Gym4ReaL: A Suite for Benchmarking Real-World Reinforcement Learning

Davide Salaorni[1] **Vincenzo De Paola**[1] **Samuele Delpero**[1] **Giovanni Dispoto**[1]
Paolo Bonetti[1] **Alessio Russo**[1] **Giuseppe Calcagno**[1] **Francesco Trovò**[1]
Matteo Papini[1] **Alberto Maria Metelli**[1] **Marco Mussi**[1] **Marcello Restelli**[1]
[1] Politecnico di Milano
mail to: davide.salaorni@polimi.it

Abstract

In recent years, *Reinforcement Learning* (RL) has made remarkable progress, achieving superhuman performance in a wide range of simulated environments. As research moves toward deploying RL in real-world applications, the field faces a new set of challenges inherent to real-world settings, such as large state-action spaces, non-stationarity, and partial observability. Despite their importance, these challenges are often underexplored in current benchmarks, which tend to focus on idealized, fully observable, and stationary environments, often neglecting to incorporate real-world complexities explicitly. In this paper, we introduce `Gym4ReaL`, a comprehensive suite of realistic environments designed to support the development and evaluation of RL algorithms that can operate in real-world scenarios. The suite includes a diverse set of tasks that expose algorithms to a variety of practical challenges. Our experimental results show that, in these settings, standard RL algorithms confirm their competitiveness against rule-based benchmarks, motivating the development of new methods to fully exploit the potential of RL to tackle the complexities of real-world tasks.

1 Introduction

In the past few years, *Reinforcement Learning* (RL) [27] has demonstrated above-human performance across different challenges, ranging from playing Atari games [13] to beating world champions of Chess and Go [24, 25], achieving impressive results also in the field of robotic control [10]. However, despite these promising advances, RL still struggles to gain traction in many real-world applications, where systems are often subject to uncertainties and unpredictable factors that complicate accurate physical modeling. An additional limitation lies in the fact that RL algorithms are typically validated on idealized environments, such as those provided by Gymnasium [29] and MuJoCo [28]. Despite their great contribution to RL research, such libraries provide artificial playgrounds able to generate infinite samples, adapt to any desired configuration, and grant harmless exploration. However, learning and overfitting these environments does not necessarily reflect skillfulness in real-world tasks, where data is limited, dynamics change, and exploration does not come for free.

As a step towards bridging the gap between simulated and real-world settings and promoting the deployment of RL in practical applications, we present `Gym4ReaL`[1], a benchmarking suite designed to realistically model several real-world environments specifically tailored for RL algorithms. The selected tasks span multiple application domains. In particular, the suite includes:

- `DamEnv`, which models a dam control system responsible for releasing the appropriate amount of water to meet residential demand;

[1] Codebase available here: `https://github.com/Daveonwave/gym4ReaL`

Preprint.

Table 1: *Characteristics* and *RL paradigms* covered by each environment provided by `Gym4ReaL`.

	Characteristics						**RL Paradigms**					
	Cont. States	Cont. Actions	Part. Observable	Part. Controllable	Non-Stationary	Visual Input	Freq. Adaptation	Hierarchical RL	Risk-Averse	Imitation Learning	Provably Efficient	Multi-Objective RL
`DamEnv`	✓	✓		✓						✓		✓
`ElevatorEnv`				✓							✓	
`MicrogridEnv`	✓	✓		✓			✓					✓
`RoboFeederEnv`	✓	✓				✓		✓				
`TradingEnv`	✓		✓	✓	✓		✓		✓			
`WDSEnv`	✓			✓						✓		✓

- `ElevatorEnv`, which addresses a simplified version of the elevator dispatching problem under dynamic request patterns;
- `MicrogridEnv`, which focuses on optimal energy management within a local microgrid, balancing supply, demand, and storage;
- `RoboFeederEnv`, which simulates a robotic work cell tasked with isolating and picking small objects, including both picking and planning challenges;
- `TradingEnv`, which aims to develop optimized trading strategies for the foreign exchange (Forex) market;
- `WDSEnv`, which models a municipal water distribution system, where the objective is to ensure a consistent supply to meet fluctuating residential demand.

While the tasks of these environments can be modeled in various ways, `Gym4ReaL` provides their standardized implementation compatible with Gymnasium [29] interfaces. Our selection of real-world environments allows for training agents that specifically address such practical problems. However, `Gym4ReaL` is also designed to serve as a methodologically agnostic suite, enabling RL researchers to evaluate and benchmark algorithms regardless of any specific domain knowledge.

The primary goal of `Gym4ReaL` is not merely to supply environments for solving specific domain tasks, but to offer a curated suite of environments that encapsulates the fundamental challenges inherent to these real-world environments. A comprehensive summary of the suite's features is presented in Table 1. In particular, we distinguish between two key aspects: *characteristics*, which refer to modeling properties specific to each environment, and *RL paradigms*, which denote the classes of RL techniques that can be effectively tested and benchmarked within these environments beyond the classical RL approaches. Furthermore, `Gym4ReaL` offers a high degree of configurability. Users can customize input parameters and environmental dynamics to better reflect domain-specific requirements, thus extending the suite's usability to researchers from the respective application domains. Through this combination of realism, diversity, and flexibility, `Gym4ReaL` supports a wide spectrum of research efforts, from benchmarking general-purpose RL algorithms under realistic conditions to developing domain-specific controllers.

Related works. Several prior works have introduced benchmarking suites aimed at evaluating RL algorithms in realistic or application-specific scenarios. For example, SustainGym [31] provides a suite of energy system simulations focused on sustainability and grid optimization, designed to assess RL agents under real-world constraints such as energy storage dynamics and carbon emissions. Similarly, SustainDC [16] targets sustainable control in data centers, offering environments that capture the complexity of workload scheduling and energy-efficient operations. In the domain of autonomous driving, MetaDrive [11] offers a customizable simulator for training RL agents in diverse driving scenarios, testing generalization and robustness capabilities by supporting procedurally generated environments. Regarding financial problems, FinRL [12] proposes a framework that reduces the complexity of training RL agents to perform different financial tasks (e.g., trading and portfolio allocation) on different markets. In contrast to the aforementioned libraries, which focus

on domain-specific tasks, `Gym4ReaL` is designed as a comprehensive benchmarking suite spanning multiple domains and aiming to serve as a reference point for RL research in real-world scenarios.

Beyond domain-specific suites, several other platforms aim to expose RL algorithms to broader real-world challenges. Real-World RL (RWRL) Suite [5] focuses on incorporating key aspects of real-world deployment such as partial observability, delayed rewards, and non-stationarity. Robust RL Suite (RRLS) [34] introduces environmental uncertainty to evaluate the robustness of RL methods in continuous control tasks. POPGym [14] presents a set of environments centered on partially observable Markov decision processes (POMDPs), making it useful for studying memory and inference in RL. Finally, Safe-Control-Gym [33] provides environments tailored for the safe RL paradigm in robotics and control, incorporating constraints and safety-aware objectives. Unlike `Gym4ReaL`, such suites do not aim to address real-world problems directly. Instead, they leverage virtual environments from Gymnasium and Mujoco to simulate some of the challenges of real-world tasks.

2 Environments and Benchmarking

This section introduces `Gym4ReaL` environments, describing the state space, the action space, and the reward function. Test results associated with each task are included, comparing rule-based expert policies with RL-based agents. Further details on environments and results are in the Appendix.

2.1 `DamEnv`

`DamEnv` is designed to model the operation of a dam connected to a water reservoir. By providing the amount of water to be released as an action, the environment simulates changes in the water level, considering inflows, outflows, and other physical dynamics. The agent controlling the dam aims to plan the water release in order to satisfy the daily water demand while preventing the reservoir from exceeding its maximum capacity and causing overflows. Formally, the objective is:

$$\max \sum_{t=1}^{T} [r_d(a_t) + r_{\text{of}}(a_t) + r_{\text{st}}(a_t)], \tag{1}$$

where r_d favors actions that meet daily demand, r_{of} actions that prevent water overflows, and r_{st} those that avoid starvation effects along the time horizon T. The daily control frequency adopted depends on the data granularity. Moreover, the available historical data derived from human-expert decisions allows for the development of imitation learning studies.

Observation Space. The observation space is composed as follows:

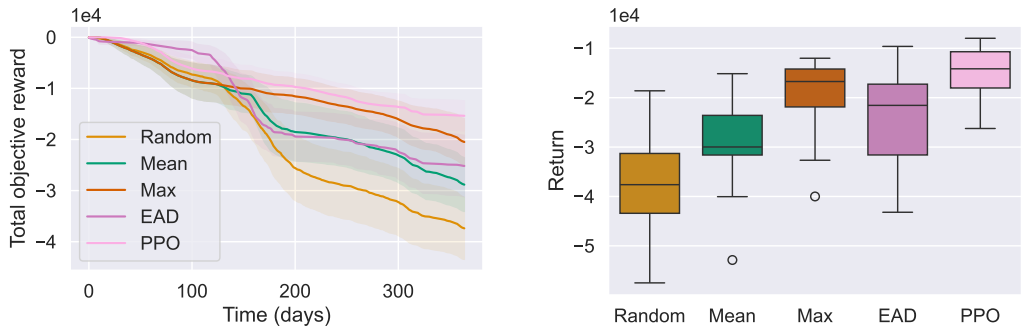
$$s_t = \left(l_t, \bar{d}_t, \cos(\varphi_t^y), \sin(\varphi_t^y)\right), \tag{2}$$

where l_t is the water level at time t, $\bar{d}_t$ is the moving average of past water demands, and $\varphi_t^y \in [0, 2\pi]$ represents the angular position of the current time over the entire year, given by $\varphi^y = \frac{2\pi\tau_y}{T_y}$, where $\tau_y \in [0, T_y]$ is the current time in seconds and T_y is the total number of seconds in a year.

Action Space. The action is a continuous variable $a_t \in \mathbb{R}^+$, representing the amount of water to release per unit of time.

Reward Function. The reward at time t is $r_t = [r_d(a_t) + r_{\text{of}}(a_t) + r_{\text{st}}(a_t)] + \lambda_1 r_{\text{clip}}(a_t) + \lambda_2 r_w(a_t)$, where $r_d(a_t)$, $r_{\text{of}}(a_t)$ and $r_{\text{st}}(a_t)$ are the quantities in Equation 1, while $r_{\text{clip}}(a_t)$ and $r_w(a_t)$ are two terms designed to discourage actions beyond the physical constraints of the environment and to discourage water releases that are higher than the daily demand, respectively. The two positive hyperparameters λ_1 and λ_2 regulate the importance of these two additional penalty terms. The presence of multiple contrastive components enables the development of MORL paradigms.

Benchmarking. We employed an off-the-shelf implementation of the Proximal-Policy Optimization (PPO) [22] algorithm as a benchmark state-of-the-art RL approach for the `DamEnv` task. We evaluated the trained agent against four rule-based baselines: the *Random* policy, which selects actions uniformly at random; the *Mean* policy, which selects the mean value of the action space; the *Max* policy, which selects the maximum value of the action space; and the *EAD* policy, which sets actions based on an exponential moving average of previous demands. The experiments conducted on 13 test episodes highlight the capability of the PPO agent to perform better than rule-based strategies. In particular,



(a) Mean cumulative rewards.

(b) Boxplot of returns.

Figure 1: Test performances with confidence intervals on `DamEnv`. Thirteen different episodes have been considered with a time horizon of one year.

we can observe a better daily control of the PPO agent throughout one year, as shown in Figure 1a, and a larger average return with small variability, as highlighted in Figure 1b. Detailed results show that PPO avoids dam overflows much more effectively than the baselines, as detailed in the Appendix.

2.2 `ElevatorEnv`

`ElevatorEnv` is a simplified adaptation of the well-known elevator scheduling problem introduced by Crites and Barto [3]. Similarly to a subsequent work [32], we design a discrete environment that simulates *peak-down traffic*, typical of scenarios such as office buildings at the end of a workday. In this environment, a single elevator serves a multi-floor building with F floors and is tasked with transporting employees to the ground floor ($f = 0$). The episode unfolds over T discrete time steps. At each floor $f \in \{1, \dots, F\}$, new passengers arrive according to a *Poisson process* with rate λ_f.

Arriving passengers join a queue on their respective floor, provided the queue length is below a predefined threshold $W_{f,\text{max}}$. Otherwise, they opt to take the stairs. The *goal* of the elevator controller is to minimize the cumulative *waiting time* of all transported passengers throughout the episode. This can be formalized as minimizing the cost:

$$\min \sum_{t=1}^{T} \Big(\sum_{f=1}^{F} w_{f,t} + c_t \Big), \tag{3}$$

where $w_{f,t}$ denotes the total waiting time of individuals at floor f at time t. This setting defines a challenging *load management problem*, involving a trade-off between serving higher floors with longer queues and minimizing elevator travel time. Furthermore, the discrete and restrained formulation of `ElevatorEnv` facilitates the development of provably efficient RL methods, without losing the connection with the underlying real-world task.

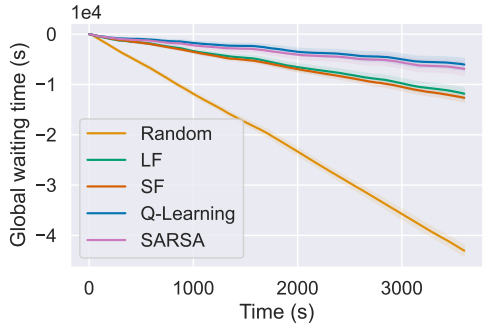
Observation Space. The observation space is structured as follows:

$$s_t = (h_t, c_t, \mathbf{w}_t, \mathbf{k}_t), \tag{4}$$

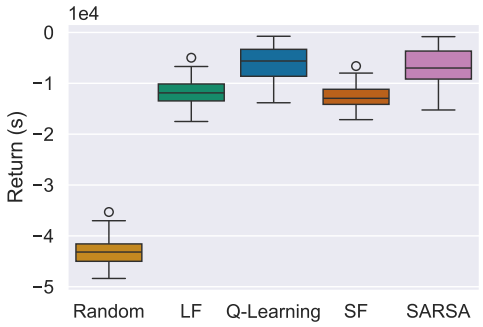
where $h_t \in \{0, \dots, H\}$ denotes the vertical position of the elevator within the building at time t, being H the maximum reachable height, $c_t \in \{0, \dots, C_{\text{max}}\}$ indicates the current load of the elevator, in number of passengers, up to the maximum capacity C_{max}, and $\mathbf{w}_t \in \mathbb{N}^F$ and $\mathbf{k}_t \in \mathbb{N}^F$ represent the actual number of people waiting in the queue and the new arrivals at each floor.

Action Space. The action space is defined by the discrete action variable $a_t \in \{u, d, o\}$ which indicates whether the elevator has to move upwards (u), move downwards (d), or stay stationary and open (o) the doors. Actions are mutually exclusive and applied at each time step t.

Reward Function. The instantaneous reward is $r_t = -(\sum_f w_{f,t} + c_t) + \mathbb{1}_{\{c_t = 0\}} \beta\, c_{t-1}$, i.e., at each step t we penalize the presence of individuals, either waiting in queues ($w_{f,t}$) or inside the elevator (c_t), as in Equation (4). In addition, we grant a positive reward when passengers are successfully delivered to the ground floor, i.e., when the elevator becomes empty. The positive hyperparameter $\beta > 0$ controls the reward magnitude for offloading c_{t-1} passengers.



(a) Mean cumulative rewards.



(b) Boxplot of returns.

Figure 2: Performance of baselines in terms of mean cumulative reward (a) and average return (b) on `ElevatorEnv`. Results collected over 30 different episodes.

Benchmarking. For the `ElevatorEnv` task, we adopt two well-known tabular RL algorithms: Q-Learning [30] and SARSA [27]. Such methods are evaluated against different rule-based strategies, i.e., the *Random* policy, and the *Longest-First* (LF) and the *Shortest-First* (SF) policies, which prioritize the floor with a higher or lower number of waiting people, respectively. As shown in Figure 2a, both RL algorithms consistently outperform the other rule-based solutions, considerably reducing the global waiting time. In particular, as reported in Figure 2b, Q-Learning shows higher performance than SARSA, which, due to its inherent nature, tends to play more conservative actions.

2.3 `MicrogridEnv`

`MicrogridEnv` simulates the operation of a microgrid within the context of electrical power systems. Microgrids are decentralized components of the main power grid that can function either in synchronization or in islanded mode. In this scenario, the control point is placed on the battery component, which must find the best strategy to manage the accumulated energy over time optimally. Formally, the controller wants to maximize its total profit over a time horizon of T. Hence, the objective is:

$$\max \sum_{t=1}^{T} [r_{\text{trad}}(a_t) + r_{\text{deg}}(a_t)], \tag{5}$$

where $r_{\text{trad}}(a_t) \in \mathbb{R}$ is the reward/cost gained from the exchanges of energy with the market, and $r_{\text{deg}}(a_t) < 0$ is the cost due to battery degradation. The benchmark leverages real-world datasets, as detailed in the Appendix, and the battery behavior is modeled using a digital twin of a BESS [20]. Each episode is formulated as an infinite-horizon problem and terminates either when the dataset is exhausted or the battery reaches its end-of-life condition. Moreover, the presence of energy market trends allows the usage of `MicrogridEnv` for frequency adaptation analysis.

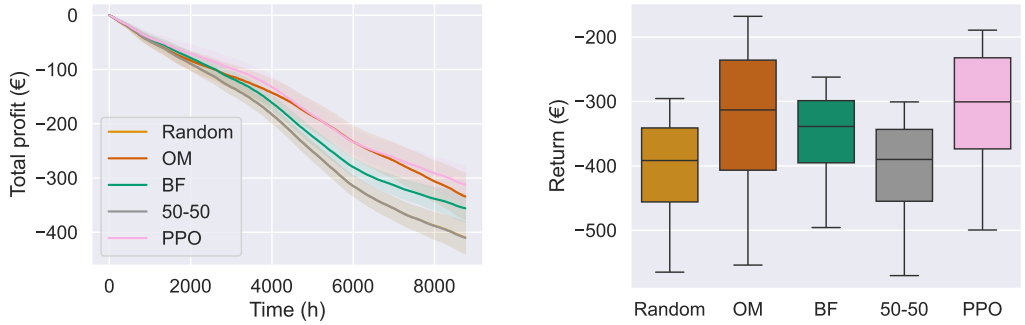
Observation Space. The observation space comprises variables regarding the internal state of the system and uncontrollable signals received from the environment. Formally:

$$s_t = \left(\sigma_t, K_t, \widehat{P}_{D,t}, \widehat{P}_{G,t}, p_t^{\text{buy}}, p_t^{\text{sell}}, \cos(\varphi_t^d), \sin(\varphi_t^d), \cos(\varphi_t^y), \sin(\varphi_t^y)\right), \tag{6}$$

where σ_t is the storage state of charge, K_t is the battery temperature, $\widehat{P}_{D,t}$ is the estimate of energy demand $P_{D,t}$, $\widehat{P}_{G,t}$ is the estimate of energy generation $P_{G,t}$, p_t^{buy} and p_t^{sell} are the buying and selling energy market prices, respectively, $\varphi_t^d \in [0, 2\pi]$ is the angular position of the clock in a day, and $\varphi_t^y \in [0, 2\pi]$ is the angular position of the time over the entire year.

Action Space. The action space is determined by the continuous action variable $a_t \in [0, 1]$, representing the proportion of energy to *dispatch* (*take*) to (from) the BESS. The action operates with the net power computed as $P_{N,t} = P_{G,t} - P_{D,t}$. If $P_{N,t} > 0$, it regulates the proportion of energy used to charge the battery or sold to the main grid. Conversely, if $P_{N,t} < 0$, the action balances the proportion of energy taken from the energy storage or bought from the market.

Reward Function. The instantaneous reward is $r_t = [r_{\text{trad}}(a_t) + r_{\text{deg}}(a_t)] + \lambda r_{\text{clip}}(a_t)$, where $r_{\text{clip}}(a_t)$ is a penalty that discourages actions that do not respect physical constraints, weighted by the



(a) Mean cumulative rewards.

(b) Boxplot of returns.

Figure 3: Performance of baselines in terms of mean cumulative reward (a) and average return (b) on `MicrogridEnv`. Results have been collected over 28 different episodes.

hyperparameter λ. The first two elements, instead, are the same components of the objective function in Equation (5), whose contrastive optimization enables multi-objective RL approaches.

Benchmarking. For the `MicrogridEnv`, we compare an RL agent trained with PPO [22] against several rule-based policies: the *Random* policy; the *Only-market* (OM) policy, which forces the interaction with the grid without using the battery; the *Battery-first* (BF) policy, which fosters the battery usage; and the *50-50* policy, which adopts a behavior in the middle between OM and BF. Figure 3a shows that, during testing, PPO achieves higher profit than rule-based strategies. However, as reported in Figure 3b, PPO has a large variance, suggesting the need for novel RL algorithms to achieve more consistent behavior.

2.4 `RoboFeederEnv`

`RoboFeederEnv` is a collection of environments designed to pick small objects from a workspace area with a 6-degree-of-freedom (6-DOF) robotic arm. This task involves two primary challenges: determining the *picking order* of the objects and identifying the precise *grasping point* on each object for successful pickup and placement. To closely mimic the behavior of the commercial robotic system, a simulation emphasizing contact interactions is conducted using MuJoCo [28]. This environment supports goal-oriented training, enabling the robot to learn how to identify the appropriate grasping points and, more broadly, to determine the most efficient order of picking. Unlike most robotic simulators, `RoboFeederEnv` is uniquely tailored to operate at the trajectory planning level rather than through low-level joint control, which is more realistic in industrial applications, given the impossibility of accessing and modifying proprietary kinematic controllers.

Due to the hierarchical nature of the problem, we split the setting into two underlying environments: `RoboFeeder-picking` and `RoboFeeder-planning`.

2.4.1 `RoboFeeder-picking`

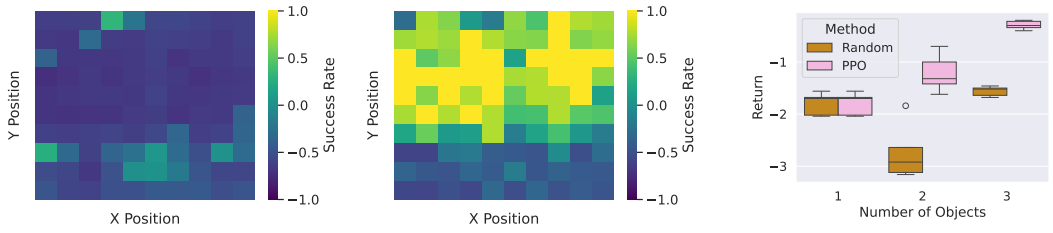
`Gym4ReaL` includes two types of picking environments of increasing difficulty:

- `picking-v0`: a simpler environment where the top-down image is pre-processed by cropping around detected objects, reducing the complexity of the visual input, thus of the observation space;
- `picking-v1`: a more challenging environment where the observation is the full camera image.

Observation Space. The observation is defined by the visual input $s_t = \mathbf{X}_t \in \mathbb{R}^{H \times W \times C}$, where each image $\mathbf{X}_t$ is represented by a tensor of height H, width W, and channel C, and is captured by a camera positioned on top of the working area. Within the `picking-v0` environment, the image tensor is restricted to $\widehat{\mathbf{X}}_t \in \mathbb{R}^{\widehat{H} \times \widehat{W} \times C}$, with $\widehat{H}$ and $\widehat{W}$ cropped image dimensions.

Action Space. The action space is determined by the continuous action $a_t = (x_t, y_t)$, where (x_t, y_t) are relative coordinates within the segmented image, corresponding to the target grasping point.

Reward Function. The reward function is designed to foster successful object picking while penalizing unfeasible or suboptimal actions. Formally, the instantaneous reward is $r_t = 1$ if the object



(a) `picking-v0`: baseline. (b) `picking-v0`: PPO. (c) `RoboFeeder-planner`.

Figure 4: Heatmap of the success rate of picking tasks across the entire workspace with baseline (a) and PPO (b) (the higher, the better). Comparison between *Random* policy and PPO within the planning problem (c) (average return over 50 episodes and 5 different random seeds).

is correctly picked up, $r_t = -1$ if the action is unfeasible, or $r_t = -1 + r_{d,t} + r_{\theta,t}$ otherwise, where $r_{d,t}$ is a distance-based shaping term that rewards proximity of the end-effector to the object, and $r_{\theta,t}$ is a rotation-based shaping term that incentivizes alignment with the desired grasping orientation.

Benchmarking. We evaluate the performance of a trained PPO [22] agent against a fixed action rule-based strategy on the `picking-v0` environment. The task involves objects uniformly distributed within the workspace, requiring non-trivial generalization capabilities. Figures 4a and 4b report how the baseline exhibits consistently poor performance, while the PPO agent achieves higher and more evenly distributed success rates, highlighting its capability to learn an effective picking strategy.

2.4.2 `RoboFeeder-planning`

The `RoboFeeder-planning` is an environment aiming to decide the order to follow for picking the objects in the work area. It is a high-level task w.r.t. `RoboFeeder-picking`, not involving the direct control of the robot, but only concerning the optimal picking schedule.

Observation Space. The observation space is defined by the vector of visual input $s_t = [\mathbf{X}_{1,t}, \ldots, \mathbf{X}_{N,t}]$, with $\mathbf{X}_{i,t} \in \mathbb{R}^{H \times W \times C}$, where N is the maximum number of images that can be processed and $\mathbf{X}_{i,t}$ is an image defined as in the `picking-v0` task. Each of the N image patches corresponds to a cropped and scaled region of a detected object.

Action Space. The action space is determined by the discrete action $a_t \in \{0, 1, \ldots, N\}$, selecting the image from 1 to N containing the object to pick. Action 0, instead, is a special *idle* action that can be chosen when no graspable objects are available. This formulation enables continuous deployment since the robot can remain idle while waiting for the arrival of new objects.

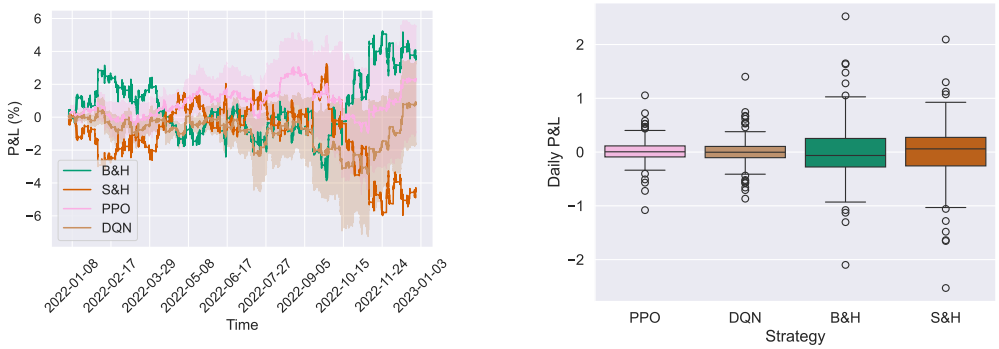
Reward Function. The immediate reward is $r_t = 1$, if the selected object is correctly picked, $r_t = -1$ if it is not picked, and $r_t = -\sum_{i=1}^{M} \mathbb{1}_{\{\text{obj}_i \text{ not picked but graspable}\}}$ if the agent plays the *idle* action $a_t = 0$ while graspable objects are present, with M being the currently available objects.

Benchmarking. In Figure 4c, we compare the efficiency of a trained PPO [22] agent against a *Random* strategy. Results highlight the agent's capability to determine an optimal picking schedule by distinguishing objects placed in a favorable position to be picked up. Moreover, as the number of objects increases, the gap between the average return of PPO and the baseline increases too.

2.5 `TradingEnv`

`TradingEnv` provides a simulated market environment, trained with historical foreign exchange (Forex) data relative to the EUR/USD currency pair, where the objective is to learn a profitable intraday strategy. The problem is framed as episodic: each episode starts at 8:00 EST and ends at 18:00 EST when the position must be closed. At each step, based on its expectations, the agent can open a *long* position (i.e., buy a fixed amount of the asset), remain *flat* (i.e., take no action), or open a *short* position (i.e., short sell a fixed amount of the asset). Typical baselines include passive strategies, such as *Buy&Hold* (B&H) and *Sell&Hold* (S&H), which consist of maintaining fixed positions.

Trading tasks are typically subjected to several challenges. For example, the state has to be carefully designed to deal with the low signal-to-noise ratio, and it is typically large-dimensional, including past prices and temporal information. Moreover, the environment is partially observable, and financial



(a) P&L curve on test set.

(b) Boxplot of P&L on test set.

Figure 5: Performances of PPO and DQN against baselines B&H and S&H on Test (a) Daily Performance on Test (b) on `TradingEnv`. Mean and standard deviation are computed over 6 seeds.

markets are non-stationary. Another relevant aspect is the calibration of the trading frequency, considering the amount of noise and transaction costs. In addition, risk-aversion approaches can be of interest, considering not only the profit-and-loss (P&L) but also the variance among episodes.

Observation Space. The observation space is composed of two components: *market state* and *agent state*. The *market state* includes calendar features and recent price variations, namely the last 60 delta mid-prices, where a delta mid-price is defined as $d_{k,t} = \frac{p_{t-k} - p_{t-k-1}}{p_{t-k-1}}$, with $k \in \{0, \dots, 59\}$. The *agent state* component, on the contrary, includes the current position z_t, that is, the action that was previously played. Formally, the state in this setting is:

$$s_t = \left(\mathbf{d}_t, \cos(\varphi_t^{day}), \sin(\varphi_t^{day}), z_t\right), \tag{7}$$

where $\mathbf{d}_t = [d_{0,t}, \dots, d_{59,t}]$ is the vector of the last 60 delta mid prices at time t, $\varphi_t^{day} \in [0, 2\pi]$ is the angular position of the current time over the trading period, and $z_t = a_{t-1}$ is the agent position.

Action Space. The action space is determined by a discrete variable $a_t \in \{s, f, l\}$, where s (*short*) indicates that the agent is betting against EUR, supposing a decline in the value relative to USD; f (*flat*) indicates no market exposition; and l (*long*) means that the agent expects that the relative EUR value will increase. Each action refers to a fixed amount of capital C to trade.

Reward Function. The immediate reward at time t is the signal $r_t = a_{t-1}(p_t - p_{t-1}) - \lambda|a_t - z_t|$, where the first term is related to the P&L obtained from a price change, and the second component regards the commissions paid when the agent changes its position, being λ, a constant transaction fee.

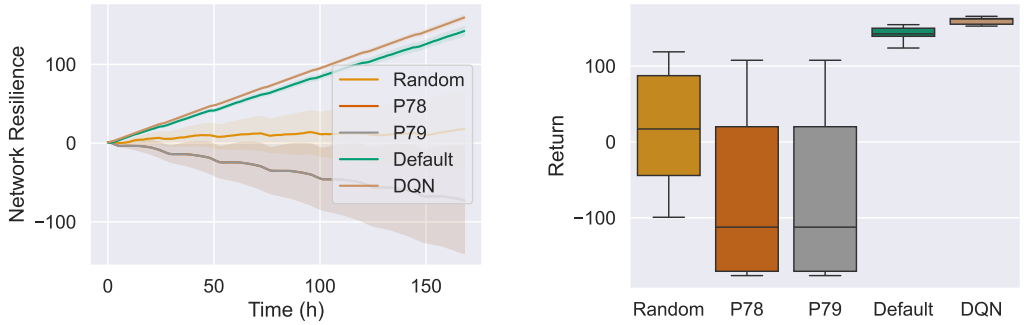
Benchmarking. We trained agents using off-the-shelf implementations of PPO [22] and DQN [13] on `TradingEnv`. Their performance against common passive baselines, B&H and S&H, are evaluated on a test year (Figure 5a). As expected, neither PPO nor DQN is able to consistently outperform the baselines, due to the complexity of the problem. However, RL remains a valid candidate to tackle trading tasks, as it significantly reduces the daily variability of the P&L (Figure 5b).

2.6 `WaterDistributionSystemEnv`

`WaterDistributionSystemEnv` simulates the evolution of a hydraulic network in charge of dispatching water across a residential town. A network is composed of different entities, such as storage tanks, pumps, pipes, junctions, and reservoirs, and the main objective of the system is the safety of the network. To achieve such a goal, we have to ensure optimal management of hydraulic pumps, which are in charge of deciding how much water should be collected from reservoirs and dispatched to the network. The pumps' controller must guarantee network resilience by maximizing the demand satisfaction ratio (DSR) while minimizing the risk of overflow. Formally, the objective is

$$\max \sum_{t=1}^{T} [r_{\text{DSR}}(a_t) + r_{\text{of}}(a_t)], \tag{8}$$

where $r_{\text{DSR}}(a_t) \in [0, 1]$ is the ratio between the supplied demand on the expected demand at time t, and $r_{\text{of}}(a_t) \in [0, 1]$ is a normalized penalty associated with the tanks' overflow risk.



(a) Mean cumulative reward.

(b) Boxplot of returns.

Figure 6: Performance of baselines in terms of mean cumulative resilience (a) and average return (b) on `WDSEnv`. Results have been collected over 20 different episodes.

The environment leverages the hydraulic analysis framework Epanet [19], which provides the mathematical solver for water network evolution, and realistic datasets of demand profiles. Therefore, `WDSEnv` may also be suitable to test imitation learning methods, having at disposal an expert policy from the *.inp* configuration file of networks read by Epanet.

Observation Space. The observation space includes the internal state of the network and an estimation of the global demand profile that the system is asked to deal with. Formally:

$$s_t = \left(\mathbf{h}_t, \mathbf{p}_t, \widehat{d}_t, \cos(\varphi_t^d), \sin(\varphi_t^d)\right), \tag{9}$$

where $\mathbf{h}_t \in \mathbb{R}^L$ is the vector of L tank levels at time t, $\mathbf{p}_t \in \mathbb{R}^J$ is the vector of J junction pressures at time t, $\widehat{d}_t$ is the estimated total demand at time t, and $\varphi_t^d \in [0, 2\pi]$ is the angular position of the clock in a day. Finally, although all tanks must be monitored, we can reduce the dimensionality of the observation space by considering only junctions placed in strategic positions.

Action Space. The discrete action variable $a_t \in \mathbb{N}$ can assume values in $\{0, \dots, 2^P - 1\}$, with P number of pumps within the system. The action determines the combination of open/closed pumps.

Reward Function. The instantaneous reward given by the environment is $r_t = r_{\text{DSR},t}(a_t) + r_{\text{of},t}(a_t)$, where the terms are those described in the objective function in Equation (8).

Benchmarking. The `WDSEnv` is benchmarked adopting DQN [13], which is compared with different rule-based baselines: the *Random* policy, *P78* and *P79* policies, which act by keeping active only the relative pump (namely P78 or P79, respectively), and the *Default* policy, which executes the default control rules contained within the *.inp* configuration file of the network, changing the control action depending on the current tank level. As depicted in Figure 6a, DQN achieves a higher level of resilience with respect to other baselines. Moreover, Figure 6b shows that it has a more consistent behavior and low variance, a crucial characteristic for the resilience and safety of the water network.

3 Discussion and Conclusion

In this work, we presented `Gym4ReaL`, a benchmarking suite designed to model several real-world environments specifically tailored for RL algorithms. Unlike standard benchmarking suites, which often rely on idealized tasks, `Gym4ReaL` represents a novel library that allows for evaluating new RL methods in realistic playgrounds. Notably, the `Gym4ReaL` suite includes environments designed to capture common real-world challenges, such as limited data availability, realistic assumptions about physical process dynamics, and constrained exploration, fostering research toward broader adoption of RL methods in practical applications. For this reason, we selected a pool of environments incorporating different *characteristics* and addressing various *RL paradigms*, as shown in Table 1.

Given the standardized and flexible interface offered by our suite, a broader range of real-world problems and challenges could be easily integrated into our framework. We believe that a collective effort from the RL community can significantly advance the development of realistic, impactful

benchmarks. Hence, we encourage researchers and practitioners to explore, contribute to, and adopt `Gym4ReaL` to evaluate RL algorithms in real-world scenarios.

References

[1] R. Almgren and N. A. Chriss. Optimal execution of portfolio trans-actions. 2000. URL `https://api.semanticscholar.org/CorpusID:15502295`.

[2] A. Cominola, M. Giuliani, A. Castelletti, A. Abdallah, and D. E. Rosenberg. Developing a stochastic simulation model for the generation of residential water end-use demand time series. 2016.

[3] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1995. URL `https://proceedings.neurips.cc/paper_files/paper/1995/file/390e982518a50e280d8e2b535462ec1f-Paper.pdf`.

[4] V. De Paola, G. Calcagno, A. M. Metelli, and M. Restelli. The power of hybrid learning in industrial robotics: Efficient grasping strategies with supervised-driven reinforcement learning. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9, 2024. doi: 10.1109/IJCNN60899.2024.10650627.

[5] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester. An empirical investigation of the challenges of real-world reinforcement learning. *arXiv preprint arXiv:2003.11881*, 2020.

[6] D. Fioriti, L. Pellegrino, G. Lutzemberger, E. Micolano, and D. Poli. Optimal sizing of residential battery systems with multi-year dynamics and a novel rainflow-based model of storage degradation: An extensive italian case study. *Electric Power Systems Research*, 203, 2022. ISSN 0378-7796. doi: https://doi.org/10.1016/j.epsr.2021.107675.

[7] Gestore dei Mercati Energetici S.p.A. Historical data mgp, 2015-2020. Data retrieved from GME: https://www.mercatoelettrico.org/it/download/DatiStorici.aspx.

[8] A. Heinsbroek. Epynet. `https://github.com/Vitens/epynet`, 2016.

[9] K. A. Klise, R. Murray, and T. Haxton. An overview of the water network tool for resilience (WNTR), 2018.

[10] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[11] Q. Li, Z. Peng, L. Feng, Q. Zhang, Z. Xue, and B. Zhou. Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[12] X.-Y. Liu, H. Yang, J. Gao, and C. D. Wang. FinRL: Deep reinforcement learning framework to automate trading in quantitative finance. *ACM International Conference on AI in Finance (ICAIF)*, 2021.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL `https://doi.org/10.1038/nature14236`.

[14] S. Morad, R. Kortvelesy, M. Bettini, S. Liwicki, and A. Prorok. POPGym: Benchmarking partially observable reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=chDrutUTs0K`.

[15] A. Murillo, R. Taormina, N. O. Tippenhauer, D. Salaorni, R. van Dijk, L. Jonker, S. Vos, M. Weyns, and S. Galelli. High-fidelity cyber and physical simulation of water distribution systems. i: Models and data. *Journal of Water Resources Planning and Management*, 149 (5):04023009, 2023. doi: 10.1061/JWRMD5.WRENG-5853. URL `https://ascelibrary.org/doi/abs/10.1061/JWRMD5.WRENG-5853`.

[16] A. Naug, A. Guillen, R. Luna, V. Gundecha, C. Bash, S. Ghorbanpour, S. Mousavi, A. R. Babu, D. Markovikj, L. D. Kashyap, D. Rengarajan, and S. Sarkar. Sustaindc: Benchmarking for sustainable data center control. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 100630–100669. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/b6676756f8a935e208f394a1ba47f0bc-Paper-Datasets_and_Benchmarks_Track.pdf.

[17] S. Pfenninger and I. Staffell. Long-term patterns of european pv output using 30 years of validated hourly reanalysis and satellite data. *Energy*, 114:1251–1265, 2016. ISSN 0360-5442. doi: https://doi.org/10.1016/j.energy.2016.08.060.

[18] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22 (268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

[19] L. A. Rossman. *EPANET 2: Users Manual*. U.S. Environmental Protection Agency, Cincinnati, OH, 2000. https://www.epa.gov/water-research/epanet.

[20] D. Salaorni. Ernesto-dt, 2023. https://github.com/Daveonwave/ErNESTO-DT.

[21] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. URL http://arxiv.org/abs/1801.04381.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[23] A. Serrano-Muñoz, D. Chrysostomou, S. Bøgh, and N. Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24(254): 1–9, 2023. URL http://jmlr.org/papers/v24/23-0112.html.

[24] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[25] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

[26] I. Staffell, S. Pfenninger, and N. Johnson. A global model of hourly space heating and cooling demand at multiple spatial scales. *Nature Energy*, 8, 09 2023. doi: 10.1038/s41560-023-01341-5.

[27] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[28] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

[29] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[30] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[31] C. Yeh, V. Li, R. Datta, Y. Yue, and A. Wierman. SustainGym: A benchmark suite of reinforcement learning for sustainability applications. In *NeurIPS 2022 Workshop on Tackling Climate Change with Machine Learning*, New Orleans, LA, USA, 12 2022. URL https://www.climatechange.ai/papers/neurips2022/38.

[32] X. Yuan, L. Buşoniu, and R. Babuška. Reinforcement learning for elevator control. *IFAC Proceedings Volumes*, 41(2):2212–2217, 2008. ISSN 1474-6670. doi: https://doi.org/10.3182/20080706-5-KR-1001.00373. URL https://www.sciencedirect.com/science/article/pii/S1474667016392783. 17th IFAC World Congress.

[33] Z. Yuan, A. W. Hall, S. Zhou, L. Brunke, M. Greeff, J. Panerati, and A. P. Schoellig. Safe-control-gym: A unified benchmark suite for safe learning-based control and reinforcement learning in robotics. *IEEE Robotics and Automation Letters*, 7(4):11142–11149, 2022. doi: 10.1109/LRA.2022.3196132.

[34] A. Zouitine, D. Bertoin, P. Clavier, M. Geist, and E. Rachelson. Rrls : Robust reinforcement learning suite, 2024. URL https://arxiv.org/abs/2406.08406.

A Metadata

A.1 Hosting and Maintenance

Gym4ReaL is distributed as a Python package and is publicly available on both GitHub and PyPI. The library follows semantic versioning, with the version associated with this publication labeled as `v1.0`. A contributor guide is provided in the GitHub repository to support community involvement. The following authors serve as maintainers for the individual environments included in Gym4ReaL:

- Davide Salaorni (davide.salaorni@polimi.it): `ElevatorEnv`, `MicrogridEnv`, `WDSEnv`
- Vincenzo De Paola (vincenzo.depaola@polimi.it): `RoboFeederEnv`
- Samuele Delpero (samuele.delpero@mail.polimi.it): `DamEnv`
- Giovanni Dispoto (giovanni.dispoto@polimi.it): `TradingEnv`.

A.2 Licenses and Responsibility

Gym4ReaL as a whole is released under a Apache-2.0 license [2]. The authors bear all responsibility in case of violation of rights. Gym4ReaL does not contain any personally identifiable data, nor any offensive content.

A.3 Reproducibility

The GitHub repository includes tutorial notebooks demonstrating how to train off-the-shelf RL agents on the various environments, as well as the notebooks used to generate the results we present in this work. Additionally, detailed installation and usage instructions for each notebook are available in the Gym4ReaL wiki: `https://daveonwave.github.io/gym4ReaL/`.

A.4 Intended Usage

Gym4ReaL is designed as a benchmarking suite for evaluating RL algorithms across a diverse set of real-world-inspired tasks. While substantial effort has been made to ensure that the environments closely reflect the dynamics and constraints of real-world systems, strong performance in Gym4ReaL does not necessarily guarantee equivalent performance in actual real-world deployments.

A.5 Software and Hardware Requirements

Gym4ReaL is implemented in Python 3.12. Each environment may require specific dependencies, which are listed in the documentation and provided in the corresponding requirements files.

Gym4ReaL does not require a specific hardware configuration and is portable across all major commercial operating systems. An exception is `WDSEnv`, which relies on the EPANET simulator since the version that we adopted is not compatible with Apple Silicon processors and requires an Intel-compatible environment. For macOS users, we recommend setting up a dedicated virtual environment to ensure compatibility.

All reported experiments were conducted on an Apple M1 chip (8-core CPU, 8GB of RAM). The per-episode running time for each environment is detailed in Table 2. As can be observed, the `WDSEnv` is computationally expensive, and this is due to the inherent design of the underlying simulator, which was not originally intended for real-time control or RL applications. We further discuss this in `WDSEnv` Section D.6.

B *Characteristics* and *RL paradigms*

In this section, we define and analyze in detail the environment-specific categories identified in Table 1. The table is divided into two main sections:

[2] `https://www.apache.org/licenses/LICENSE-2.0`

Table 2: Running time required per episode for the different environments.

Environment	Time per Episode (s)
DamEnv	0.045
ElevatorEnv	0.035
MicroGridEnv	2.781
RoboFeeder-picking-v0	0.266
RoboFeeder-picking-v1	0.207
RoboFeeder-planning-v0	0.076
TradingEnv	0.007
WDSEnv	72.932

1. **Characteristics.** In the first part of the table, we report the key features of each environment with respect to the standard RL taxonomy. These *Characteristics* are inherent to the design and implementation of the environments provided by Gym4ReaL.
2. **RL Paradigms.** This second part of the table describes the RL subfields that can be associated with each environment. Specifically, although the benchmarking results provided in the main paper employ standard RL approaches, the environments can be adapted to test novel algorithms related to the respective *RL paradigms* mentioned in the table.

We now elaborate on each of the *Characteristics* and *RL paradigms* reported in the table.

Characteristics.

- **Continuous States.** The environments match this characteristic if the associated state space is continuous. Among the environments provided in Gym4ReaL, 5 out of 6 satisfy this feature, thus reflecting their realism. A notable exception is represented by ElevatorEnv, which allows testing algorithms devised for environments with finite state spaces.
- **Continuous Actions.** This characteristic is matched by the environments when the associated action space is continuous. Gym4ReaL environments cover both the finite and continuous cases.
- **Partially Observable.** This characteristic is associated with environments where only partial information is available. Partial information can be related to the difficulty of modeling or accessing relevant state features. This scenario typically happens in trading contexts, such as the one modeled in the proposed TradinvEnv.
- **Partially Controllable.** This characteristic refers to environments where part of the state evolves due to external, uncontrollable dynamics. These components, present in the observation space, are independent of the agent's actions. Examples are precipitation in a water control system (DamEnv) or market price variations (MicrogridEnv, TradingEnv).
- **Non-Stationary.** This characteristic refers to an (abrupt or gradual) variation of the underlying data distribution, which introduces additional challenges caused by the need for continuous adaptation. This feature is intrinsically related to trading contexts, such as TradingEnv.
- **Visual Input.** This characteristic is satisfied when a visual component constitutes a portion of the state space. For example, in RoboFeederEnv, the observation space is defined by camera images.

RL Paradigms.

- **Frequency Adaptation.** Given that many real-world problems can be naturally defined in the *continuous time* domain, the environments within this paradigm allow selecting actions at different frequencies. Increasing the control frequency of the system offers the agent more control opportunities, at the cost of higher computational and sample complexity. MicrogridEnv and TradingEnv are natural examples in which different market opportunities can be exploited at various frequencies.

- **Hierarchical RL.** The hierarchical RL (HRL) paradigm allows modeling, at different specificity levels, complex real-world problems presenting an inherent stratified structure. HRL divides the problem into simpler subtasks arranged in a hierarchy, thus enabling more efficient learning. Notably, `RoboFeederEnv` represents an example where the tasks of ordering (`RoboFeeder-Planning`) and collecting (`RoboFeeder-Picking`) items are hierarchically decoupled.
- **Risk-Averse.** Risk-averse RL focuses on mitigating uncertainty by favoring policies that yield more predictable and less variable outcomes. This paradigm is especially relevant in high-stakes real-world domains, such as finance (`TradingEnv`) or healthcare, where large losses can have significant consequences.
- **Imitation Learning.** Imitation Learning in RL refers to a class of methods in which an agent learns to perform tasks by mimicking expert behavior. To this extent, a dataset comprising expert choices is needed. Within our `DamEnv` and `WDSEnv` environments, we provided such data information.
- **Provably Efficient.** Provably Efficient RL refers to a class of algorithms that come with theoretical guarantees on their performance, typically in terms of sample or computational efficiency. These guarantees can be tested more easily in tabular settings, such as the one provided in `ElevatorEnv`.
- **Multi-Objective RL.** Multi-Objective RL (MORL) is a paradigm where the agent must optimize multiple contrastive objectives simultaneously, instead of a single scalar reward. Multiple contrastive objectives can be identified in `DamEnv`, `MicrogridEnv`, and `WDSEnv`, where a trade-off between competing interests needs to be simultaneously optimized.

C Datasets

C.1 `DamEnv`

The dataset underlying the current training of the `DamEnv` environment is related to Lake Como, Northern Italy. The dataset includes historical daily records of water level, demand, and inflow from 1946 to 2010. The 65 one-year-long time series have been split into train and test sets in an 80%–20% proportion, resulting in 52 years for training and 13 subsequent years for testing. Additionally, the minimum required water release for each day of the year can be integrated into the dataset. Furthermore, given the information of the lake water level, it is possible to reconstruct the control action performed by experts, therefore enabling the use of this environment to perform Imitation Learning or Inverse RL tasks. Data are released under a Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0) license.[3]

C.2 `ElevatorEnv`

`ElevatorEnv` does not rely on external datasets. Instead, passenger arrival profiles are synthetically generated using a Poisson process, with the arrival rate for each floor independently sampled from a uniform distribution within a predefined range. This range is carefully selected to avoid unrealistic scenarios, such as excessively high arrival rates that exceed the physical handling capacity of the elevator. Specifically, the arrival rate λ_f for each floor f is sampled from the interval $[0.01, 0.1]$, resulting in a minimum and maximum expected number of arrivals of approximately 43 and 372, respectively, over a one-hour window with arrivals occurring at one-second intervals.

C.3 `MicrogridEnv`

`MicrogridEnv` integrates diverse datasets covering energy consumption, renewable energy generation, market pricing, and ambient temperature. Most datasets are provided at hourly resolution, except for ambient temperature, which is available daily. The data span the period from 2015 to 2021. We use the first four years for training and reserve the final year for testing.

[3] `https://creativecommons.org/licenses/by-nc/4.0/`

Energy Consumption. This dataset includes 398 realistic demand profiles representing residential electricity usage across different Italian regions (North, Center, South, and islands), obtained from Fioriti et al. [6]. We split these demand time-series in 370 profiles for training and the remaining 28 for testing. The dataset is released under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.[4]

Energy Generation. Hourly renewable generation profiles are computed for a 3 kW residential photovoltaic installation located in northern Italy. Capacity factors are derived from Pfenninger and Staffell [17] and retrieved via `https://www.renewables.ninja`, under a CC BY-NC 4.0 license[3].

Energy Market. Electricity prices are based on hourly data from the Italian Gestore Mercati Energetici (GME) [7], including the National Single Price and fixed tax-related and operational fees. These data, available under the CC BY-NC 4.0 license, are used to define the energy selling price. The buying price includes an additional empirical surcharge to account for taxes and provider fees.

Ambient Temperature. Daily temperature data are sourced from Staffell et al. [26] and obtained from `https://www.renewables.ninja`, also under the CC BY-NC 4.0 license[3].

C.4 `RoboFeederEnv`

The `RoboFeederEnv`'s tasks do not rely on any external dataset. In particular, in the `picking-v0` environment, the object detection module is trained offline using a set of 300 synthetic images generated directly from the simulator. The geometry of the object is not synthetic or artificially designed, but it is derived directly from real-world components used in the automotive industry. Specifically, it reflects the exact structure of metallic hinges employed in assembly applications, ensuring that all evaluations are grounded in practical, industrially relevant scenarios. This training process is independent of the environment's dynamics and interaction loop, ensuring a clear separation between perception and control.

C.5 `TradingEnv`

TradingEnv is built using historical market data. In this suite, we consider the Forex market, in which currencies are traded. In particular, in the environment, we provide an implementation for the EUR/USD currency pair. The Forex market is open 24 hours a day, 5 days a week. In this environment, we use tick-level data freely available at `https://www.histdata.com/`, which provides the highest level of granularity by recording every individual price change. Each entry in the dataset contains the datetime with nano-second resolution, bid price, ask price, and volume, which is then resampled to a minute resolution.

C.6 `WDSEnv`

`WDSEnv` relies on synthetically generated residential water demand datasets using STREaM [2], a stochastic simulation model to generate synthetic time series of water end uses with diverse sampling resolutions. STREaM is calibrated on a large dataset that includes observed and disaggregated water end-uses from U.S. single-family households. The STREaM simulator is available under the GNU General Public License 3.0 (GPL-3.0)[5].

We categorized the generated profiles into one of three classes, representing increasing levels of stress imposed on the water distribution system: *normal*, *stressful*, or *extreme*. These classes are designed to facilitate the analysis of network resilience under varying conditions.

During training, users can specify the probability distribution over the three demand types via the environment's configuration file. Specifically, the dataset includes 21 files corresponding to *normal* operating conditions, 5 files reflecting *stressful* scenarios, and 5 files depicting *extreme* demand situations. Each file comprises a time series of weekly demand values over a period of 53 weeks. The test set comprises 1,000 synthetic profiles, held out from training and randomly sampled across the defined demand categories.

[4] `https://creativecommons.org/licenses/by/4.0/`

[5] `https://www.gnu.org/licenses/gpl-3.0.html`

D Environment Details

D.1 `DamEnv`

In addition to the description in Section 2.1, in this section we provide further details about the `DamEnv` environment. This environment models a water reservoir with the aim of controlling the water release to meet the demand while avoiding floods and starvation. The action is provided daily, and it represents the amount of water per second to be released during the day.

Simulator. The dynamics of the reservoir are modeled with greater granularity with respect to the action, which is provided daily. This is controlled by the parameter `integration` in the environment configuration file. It is set by default to 24, which corresponds to one iteration per hour.

For each iteration, the bounds of the feasible amount of water to release are computed as explained in the following paragraph. Those bounds then clip the action, and the new water level is calculated. In this calculation, the water inflow and the evaporation (if provided at initialization) are also taken into account.

In Table 3 we report the default parameters of the simulator, which refer to the Lake Como basin, in Northern Italy. We set `evaporation` to `False`. If set to `True`, `evaporation_rates` must be provided in the form of a csv file path with a value for each day of the year. Many of these parameters are needed for clipping the action, and their role will be clarified in the following paragraph.

Table 3: `DamEnv`'s simulator parameters.

Parameter	Symbol	Value
Surface	A	145.9 [km^2]
Evaporation	-	False [-]
Initial level	l_0	0.35 [m]
Minimum flow	r_{min}	5 [m^3/s]
Minimum level	l_{min}	-0.5 [m]
Maximum level	l_{max}	1.25 [m]
Zero-flow level	α	-2.5 [m]
Rating exponent	β	2.015 [-]
Discharge coefficient	C_r	33.37 [m$^{3-\beta}$/s]
Linear slope	k	1488.1 [m^2/s]
Linear intercept	c	744.05 [m^3/s]
Linear limit	l_{lim}	-0.4 [m]

Feasible Actions. For each iteration of the lake simulator, the action is clipped in the feasible range, which is bounded by two piecewise functions. Both the maximum and the minimum account for operational and physical constraints, such as minimum water release requirements and hydraulic limits. These bounds are functions of the water level and the minimum daily release r_t^{min}.

The *minimum release* $q_{\text{min}}(l, t)$ on day t is defined as:

$$q_{\text{min}}(l,t) = \begin{cases} 0 & l \leq l_{\text{min}}, \\ r_{\text{min},t} & l_{\text{min}} < l \leq l_{\text{max}}, \\ C_r(l-\alpha)^{\beta} & l > l_{\text{max}}\,. \end{cases}$$

This structure ensures that: no water is released when the water level is below the critical level l_{min}, the minimum release $r_{\text{min},t}$ is maintained within the normal operating range, and a nonlinear release based on a rating curve applies when the level exceeds h_{max}.

The *maximum release* $q_{\text{max}}(h)$ is defined as:

$$q_{\max}(l) = \begin{cases} 0 & l \leq l_{\min}, \\ kl + c & l_{\min} < l \leq l_{\lim}, \\ C_r(l - \alpha)^{\beta} & l > l_{\lim}. \end{cases}$$

This formulation reflects operational policies where: no release occurs below the critical level $h_{\min}$, a linear release policy applies in the intermediate range, and a nonlinear rating curve governs the release when the water level exceeds $l_{\lim}$.

Reward Function. The reward function of `DamEnv` presents five components:

- **Daily deficit**: it penalizes actions that do not meet the daily demand. It is computed as $-\max\left(d_t - \max\left(r_t - r_{\min,t}, 0\right), 0\right)$, where at time t, d_t is the demand, r_t is the water that is being released, and $r_{\min,t}$ is the minimum amount of water that needs to be released.
- **Overflow**: it penalizes actions that lead the water level to go beyond the overflow threshold. If this happens, its value is -1, otherwise 0.
- **Starving**: similarly, it penalizes actions that lead the water level to go below the starving threshold. If this happens, its value is -1, otherwise 0.
- **Wasted water**: it penalizes actions that release more water than the demand. It is computed as $-\max\left(r_t - d_t, 0\right)$, where r_t and d_t are defined as above.
- **Clipping**: it penalizes actions that do not fulfill the constraints of the environment. It is calculated as $-\left(a_t - r_t\right)^2$ where a_t is the action performed at time t.

Every component is weighted by a coefficient that can be set when the environment is created, balancing the influence of each component depending on the specific characteristics of the water reservoir considered.

Limitations. Simulating a different water reservoir with `DamEnv` requires domain-specific expertise to configure realistic environment parameters. Similarly, acquiring high-quality, multi-year data for other reservoirs, essential for effective learning, may be difficult.

D.2 `ElevatorEnv`

This section provides additional details on the `ElevatorEnv` w.r.t. the main paper. The environment simulates the operation of an elevator system under a specific and realistic traffic pattern known as *peak-down traffic*, which typically occurs during limited periods of the day (e.g., lunch breaks or end-of-work shifts), when the majority of users exit upper floors toward the ground level.

We adopt an episodic setting lasting 3600 seconds (i.e., one hour), with control actions executed every second. While the environment does not include a full physical simulator, we deliberately model it in a fully discrete manner. This choice enables a compact formalization of the dynamics while preserving the core challenges of the task and allows us to use tabular policies, which are both simple and interpretable. This is particularly advantageous for benchmarking, as it avoids the complexity of large-scale function approximation while still offering meaningful learning dynamics. By carefully designing the observation space and controlling its dimensionality, we ensure that the problem remains tractable, yet rich enough to pose a non-trivial challenge. Additionally, considering the six `Gym4ReaL` environments, this is the only one that allows users to test RL algorithms designed for tabular settings, as typical of provably efficient approaches, thus broadening the applicability of our library.

The configurable parameters used in this task are reported in Table 4.

Assumptions. The following assumptions are made to simplify the dynamics of the `ElevatorEnv`, allowing for an efficient yet representative modeling of the task:

- **No acceleration dynamics.** We do not model acceleration or deceleration; the elevator is assumed to move with uniform rectilinear motion;
- **Instantaneous boarding.** We assume that the entry and exit of passengers inside the elevator occur instantaneously;

Table 4: `ElevatorEnv` parameters.

Parameter	Symbol	Value
Floors	f	$\{0, \dots, 4\}$ [-]
Maximum capacity	C_{max}	4 [people]
Movement speed	v	3 [m/s]
Floor height	-	6 [m]
Maximum queue length	$W_{f,\text{max}}$	3 [people]
Maximum new arrivals	-	2 [people]
Arrival rate	λ_f	$[0.01, 0.1]$ [-]

- **Uniform floor spacing.** All floors are equidistant, and the total height of the building is assumed to be divisible by the elevator's speed. This ensures that the elevator can only occupy a finite number of discrete vertical positions;
- **Elevator height.** For the same reason as the previous bullet point, we assume that the height of the elevator is equal to the floor height;
- **Non-floor stopping.** We allow the elevator to open its doors even between two floors, unlike in real-world systems. While this behavior is physically unrealistic, it does not provide any advantage. The agent is penalized for unnecessary stops and will learn to avoid such actions through experience.

Feasible Actions. We adopt a permissive design where all control actions are considered feasible at any state. In particular, we do not explicitly penalize the agent for executing physically invalid actions, such as opening doors between two floors (e.g., when the floor height is smaller than the elevator's movement speed), or attempting to move upward at the top floor ($a_t = u$ when $h_t = H$), or downward at the ground floor ($a_t = d$ when $h_t = 0$). These actions do not result in termination or constraint violations but are treated as valid no-ops since the agent is still implicitly penalized through time progression, as ineffective or wasteful actions delay task completion and reduce the cumulative reward.

Reward Function. Although not strictly necessary to solve the task, the β hyperparameter plays a crucial role in accelerating the learning process. Specifically, β modulates the only source of positive reward available to the agent, i.e., passenger delivery. This design encourages the agent to recognize that minimizing the cumulative waiting time inherently requires completing passenger trips by bringing them to their destinations. In doing so, β reinforces the primary operational goal of an elevator system: transporting users efficiently rather than merely avoiding penalties or idle behavior.

Limitations. The principal limitation of `ElevatorEnv` arises from its scalability. While the environment is inherently finite and conceptually straightforward, the dimensionality of its observation space can increase rapidly, rendering it intractable for tabular RL algorithms. Consequently, it is essential to carefully configure the environment, adjusting its parameters to ensure that the resulting problem is neither trivial nor computationally prohibitive.

D.3 `MicrogridEnv`

This section provides further details on `MicrogridEnv`. As outlined in the main paper, this environment models a control problem centered on optimal electrical energy management in a residential setting equipped with a PV implant, a battery system, and access to the main grid for energy trading. At each time step t, the controller must decide how to allocate the net power, $P_{N,t} = P_{G,t} - P_{D,t}$ where $P_{G,t}$ is the power generated by the PV system and $P_{D,t}$ is the residential electricity demand. The agent must choose whether to address the net power to the battery or trade it with the grid. Importantly, this decision is made based on estimates of $P_{G,t}$ and $P_{D,t}$, as the actual values are unknown at decision time.

Consequently, the decision-making process is inherently influenced by uncertainty in both demand and generation forecasts, as well as by exogenous factors such as electricity prices and ambient

temperature, each exhibiting cyclo-stationary patterns. Notably, ambient temperature significantly impacts battery degradation, creating a long-term trade-off between maximizing short-term energy efficiency and preserving battery health.

The problem is formulated as an infinite-horizon task, which terminates when the battery reaches its end-of-life (EOL) condition. Control actions are taken every 3600 seconds, matching the finest available data granularity.

Simulator. We simulate the dynamics of the BESS using a digital twin [20], which models the evolution of the battery's state of charge (SoC), temperature, and state of health (SoH) over time. The simulator emulates a realistic lithium-ion battery pack suitable for residential applications. The simulator operates in a step-wise manner consistent with the Gymnasium interface. At each time step, it receives the input power to store or retrieve from the battery, computes the resulting voltage, thermal dynamics, SoC update, and degradation, and returns a snapshot of the battery's status to the RL agent.

Model parameters and configurations are based on expert consultation and are preconfigured for direct use. However, users can easily modify or replace the battery model configuration to experiment with different setups or refer directly to the full simulator details in [20]. The specific parameters used in this work are summarized in Table 5.

Table 5: `MicrogridEnv`'s simulator parameters.

Parameter	Symbol	Value
Nominal capacity	C_N	60.0 [Ah]
Maximum voltage	V_{max}	398.4 [V]
Minimum voltage	V_{min}	288.0 [V]
Replacement cost	$\mathcal{R}$	3000 [€]
SoC range	-	$[0.2, 1]$ [-]

Observation Space. The observation space includes the variables $\widehat{P}_{D,t}$, $\widehat{P}_{G,t}$, p_t^{buy}, and p_t^{sell}, which encapsulate the exogenous factors influencing the system, namely, energy demand, renewable energy generation, and the buying/selling energy prices. Ideally, the control decision at time t should rely on the actual values of demand $P_{D,t}$ and generation $P_{G,t}$. However, these values are not available at decision time. To overcome this, we approximate them using their most recent observations at time $t-1$, denoted as $\widehat{P}_{D,t}$ and $\widehat{P}_{G,t}$. This assumption is reasonable in typical microgrid scenarios, where energy demand and generation profiles exhibit temporal smoothness or autocorrelation, allowing previous values to serve as informative estimators. On the other hand, energy market prices p_t^{buy} and p_t^{sell} are assumed to be known in advance. This assumption is justified by the common practice in energy markets where prices are typically set a day ahead, enabling the agent to access this information at decision time.

To further enhance the expressiveness of the observation space, we introduce two time-based features that encode temporal periodicity. These are defined as:

- $\varphi^d = \frac{2\pi\tau_d}{T_d}$, where $\tau_d \in [0, T_d]$ represents the current time of day in seconds, and T_d is the total number of seconds in a day. This captures daily periodicity (e.g., day-night cycles).
- $\varphi^y = \frac{2\pi\tau_y}{T_y}$, where $\tau_y \in [0, T_y]$ represents the current time of year in seconds, and T_y is the total number of seconds in a year. This accounts for seasonal patterns within datasets.

Feasible Actions. The controller's action a_t defines the fraction of the net power $P_{N,t}$ to allocate to the battery at time step t. We can formally express this decision as:

$$P_{B,t} = a_t P_{N,t}, \tag{10}$$

$$P_{E,t} = (1 - a_t) P_{N,t}, \tag{11}$$

where $P_{B,t}$ is the amount of power stored in (or retrieved from) the battery, and $P_{E,t}$ is the portion traded with the main grid, either sold or purchased.

The controller's behavior depends on the sign of $P_{N,t}$:

- **Deficit case** ($P_{N,t} < 0$): The generated PV power does not meet the demand. The controller determines $P_{B,t}$, the amount to draw from the battery, and covers the remaining deficit $P_{E,t}$ by purchasing from the grid.
- **Surplus case** ($P_{N,t} > 0$): There is excess generation relative to demand. The controller allocates $P_{B,t}$ to the battery and sells the remaining $P_{E,t}$ to the grid.

The chosen action must respect the physical constraints of the BESS, such as bounds on charging/discharging power and SoC limits. At time t, SoC denoted as $\sigma_t \in [0, 1]$ is defined by:

$$\sigma_t := \sigma_1 + \sum_{h=2}^{t} \frac{i_h \Delta\tau}{C_h}, \tag{12}$$

where σ_1 is the initial SoC, i_h is the current, C_h is the internal battery capacity at time h, and $\Delta\tau$ is the time step in hours. To ensure valid operations, $P_{B,t}$ must satisfy:

$$P_{dch} \leq P_{B,t} \leq P_{ch}, \tag{13}$$

$$\frac{(\sigma_{\min} - \sigma_t)}{\Delta\tau} C_t V_t \leq P_{B,t} \leq \frac{(\sigma_{\max} - \sigma_t)}{\Delta\tau} C_t V_t, \tag{14}$$

where $P_{dch} < 0$ and $P_{ch} > 0$ are the maximum discharging and charging powers, $\sigma_{\min}$ and $\sigma_{\max}$ are the minimum and maximum SoC levels, and V_t is the battery voltage at time t.

Reward Function. The reward signal consists of three components: trading profit, battery degradation cost, and a penalty for violating physical constraints. Formally, the trading component $r_{\text{trad}}(a_t)$ of the reward is defined as:

$$r_{\text{trad}}(a_t) = \left(p_t^{\text{sell}} P_{E,t}^{+} + p_t^{\text{buy}} P_{E,t}^{-} \right) \Delta\tau \,, \tag{15}$$

where p_t^{sell} and p_t^{buy} are the unit energy prices for selling and buying electricity at time t, respectively, with $p_t^{\text{sell}} < p_t^{\text{buy}}$. The terms $P_{E,t}^{+}$ and $P_{E,t}^{-}$ denote the positive and negative parts of $P_{E,t}$, respectively.[6]

The degradation component $r_{\text{deg}}(a_t)$ accounts for battery aging and is linked to the SoH, which monotonically decreases from 100% to an application-specific EOL threshold (typically 60–80%). Let $\rho_t \in [0, 1]$ denote the SoH at time t, defined as:

$$\rho_t := \frac{C_t}{C_N}, \tag{16}$$

where C_t is the internal capacity at time t, and $C_N \in \mathbb{R}^+$ is the nominal capacity of the system. The SoH decay incorporates both calendar aging and usage-related degradation. Thus, the degradation cost is computed as:

$$r_{\text{deg}}(a_t) = \frac{\rho_t - \rho_{t-1}}{1 - \rho_{\text{EOL}}} \mathcal{R}, \tag{17}$$

where $\rho_{\text{EOL}} \in (0, 1)$ is the SoH value at end-of-life, and $\mathcal{R}$ is the BESS replacement cost. Note that since $\rho_t < \rho_{t-1}$, this term yields a negative reward.

Finally, $r_{\text{clip}}(a_t)$ penalizes actions that violate the system's operational constraints and must be clipped. In our formulation, we only penalize violations of the tighter constraint (14), which is stricter than (13). Hence, formally, the penalty is defined as:

$$r_{\text{clip}}(a_t) = -\max\left\{0,\, a_t P_{N,t} + \frac{(\sigma_t - \sigma_{\max})}{\Delta\tau} C_t V_t,\, \frac{(\sigma_{\min} - \sigma_t)}{\Delta\tau} C_t V_t - a_t P_{N,t}\right\}. \tag{18}$$

Limitations. The main limitation of `MicrogridEnv` lies in its data and configuration specificity. The environment is built upon data representative of a single geographical and regulatory context (Italy), which includes particular consumption patterns, solar generation profiles, and market prices. Additionally, the battery configuration models a specific lithium-ion storage system with fixed

[6] For a real-valued quantity $q \in \mathbb{R}$, its positive and negative parts are defined as $q^+ := \max\{0, q\}$ and $q^- := \min\{0, q\}$, respectively.

chemical composition, physical dimensions, and operational parameters. As a result, the environment may not generalize directly to other geographical regions or technological setups without additional data integration or customization.

D.4 `RoboFeederEnv`

In addition to the description provided in Section 2.4, we present further details used in designing the `Robofeeder` environment. `Robofeeder` (Fig. 7) is designed to handle small objects like those shown in Figure 8, using a commercial 6-DOF Staubli robot with a predefined kinematic cycle that enables reliable pick-and-place operations. The environment includes a configuration file that allows users to specify the number of objects in the scene and their initial orientations. Notably, since `RoboFeederEnv` is built on MuJoCo, the object geometries can be easily modified, making it possible to extend the simulator's capabilities far beyond the examples presented here. In Table 6 we report the main parameters of the `RoboFeederEnv` environments.

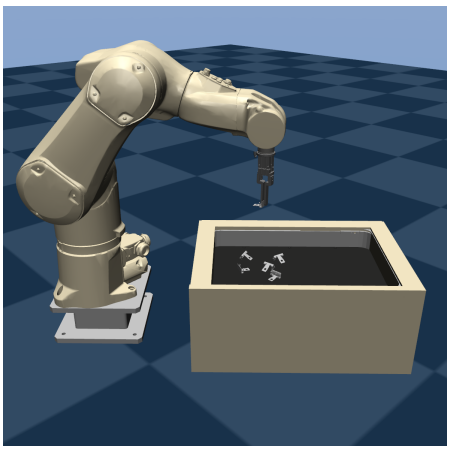


Figure 7: Rendering of MuJoCo simulator from `RoboFeederEnv`.

Figure 8: the objects considered, metallic hinges from the automotive sector ([4]).

Table 6: `RoboFeeder` Parameters.

Parameter	**Symbol**	`picking-v0`	`picking-v1`	`planning`
# objects	N	1	1	3
Observation shape	-	$1 \times 50 \times 50 \times 1$ [px]	$1 \times 300 \times 300 \times 1$ [px]	$3 \times 50 \times 50 \times 1$ [px]
Distance-rotation weight	λ	0.85	0.85	-
Threshold distance	δ_d	0.012	0.012	-
Threshold rotation	δ_θ	0.35	0.35	-
Distance decay rate	α_d	$-\frac{\ln(0.5)}{(\delta_d)^2}$	$-\frac{\ln(0.5)}{(\delta_d)^2}$	-
Rotation decay rate	α_θ	$-\frac{\ln(0.5)}{(\delta_\theta)^2}$	$-\frac{\ln(0.5)}{(\delta_\theta)^2}$	-
Timestamps	T	2	2	5

D.4.1 `RoboFeeder-picking`

In each episode, the agent interacts with the family of `RoboFeeder-picking` environments by selecting an action a_0 based on the initial state s_0 of the system, where a single object is placed within the robot's working area. The action $a_0 \in [-1, 1]$ is rescaled with respect to the observation space to map it onto the u-v image coordinates, representing a precise position in the MuJoCo robotic world. After executing the action, the agent receives a reward $r(s_0, a_0)$, and the simulator transitions to state s_1 depending on the success of the robot's motion. This process repeats at each timestep until the episode reaches the terminal step $t = T$.

Assumptions. The picking environments are designed to train a specialized robot to grasp a single, small, and complex-geometry object. To ensure the robot can consistently focus on the picking task, the environment contains only one object, which is placed in a way that always allows it to be successfully grasped. Even though the object's position is randomized at the start of each episode, its orientation is carefully set to ensure that at least one feasible robot pose exists from which it can be picked. When working with 6-DOF robots, mechanical constraints may prevent the end-effector from reaching every position in Cartesian space. Therefore, robot programming should avoid assigning target poses that are near kinematic singularities or that would place excessive stress on the joints. To replicate reality, we exclude from the feasible action space the upper part of the workspace area, not reachable by the robot. This assumption could be changed from the base configuration file provided in the environment implementation.

Observation Space. `Gym4Real` implements two versions of this environment. `picking-v0` represents the simpler of the two, as the region of interest in which the agent must search is limited to the object's neighborhood, thanks to the assistance of a pretrained object detection neural network. By leveraging an SSD MobileNet V2 network [21], the observation space is significantly reduced to a cropped image of dimensions $1 \times H_{\text{cropped}} \times W_{\text{cropped}} \times C$, where $H_{\text{cropped}} = W_{\text{cropped}} = 50\,\text{px}$, and $C = 1$. This reduction simplifies the task, as the reward function is designed to evaluate the distance between the robot's selected grasp point and the actual object. By narrowing the search area, the agent can focus on a more relevant region of the scene, reducing training time and avoiding sparse experience with little to no learning signal.

Feasible Actions. Actions coincide with a continuous value that normalizes the choice of where to grasp, depending on the observation space definition. It means that based on the observation, i.e., the actual visual input, the agent selects the portion of the image to identify as a grasping point. From the image projection, the environment converts the agent's choice to a unique robot world position, which is relative to the fixed position of the top-down camera in the world. The latter is set parallel to the object's *home* position, acquiring a fixed portion of the scene. This design allows for playing an action with a granularity definition in the continuous domain.

Reward Function. The reward function for the `picking` environments is defined as a weighted sum of the distance and orientation alignment between the robot's end-effector (TCP) and the target object.

Let d denote the Euclidean distance between the TCP and the object, and $\Delta\theta$ the angular misalignment (in radians) between the TCP and the object. Define δ_{lim} as the distance threshold for soft negative feedback, and δ_θ as the angular threshold for rotational alignment. A weighting parameter $\lambda \in [0, 1]$ balances the contribution of the distance and rotational terms.

The reward function $r(d, \Delta\theta)$ is then given by:

$$r_t(d, \Delta\theta) = \begin{cases} 1 - \left[\lambda \cdot \exp\left(\alpha_{\text{d}} \cdot d^2\right) + (1-\lambda) \cdot \exp\left(\alpha_\theta \cdot (\Delta\theta)^2\right)\right] & \text{picking failed} \\ 1 & \text{picking successfull} \end{cases} \tag{19}$$

α_d and α_θ act as decay rates to incentivize the agents to correctly pick the object, where $\alpha_d = -\frac{\ln(0.5)}{(\delta_d)^2}$ and $\alpha_\theta = -\frac{\ln(0.5)}{(\delta_\theta)^2}$.

This formulation ensures that when $d = \delta_{\text{d}}$ or $\Delta\theta = \delta_\theta$, the respective exponential term evaluates to 0.5. The parameter λ thus tunes the agent's sensitivity between spatial precision and angular alignment during grasping attempts. Both versions of the picking environments share the parameter λ, which expresses the preference of the simulator for giving a higher reward when correctly approaching the theoretical point of grasp in the Euclidean space with respect to its correct orientation. λ is set to encourage the agent to reach a limit distance in terms of space and angle, so that it will be able to correctly pick the object. The cumulative reward is updated as $\sum_t^T r_t(d, \Delta_\theta)$.

Limitations. The `picking` environments are well-suited for scenarios where the user needs to adjust the object geometry. However, the embedded kinematic solver is specifically designed for the

robot being used, which means it does not support modifications to the robot model. This limitation extends to the two-finger grasping tool and its pick-and-place cycle.

Why RL is Needed. Although the `RoboFeeder-picking` problem may initially appear solvable through supervised learning, given that the object is always placed in a pickable position, the task fundamentally requires RL. This is due to the need for the agent to learn grasping as a procedural strategy, rather than a simple input-output mapping. In practice, it is not sufficient to identify where the object is located; the agent must also learn how to approach and align itself to perform a successful grasp under realistic constraints, such as occlusions, partial views, and kinematic limitations of the robot. While in a real-world setting, a human expert could manually specify the grasp points, the objective of this environment is to enable the robot to autonomously learn the entire grasping process—from perception to actuation—without supervision. This involves discovering optimal action sequences and dealing with delayed rewards and sparse success signals, challenges that are inherently better addressed through RL frameworks rather than supervised data-driven approaches.

D.4.2 `RoboFeeder-planning`

In each episode, the agent interacts with the `RoboFeeder-planning` environment by selecting a discrete action a_0 based on the initial state s_0, where multiple objects are placed within the robot's working area. The action $a_0 \in \{0, \ldots, n\}$, with $n \in \mathbb{N}$, corresponds to a relative index in a vector of cropped images representing the state s_0. From s_0, the selected cropped image, of shape $1 \times H_{\text{cropped}} \times W_{\text{cropped}} \times C$, is used as input to a pre-trained agent that determines a grasp point to reach an object, if one is present in the selected region. Depending on the success of the grasp attempt, the agent receives a reward $r(s_0, a_0)$, and the simulator transitions to a new state s_1. This process is repeated at each time step $t \in [T] \coloneqq \{1, \ldots, T\}$ until the episode reaches the final step $t = T$.

Observation Space. The environment is designed to handle a variable number of objects within the robot's workspace. This is achieved by maintaining a fixed-size array of detected objects. The visual observation at each timestep is represented as $s_t = [\mathbf{X}_{1,t}, \ldots, \mathbf{X}_{N,t}], \quad \mathbf{X}_{i,t} \in \mathbb{R}^{H \times W \times C}$ where each $\mathbf{X}_{i,t}$ corresponds to the image crop of the i-th detected object. Although the environment can potentially detect many objects, we define an upper bound $N = 3$ on the number of objects processed in each observation, ensuring a consistent and manageable input size for the agent. Each cropped image, used in observation space, contains the result of the object detection stage, capturing only one object at a time and creating a totally black image when the number of objects present in the robot workspace is less than N. This would happen if the environment is loaded with fewer objects or during the episode, simulating with the robot capable of placing the objects in the final position.

Feasible Actions. Based on the construction of the observation space, the agent can only select an index $i \in [1, N]$ with N the maximum fixed number of elements. object processed, to attempt a demanding grasping of the inner picking agent. The action $a_t = 0$ corresponds to a reset request.

Limitations. This environment strongly depends on the performance of a low-level pre-trained agent capable of solving the underlying `picking` tasks. The optimal policy $\pi^*(a, s)$ is one that can correctly infer when an object is properly oriented for picking, or when a sequence of grasp attempts will eventually allow the robot to feed out the remaining objects.

D.5 `TradingEnv`

In this section, we provide additional details about `TradingEnv` introduced in Section 2.5. The environment, for each episode, simulates a trading day, and at every step, the agent decides which action to perform to maximize the profit-and-loss (P&L) net of transaction costs. The configuration parameters considered in the presented setting are reported in Table 7.

Assumptions. As stated in the main paper, we restricted the trading hours from 8:00 EST to 18:00 EST, from Monday to Friday, due to low liquidity outside this interval. Hence, since we are learning an intra-day trading strategy, the position of the agent is opened at 8:00 EST and forced to be closed at 18:00 EST.

Observation Space. In this section, we want to give more information about the considered observation space. As said, the variable $\mathbf{d}_t$ is the vector of the delta mid-prices. The mid-price p_t is defined as the average price between bid and ask $p_t = \frac{p_{bid}+p_{ask}}{2}$.

Furthermore, the observation space can be customized, including more and different market information, such as:

- **Number of delta mid-prices:** Setting a higher number of delta mid-prices could allow for considering longer price evolution. However, this could increase noise in the observations, negatively impacting the learning.
- **Delta mid-prices offset:** Default is 1 minute, but it is possible to enlarge it to consider the return obtained with different intervals (e.g., by considering 60 delta mid-prices with 1-minute offset, we can observe information on the last hour price variation).
- **Temporal information:** By default, only the timestamp is considered, but it is possible to consider also Month, Day, and Day of the Week to capture more complex temporal patterns.

Table 7: `TradingEnv` parameters.

Parameter	Symbol	Value
Number of deltas	$\mathbf{d}_t$	60 [-]
Offset	-	1 [min]
Persistence	-	5 [min]
Capital	C	100k [€]
Fees	λ	1 [€]

Regarding the temporal features, it is also possible to enable the cyclic encoding, using sine and cosine transformations. Cyclic encoding emphasizes that time-related quantities, although numerically distant, can be close in a cyclical sense. This type of encoding is typically preferred when using function approximators such as neural networks. In contrast, it is not recommended in some cases, such as with tree-based models, which are sometimes considered in trading settings, where the encoding can obscure the natural ordering of features.

Feasible Actions. At each decision step, the agent can perform 3 actions: As explained in Section 2.5, at each decision step, the agent can perform 3 actions: *long*, *short*, and *flat*. Each action involves a fixed amount of capital C that in the experiments we set to €100k.

In this environment, it is possible to specify the action frequency (or *persistence*), as time is discretized. The default frequency is 5 minutes, meaning that the agent reassesses its position every 5 minutes. It is important to emphasize that the persistence should be adjusted in conjunction with the observed delta mid-prices, both in terms of their quantity and the temporal offset used.

Reward Function. In the presented reward formulation, λ is the transaction fees that in our experiments is set to 1$, which is paid for every position change. Modification to the fees parameter could lead to more conservative or aggressive strategies.

Limitations. The main limitation of `TradingEnv` is that, being built using historical data, the effective quality of the learning environment is highly linked to the quality of those. Such data typically contains missing values that must be appropriately handled and preprocessed. In our case, missing values up to 5 minutes (this is a customizable threshold) are forward-filled. Instead, all the days with wider gaps are removed from the data. In addition, the market impact is not taken into account. This is not a substantial limitation in high liquid settings, as the EUR/USD currency pair, considered in this paper.

We refer to Optimal Execution approaches for further details [1], since this is a well-established research area that aims to minimize market impact and prevent significant price movement during trade execution.

D.6 WDSEnv

WDSEnv addresses the problem of enhancing the resilience of residential water distribution networks. As discussed in the main paper, we rely on the EPANET simulator [19] to model the evolution of the network and to compute the fluid dynamics within the system. The task is formulated as an infinite-horizon decision-making problem, with each episode truncated after one week of simulated time. Both the hydraulic step and the demand pattern step are set to one hour, aligning with the temporal resolution typically used in real-world water management systems.

Simulator. To simulate the environment, we rely on `Epynet` [8], a Python wrapper for EPANET, upon which we built our Gymnasium-based environment. While the most widely adopted Python interface for EPANET is currently `WNTR` [9], it lacks support for step-wise (i.e., discrete-time) simulation of the hydraulic network [15], which is essential for reinforcement learning tasks. In contrast, `Epynet` includes built-in features that facilitate step-wise simulation with only minimal modifications required on our end.

The simulator models the water distribution system as a graph composed of nodes and links. Among the nodes, tanks, and junctions play a critical role: tanks store water and help meet demand during peak usage or scarcity and must maintain their water levels within operational boundaries; junctions, on the other hand, are expected to maintain adequate pressure to ensure service reliability.: tanks store water and help meet demand during peak usage or scarcity, and must maintain their water levels within operational boundaries; junctions, on the other hand, are expected to maintain adequate pressure to ensure service reliability. Each junction is assigned a base demand value in the `.inp` file, which represents the nominal water demand at that location. The *Demand Satisfaction Ratio* (DSR) is used as a key indicator of how well the actual demand is met over time.

It is important to note that the demand values in `.inp` files are static. To simulate dynamic and diverse operational conditions, we introduce time-varying demand profiles that modulate the base demand values to reflect *normal*, *stressful*, and *extreme* scenarios. These profiles allow us to simulate periods of increased stress on the network, such as high usage due to heatwaves or population surges. In Table 8, we summarize the main parameters used to configure the water distribution simulator.

The network chosen for the experiment is the *Anytown* system, which has the following composition: 22 junctions (from J_1 to J_22), 2 tanks (*T41* and *T42*), 2 pumps (*P78* and *P79*), and 1 reservoir (*R40*). Further details on the network configuration are available in the relative *.inp* file.

Table 8: `WDSEnv`'s simulator parameters.

Parameter	Symbol	Value
Town	-	*anytown.inp*
Demand estimation	$\widehat{d}_t$	SMA
Normal demand prob.	-	0.6 [-]
Stressful demand prob.	-	0.35 [-]
Extreme demand prob.	-	0.05 [-]
Overflow risk	λ_{of}	0.9 [-]

Observation Space. The observation space considered in the reported experiment includes both the tank level ($L = 2$) and all the junctions of the network ($J = 22$). The estimated total demand pattern $\widehat{d}_t$ is computed as a moving average over the previous quarter of the day, corresponding to a 6-hour window. This can be implemented either as a simple moving average (SMA) or using an exponentially weighted moving average (EWMA), depending on the desired responsiveness to recent variations. Additionally, the observation space includes a time-based feature to capture intra-day periodicity. Specifically, we define $\varphi^d = \frac{2\pi\tau_d}{T_d}$, where $\tau_d \in [0, T_d]$ denotes the current time of day in seconds, and T_d is the total number of seconds in a day (i.e., $T_d = 86400$).

Reward Function. The reward function consists of two main components: the DSR term and the overflow penalty term. These components jointly encourage the agent to maintain an adequate water supply across the network while avoiding risky conditions such as tank overflows.

The first component, the DSR term $r_{\text{DSR},t}(a_t)$, quantifies how effectively the agent satisfies the expected demand across all junctions at time t. It is defined as:

$$r_{\text{DSR},t}(a_t) = \frac{\sum_{j=1}^{J} d_{j,t}}{\sum_{j=1}^{J} \overline{d_{j,t}}}, \tag{20}$$

where $d_{j,t}$ denotes the actual supplied demand at junction j and time t, and $\overline{d_{j,t}}$ is the corresponding expected demand. This term rewards the agent proportionally to the fraction of demand satisfied, with $r_{\text{DSR},t}(a_t) = 1$ indicating perfect demand satisfaction.

The second component, the overflow penalty term $r_{\text{of},t}(a_t)$, is designed to discourage unsafe operational states where tank levels approach their capacity limits. It is defined as:

$$r_{\text{of},t}(a_t) = \begin{cases} \sum_{l=1}^{L} \frac{h_{l,t} - \lambda_{\text{of}}\, h_{l,\max}}{(1-\lambda_{\text{of}})\, h_{l,\max}} & \text{if } h_{l,t} > \lambda_{\text{of}}\, h_{l,\max}, \\ 0 & \text{otherwise,} \end{cases} \tag{21}$$

where $h_{l,t}$ is the current level of tank l, $h_{l,\max}$ is its maximum allowable level, and $\lambda_{\text{of}} \in (0, 1)$ is a configurable threshold that defines the critical level for triggering the penalty. The penalty increases linearly beyond this threshold, encouraging the agent to avoid risky overfill conditions before they become critical.

Limitations. The main limitations of `WDSEnv` stem from the underlying simulator. First, from a modeling perspective, the environment does not allow for fine-grained control of pumps beyond simple on/off status, since features such as variable speed control are not supported by either Epynet or Epanet.

Second, running experiments within this environment is computationally expensive, as reported in Table 2. While our experiments are conducted on a relatively small network with a limited number of nodes and links, scaling to larger networks for benchmarking would require significantly more powerful hardware and the use of parallelization strategies. This challenge is inherent in the design of Epanet, which was not originally intended for real-time control or RL applications. Instead, Epanet was primarily developed to simulate the hydraulic behavior of water networks in response to predefined scenarios and configurations, such as in *what-if* analyses.

Given these limitations, there is an opportunity for closer collaboration between the RL and hydraulic communities. A potential direction could involve the development of a lightweight, RL-friendly Python wrapper for Epanet, optimized for efficient, step-wise simulations and scalable training pipelines. Such a tool would significantly facilitate the integration of advanced learning algorithms into water system management tasks.

E Experiment Details

E.1 `DamEnv`

In this section we provide more details on the benchmarking experiments performed on the `DamEnv` environment. In this setting we compare the SkRL [23] version of PPO with some rule-based strategies described in Section 2.1. The parameters used for the PPO results reported in the main paper can be found in Table 9.

Additionally, Figure 9 shows the validation performances during the training of PPO. In particular, the y-axis represents the mean return over 13 year-long episodes, with 95%-confidence intervals. From the figure, we can observe the proper learning and convergence of the approach.

E.2 `ElevatorEnv`

Within the `ElevatorEnv`, we compare custom implementation of tabular RL algorithms, i.e., Q-Learning and SARSA, against several rule-based strategies. Besides the trivial *Random* policy, we wanted to provide also other two intuitive rule-based solutions that could be easily implemented in a physical system. Below, we briefly describe the rational behind such rule-based policies.

- *Random* policy: the agent selects an action uniformly at random, without considering the state of the system.

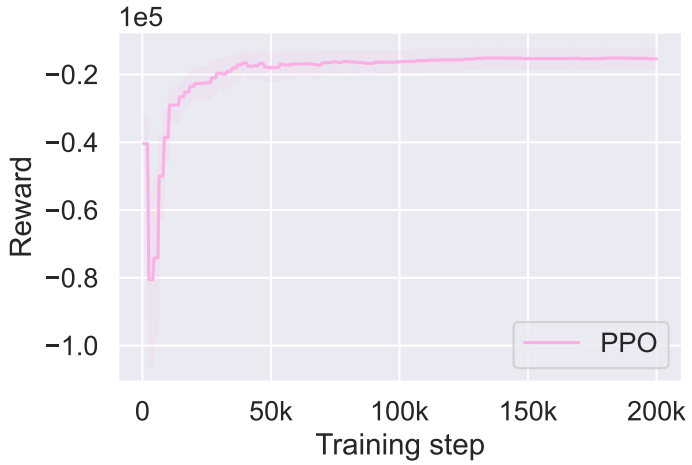


Figure 9: Learning curves of PPO on `DamEnv`.

Table 9: Parameter configuration for PPO on `DamEnv`.

Parameter	**Value**
Training Steps	200k
Batch Size	32
# Epochs	10
Rollouts	2048
Gamma	0.995
Learning Rate	$8e-6$
Policy Network Size	$[16, 16]$
Initial log. std.	-0.5
Normalize obs.	True
Seed	123

- *Longest-first* (LF) policy: the agent prioritizes serving passengers on the floor with the longest waiting queue. Once passengers are picked up, the elevator travels directly to their destination, skipping any intermediate stops.

- *Shortest-first* (SF) policy: the agent prioritizes serving passengers on the floor with the shortest waiting queue. Similarly, it proceeds directly to their destination floor after onboarding passengers without making additional stops.

As evidenced in Figures 2a and 2b, both the LF and SF policies outperform the *Random* one, demonstrating to be reasonable in this setting. However, Q-Learning and SARSA are able to achieve even higher performances both in their vanilla version and with a modest training. We expect even better performances from ad-hoc algorithms specifically designed to address this problem and its challenges.

We provide information on training configurations in Table 10 for both algorithms, as well as their learning curves in Figure 10. In particular, y-axis reports the mean return, i.e., the *global waiting time*, over 30 episodes for each evaluation epoch, with a 95%-confidence interval. Notably, we trained Q-Learning and SARSA for $100,000$ episodes with an early-stopping mechanism that alts the training if after 10 subsequent episodes no improvement grater than a threshold (*tolerance*) occurs.

Table 10: Parameter configuration for Q-Learning and SARSA on `ElevatorEnv`.

Parameter	Q-Learning	SARSA
# Episodes	100k	100k
# Envs	1	1
Gamma	1.0	1.0
Epsilon	1.0	1.0
Epsilon decay	0.99	0.99
Minimum Epsilon	0.05	0.05
Tolerance	0.1	0.1
Stop after # eps. with no improvement	10	10
Seed	42	42

Figure 10: Learning curves of Q-Learning and SARSA on `ElevatorEnv`.

E.3 `MicrogridEnv`

Within the `MicrogridEnv`, we compare the Stable-Baselines3 [18] version of PPO against several rule-based strategies. Below, we briefly describe each of them.

- *Random* policy: the agent selects an action uniformly at random, without considering the current state of the system.
- *Battery-first* (BF) policy: the agent prioritizes the use of the battery, attempting to store or retrieve energy as much as possible. The main grid is used only when the battery is insufficient ($a_t = 1$).
- *Only-market* (OM) policy: the agent exclusively interacts with the main grid, buying and selling energy without utilizing the battery ($a_t = 0$). Note that the battery will still degrade over time due to temporal aging, even though it is not actively used.
- *50-50* policy: the agent consistently splits the net power $P_{N,t}$ equally between the battery and the grid, by selecting $a_t = 0.5$ at every time step.

We trained PPO over 100 episodes on 8 parallel environments. The training span 4-year of data (from 2015 to 2019 included), while testing is done on year 2020, all with a resolution of 1 hour. Parameters of PPO used to obtained the results presented in the main paper are reported in Table 11. The learning curve of PPO is depicted in Figure 11 with on y-axis the is reported the normalized mean episode return over 10 demand profiles, with 95%-confidence interval.

Table 11: Parameter configuration for PPO on `MicrogridEnv`.

Parameter	Value
# Episodes	100
# Envs	8
Policy Network Size	$[64, 32]$
Gamma	0.99
Learning Rate	$5e{-5}$
Batch size	512
# Epochs	10
Rollouts	8912
Initial log. std.	-1
Normalize obs.	True
Seed	42

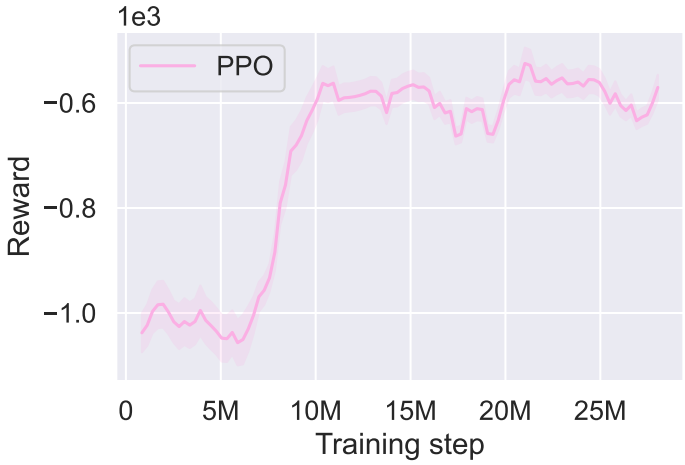


Figure 11: Learning curves of PPO on `MicrogridEnv`.

E.4 `RoboFeederEnv`

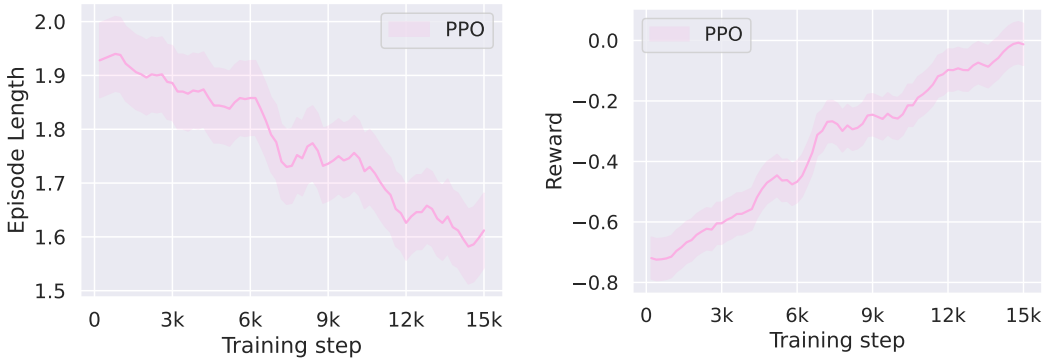
This section gives further details on the training of RL agents on `RoboFeeder` for reproducibility purposes. Table 12 enumerates the entire list of hyperparameters used to train the Stable-Baselines3 [18] implementations of the PPO algorithm. This algorithm is used for both the `picking` and `planning` environments.

E.4.1 `RoboFeeder-picking`

Figures 12a and 12b illustrate the training curve of the PPO algorithm, interacting with the `RoboFeeder-picking-v0` environment. Based on 15k timestamps, using a single object to train the robot to learn to pick the object, it is possible to notice how the distance-based reward helps the agent to learn the right approaching point to the object. The mean reward, over the multiple parallel environments considered, is normalized with respect to the total number of timesteps $T = 2$, to have a value in the range $r \in [-1, 1]$. The training curves of the `robofeeder-picking-v1` are not provided, since they are in line with the ones provided for `v0`. The training curve reported and the related confidence intervals are based on 5 different seeds.

Table 12: Parameters configuration for PPO on `RoboFeederEnv`.

Parameter	**PPO** (`picking`)	**PPO** (`planning`)
# Episodes	15	10
# Envs	20	6
Policy Network Size	$[512, 128]$	$[64, 64]$
Gamma	0.99	0.99
Learning Rate	0.001	0.0003
Log Rate	10	10
Batch Size	128	128
# Steps	15k	10k
Ent. Coeff.	0.01	–
Clip fraction	0.2	0.2
Seed	[0,1,2,56,123]	[0,1,2,56,123]



(a) Mean Episode Length during training.

(b) Mean reward during training.

Figure 12: `RoboFeeder-picking-v0`, PPO training curve of mean episode length and reward over 5 seeds.

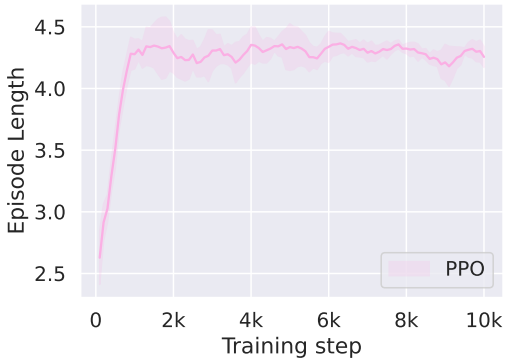
E.4.2 `RoboFeeder-planning`

Figures 13a and 13b present the results of training the PPO algorithm over 10k iterations. The environment is set up with three objects positioned so they can be successfully picked. Despite the limited number of training steps, the learning curve for the mean reward demonstrates the agent's ability to select effective actions. With a total time horizon of $T = 5$, the agent achieves positive rewards, indicating successful object picking. Initially, episodes end prematurely, as shown in the mean episode length plot. However, over time, the agent converges to an optimal time horizon of $T = 4$, which aligns with picking the three available objects, followed by a reset action once the scene is empty. The resulting training curves are based on 5 different seeds.

E.5 `TradingEnv`

This section gives further details on the training of RL agents on `TradingEnv` for reproducibility purposes. Regarding the environment configuration, we considered the base setting, as referred in the Appendix E.5, using 60 delta mid-prices, the timestamp, and the position of the agent. The trading activity is allowed from 8:00 EST to 18:00 EST. Due to the use of Deep RL algorithms as PPO [22] and DQN [13], the timestamp is transformed using cyclic encoding.

It is important to notice that training, validation, and tests were performed using different sets of years on 6 seeds. In particular, the model was trained from 2019 to 2020, validated in 2021, where the best model is selected, and tested in 2022. The hyperparameter configurations chosen for DQN and PPO are reported in Table 13, while the related training curves can be found in Figure 14



(a) Mean Episode Length during training.

(b) Mean reward during training.

Figure 13: `RoboFeeder-planning`, training curve of mean episode reward and episode length over 5 seeds.

and 15. We used the Stable-Baselines3 [18] implementations of the algorithms, specifically the SBX (Stable-Baselines3 in JAX) implementation of PPO and the standard SB3 implementation of DQN.

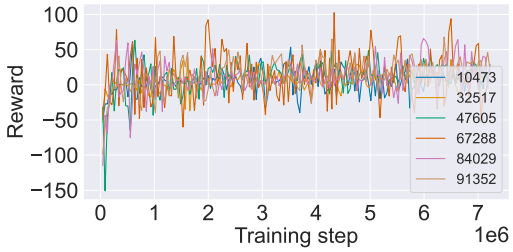
Table 13: Parameter configuration for PPO and DQN on `TradingEnv`.

Parameter	**PPO**	**DQN**
# Episodes	30	30
# Envs	6	6
Policy Network Size	$[512, 512]$	$[512, 512]$
Gamma	0.90	0.90
Learning Rate	0.0001	0.0001
Log Rate	10	-
Batch Size	236	64
# Steps	708	-
Entropy Coeff.	0.0	-
Buffer Size	-	1M
Learning Starts	-	100
Exploration Fraction	-	0.2
Exploration Final Eps.	-	0.05
Polyak Update (tau)	-	1.0
Train Frequency	-	4

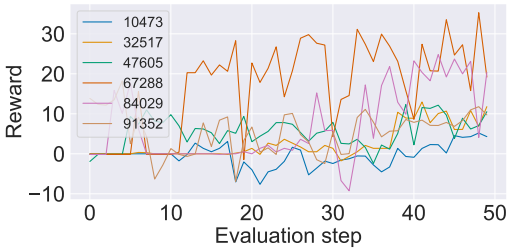
Standard hyperparameters have been used for benchmarking, hence we encourage the use of hyperparameter tuning techniques to better calibrate the model. Indeed, without a proper methodology for hyperparameter tuning and model selection, it is very easy to overfit when manually selecting their values.

In Figure 16, the P&L performances for Training, Validation, and Test years are reported in addition to common passive strategies. The training was performed for approximately 7M steps. Unlike other domains, these baseline strategies are particularly strong. Indeed, people who invest in the stock market typically adopt the Buy&Hold strategy, buying the stock and keeping the position long for years. In this environment, the Buy&Hold policy corresponds to consistently selecting the *Long* action, while the Sell&Hold policy corresponds to consistently selecting the *Short* action.

Figure 17 shows a sample of three policies on the validation set with PPO. We can observe that three different seeds lead to three significantly different policies, underlying the high stochasticity of the environment.

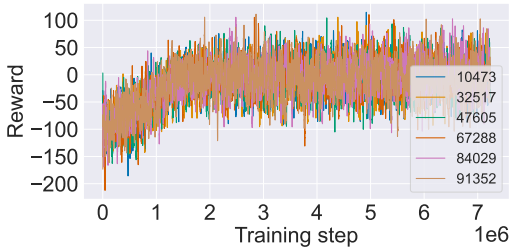


(a) Episode Reward Mean during PPO Training.

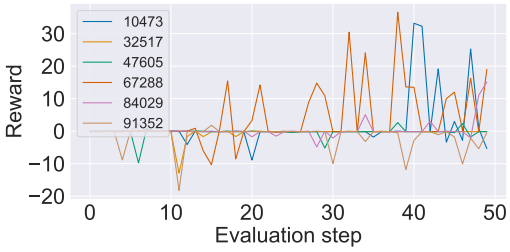


(b) Cum. Reward of PPO on validation data.

Figure 14: Training Curves PPO of the 6 seeds considered for `TradingEnv`, (a) episode reward mean on Training and (b) cumulative reward on the entire validation year.



(a) Episode Reward Mean during DQN Training.



(b) Cum. Reward of DQN on Validation.

Figure 15: Training Curves DQN of the 6 seeds considered for `TradingEnv`, (a) episode reward mean on Training and (b) cumulative reward on the entire validation year.

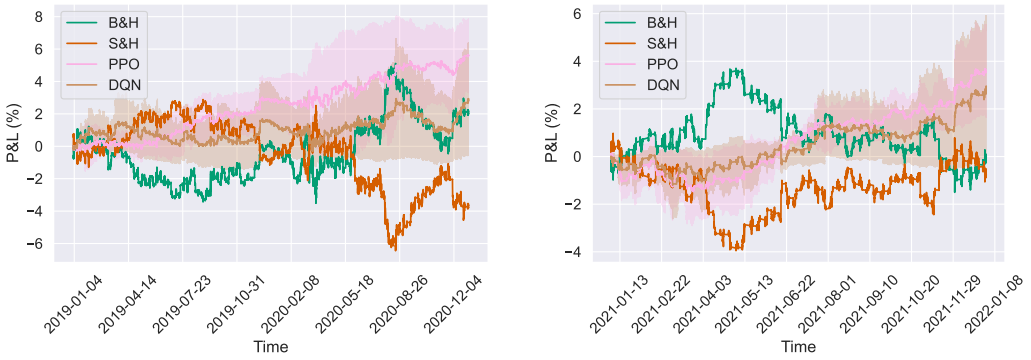
E.6 `WDSEnv`

Within the `WDSEnv`, we compare the Stable-Baselines3 [18] version of DQN against different rule-based strategies. Hereafter, we spend a few words to describe each.

- *Random* policy: the agent selects an action uniformly at random, without considering the state of the system;
- *P78* policy: the agent plays a fixed action $a_t = 2$, keeping the pump P78 always open and the pump P79 always closed;
- *P79* policy: conversely from the previous policy, the agent plays a fixed action $a_t = 1$, keeping the pump P79 always open and the pump P78 always closed;
- *Default* policy: the agent plays a strategy defined in the *[CONTROLS]* section of the *.inp* file. This represents an expert policy adopted by default within Epanet. In this case, the strategy requires the following actions:
  - with l being tank *T41*, open P78 if $h_{l,t} < 5.0$ and close P78 if $h_{l,t} > 8.0$;
  - with l being tank *T42*, open P79 if $h_{l,t} < 5.0$ and close P79 if $h_{l,t} > 8.0$.

In this environment, we adopt the DQN algorithm, which is well-suited to the setting due to the continuity of the observation space and the finite nature of the action space. Training is carried out over episodes of one-week duration, with a control time step of one hour. Each episode presents the agent with varying demand profiles, sampled according to the occurrence probabilities reported in Table 8. Experimental results show that DQN consistently outperforms all rule-based strategies, as it learns to execute more conservative control actions that effectively reduce the risk of tank overflow, while still maintaining a high DSR.

The full training configuration for DQN is detailed in Table 14, while the convergence of the algorithm is illustrated in Figure 18. The y-axis in the figure reports the mean episode return over 10 evaluation episodes, with a 95% confidence interval.



(a) Training Years. (b) Validation Years.

Figure 16: Performances of PPO and DQN w.r.t. common baselines (i.e., passive strategies) on Training (2019-2020) and Validation (2021) for `TradingEnv`. Confidence intervals obtained with 6 seeds.

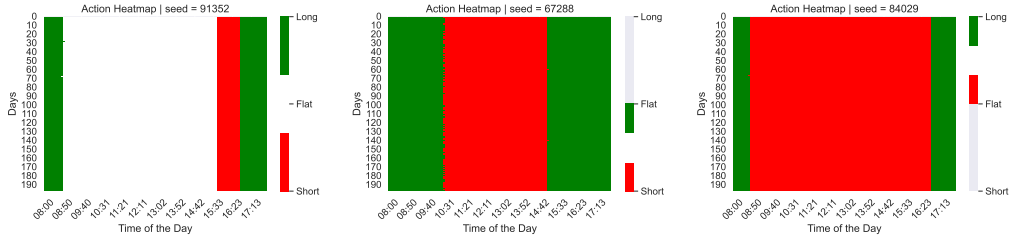


Figure 17: Policy learned by PPO on 3 different seeds in the `TradingEnv`. The x-axis and the y-axis report the time of the day and the different days, respectively. Noticeably, three different policies are learned depending on the seed.

Table 14: Parameter configuration for DQN on `WDSEnv`.

Parameter	**Value**
# Episodes	100
# Envs	8
Policy Network Size	[64, 64]
Gamma	0.99
Learning Rate	0.001
Batch Size	32
Buffer Size	1M
Learning Starts	100
Exploration Fraction	0.1
Exploration Final Eps.	0.05
Polyak update (tau)	1.0
Train Frequency	4
Seed	42

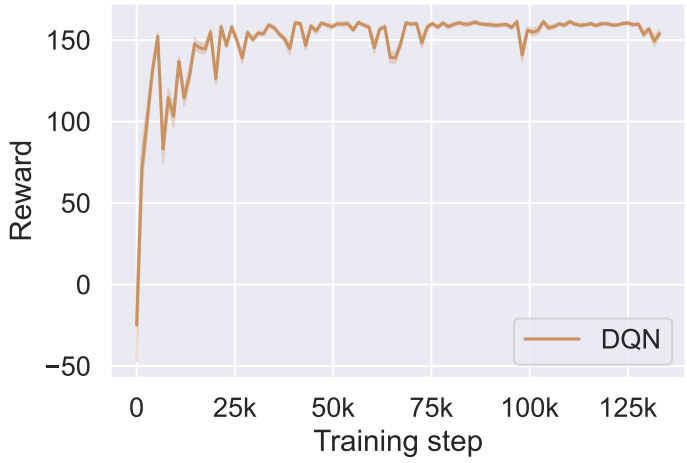


Figure 18: Learning curves of DQN on `WDSEnv`.