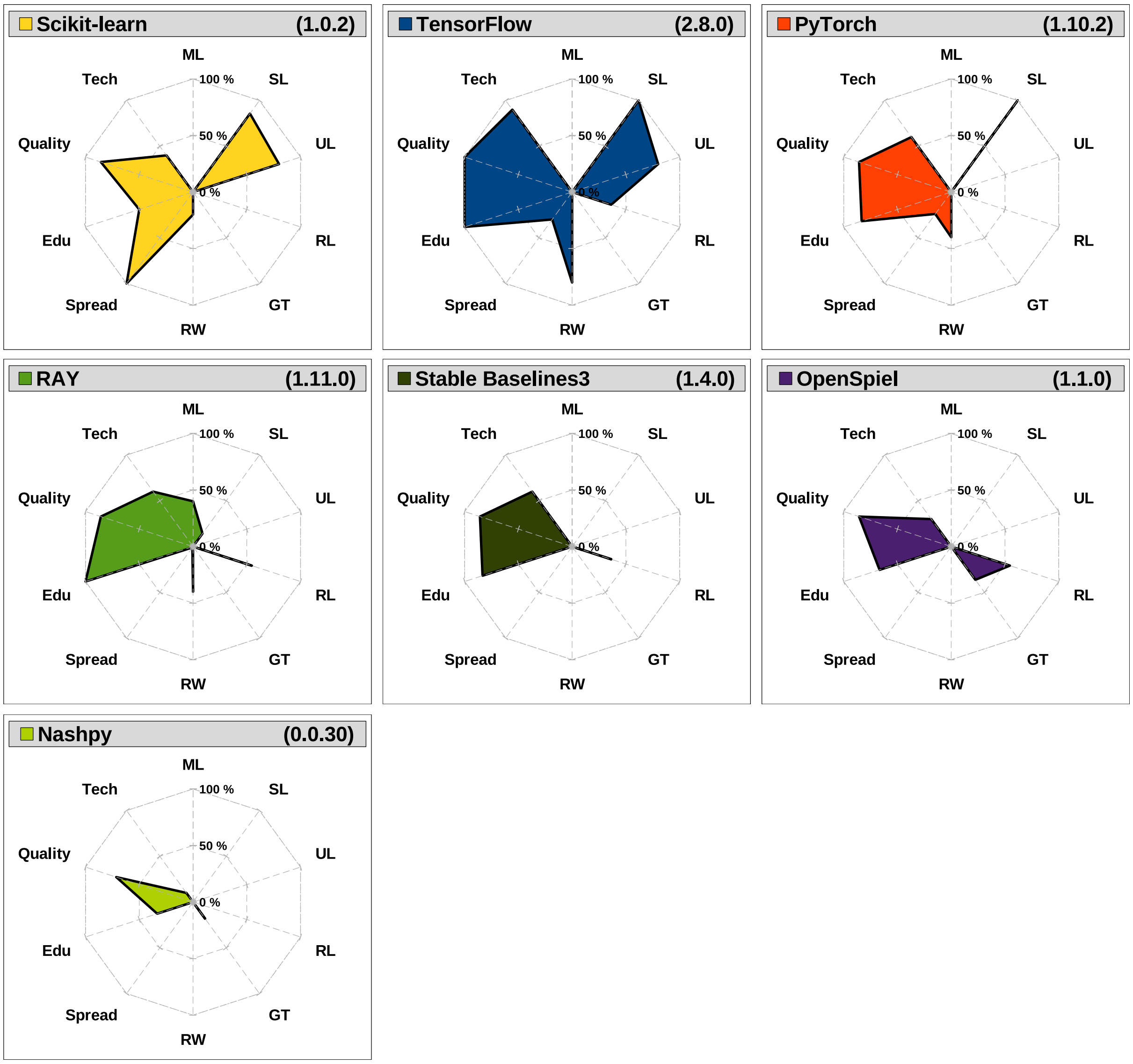


Fig. 3. Classification of ML-Frameworks.

to have better performance, making it a notable alternative. Scikit-learn impresses with its wealth of state-of-the-art algorithms for data analysis.

In the area of RL, the OpenAI Gym environment model for single-agent scenarios is widespread and is taken up by packages such as Stable Baselines3 or Ray. Numerous example environments are available, but the model itself is clearly limited to simulations, being primarily designed as a training partner for custom agents. According to the authors, it is less suitable for creating digital twins of real applications or even real applications themselves. SI units of state and action variables cannot be specified and in particular, the time management of an environment remains opaque. From the outside, it is completely unknown what period is simulated after the induction of an action. A sampling rate is neither known nor adjustable. In addition, the logging of important runtime information is the responsibility of the developer, as is the visualization, which usually takes place in a separate window due to the lack of conventions for embedding in frame applications. PettingZoo's multi-agent environment model is based on the OpenAI Gym environment model and is subject to the same limitations.

As far as the ML frameworks for RL are concerned, TensorFlow only offers standards for model-free single-agent scenarios. There are templates for agents and environments and implementations of some state-of-the-art algorithms. Instead of a standard, there are only tutorials with sample code for the training process. Hyperparameter tuning is not discussed in this context.

Ray goes further with its sub-framework Ray-RLlib. Both single- and multi-agents (MARL) are supported and corresponding environment templates, as well as a standardized training process including hyperparameter tuning, are also available. Furthermore, numerous state-of-the-art algorithms have been implemented. However, standards for model-based agents (MBRL) or action planning (MPC) are not included.

For its part, Stable Baselines3 offers numerous state-of-the-art algorithms and a standardized training process including hyperparameter tuning. The training is limited to the OpenAI Gym environment model. Both packages in combination thus represent a consistent platform for single-agent scenarios. However, more advanced concepts such as MARL, MBRL or MPC are not supported.

A hybrid framework that implements both RL and GT aspects is OpenSpiel by DeepMind. It provides a long list of cooperative and competitive games, each of which can have one or more players added to it. Both games and players are based on standard templates. The players, for whom a number of state-of-the-art algorithms from the RL context are available, are trained while playing, with hyperparameter tuning not being provided. Standards for higher RL concepts such as MBRL or MPC are also not implemented.

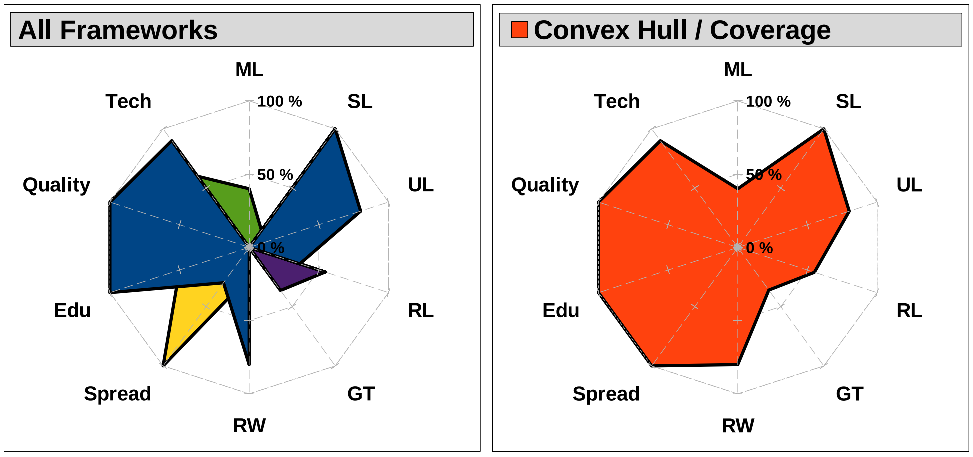


Fig. 4. Coverage of all frameworks.

Finally, with Nashpy, a thoroughbred GT framework should be mentioned. This focuses on competitive 2-player games and provides numerous state-of-the-art algorithms from the GT context for finding the Nash equilibrium. However, there is no player model in the true sense and otherwise no parallels to RL.

Fig. 4 shows the coverage of the various sub-aspects by the examined packages. In summary, it can be said that the areas of base technology and SL/DL are already well covered. In the area of UL, the coverage is already decreasing and in the field of RL must be stated that essential aspects of current research such as MBRL and MPC are not served by any framework. GT has exotic status among the frameworks and is only picked up by packages that want to solve game problems. The potential of this side branch of ML for real-world applications has not yet been tapped.

Apart from the sub-areas that are not covered, there is currently not a single package that shows a truly holistic approach. Hybrid applications often have to be designed and developed across multiple frameworks, which puts an end to any discussion of standardization or even comparability. Since none of the packages has an abstract and learning paradigm-spanning ML substructure, expanding the scope while maintaining internal integrity, reusability and recombinability will become increasingly difficult.

That was exactly our motivation to create a new ML framework that starts at a more general level in terms of design and architecture and establishes advanced standards across all ML disciplines, incorporating already available and proven detailed solutions.

3. Fundamentals of MLPro

With MLPro we present a new and freely available framework for the holistic and standardized description and solution of machine learning tasks. It provides sub-packages for dedicated subtopics of ML based on a uniform infrastructure of basic and cross-sectional functionalities. MLPro supports training in simulated environments and the control and monitoring of real configurations in robotics, automation technology, etc. State-of-the-art implementations of established packages can optionally be integrated into own MLPro scenarios.

During the implementation, great importance was attached to the completeness, correctness and flexibility of the object-oriented (OO) model landscapes and processes. Established scientific terminology has been incorporated into the naming of the development objects, which should make it easier for scientists and developers to create their applications.

3.1. Architecture and infrastructure

As shown in Fig. 5, MLPro is divided into so-called subtopic packages, which cover independent sub-areas of machine learning.

Each subtopic package provides a self-contained landscape of specific OO models and processes for the definition, training and application of ML models. In addition, reusable pool objects (algorithms, standard test arrangements, etc.), as well as executable and well-documented program examples and detailed online documentation, are part of the scope. Selected functions from third-party providers are integrated into the MLPro framework using the so-called wrapper[4] technology. Specific interactive applications are also intended based on MLPro's own sub-framework SciUI, which enables standardized user interaction for education, graphical validation or presentation purposes.

As mentioned, in the first version of MLPro, we provide subtopic packages for the areas of RL and cooperative GT, which are described in more detail in the following sections. Due to the open architecture of MLPro, an extension with further subtopic packages is possible and also planned. The recombinability of functionalities from different subtopic packages is also intended, which in particular enables the arranging of hybrid ML applications.

This is made possible by a subordinate layer of overarching basic functions, which in particular ensures the standardization of recurring subtasks. First of all, elementary functionalities for logging, data management (loading, saving, buffering, plotting) and time management on microsecond level for real or simulated processes should be mentioned. In addition, mathematical objects such as dimensions, sets, spaces and functions were implemented. The sub-framework SciUI for standardized user interaction has already been mentioned.

One special sub-module, namely that for machine learning, is of prominent importance. Here, the fundamental properties of adaptive systems are defined uniformly for all subtopic packages and are explained in more detail in the following section.

3.2. Implementation of machine learning

Regardless of any learning paradigms, we can state that ML models have the ability to adapt and that this adaptivity is in turn highly influenced by hyperparameters (e.g. the number of layers of an artificial neural network). ML models are embedded in a kind of scenario

[4] A wrapper encapsulates external functionalities or data and maps them in a standardized way into the own framework.

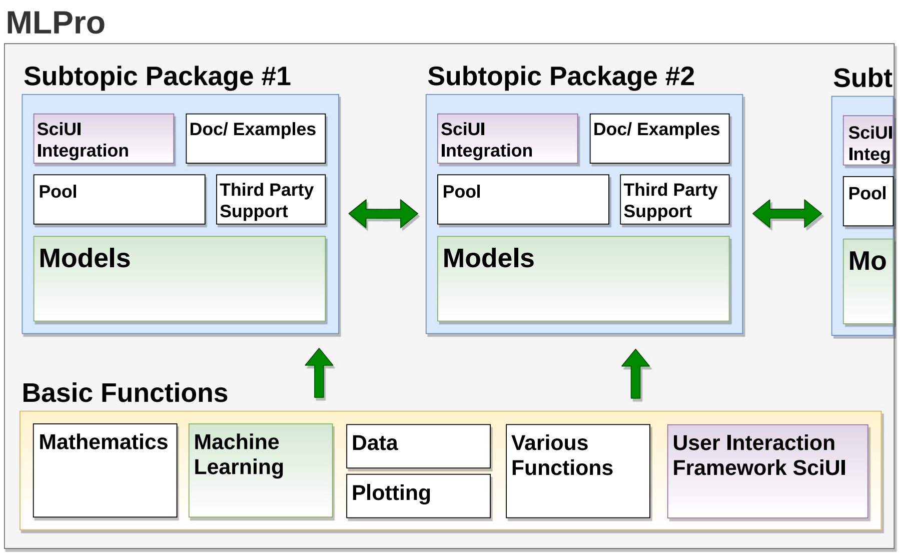


Fig. 5. Common architecture of MLPro.

in which they have to solve a concrete and well-defined task. The adaptation itself often takes place in separate training runs in which the surrounding scenario is simulated. Then the trained ML model is ready to solve the same task in the same scenario in real operation. The concrete internal structure of a scenario is not further defined on this abstract level. But what can be defined is that a scenario includes an ML model and that it can be executed. Furthermore, it can either be in simulation mode or real operation.

Applied to the SL paradigm, a scenario might consist of a ML model with a neural network for pattern recognition and a data set of sample images for training. It would also allow access to images from a camera. In the simulation, the ML model would be presented with images from the sample data set for processing, while in real operation the camera would be accessed. The training itself would be based on the sample data set. In the case of RL, a ML model (in this case an agent) interacts with an environment, which in turn simulates state transitions or induces them into a real hardware configuration via actuators.

Following these basic considerations, a hierarchy of classes was designed, which is shown in Fig. 6 in a simplified form. First of all, the template classes for the named main components ML model, scenario and training are shown in green. The class *Model* essentially contains an adaptation mechanism for later implementation by inheriting classes (blue-colored method). Besides, hyperparameter management consisting of a hyperparameter tuple as an element from a hyperparameter space is also included. Here, basic properties of mathematical dimensions, spaces and elements have been reused through inheritance. The class also offers internal mechanisms for buffering data.

The class *Scenario* serves as a template for higher-level ML applications. It standardizes the initialization and execution in the simulation or real operation mode and in particular introduces processing as a sequence of processing cycles.

The class *Training* is also a template class for implementation in higher layers. It is used to train the ML models contained in scenarios. The training result includes in particular a scalar – however calculated – high score.

Optional tuning of the hyperparameters of the embedded ML model is already fully implemented at this level. Here the hyperparameters are set by a separate tuner through repetitive execution of the training so that the high score of the training is maximized. The tuner in turn involves professional third-party tools such as Optuna (Akiba et al., 2019) or Hyperopt (Bergstra et al., 2015), for which appropriate wrapper classes are made available.

Finally, the special template class *AdaptiveFunction* (not shown in Fig. 6) recombines the properties of a mathematical function with the adaptivity of an ML model. It is the basis for the standardized creation of adaptive approximation functions.

3.3. Development, quality assurance and documentation

MLPro is consistently designed and programmed in an object-oriented manner (OOD/OOP), whereby the Python programming language was chosen due to its uninterrupted popularity and distribution, especially in the context of machine learning. Like many other open-source projects, MLPro is managed in GitHub[5] and can easily be obtained from the Python Package Index (PyPI).[6] Additionally, a comprehensive documentation is provided on the public documentation platform Read the Docs.[7] In addition to detailed explanations of all functions, numerous executable example programs, detailed class diagrams and technical API documentation are also provided here. Quality is ensured through test-driven development and automated unit tests. The source code itself is subject to the clean code paradigm.

Interested researchers, engineers or developers are welcome to visit our public project platforms and give feedback or make suggestions for improvements.

4. MLPro-RL — The sub-framework for reinforcement learning

Reinforcement learning (RL) is an evolving branch in machine learning that handles goal-oriented programming by seeking optimal sequential decision making to maximize rewards (Sutton & Barto, 2018). Here, model-based single- and multi-agent systems (MARL/MBRL) in combination with action planning (MPC) represent a current research focus. The standardization of this in a framework places increased demands on the design and architecture.

In addition to a flexible template for multi-agent-capable environments, there is also a need for templates for adaptive environment models, which in turn are based on adaptive functions for predicting state transitions and agent rewards. This in turn requires a powerful single-agent template that can internally manage both the policy itself and an optional environment model combined with an interchangeable action-planning algorithm. Another template for multi-agents to manage any number of (weighted) single-agents is needed as well.

[5] MLPro@GitHub: https://github.com/fhswf/MLPro.
[6] MLPro@PyPI: https://pypi.org/project/mlpro.
[7] MLPro@Read the Docs: https://mlpro.readthedocs.io.

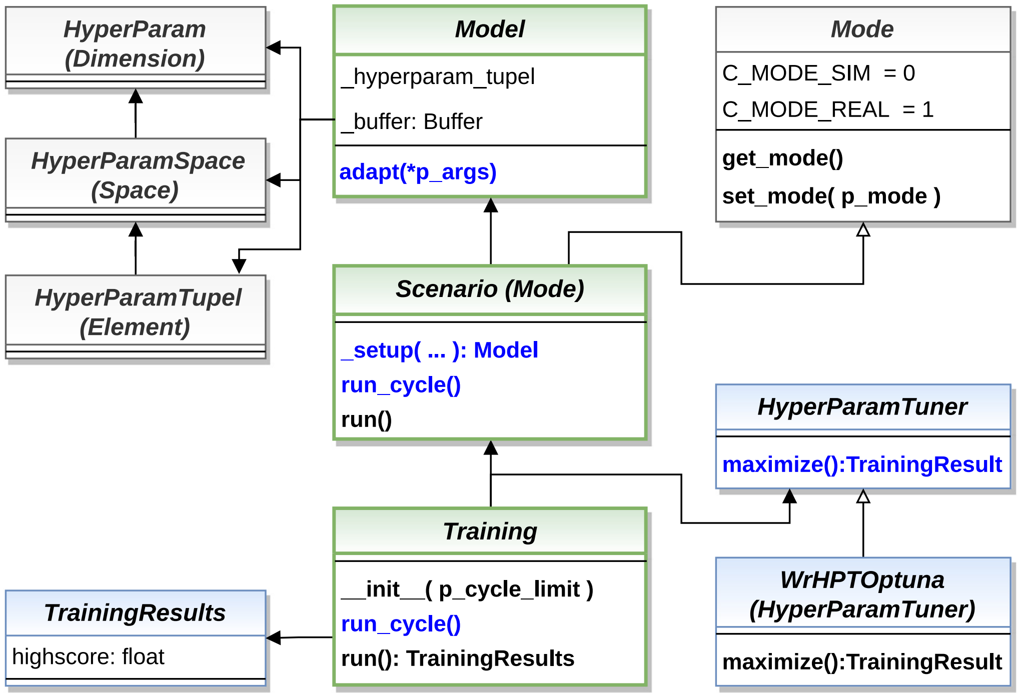


Fig. 6. Machine Learning in MLPro — Simplified class diagram. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

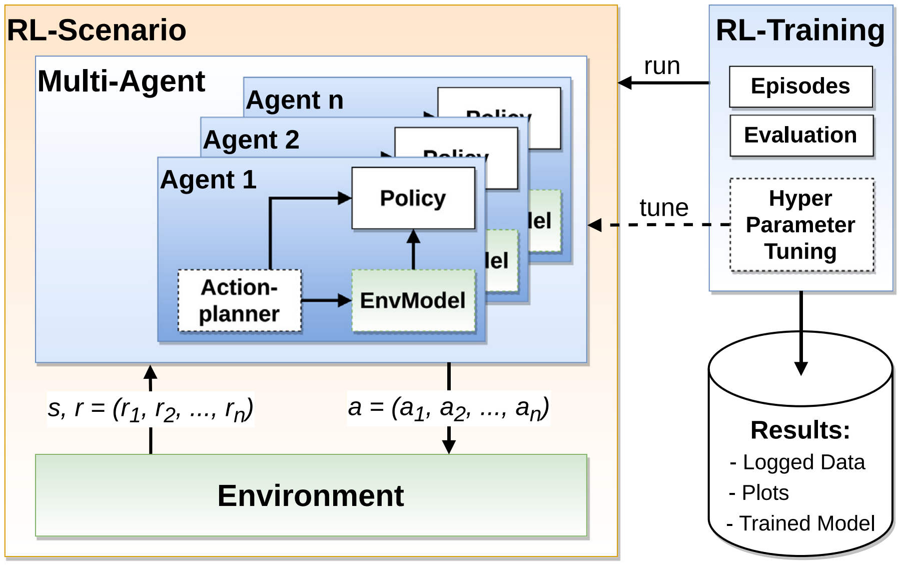


Fig. 7. Structure of MLPro-RL.

MLPro-RL makes these standards available and thus goes beyond the status quo of available OO models from other frameworks. Furthermore, MLPro-RL (and MLPro as a whole) follows the holistic approach that a real ML task consists not only of simulating a problem but also of overcoming a real requirement. In this respect, implementations for the simulation and, for example, the control of the associated real industrial plant can be placed in the same environment model.

The underlying structure of MLPro-RL is shown in Fig. 7. The following Sections 4.1 and 4.2 detail the environment and agent models. Then the concepts of episodic training with cyclical evaluation and stagnation detection are explained in Section 4.3. Finally a brief overview of the continuously growing number of sample environments and other freely reusable pool objects is given in Section 4.4.

4.1. Environments

MLPro-RL provides two main classes for environments that are colored green in Fig. 8. The first class *Environment* serves as a template for designing environments in the sense of RL. It is part of an RL scenario and exchanges state, action and reward information with an agent. The second class *EnvModel* is part of a model-based agent. It is used to automatically find an internal representation of the agent's environment and, as such, is adaptive.

Both classes inherit the basic properties of environments from a common base class *EnvBase*. This introduces the state and action space and defines methods for resetting an environment (and especially an included random number generator to a defined initial state that is

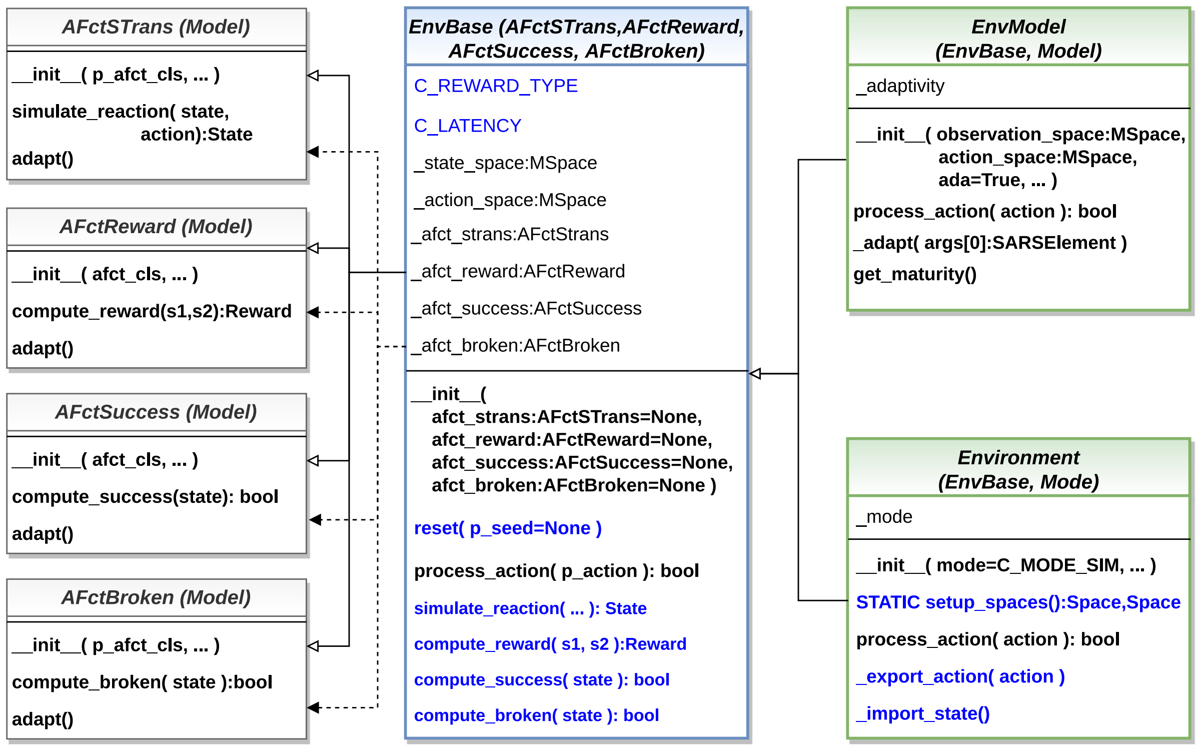

AFctSTrans (Model)
__init__(p_afct_cls, ...)
simulate_reaction(state, action):State
adapt()
AFctReward (Model)
__init__(afct_cls, ...)
compute_reward(s1,s2):Reward
adapt()
AFctSuccess (Model)
__init__(afct_cls, ...)
compute_success(state): bool
adapt()
AFctBroken (Model)
__init__(p_afct_cls, ...)
compute_broken(state):bool
adapt()
EnvBase (AFctSTrans,AFctReward, AFctSuccess, AFctBroken)
C_REWARD_TYPE
C_LATENCY
_state_space:MSpace
_action_space:MSpace
_afct_strans:AFctStrans
_afct_reward:AFctReward
_afct_success:AFctSuccess
_afct_broken:AFctBroken
__init__(afct_strans:AFctSTrans=None, afct_reward:AFctReward=None, afct_success:AFctSuccess=None, afct_broken:AFctBroken=None)
reset(p_seed=None)
process_action(p_action): bool
simulate_reaction(...): State
compute_reward(s1, s2):Reward
compute_success(state): bool
compute_broken(state): bool
EnvModel (EnvBase, Model)
_adaptivity
__init__(observation_space:MSpace, action_space:MSpace, ada=True, ...)
process_action(action): bool
_adapt(args[0]:SARSElement)
get_maturity()
Environment (EnvBase, Mode)
_mode
__init__(mode=C_MODE_SIM, ...)
STATIC setup_spaces():Space,Space
process_action(action): bool
_export_action(action)
_import_state()

Fig. 8. Environments in MLPro-RL — Simplified class diagram. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

called *seeding*) as well as for executing a state transition after the induction of an external action.

Further elementary methods for simulating a state transition, determining a reward for the agent or the qualitative assessment of the current state (done/broken) are inherited from specialized adaptive function classes (*AFct**). Their actual purpose, however, is to train an embedded universal adaptive function (see Section 3.2, class *AdaptiveFunction*) for purposes of prediction by using sample data. The *AFct* classes can in turn be used by the environments as alternative external functions for determining the subsequent status, reward, etc.

Another fundamental characteristic is the *reward type*. MLPro-RL supports a total of three types:

- Overall scalar reward
- Scalar reward per agent
- Scalar reward per action and agent

The first type corresponds to the reward of a single agent, while the other two types are intended for a more differentiated reward of multi-agents.

Finally, the class *EnvBase* class also introduces the *latency* of an environment. This is the duration it takes for an environment to respond to a new action by an agent. In a simulation, this is exactly the *virtual period of time* by which the internal system dynamics must be calculated forward to determine a subsequent state. In real operation, on the other hand, this is exactly the *real period of time* that has to be waited after the induction of an action (e.g. via actuators) until a subsequent state (e.g. via sensors) can be determined.

The class *Environment* also provides a mode with the possible values *simulation* or *real operation*. In the simulation mode, method *process_action* executes the inherited custom method *simulate_reaction*, which now calculates forward a simulation step of duration *latency*. In the real operation mode, however, the action is exported to the real environment using the custom method *_export_action*. After that, there is a real waiting period of duration *latency* before the next state is determined by calling the custom method *_import_state*. In this way, holistic environments can be implemented that contain both calculation rules for the simulation and functions for communication with the real world.

In contrast, the class *EnvModel* has no mode. Rather, it is adaptive in the sense that it uses up to four adaptive functions for the prediction of the state transition, rewarding and state assessment (done/broken). In opposite to the class *Environment*, however, these functions are not trained in advance but rather at runtime. Each adaptive function can also be replaced by an internal custom implementation, provided that a corresponding calculation rule is known. In the special case that state and action space of the environment match with the observation and action space of the agent, the environment itself can also be used instead of each of the adaptive functions. After all, it is compatible with them because of inheritance.

The provision of wrapper classes for the packages OpenAI Gym (Brockman et al., 2016) and PettingZoo (Terry et al., 2020) enables the integration of an abundance of already existing environments. For the opposite direction, further wrapper classes have been created that convert native MLPro-RL environments into the format of Gym or PettingZoo. Thus, the user is free to choose either the format of one of the named third-party providers or the native MLPro-RL format for a new environment.

In summary, however, we would like to emphasize again the advantages that qualify the native environment model of MLPro-RL in particular for real applications:

- One template for single and multi-agent scenarios
- Simulation or real operation mode
- Consideration of the reaction time (*latency*)
- Distinction of the terminal states according to *goal achieved (done)* and *environment lost (broken)*
- No reset necessary after the goal has been achieved (done state)
- Units of variables/dimensions can be defined
- Alternative setup based on adaptive functions pre-trained with sample data, in case that the calculation equations for status transition, reward or state assessment are not known

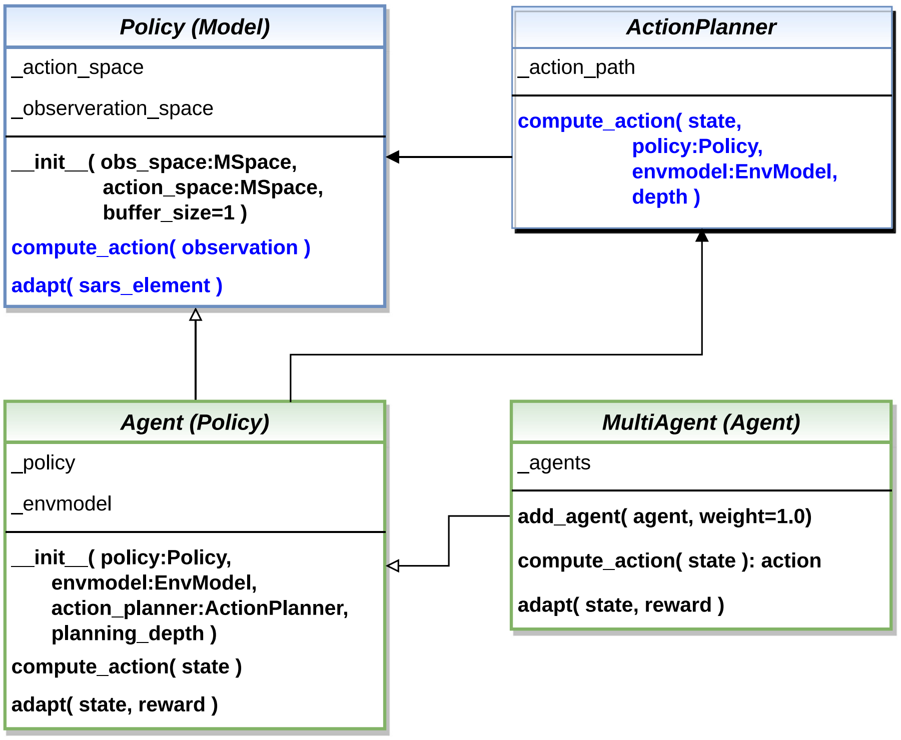

Policy (Model)
_action_space
_observeration_space
__init__(obs_space:MSpace, action_space:MSpace, buffer_size=1)
compute_action(observation)
adapt(sars_element)
ActionPlanner
_action_path
compute_action(state, policy:Policy, envmodel:EnvModel, depth)
Agent (Policy)
_policy
_envmodel
__init__(policy:Policy, envmodel:EnvModel, action_planner:ActionPlanner, planning_depth)
compute_action(state)
adapt(state, reward)
MultiAgent (Agent)
_agents
add_agent(agent, weight=1.0)
compute_action(state): action
adapt(state, reward)

Fig. 9. Agents in MLPro — Simplified class diagram.

4.2. Agents

The agent model landscape of MLPro-RL ranges from simple single-agents with adaptive policy to model-based agents (MBRL) with optional Action Planner to multi-agents (MARL), which in turn can consist of any number of single-agents (see Fig. 9).

The core component of every single-agent is the policy, which inherits the basic properties of adaptivity from the subordinate ML model from Section 3.2 and extends it by the RL-specific function of the action calculation. The class of the same name is designed as a template class. Own algorithms can thus be implemented as inheriting classes and easily incorporated into the framework. In addition, a wrapper class for algorithms of the framework Stable Baselines3 (Raffin et al., 2019) is available.

The single-agent in turn inherits the interface of the policy, since its core functionality of calculating actions and adapting ultimately corresponds to that of the policy. In the simplest case, a single-agent is actually nothing more than a wrapping around a policy.

Optionally, an environment model (see class *EnvModel* from Section 4.1) can be added to a single-agent. This is used to learn the behavior of the part of the environment that is visible to the agent in order to be able to make predictions about their reaction to calculated actions. In this sense, the environment model is integrated into the agent's adaptation mechanism and is always retrained if the predictions of the subsequent state and reward deviate too much from the actual values of the environment. An adaptation of the environment model in turn entails an adaptation of the policy. This is based on an internally running episodic training (see next Section 4.3) of the policy in interaction with the environment model.

Another optional extension of the single-agent is an *action planner*. It uses an environment model to plan the next actions in advance. The aim of advance planning is to find the shortest possible sequence of actions that maximizes the reward of the environment or, ideally, even puts it in a *state of success*. This procedure essentially corresponds to Model Predictive Control (MPC) described in Williams et al. (2017). The associated class *ActionPlanner* is also designed as a template class, so that own classes inherited from it can be seamlessly integrated.

The multi-agent model rounds off the scope of the agent landscape. It is compatible with single-agent, but does not have its own policy. Rather, it is used to incorporate and manage any number of single-agents that cooperatively take over the action calculation. Here, each single-agent operates on a disjoint subspace of the observation and action space of the surrounding multi-agent. Multi-agents interact with corresponding environments that support the reward type *scalar reward per agent*. These are native developments that use the MLPro environment template or environments from the PettingZoo (Terry et al., 2020) ecosystem that can be integrated using the associated wrapper class provided by MLPro.

4.3. Episodic training with evaluation and hyperparameter tuning

As already described in Section 3.2, the basics for the definition of ML scenarios and the training of the ML models contained therein are laid in MLPro on a lower and correspondingly fundamental level. In MLPro-RL, classes are provided that take up these basic concepts and expand them for the corresponding requirements of RL. First of all, a class *RLScenario* is available that inherits from the template class *Scenario*. This combines an environment and a (multi-)agent into an executable unit that serves as the basis for both training and real operation.

The class *RLTraining* is available for training and hyperparameter tuning of agents, which in turn inherits from the template class *Training*. It implements episodic training algorithms tailored to RL and makes the corresponding extended training data and results as well as the trained agents available in the file system.

In the simplest case, the training itself can be limited by a predetermined number of training cycles. As usual in RL, these are grouped into episodes that begin with a (defined) random initial state of the environment and end with at least one of the following three events:

1. Event *Success*
   The environment reaches a defined target state. Whether this also ends the episode depends on the specific task of the environment.

2. Event *Broken*
   The environment reaches a state from which the control objective can no longer be achieved. The episode definitely ends here.
3. Event *Timeout*
   A maximum number of training cycles per episode specified by the environment has been exceeded. The episode ends here too.

At the end of the training, detailed data logs for the course of actions, states and rewards as well as a summary are made available in the file system.

Optionally, a maximum number of adaptations of the agent can also be specified. In addition, the class includes what is known as stagnation detection. In this special mode of operation, the class is alternately in either the training or the evaluation phase. In the training phase, a defined number of episodes is completed in which the agent is adaptive. As a result, each episode begins with a reproducible individual random state. A training phase is followed by an evaluation phase in which the adaptivity of the agent is deactivated. Episodes are now completed which are used solely to evaluate the current training status. Each episode of an evaluation starts in an individual – but always the same – random state, which enables direct comparability of the results of all evaluations.

A scalar rating (score) is determined for each evaluation. To this end, the rewards of all the m agents involved are first added up for each episode e to form an m-tuple

$$r_e = \left(\sum_{c=1}^{n_e} r_c^{(a)} \right)_{a=1}^{m} \tag{1}$$

where n_e is the number of cycles of the episode e and $r_c^{(a)}$ is the scalar reward of agent a in cycle c. At the end of the evaluation, the score is calculated as the average reward across all o episodes and m agents by

$$s = \frac{1}{m} \sum_{a=1}^{m} \left(\frac{w_a}{o} \sum_{e=1}^{o} r_e^{(a)} \right) \tag{2}$$

where $w_a \in [0, 1]$ is the weight of agent a. In the special case of a single-agent scenario this weight is 1 and the score is calculated in a simplified way by

$$s = \frac{1}{o} \sum_{e=1}^{o} \sum_{c=1}^{n_e} r_c \tag{3}$$

The goal of the training is now to maximize the score of the repetitive evaluations. There is a *stagnation* when an evaluation does not achieve a higher score than the previous one. The training ends after reaching a specified number of consecutive stagnations. The result is the highscore achieved up to that point and the number of training cycles required for this, which enables different agent configurations to be compared directly (benchmarking).

Optionally, the agent's hyperparameters can also be tuned. The basic mechanism is inherited from the template class *Training* described in Section 3.2. In the course of the tuning, the highscore of the training is maximized depending on the values of the hyperparameters.

4.4. Pool objects

MLPro-RL provides a pool of objects that consists of a collection of ready-to-use RL environments, RL algorithms, RL scenarios, buffering methods and other reusable objects. The main objectives are to enable users to straightforwardly and effectively utilize the available objects for their work or research, apply their developed learning algorithms into different environments, measure the performance of the applied approaches in a standardized way, and compare them to the state-of-the-art algorithms. Therefore, MLPro supports researchers in machine learning study to validate their approaches, present their results accurately, and get accepted by the communities.

To this extent, MLPro-RL provides a native implementation of several RL environments as well as an extension of third-party environments, for instance, a simple grid-world problem, UR5 joint control, a bulk good system (Schwung, Yuwono, Schwung, & Ding, 2021), and a multi-agent cart pole based on CartPole-v1 of OpenAI Gym Environment (Brockman et al., 2016). When implementing pool objects, we apply the same high standards of quality and code hygiene as to the framework as a whole. Developers and researchers are welcome to give feedback on existing objects or propose new ones.

5. MLPro-GT — The sub-framework for game theory

The second subtopic package of MLPro is committed to Game Theory (GT) for solving decision-making problems in primarily multi-player systems. GT is a notable branch in economic studies as a hypothetical approach to model the strategic interaction between numerous players or individuals in a particular circumstance (Sevgi & Orbay, 2014). Recent years have witnessed a growing academic interest in GT with engineering applications (Bauso, 2016). Particularly, the cooperative GT-based learning algorithms with the potential game (PG) has been proven to successfully solve MARL problems (Schwung, Schwung, & Ding, 2020), since the interaction between each player and also the related state information can be properly incorporated into the games to enhance the optimization efficiency (Marden, 2012). The GT-based approach could fill the gap in real industrial cases due to lengthy training time and the necessity of high-sophisticated tools to train the ML models. This approach requires less computation and intricacy in optimization of the ML models rather than a deep RL-based approach, in particular learning from the deep neural network does not appear to be generally conceivable in every real production environment.

The fundamental of the RL framework contains interactions between the environment and the agent, where the agent selects an action according to the actual state of the environment and optimize its policy based on the handed over reward value from the environment. This process is repeated again and again. The agent aims to find a way to maximize its individual long-term reward regardless he/she deals with a single-agent system or multi-agents system. Thus, we could conclude that the agent's policy is the decision rule that offers which best action to be selected in which state. Meanwhile, in the cooperative GT, a player aims to find an optimal policy that benefits all corresponding players that share the same game board instead of its individual advantage. The main idea of GT itself is to design effective strategic interactions between players as a game. Furthermore, single-agent applications of RL might not need any GT framework. For multi-agent applications, however, the GT techniques can be incorporated, e.g. in the reward function, policy adaptation, states generalization, etc., using strategic-form, extensive-form, state-based games, and many more.

GT framework can be incorporated into the multi-agent case of a Markov game. However, different terminologies are applied to transform a Markov game into a GT-based game. Fig. 10 displays the multi-agent Markov decision process. The equivalences of RL and GT in the context of the Markov game are each agent/player select an action independently and the interaction between environment-agents and game board-players. However, in the GT-based game, strategic interaction between the players is automatically involved through implemented GT techniques, which is not available in the basic Markov game, see Fig. 11 for overview.

Therefore, we believe that a sub-framework for the cooperative GT approach that focuses on Markov games is necessary, which is called MLPro-GT. We also would like to highlight that our framework is reusable by developing the sub-framework that reuses several functionalities of MLPro-RL. A Markov game consists of a set of independent players, which make decisions simultaneously. The existing RL sub-framework can be reused because the ML models and training procedures between both approaches are comparable despite different terminologies, distinctive rewarding functions, and some additional GT

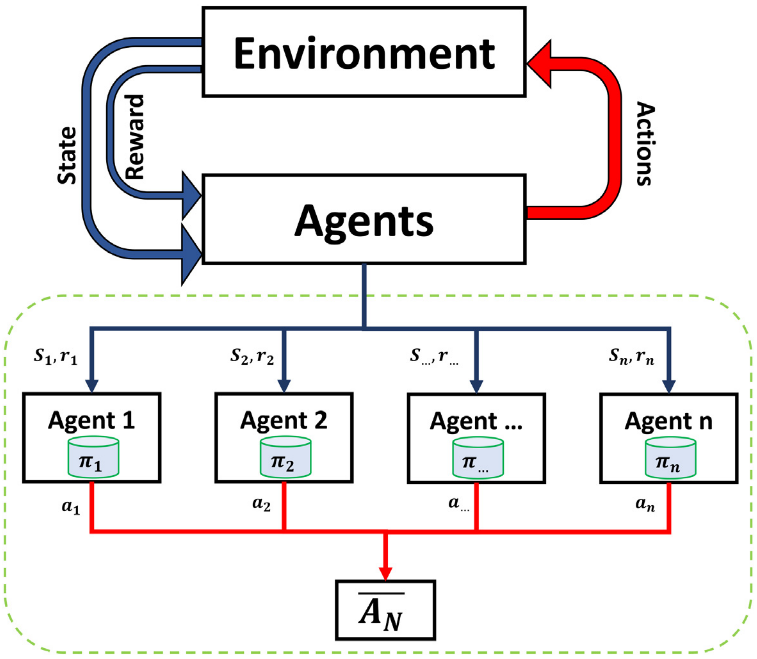


Fig. 10. A multi-agent and environment interaction in the Markov decision process.

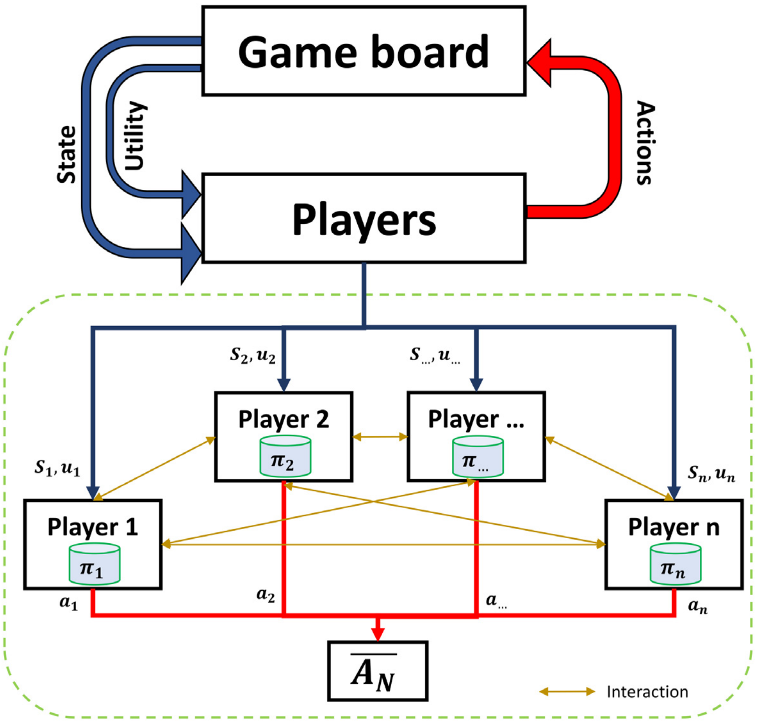


Fig. 11. Incorporation of GT framework in a Markov game.

specifications. For instance, in the field of GT, we define *Environment* as *Game Board*, *Agent* as *Player*, and *Scenario* as *Game*. As a consequence, we can basically inherit a few functionalities provided by MLPro-RL to MLPro-GT, e.g. class *Game Board* of MLPro-GT gets an inheritance from class *Environment* of MLPro-RL. To be noted, the further details of MLPro-RL are comprehensively described in Section 4.

Fig. 12 displays the integration between MLPro-GT as a child package and MLPro-RL as a parent package in a simplified form, where a portion of functionalities of the child package inherits from its parent class. The main advantage of the GT-based approach is the capability to studies on convergence, especially for searching the local and/or global optima of a system. Of course, this leads to a different implementation strategy than RL. We can infer that the global objective of a game is to maximize the overall utility $\max_{a \in A} \phi(a, S)$ of the system, i.e. potential value, that depends on the action profile and state combination by jointly maximizing local utilities $U_i(a_i, S^{A_i})$ for player i (Schwung et al., 2020).

This sub-package provides a complete set of template classes to run a GT scenario or a game, including class *Player*, class *GameBoard*, class *GTTraining*, etc. Hyperparameter tuning of players is also possible to be executed using the provided functionality by the wrapper class of hyperparameter tuners. GT-approach is mainly intended for multi-agents

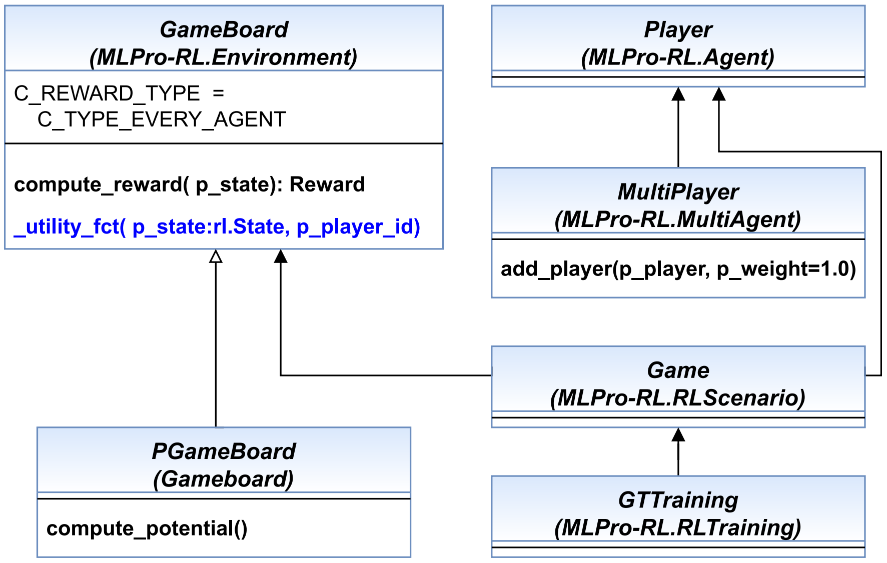

GameBoard (MLPro-RL.Environment)
C_REWARD_TYPE = C_TYPE_EVERY_AGENT
compute_reward(p_state): Reward
_utility_fct(p_state:rl.State, p_player_id)
Player (MLPro-RL.Agent)
MultiPlayer (MLPro-RL.MultiAgent)
add_player(p_player, p_weight=1.0)
Game (MLPro-RL.RLScenario)
PGameBoard (Gameboard)
compute_potential()
GTTraining (MLPro-RL.RLTraining)

Fig. 12. Simplified class diagram of MLPro-GT.

```
from mlpro.gt.pool.boards.bglp import BGLP_GT
```

Listing 1: Importing BGLP gameboard from the pool of objects.

systems. However, we have not excluded the possibility of processing single-player as well as model-based learning. Thus, the scope is not limited to multi-player problems. MLPro-GT provides reusable pool objects (see Section 4.4) that consist of ready-to-use GT game boards and algorithms. Alternatively, the user can straightforwardly import an RL environment from the pool objects and reuse it as a game board, as demonstrated in BGLP[8] and multi-cart–pole[9] game boards, or from the existing packages OpenAI Gym (Brockman et al., 2016) and PettingZoo (Terry et al., 2020) via the wrapper classes. Additionally, this framework allows us to compare the performance between the cooperative GT-based algorithms and the state-of-the-art RL algorithms in a standardized way.

To conclude this section, MLPro-GT has shown the potential of reusing existing functionalities from another sub-framework within MLPro and extending the developed sub-framework into a more extensive region. An augmentation to other subtopics inside this framework is practicable in light of the fact that the engineering of MLPro permits us to do as such, as it is comprehensively explained in Section 3.1.

6. MLPro in Practice

In this section, we explain the processes of implementing own RL/GT scenarios in MLPro. Furthermore, we describe two sample applications that we elaborated in MLPro to validate and demonstrate the core functionalities of the framework, such as a simulation of a bulk good system and control of a UR5 Robot Arm.

[8] BGLP@GitHub: https://github.com/fhswf/MLPro/blob/main/src/mlpro/gt/pool/boards/bglp.py.

[9] multi-cart–pole@GitHub: https://github.com/fhswf/MLPro/blob/main/src/mlpro/gt/pool/boards/multicartpole.py.

```
from stable_baselines3 import PPO
from mlpro.wrappers.sb3 import WrPolicySB32MLPro

self._env = UR5JointControl(p_logging=p_logging)

policy_sb3 = PPO(
        policy="MlpPolicy",
        n_steps=20,
        env=None,
        _init_setup_model=False,
        device="cpu",
        seed=1)

policy_wrapped = WrPolicySB32MLPro(
    p_sb3_policy=policy_sb3,
    p_cycle_limit=5500,
    p_observation_space=self._env.get_state_space(),
    p_action_space=self._env.get_action_space(),
    p_ada=True,
    p_logging=True)
```

Listing 2: Reusing SB3 policy.

6.1. Implementation of own RL/GT scenarios

MLPro 1.0 offers a complete solution of RL and GT tasks with standardized processes. In this subsection, we would like to describe the

```python
from mlpro.rl.models import *

class SbPG_GlobI(Policy):

    def __init__(self, p_observation_space:MSpace,
                p_action_space:MSpace,
                p_buffer_size=1,
                p_ada=1.0,
                p_logging=True):
        .....

    def _init_hyperparam(self):
        # Method to set up hyperparameters
        self._hyperparam_space.add_dim(HyperParam('lr_rate','R'))
        self._hyperparam_space.add_dim(HyperParam('num_states','Z'))
        self._hyperparam_tuple = HyperParamTuple(self._hyperparam_space)

        ids_ = self._hyperparam_tuple.get_dim_ids()
        self._hyperparam_tuple.set_value(ids_[0], 0.01)
        self._hyperparam_tuple.set_value(ids_[1], 40)

    def compute_action(self, p_state: State) -> Action:
        ....
        self.action_current = ....
        return Action(self._id,
                      self._action_space,
                      self.action_current)

    def _adapt(self, *p_args) -> bool:
        if not self._buffer.is_full():
            return False

        ....
        return True
```

Listing 3: Creating a custom policy.

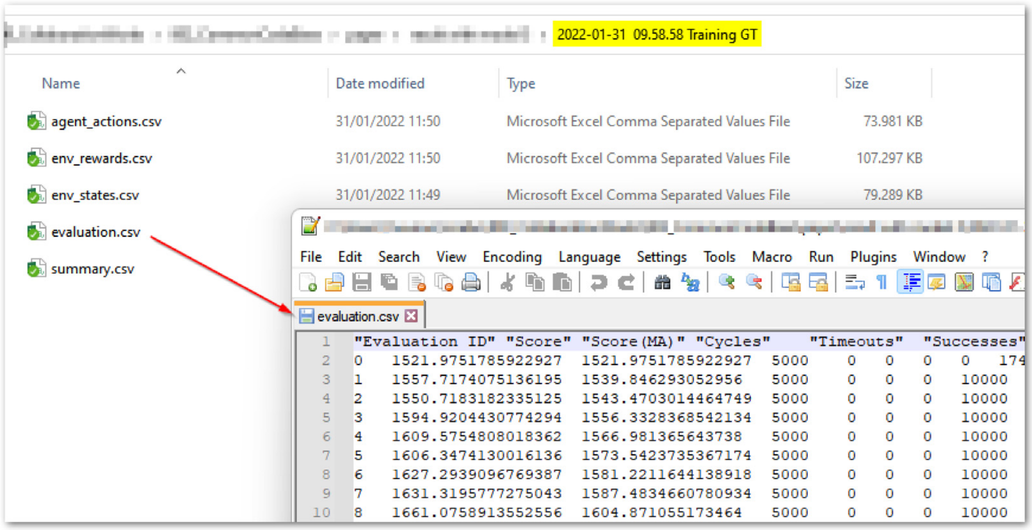
implementation steps of developing and executing RL/GT scenarios using the framework. Table 1 presents an overview of the implementation steps.

The implementation steps for both RL and GT training are almost identical despite different used packages. First of all, we need an environment (or a game board) that is compatible with the MLPro framework. There are three possibilities to build the environment. The first possibility is by loading an environment from the environment pool. For example, in the first sample application, the BGLP environment is a native environment of MLPro and has been automatically bundled into the framework. Thus, we can easily import the environment from the environment pool, as shown in Listing 1. Additionally, we also provide wrapper classes to translate Gym or PettingZoo environments to MLPro compatible environments, i.e. Class *WrEnvGYM2MLPro* and *WrEnvPZOO2MLPro* respectively. The last possibility is by creating a custom environment by inheriting from class *Environment* for RL or class *GameBoard* for GT.

Second, a policy has to be defined. There are three main possibilities to set up a policy, such as (1) an SB3 policy, (2) a native MLPro policy,

Table 1
An overview of implementing own RL/GT scenarios.

No.	Step	Remarks
1	Building an environment/game board.	The environment/game board can be customized, loaded from MLPro's pool of object, or adopted from third party packages via wrapper classes, e.g. OpenAI Gym environment.
2	Implementing an agent policy.	The policy can be customized, loaded from MLPro's pool of object, or adopted from third party packages via wrapper classes, e.g. Stable Baselines3.
3	Setting up a scenario or a game.	Inheriting from class *Scenario* for RL or class *Game* for GT.
4	Setting up a training and executing the scenario.	Inheriting from class *RLTraining* for RL and *GTTraining* for GT.



[redacted] › 2022-01-31 09.58.58 Training GT

Name	Date modified	Type	Size
agent_actions.csv	31/01/2022 11:50	Microsoft Excel Comma Separated Values File	73.981 KB
env_rewards.csv	31/01/2022 11:50	Microsoft Excel Comma Separated Values File	107.297 KB
env_states.csv	31/01/2022 11:49	Microsoft Excel Comma Separated Values File	79.289 KB
evaluation.csv			
summary.csv			

[redacted]

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
evaluation.csv

```
"Evaluation ID" "Score" "Score (MA)" "Cycles"     "Timeouts"  "Successes"
0    1521.9751785922927  1521.9751785922927  5000    0    0    0    0    174
1    1557.7174075136195  1539.846293052956   5000    0    0    0    10000
2    1550.7183182335125  1543.4703014464749  5000    0    0    0    10000
3    1594.9204430774294  1556.3328368542134  5000    0    0    0    10000
4    1609.5754808018362  1566.981365643738   5000    0    0    0    10000
5    1606.3474130016136  1573.5423735367174  5000    0    0    0    10000
6    1627.2939096769387  1581.2211644138918  5000    0    0    0    10000
7    1631.3195777275043  1587.4834660780934  5000    0    0    0    10000
8    1661.0758913552556  1604.871055173464   5000    0    0    0    10000
```

Fig. 13. An example of generated CSV files after a training.

```python
from mlpro.rl.models import *
from mlpro.rl.pool.envs.ur5jointcontrol import UR5JointControl
from stable_baselines3 import PPO
from mlpro.wrappers.sb3 import WrPolicySB32MLPro

class ScenarioUR5A2C(RLScenario):
    C_NAME = 'Matrix'

    def _setup(self, p_mode, p_ada, p_logging):
        self._env = UR5JointControl(p_logging=p_logging)

        policy_sb3 = PPO(....)
        policy_wrapped = WrPolicySB32MLPro(
                        p_sb3_policy=policy_sb3,
                        ....
                        )

        self._agent = Agent(
                    p_policy=policy_wrapped,
                    p_envmodel=None,
                    p_name='Smith',
                    p_ada=p_ada,
                    p_logging=p_logging
                )

        return self._agent
```

Listing 4: Building a single-agent scenario.

```python
class BGLP_SbPG_GlobI(Game):

    def _setup(self, p_mode, p_ada, p_logging):
        self._env = BGLP_GT(p_logging=True)
        self._agent = MultiPlayer(
                        p_name='SbPG - GlobI',
                        p_ada=1,
                        p_logging=False)
        state_space = self._env.get_state_space()
        action_space = self._env.get_action_space()

        # Player 1
        pl_ospace = state_space.spawn([0,1])
        pl_aspace = action_space.spawn([0])
        pl_policy = SbPG_GlobI(
                    p_observation_space=pl_ospace,
                    p_action_space=pl_aspace,
                    p_buffer_size=1,
                    p_ada=1,
                    p_logging=False)
        self._agent.add_player(
            p_player=Player(
                p_policy=pl_policy,
                p_envmodel=None,
                p_name='BELT_CONVEYOR_A',
                p_id=0,
                p_ada=True,
                p_logging=True),
            p_weight=1.0
            )

        # Player i
        ....

        return self._agent
```

Listing 5: Building a multiplayer game.

and (3) a custom policy. For the first option, we can simply use a wrapper class *WrPolicySB32MLPro* to make the SB3 policy algorithm compatible with MLPro, as implemented for the second sample application, see Listing 2. Meanwhile, native MLPro policy can be imported directly from the pool of objects. Then, in the first sample application, we establish a custom policy (Schwung et al., 2020) by inheriting a template class *Policy* and redefined few methods, as described in Listing 3. To be noted, the following listing describes the general implementation idea, but this is not the complete code.

Third, we must prepare a training scenario, which can be a single agent (or a player) scenario or a multi-agent (or multiplayer) scenario by inheriting from the template class *Scenario* for RL and *Game* for GT, in which the prepared environment (or game board) and policy are incorporated. Building a single agent scenario is straightforward, as you can see from Listing 4 that describes the implementation of an RL scenario with a standard single agent. To be noted, *p_envmodel* indicates whether it is a model-free or model-based training. Meanwhile, in building a multiplayer game, we need to begin with the initiation of *MultiPlayer* instead of *Player*. Then, the single players are added to the class as part of the multiplayer, see Listing 5 for more overview.

Fourth, we need to set up a training and run the scenario by utilizing *RLTraining* or *GTTraining* classes with predefined training parameters. We allow to enable or disable stagnation detection and visualization, limit the logged information to speed up training, and select desired information to be stored, e.g. states, actions, rewards, and training. Listing 6 provides an example to execute the game of *BGLP_SbPG_GlobI*.

Lastly, by default, MLPro provides logging and visualization (if available) during the training that shows the happening training processes. We are allowed to turn on/off the logging and visualization before the training is started. At the end of the training, several CSV files are automatically generated that contain information stored by *DataStoring*, e.g. actions, score, time, done, and training summary. Fig. 13 shows an example of generated CSV files after a GT training.

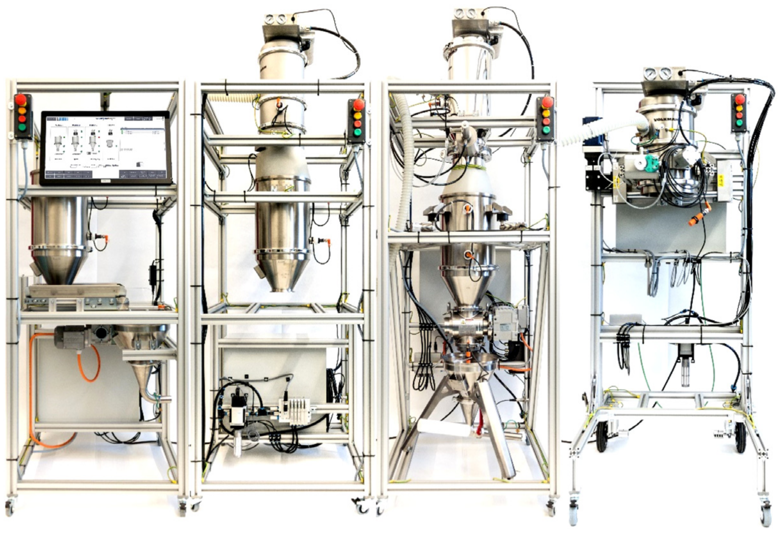


Fig. 14. Illustrative of the BGLP (Schwung, Kempe, Schwung, & Ding, 2017).

```
from mlpro.gt.models import *
from pathlib import Path

training        = GTTraining(
                        p_game_cls=BGLP_SbPG_GlobI,
                        p_cycle_limit=2000000,
                        p_cycles_per_epi_limit=1000,
                        p_eval_frequency=10,
                        p_eval_grp_size=5,
                        p_adaptation_limit=0,
                        p_stagnation_limit=5,
                        p_score_ma_horizon=5,
                        p_collect_states=True,
                        p_collect_actions=True,
                        p_collect_rewards=True,
                        p_collect_training=True,
                        p_visualize=False,
                        p_path=str(Path.home()),
                        p_logging=Log.C_LOG_WE
                  )

training.run()
```

Listing 6: Executing a GT training.

6.2. Sample Application: a Bulk Goods Laboratory Plant

The Bulk Goods Laboratory Plant (BGLP) (Schwung et al., 2017, 2021) illustrates a smart production system with high flexibility and distributed control to transport bulk raw materials. One of the major advantages of the system is the modularity in design, as depicted schematically in Fig. 14. In this manner, each module can be exchanged, independently replaced or based on with different modules inside the system. The BGLP comprises of four modules in a default setting, which are loading, storing, weighing, and filling stations respectively, and has dosing on and conveying units as fundamental pieces of the framework.

The interface between the modules is assembled via a mini hopper placed in the prior module. Then, the next module is fed by a vacuum pump, which operates in a discontinuous manner, before the goods are temporarily stored in a silo of the next module. The filling station has no silo because the main purpose of the station is to occupy the transport containers. Additionally, we utilize dissimilar actuators in modules 1–3 to transport the goods from the silo to the mini hopper. Module 1 utilizes a belt conveyor, that operates between 0 and 1800 rpm. Module 2 uses a vibratory conveyor, which can be completely switched on and off. Lastly, Module 3 utilizes a rotary feeder, that operates between 0 and 1450 rpm. We employ a bunch of sensors to monitor the actual status of every module and control them. Every module is furnished with an independent PLC control system (Siemens ET200SP) and imparts between the modules utilizing Profinet.

We consider the BGLP as a multi-agents and multi-players system in the RL and GT context respectively. The environment of BGLP[10] with default configurations as well as the game board[11] for GT-based approach are both available in the first version of MLPro, where the user can select either continuous or batch production scenarios. However, it is still limited to the simulation up to this point. The BGLP is naturally constructed using class *Environment* as a template for designing the RL environment, instead of using any wrappers, that inherits the basic properties from a base class *EnvBase*. Therefore, the BGLP environment is a native environment of MLPro. In addition, the BGLP validates the multi-agents functionality in MLPro-RL and multi-players functionality in MLPro-GT. One of the main benefits of implementing an RL environment or a GT game board in MLPro is that the package offers a standardized procedure to perform the training and interaction between environment-and-agents or game board-and-players with comprehensively complete features. Aside from template class *Environment*, we also utilize the elementary functionalities provided by MLPro's basic functions, i.e. load/save, logging, timer, scientific object, basic mathematical classes, data handling, etc.

Furthermore, we conduct a test experiment to test, analyze, and validate our framework by training a developed GT-based learning algorithm on BLGP game board using MLPro-GT sub-framework. Generally, each actuator of the system is pointed as a player. Thus, we have 5 players on the field. The states information for each player is the fill levels of the prior reservoir and next reservoir. We refer to the published research in Schwung et al. (2020) as the baseline, which implements a basic potential game with a state-based approach. We

[10] BGLP Environment@Read the Docs: https://mlpro.readthedocs.io/en/latest/content/rl/env/pool/bglp.html.

[11] BGLP Game Board@Read the Docs: https://mlpro.readthedocs.io/en/latest/content/gt/gameboard/pool/bglp.html.

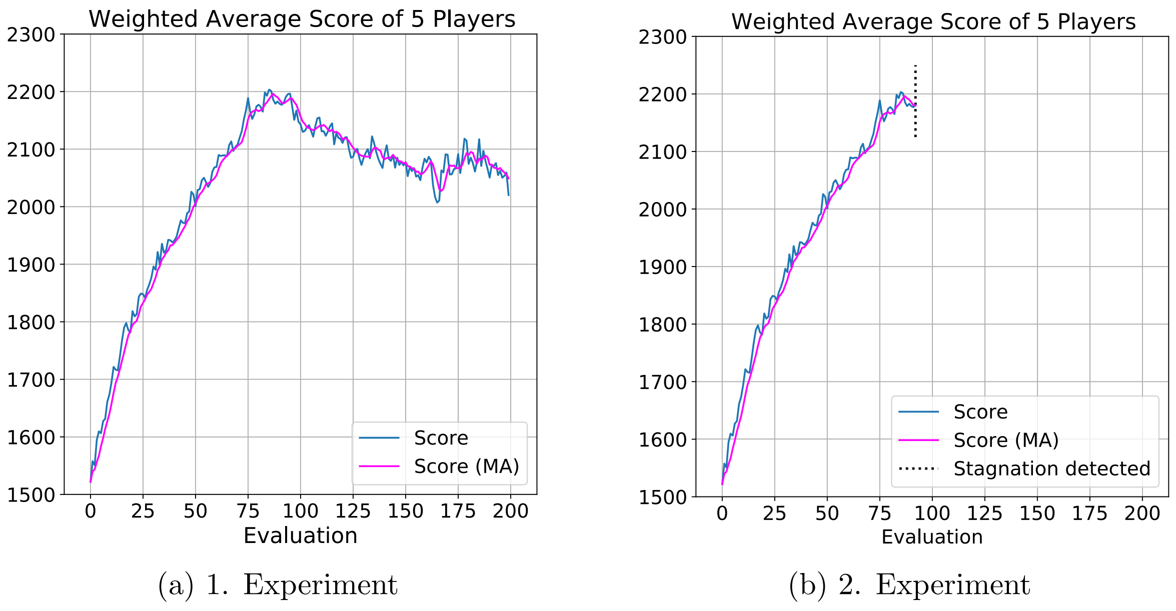


Fig. 15. Results of the basic SbPG approach on the BGLP environment using MLPro without (a) and with (b) stagnation detection.

simply utilize the class *Policy* for implementing the learning algorithm instead of using any wrappers. Unfortunately, we could not provide the source code or a how-to file of the related game as a part of MLPro because we would like to keep the policy algorithm private.

For the first experiment, we set up the basic state-based potential game (SbPG) approach on the BGLP (Schwung et al., 2020) with the global interpolation method and a continuous production scenario. The training duration is limited to 2,000,000 cycles with a maximum of 1,000 cycles per episode, e.g. equals 20,000,000 s in the BGLP game board. The stagnation detection is not applied in the first experiment. We apply the same hyperparameters of the learning algorithm, such as smoothing parameter, exploration decay rate, learning rate for each utility function, and the number of discretized states of the performance maps, as they are proposed in Schwung et al. (2020). Fig. 15 describes the training results of the first experiment with the MLPro framework. For more information, the data to plot the graph are stored and plotted by utilizing the data storing and plotting functionalities of MLPro.

From the first experiment, we could visually detect stagnation in the course of the training. Therefore, we include stagnation detection in the second experiment according to Section 4.3. We set the stagnation limit to 5, which means that the simulation will stop when there is no progress in 5 consecutive evaluations. Additionally, we set the moving average horizon to 5, evaluation frequency to 10 and evaluation group size to 5. It means that an evaluation is performed every 10 episodes with 5 evaluation episodes for each event. Fig. 15 describes the training results of the second experiment and highlights that the stagnation is successfully detected after 920 training episodes (e.g. 920,000 s in the BGLP game board) and 93 evaluations.

Those results are comparable with the native implementation (Schwung et al., 2020), i.e. without MLPro. The author limited the training time to 1,000,000 s manually after observing several test runs that more training will not bring any progress and lead to over-fitting. In addition, Fig. 16 presents the weighted average score of each player and emphasizes the advantage of cooperative GT learners where one agent gave up his local objective (i.e. Player 5) to maximize the global objective. Since a selfish player could cause a negative influence on the whole performance of the system. This phenomenon often occurs in the cooperative game. An insignificant difference between both of them understandably occurs due to the random initialization and the continuous production process in a long period cause the minor difference. However, we can conclude that the training sequence works properly, including the player-game board interaction, and MLPro does not lead to any performance loss in any event.

6.3. Sample Application: a UR5 Robot Arm

Universal Robots is a company that manufactures collaborative robots (cobots). One of their products is called UR5, which comprises 6 joints. An environment setup is developed based on our current situation in our lab, where the robot is placed on top of a table. A gripper from Robotiq is attached to the end-effector of the robot. A camera is placed in front of the robot. The robot's main task in our lab is to pick an object from a pick area and place it in the place area. The pick area is the area with the blue tape. The place area is the area with the red tape. The environment setup in our lab is shown in (Fig. 17a). Its digital twin (Fig. 17b) was set up in the Gazebo Simulator. The simulation environment is developed on top of the OpenAI Gym environment, based on the Robot Operating System (ROS) (Stanford Artificial Intelligence Laboratory et al., 2018). The movement of the robot is controlled by MoveIt (Coleman, 2014), where it provides the path planning algorithm to move the end-effector of the robot from one point to another point. The simulated environment is developed in such a way that it is the same as the real situation. Hence, reducing the complexity of the sim-to-real transition.

The RL scenario presented here consists of the digital twin for the UR5 robot as described and a single-agent based on a proximal policy optimization algorithm (PPO) (Schulman et al., 2017). Here the SB3 implementation of PPO was taken and incorporated by using the MLPro wrapper for SB3. The Gym-style UR5 environment was in turn embedded by using the MLPro wrapper for Gym environments.

The simulated environment starts in a predefined initial position. To reduce the complexity of the task, the agent's task is not to pick and place an object as in our lab. Instead, its task is to reach a fixed destination position p_d (blue ball in Fig. 17b) with the end effector position p_t of the robot (red ball in Fig. 17b) by stepwise providing angle changes for each joint. The reward function is shown in Eq. (4) as r_t.

$$r_t = -\frac{de_t}{di} - pe \tag{4}$$

where t is the timestep, de_t is the distance error per timestep, di is initial distance and pe is the penalty. The penalty value is an empirical constant. The distance error can be calculated as in Eq. (5)

$$de_t = p_t - p_d \tag{5}$$

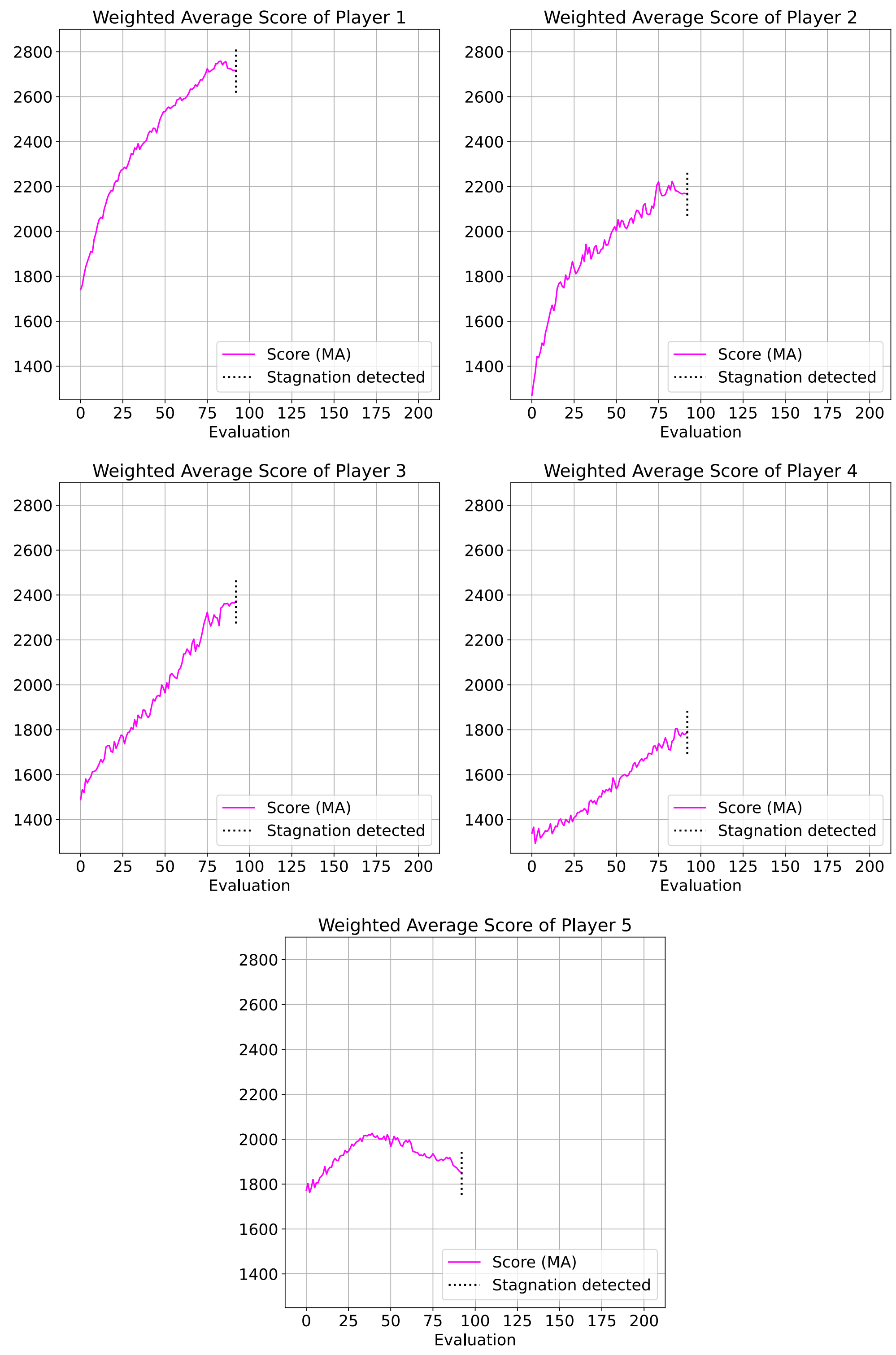


Fig. 16. Weighted average score of each player (2. Experiment).

where p_t is the end effector position of the robot at t. The initial distance can be calculated as in Eq. (6).

$$di = p_0 - p_d \tag{6}$$

where p_0 is the end effector position of the robot at $t = 0$. The ratio between the distance error at t and the initial distance is calculated to avoid differences in reward at $t = 0$.

The training was set up in a special mode in which the stagnation detection is active but does not end the training. This allows to gain detailed evaluation data until stagnation was detected and the regular end by reaching the training cycle limit in just one run. This enables examining the behavior of the stagnation detection itself in a more efficient way. The training parameters in detail were:

- Overall training cycle limit: 5500
- Cycle limit per episode: 20
- Moving average score horizon: 20
- Stagnation limit: 5

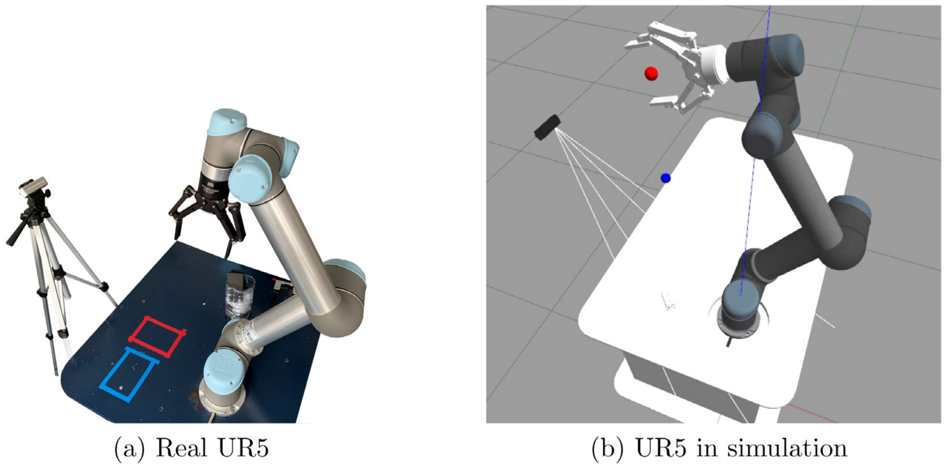


Fig. 17. UR5 Environment. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

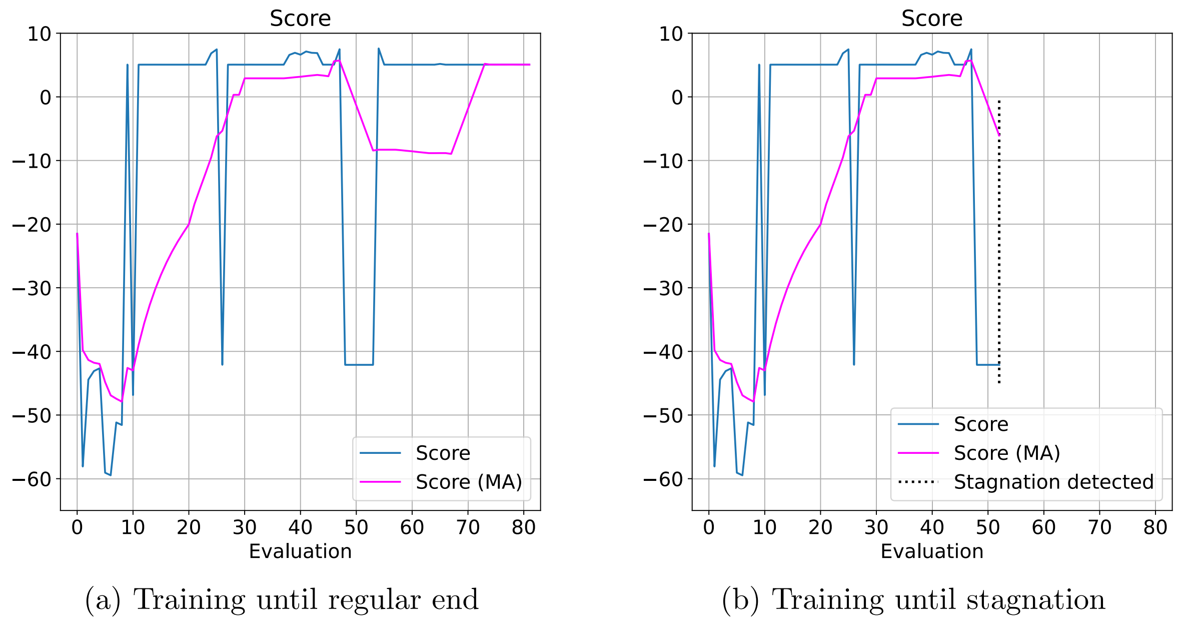


Fig. 18. Results of wrapped PPO training in wrapped UR5 environment using MLPro until (a) regular end and (b) stagnation was detected.

- Evaluation frequency: 10
- Stagnation detection starts after 15 evaluations
- Reward penalty value (pe): $10e^{-2}$

The results (shown in Fig. 18) demonstrate, that the training itself was successful and that stagnation detection would have terminated the training at a senseful point.

Actually, it took a few tries to find suitable parameters for the moving average score horizon, the stagnation limit and the number of ignored evaluations at the beginning of the training. All of those parameters highly influence the stagnation detection and here is potential for future improvements that automatically determine these parameters.

7. Conclusion and future work

In this paper we presented a new software package called MLPro for the standardized and holistic definition and solution of machine learning tasks in Python. The package comes up with carefully developed advanced ML models and processes and supports both the training of learning algorithms in simulations and the control and monitoring of real industrial applications. The native implementation of own ML scenarios is just as possible as the needs-based integration of existing state-of-the-art implementations from third-party providers.

As part of the first version of MLPro, an open and modular architecture are established, which is based on a cross-sectional infrastructure of reusable basic functions. In particular, the fundamentals of machine learning are anchored here: the abstract adaptive ML model, which interacts with its surroundings in a defined scenario and is trained and tuned on it. At a higher level, these basic elements are then specialized and concretized in so-called subtopic packages.

In the first subtopic package MLPro-RL for reinforcement learning, the abstract ML model becomes a landscape of model-free or model-based single-/multi-agents and the surroundings become environments. In the second subtopic package MLPro-GT for cooperative game theory, large parts of MLPro-RL are taken up, transferred to the established terminology and further specialized.

As illustrated in Fig. 19, MLPro outperforms the current frameworks in the generalization and standardization of overarching ML concepts and in the provision of advanced ML models and processes for RL/GT.

The authors are firmly convinced that the first aspect in particular will greatly simplify and thus accelerate the development of new subtopics and significantly extend the life cycle of the framework.

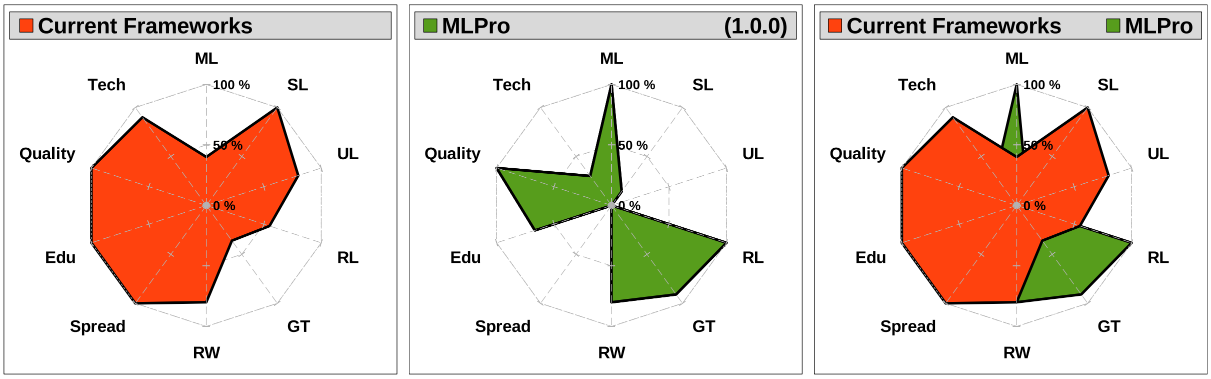


Fig. 19. Coverage of current frameworks and MLPro's contribution.

Table A.1
Data Basis for all Spider Diagrams.

Framework	Inspected Release	Release Date	Project started in	ML	Tech	Quality	Edu	Spread	RW	GT	RL	UL	SL
Hyperopt	0.2.7	2022	2017	20%	50%	71%	67%	1%	0%	0%	0%	0%	0%
MLflow	1.24.0	2022	2020	20%	50%	86%	67%	0%	20%	0%	0%	0%	0%
MLPro	**1.0.0**	**2022**	**2021**	**100%**	**30%**	**100%**	**67%**	**0%**	**80%**	**91%**	**100%**	**0%**	**14%**
Nashpy	0.0.30	2021	2018	0%	10%	71%	33%	0%	0%	18%	0%	0%	0%
OpenAI Gym	0.23.0			0%	30%	71%	67%	6%	0%	0%	18%	0%	0%
OpenML				0%	40%	86%	67%	2%	0%	0%	0%	20%	29%
OpenSpiel	1.1.0	2022	2020	0%	30%	86%	67%	0%	0%	36%	55%	0%	0%
Optuna	2.10.0	2022	2016	20%	50%	86%	83%	1%	0%	0%	0%	0%	0%
PettingZoo	1.16.0	2021	2007	0%	20%	86%	50%	0%	0%	0%	18%	0%	0%
PyTorch	1.10.2	2022	2018	0%	60%	86%	83%	24%	40%	0%	0%	0%	100%
RAY	1.11.0	2021	2017	40%	60%	86%	100%	1%	40%	0%	55%	0%	14%
Scikit-learn	1.0.2	2022	2021	0%	40%	86%	50%	100%	20%	0%	0%	80%	86%
Stable Baselines3	1.4.0	2022	2019	0%	60%	86%	83%	0%	0%	0%	36%	0%	0%
TensorFlow	2.8.0	2021	2013	0%	90%	100%	100%	30%	80%	0%	36%	80%	100%

In two selected industrial applications of automation technology and robotics, we were finally able to demonstrate the practical use of MLPro. In particular, the novel stagnation detection of the training process was illustrated here, which will enable a more detailed benchmarking of state-of-the-art algorithms in the future. In this way, they can be compared not only in terms of their maximum score but also in terms of their learning performance — i.e. the earliest attainment of stagnation in the training.

The continuous expansion and optimization of the existing functionalities is planned for the future. In particular, further pool objects such as environments and algorithms as well as wrappers for further third-party packages are planned. The functionality already implemented for user interaction will be introduced into the subtopic packages so that scientific work can soon be experienced visually and interactively. In addition, further subtopic packages for supervised learning and data stream mining are being planned.

The declared goal is to expand MLPro to a complete and holistic framework in which even advanced hybrid ML applications can be implemented on the basis of standardized ML models and processes at a scientific level across all learning paradigms.

Funding

This Article is funded by the Open Access Publication Fund of South Westphalia University of Applied Sciences.

CRediT authorship contribution statement

Detlef Arend: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Project administration. **Steve Yuwono:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Mochammad Rizky Diprasetya:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Andreas Schwung:** Resources, Supervision, Funding acquisition, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Data basis for framework evaluation

Table A.1 shows the results of our evaluation of all frameworks considered in this paper. All spider diagrams in Sections 2.2, 2.3 and 7 are based on this table.

The following subsections detail the evaluations of all frameworks for each criteria class.

Table A.2
Evaluation results for criteria class ML.

Framework	Inspected Release	ML	1.1 Prototype of an adaptive ML-Model	1.2 Abstract Surroundings Model	1.3 Abstract ML-Scenario Model	1.4 General Training Process	1.5 General Hyperparameter Tuning Process
Hyperopt	0.2.7	20%					1
MLflow	1.24.0	20%	1				
MLPro	**1.0.0**	**100%**	**1**	**1**	**1**	**1**	**1**
Nashpy	0.0.30	0%					
OpenAI Gym	0.23.0	0%					
OpenML		0%					
OpenSpiel	1.1.0	0%					
Optuna	2.10.0	20%					1
PettingZoo	1.16.0	0%					
PyTorch	1.10.2	0%					
RAY	1.11.0	40%				1	1
Scikit-learn	1.0.2	0%					
Stable Baselines3	1.4.0	0%					
TensorFlow	2.8.0	0%					

Table A.3
Evaluation results for criteria class SL.

Framework	Inspected Release	SL	2.1 Specific adaptive Models	2.2 Specific adaptive Models for Deep Learning	2.3 Data Set Management	2.4 State-of-the-Art Algorithms	2.5 Standardized Training	2.6 Standardized Hyperparameter Tuning	2.7 Sample Data Sets
Hyperopt	0.2.7	0%							
MLflow	1.24.0	0%							
MLPro	**1.0.0**	**14%**	**1**						
Nashpy	0.0.30	0%							
OpenAI Gym	0.23.0	0%							
OpenML		29%			1				1
OpenSpiel	1.1.0	0%							
Optuna	2.10.0	0%							
PettingZoo	1.16.0	0%							
PyTorch	1.10.2	100%	1	1	1	1	1	1	1
RAY	1.11.0	14%			1				
Scikit-learn	1.0.2	86%	1		1	1	1	1	1
Stable Baselines3	1.4.0	0%							
TensorFlow	2.8.0	100%	1	1	1	1	1	1	1

A.1. ML — Abstract/overarching ML concepts

See Table A.2.

A.2. SL — Supervised learning

See Table A.3.

A.3. UL — Unsupervised learning

See Table A.4.

A.4. RL — Reinforcement learning

See Table A.5.

Table A.4
Evaluation results for criteria class UL.

Framework	Inspected Release	UL	3.1 Specific adaptive Models	3.2 State-of-the-Art Algorithms	3.3 Standardized Training	3.4 Standardized Hyperparameter Tuning	3.5 Sample Data
Hyperopt	0.2.7	0%					
MLflow	1.24.0	0%					
MLPro	**1.0.0**	**0%**					
Nashpy	0.0.30	0%					
OpenAI Gym	0.23.0	0%					
OpenML		20%					1
OpenSpiel	1.1.0	0%					
Optuna	2.10.0	0%					
PettingZoo	1.16.0	0%					
PyTorch	1.10.2	0%					
RAY	1.11.0	0%					
Scikit-learn	1.0.2	80%	1	1	1		1
Stable Baselines3	1.4.0	0%					
TensorFlow	2.8.0	80%	1	1	1	1	

Table A.5
Evaluation results for criteria class RL.

Framework	Inspected Release	RL	4.1 Single-Agent Model	4.2 Multi-Agent Model (MARL)	4.3 Model-based Agent Model (MBRL)	4.4 Action Planning (MPC)	4.5 State-of-the-Art Algorithms	4.6 Single-Agent Environment Model	4.7 Multi-Agent Environment Model	4.8 Standardized Training	4.9 Standardized Hyperparameter Tuning	4.10 Sample Single-Agent Environments	4.11 Sample Multi-Agent Environments
Hyperopt	0.2.7	0%											
MLflow	1.24.0	0%											
MLPro	**1.0.0**	**100%**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Nashpy	0.0.30	0%											
OpenAI Gym	0.23.0	18%						1				1	
OpenML		0%											
OpenSpiel	1.1.0	55%	1				1	1	1			1	1
Optuna	2.10.0	0%											
PettingZoo	1.16.0	18%							1				1
PyTorch	1.10.2	0%											
RAY	1.11.0	55%	1	1			1	1	1	1			
Scikit-learn	1.0.2	0%											
Stable Baselines3	1.4.0	36%	1				1			1	1		
TensorFlow	2.8.0	36%	1				1	1				1	

A.5. GT — Game theory

See Table A.6.

A.6. RW — Real world application support

See Table A.7.

A.7. Spread in research and science

See Table A.8.

A.8. Edu — Educational features

See Table A.9.

Table A.6
Evaluation results for criteria class GT.

Framework	Inspected Release	GT	5.1 Cooperative Game Theory	5.2 Competitive Game Theory	5.3 Single-Player Model	5.4 Multi-Player Model	5.5 Model-based Player Model	5.6 Action Planning (MPC)	5.7 State-of-the-Art Algorithms	5.8 Multi-Player Game Board Model	5.9 Standardized Training	5.10 Standardized Hyperparameter Tuning	5.11 Sample Game Boards
Hyperopt	0.2.7	0%											
MLflow	1.24.0	0%											
MLPro	**1.0.0**	**91%**	**1**		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Nashpy	0.0.30	18%		1					1				
OpenAI Gym	0.23.0	0%											
OpenML		0%											
OpenSpiel	1.1.0	36%	1	1					1				1
Optuna	2.10.0	0%											
PettingZoo	1.16.0	0%											
PyTorch	1.10.2	0%											
RAY	1.11.0	0%											
Scikit-learn	1.0.2	0%											
Stable Baselines3	1.4.0	0%											
TensorFlow	2.8.0	0%											

Table A.7
Evaluation results for criteria class RW.

Framework	Inspected Release	RW	6.1 Real Operation Support	6.2 Support of physical units	6.3 Precise time management	6.4 Detailed logging	6.5 Data Preprocessing
Hyperopt	0.2.7	0%					
MLflow	1.24.0	20%	1				
MLPro	**1.0.0**	**80%**	**1**	**1**	**1**	**1**	
Nashpy	0.0.30	0%					
OpenAI Gym	0.23.0	0%					
OpenML		0%					
OpenSpiel	1.1.0	0%					
Optuna	2.10.0	0%					
PettingZoo	1.16.0	0%					
PyTorch	1.10.2	40%				1	1
RAY	1.11.0	40%	1				1
Scikit-learn	1.0.2	20%					1
Stable Baselines3	1.4.0	0%					
TensorFlow	2.8.0	80%	1		1	1	1

A.9. Quality

See Table A.10.

A.10. Tech — Technical features

See Table A.11.

Table A.8
Evaluation results for criteria class Spread.

Framework	Inspected Release	Spread	7.1 Citations of core papers (Google Scholar)
Hyperopt	0.2.7	1%	454
MLflow	1.24.0	0%	193
MLPro	**1.0.0**	**0%**	**0**
Nashpy	0.0.30	0%	8
OpenAI Gym	0.23.0	6%	3606
OpenML		2%	867
OpenSpiel	1.1.0	0%	90
Optuna	2.10.0	1%	810
PettingZoo	1.16.0	0%	39
PyTorch	1.10.2	24%	13359
RAY	1.11.0	1%	594
Scikit-learn	1.0.2	100%	56636
Stable Baselines3	1.4.0	0%	194
TensorFlow	2.8.0	30%	16926

Table A.9
Evaluation results for criteria class Edu.

Framework	Inspected Release	Edu	8.1 Comprehensive Documentation	8.2 Ready to run sample codes	8.3 Tutorials	8.4 Training Programs / Certification	8.5 Descriptive Publications	8.6 Interactive User Experience
Hyperopt	0.2.7	67%	1	1			1	1
MLflow	1.24.0	67%	1	1	1		1	
MLPro	**1.0.0**	**67%**	**1**	**1**			**1**	**1**
Nashpy	0.0.30	33%	1	1				
OpenAI Gym	0.23.0	67%	1	1			1	1
OpenML		67%	1	1	1		1	
OpenSpiel	1.1.0	67%	1	1			1	1
Optuna	2.10.0	83%	1	1	1		1	1
PettingZoo	1.16.0	50%	1	1			1	
PyTorch	1.10.2	83%	1	1	1		1	1
RAY	1.11.0	100%	1	1	1	1	1	1
Scikit-learn	1.0.2	50%	1	1			1	
Stable Baselines3	1.4.0	83%	1	1	1		1	1
TensorFlow	2.8.0	100%	1	1	1	1	1	1

Table A.10
Evaluation results for criteria class Quality.

Framework	Inspected Release	Quality	9.1 Professional Source Code Management	9.2 Professional Project Management	9.3 Test Automation	9.4 CI/CD	9.5 API Documentation	9.6 Clean Code Paradigm	9.7 Publications
Hyperopt	0.2.7	71%	1	1	1	1			1
MLflow	1.24.0	86%	1	1	1	1	1		1
MLPro	**1.0.0**	**100%**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Nashpy	0.0.30	71%	1	1	1	1			1
OpenAI Gym	0.23.0	71%	1	1	1	1			1
OpenML		86%	1	1	1	1	1		1
OpenSpiel	1.1.0	86%	1	1	1	1		1	1
Optuna	2.10.0	86%	1	1	1	1	1		1
PettingZoo	1.16.0	86%	1	1	1	1	1		1
PyTorch	1.10.2	86%	1	1	1	1	1		1
RAY	1.11.0	86%	1	1	1	1	1		1
Scikit-learn	1.0.2	86%	1	1	1	1	1		1
Stable Baselines3	1.4.0	86%	1	1	1	1	1		1
TensorFlow	2.8.0	100%	1	1	1	1	1	1	1

Table A.11
Evaluation results for criteria class Tech.

Framework	Inspected Release	Tech	10.1 GPU/CUDA Support	10.2 Advanced Mathematics	10.3 Model Deployment	10.4 Data Management	10.5 Data Plotting	10.6 Python API	10.7 Java API	10.8 C++ API	10.9 APIs for further Programming Languages	10.10 REST API
Hyperopt	0.2.7	50%	1	1		1	1	1				
MLflow	1.24.0	50%			1			1	1		1	1
MLPro	**1.0.0**	**30%**				**1**	**1**	**1**				
Nashpy	0.0.30	10%						1				
OpenAI Gym	0.23.0	30%	1	1				1				
OpenML		40%				1		1	1		1	
OpenSpiel	1.1.0	30%						1		1	1	
Optuna	2.10.0	50%	1	1		1	1	1				
PettingZoo	1.16.0	20%	1					1				
PyTorch	1.10.2	60%	1	1		1	1	1		1		
RAY	1.11.0	60%	1	1	1	1		1	1			
Scikit-learn	1.0.2	40%		1		1	1	1				
Stable Baselines3	1.4.0	60%	1			1	1	1		1	1	
TensorFlow	2.8.0	90%	1	1	1	1	1	1	1	1	1	

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. URL https://www.tensorflow.org/, Software available from tensorflow.org.

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. arXiv:1907.10902.

Bauso, D. (2016). *Game theory with engineering applications*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Bergstra, J., Komer, B., Eliasmith, C., Yamins, D., & Cox, D. D. (2015). Hyperopt: A python library for model selection and hyperparameter optimization. *8*, (1), Article 014008, URL http://stacks.iop.org/1749-4699/8/i=1/a=014008.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., et al. (2016). Openai gym. arXiv:1606.01540.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., et al. (2013). API design for machine learning software: experiences from the scikit-learn project. arXiv:1309.0238.

Busoniu, L., Babuska, R., & De Schutter, B. (2008). A comprehensive survey of multi-agent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *38*(2), 156–172.

Casalicchio, G., Bossek, J., Lang, M., Kirchhoff, D., Kerschke, P., Hofner, B., et al. (2019). Openml: An R package to connect to the machine learning platform openml. *Computational Statistics*, *34*(3), 977–991.

Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., et al. (2020). Developments in mlflow: A system to accelerate the machine learning lifecycle. In *Proceedings of the fourth international workshop on data management for end-to-end machine learning* (pp. 1–4).

Chirodea, M. C., Novac, O. C., Novac, C. M., Bizon, N., Oproescu, M., & Gordan, C. E. (2021). Comparison of tensorflow and pytorch in convolutional neural network - based applications. In *2021 13th international conference on electronics, computers and artificial intelligence*. IEEE, http://dx.doi.org/10.1109/ecai52376.2021.9515098.

Coleman, D. T. (2014). *Reducing the barrier to entry of complex robotic software: A moveit! case study*. Universita degli studi di Bergamo, http://dx.doi.org/10.6092/JOSER_2014_05_01_P3.

Elkind, E., & Rothe, J. (2016). Cooperative game theory. In *Economics and computation* (pp. 135–193). Springer.

Feurer, M., van Rijn, J. N., Kadra, A., Gijsbers, P., Mallik, N., Ravi, S., et al. (2019). OpenML-python: An extensible python API for OpenML. arXiv:1911.02490.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*(8), 2554–2558.

Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., et al. (2019). Model-based reinforcement learning for atari. arXiv:1903.00374.

Knight, V., & Campbell, J. (2018). Nashpy: A python library for the computation of Nash equilibria. *Journal of Open Source Software*, *3*(30), 904.

Lanctot, M., Lockhart, E., Lespiau, J.-B., Zambaldi, V., Upadhyay, S., Pérolat, J., et al. OpenSpiel: A Framework for reinforcement learning in games, arXiv:1908.09453, URL http://arxiv.org/abs/1908.09453.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., et al. (2017). RLlib: Abstractions for distributed reinforcement learning. arXiv:1712.09381.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training. arXiv:1807.05118.

Marden, J. R. (2012). State based potential games. *Automatica*, *48*(12), 3075–3088.

Martin, R. C. (2011). *The clean coder: A code of conduct for professional programmers*. Pearson Education.

Matsugu, M., Mori, K., Mitari, Y., & Kaneda, Y. (2003). Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, *16*(5–6), 555–559.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., et al. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937). PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Murtagh, F. (1991). Multilayer perceptrons for classification and regression. *Neurocomputing*, *2*(5–6), 183–197.

Owen, G. (2013). *Game theory*. Emerald Group Publishing.

Pal, C.-V., & Leon, F. (2020). Brief survey of model-based reinforcement learning techniques. In *2020 24th international conference on system theory, control and computing* (pp. 92–97).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems, Vol. 32* (pp. 8024–8035). Curran Associates, Inc..

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*(Oct), 2825–2830.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. (2019). Stable Baselines3. *GitHub Repository*, https://github.com/DLR-RM/stable-baselines3.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement LearningImplementations. *22*.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arxiv.org/pdf/1707.06347v2, http://arxiv.org/abs/1707.06347[arXiv:1707.06347].

Schwung, D., Kempe, T., Schwung, A., & Ding, S. X. (2017). Self-optimization of energy consumption in complex bulk good processes using reinforcement learning. In *2017 IEEE 15th international conference on industrial informatics* (pp. 231–236).

Schwung, D., Schwung, A., & Ding, S. X. (2020). Distributed self-optimization of modular production units: A state-based potential game approach. *IEEE Transactions on Cybernetics*, 1–12.

Schwung, D., Yuwono, S., Schwung, A., & Ding, S. X. (2021). Decentralized learning of energy optimal production policies using PLC-informed reinforcement learning. *Computers & Chemical Engineering*, *152*, Article 107382. http://dx.doi.org/10.1016/j.compchemeng.2021.107382.

Sevgi, L., & Orbay, B. Z. (2014). Game theory and engineering applications [testing ourselves]. *IEEE Antennas and Propagation Magazine*, *56*(3), 255–267.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815.

Stanford Artificial Intelligence Laboratory, et al. (2018). Robotic operating system. URL www.ros.org.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second). The MIT Press.

Terry, J. K., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., et al. (2020). Pettingzoo: Gym for multi-agent reinforcement learning. arXiv:2009.14471.

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2014). OpenML: networked science in machine learning. http://dx.doi.org/10.1145/2641190.2641198, arXiv:1407.7722.

Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J. M., Boots, B., et al. (2017). Information theoretic MPC for model-based reinforcement learning. In *2017 IEEE international conference on robotics and automation*. IEEE.

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., et al. (2018). Accelerating the machine learning lifecycle with mlflow. *41*, (4), (pp. 39–45).