Expert Systems With Applications 224 (2023) 119883

Contents lists available at ScienceDirect

Expert Systems With Applications

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

ARLO: A framework for Automated Reinforcement Learning

Marco Mussi [a,*], Davide Lombarda [b], Alberto Maria Metelli [a], Francesco Trovó [a], Marcello Restelli [a]

[a] *Politecnico di Milano, Milan, Italy*
[b] *ML cube, Milan, Italy*

ARTICLE INFO

Keywords:
AutoRL
Automated Reinforcement Learning

ABSTRACT

Automated Reinforcement Learning (AutoRL) is a relatively new area of research that is gaining increasing attention. The objective of AutoRL consists in easing the employment of Reinforcement Learning (RL) techniques for the broader public by alleviating some of its main challenges, including data collection, algorithm selection, and hyper-parameter tuning. In this work, we propose a general and flexible framework, namely ARLO: Automated Reinforcement Learning Optimizer, to construct automated pipelines for AutoRL. Based on this, we propose a pipeline for offline and one for online RL, discussing the components, interaction, and highlighting the difference between the two settings. Furthermore, we provide a Python implementation of such pipelines, released as an open-source library. Our implementation is tested on an illustrative LQG domain and on classic MuJoCo environments, showing the ability to reach competitive performances requiring limited human intervention. We also showcase the full pipeline on a realistic dam environment, automatically performing the feature selection and the model generation tasks.

1. Introduction

Reinforcement Learning (RL, Sutton & Barto, 2018) has recently achieved successful results in solving several complex control problems, including autonomous driving (Wang, Jia, & Weng, 2018), robot manipulators (Nguyen & La, 2019), and finance (Zhang, Zohren, & Roberts, 2020). These outstanding achievements are rooted in the employment of powerful training algorithms combined with complex model representations, such as deep neural networks (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017). Unfortunately, empirical experience suggests that this class of approaches heavily depends on fine-tuning, where an inaccurate choice of the hyper-parameters makes the difference between learning the optimal policy and not learning at all (Buşoniu, de Bruin, Tolić, Kober, & Palunko, 2018). This represents an indubitable limitation, making this powerful tool not immediately usable by non-expert users. While this scenario is common even in general Machine Learning (ML, Bishop & Nasrabadi, 2006), the inherent complexity of RL, due to the sequential nature of the problem, exacerbates this issue even more.

The research effort toward the democratization of ML has reached a mature level of development for supervised learning. Indeed, several Automated Machine Learning (AutoML) frameworks and the corresponding libraries have been developed and tested, such as the ones proposed by Feurer, Eggensperger, Falkner, Lindauer, and Hutter (2020), Feurer et al. (2015), LeDell and Poirier (2020), Olson, Bartley, Urbanowicz, and Moore (2016). AutoML is intended to automate the whole ML pipeline, starting from the preliminary operations on the data and ending with the trained and evaluated final model. This way, the complete ML process can be regarded, by the non-expert user, as a black box, abstracting from the unnecessary details and favoring the adoption of ML as a production tool. For a detailed review of the currently available AutoML frameworks, we refer the reader to the recent survey by He, Zhao, and Chu (2021). Conversely, RL is currently far from being a tool usable by a non-expert user since a complete and reliable Automated Reinforcement Learning (AutoRL) pipeline is currently missing. Indeed, this automation gap between RL and supervised learning is even more severe from a theoretical perspective since, to the best of our knowledge, a general and flexible notion of AutoRL pipeline has not been formalized yet.

Recently, a surge of scientific works in the RL field (Afshar, Zhang, Vanschoren, & Kaymak, 2022; Parker-Holder et al., 2022) attempted to tackle either specific stages of the RL pipeline *individually* (e.g., feature construction, policy generation), or focus on specific application scenarios. While providing a vast analysis of the available approaches for every single stage, they review the state-of-the-art to solve single

* Corresponding author.
E-mail addresses: marco.mussi@polimi.it (M. Mussi), davide.lombarda@mlcube.com (D. Lombarda), albertomaria.metelli@polimi.it (A.M. Metelli), francesco1.trovo@polimi.it (F. Trovó), marcello.restelli@polimi.it (M. Restelli).

https://doi.org/10.1016/j.eswa.2023.119883
Received 1 September 2022; Received in revised form 5 March 2023; Accepted 13 March 2023
Available online 28 March 2023

tasks individually, do not propose a full pipeline, and do not study the peculiarities characterizing the *interaction* between such stages. On the other hand, a naïve adaptation of the existing automated pipelines designed for AutoML to the RL setting is not a viable approach since they fail to capture the unique characteristics of RL related to the presence of an interacting environment and the sequential nature of the learning problem.

Contributions. In this paper, we make a step toward the formalization of an AutoRL framework. The contributions of this work can be synthesized as follows.

- We propose a general and flexible formalization of a *pipeline* for AutoRL. Grounding on such a definition, we instantiate it for two different scenarios: *offline* and *online* RL.[1] The former assumes that the RL process is carried out based on a fixed batch of data. The latter takes into account the availability of an interactive environment.
- We describe the individual *stages* of the two pipelines and their respective characteristics, highlighting the *interactions* between them and focusing on their inputs and outputs. Furthermore, we discuss the corresponding *units*, i.e., possible implementations of stages, and introduce a general approach to select the best-tuned unit in a finite set.
- We provide an implementation of the framework in an open-source Python library, called ARLO.[2] The library contains the implementation of all the stages, the two RL pipelines, and the needed tools to run, optimize, and evaluate the pipelines.
- Finally, we test the implementation on the LQG and MuJoCo environments, showing the ability to reach optimal performances without requiring any manual adjustment by humans. At last, we provide an experiment on a realistic dam environment with a pipeline composed of the data generation, feature selection, policy generation, and policy evaluation stages.

Given the wide variety of RL problems and solutions, we restrict our formalization to *stationary* and *fully observable* environments. We leave the extension to complex settings (e.g., multi-objective, multi-agent, lifelong) as future work.

Outline. The paper is structured as follows. In Section 2, we present the fundamental notions of Markov Decision Processes and the basics of RL. In Section 3, we introduce a general notion of pipeline, stage, and unit. In Section 4, we present the online and offline pipelines for RL. In Section 5, we describe the details of the components included in the two pipelines. In Section 6, we report the results of the tests performed on standard benchmarks and on a realistic environment. In Section 7, we highlight the conclusions of our works, and we propose future research lines.

2. Preliminaries

A Markov Decision Process (MDP, Puterman, 2014) is defined as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu_0)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $P(s'|s,a)$ is the state transition model, specifying the probability to land in state s' starting from state s and performing action a, $R(s,a)$ is the reward function, defining the expected reward when the agent is in state s and performs action a, $\gamma \in [0,1]$ is the discount factor, and $\mu_0(s)$ is the initial-state distribution. The agent's behavior is defined in terms of a policy $\pi(a|s)$ representing the probability of performing action a in state s.

Interaction protocol. The initial state is sampled from the initial-state distribution $s_0 \sim \mu_0$, the agent selects an action based on its policy $a_0 \sim \pi(\cdot|s)$, the environment provides the agent with the reward $R(s_0, a_0)$, and the state evolves according to the transition model $s_1 \sim P(\cdot|s_0, a_0)$. The process is repeated for T steps, where $T \in \mathbb{N} \cup \{+\infty\}$ is the (possibly infinite) horizon.

Objective. The goal of RL consists in learning an *optimal* policy $\pi(a|s)$, i.e., a policy maximizing the expected discounted sum of the rewards, a.k.a. the *expected return*:

$$J(\pi) := \mathbb{E}^{\pi}\left[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)\right], \qquad (1)$$

where the expectation $\mathbb{E}^{\pi}[\cdot]$ is computed w.r.t. the randomness of environment and of the policy (Sutton & Barto, 2018).

Environments and datasets. We introduce the notion of *environment* and *dataset*. Formally, an environment $\mathcal{E}$ is a device to interact with the underlying MDP, that, given a state s_t and an action a_t, it provides the next state $s'_t \sim P(\cdot|s_t, a_t)$ and the reward $r_t = R(s_t, a_t)$. An environment is a *generative model* if it allows to freely choose the state s_t at each step, or a *forward model* if, instead, we can perform steps in the MDP ($s_{t+1} = s'_t$) or start again sampling s_t from the initial-state distribution μ_0. A dataset $\mathcal{D} := \{\tau_i\}_{i=1}^n$ is a set of trajectories τ_i, where each *trajectory* is a sequence $\tau_i = (s_i^0, a_i^0, r_i^1, \dots, s_i^{T_i-1}, a_i^{T_i-1}, r_i^{T_i}, s_i^{T_i})$ and T_i is the length of the trajectory.

Online vs. offline RL. We distinguish between two main groups of RL algorithms: *online* and *offline* RL. The online RL algorithms (Sutton & Barto, 2018) aim at learning a policy π by directly interacting with an environment $\mathcal{E}$. Typically they employ the last available policy to collect data and leverage the experience to improve it. Conversely, the offline RL paradigm (Levine, Kumar, Tucker, & Fu, 2020) consists in carrying out the policy learning on a dataset $\mathcal{D}$ previously collected.[3] The ability to learn a (near-)optimal policy heavily depends on the exploration properties of the dataset $\mathcal{D}$.

3. Framework

In this section, we present the abstract formalization of the proposed AutoRL pipeline, detailing the notions of *pipeline*, *stage*, and *unit*.

3.1. Stages and pipelines

A stage ψ represents a single component of the pipeline with a specific *purpose*. For instance, the portion of the pipeline in charge of performing feature engineering is regarded as a stage. A stage ψ interacts with the other stages of the pipeline by means of an *interface*, defining its inputs and outputs. We denote a stage's inputs with $\mathrm{In}(\psi)$ and its outputs with $\mathrm{Out}(\psi)$. A pipeline is a sequence of $m \in \mathbb{N}$ stages $\Psi = (\psi_1, \dots, \psi_m)$. The possibility of staking specific stages in a sequence depends, in general, on problem-dependent constraints.

3.2. Units

A unit constitutes the actual *implementation* of the stages corresponding to algorithms that are in charge of generating the output required by the corresponding stage.[4] We define three relevant types of units: *fixed*, *tunable*, and *automatic*.

[1] The reader might be tempted to address the offline RL setting with AutoML, given the fixed available dataset and, thus, the similarity with supervised learning. We stress that this choice is inappropriate as the peculiarities of RL are still crucial, especially the sequential properties of the problem.

[2] The library is available at https://github.com/arlo-lib/ARLO.

[3] Even in this case, we may have an environment $\mathcal{E}$ to test the performance of the learned policy. Commonly, it is a less costly version, e.g., in terms of computational or real costs, of the environment where the final policy will be applied.

[4] From a software engineering perspective, a stage is an abstract class, while a unit a concrete class.

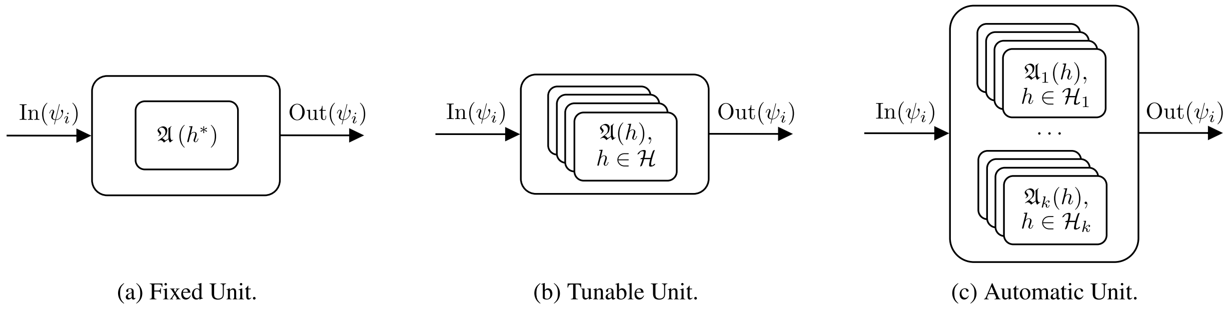

In(ψ_i)
$\mathfrak{A}(h^*)$
Out(ψ_i)
(a) Fixed Unit.
$\mathfrak{A}(h)$, $h \in \mathcal{H}$
(b) Tunable Unit.
$\mathfrak{A}_1(h)$, $h \in \mathcal{H}_1$
...
$\mathfrak{A}_k(h)$, $h \in \mathcal{H}_k$
(c) Automatic Unit.

Fig. 1. The three types of units.

Fixed unit. A fixed unit (Fig. 1(a)) corresponds to an algorithm $\psi = \mathfrak{A}(h)$, where $\mathfrak{A}(h)$ denotes algorithm $\mathfrak{A}$ that generates the stage output, instanced with hyper-parameters $h \in \mathcal{H}$ selected from an hyper-parameter set $\mathcal{H}$.

Tunable unit. A tunable unit (Fig. 1(b)) is described by a tuple $\psi = (\mathfrak{A}, \mathcal{H}, \mathfrak{T}, \ell)$ where $\mathfrak{A}(\cdot)$ is an algorithm, $\mathcal{H}$ is a *hyper-parameters set*, $\mathfrak{T}$ is a *tuner* (e.g., genetic algorithm, particle swarm, Bayesian optimizer), and $\ell(\mathfrak{A}, h) \in \mathbb{R}$ is a tuning *performance index* mapping an algorithm $\mathfrak{A}(\cdot)$ and hyper-parameters $h \in \mathcal{H}$ pair to a real number. The *tuning optimization problem* can be formulated as finding the hyper-parameters $h^* \in \mathcal{H}$ maximizing the performance index ℓ. Formally:

$$h^* \in \arg\max_{h \in \mathcal{H}} \ell(\mathfrak{A}, h).$$

This optimization is addressed by the tuner $\mathfrak{T}$. When the stage corresponding to the tunable unit is executed, it reduces to the fixed unit $\mathfrak{A}(h^*)$, and, subsequently, it generates the block outputs.

Automatic unit. An *automatic unit* (Fig. 1(c)) is a set of tunable units paired with a performance index, i.e., $\psi = (\{\psi_j\}_{j=1}^k, \ell)$, where $\psi_j = (\mathfrak{A}_j, \mathcal{H}_j, \mathfrak{T}_j, \ell_j)$, for $j \in \{1, \dots, k\}$, and $\ell(\mathfrak{A}, h) \in \mathbb{R}$ is a performance index for algorithm $\mathfrak{A}$ with hyper-parameters h. The goal of an automatic unit consists in selecting the best-tuned algorithm among the available ones by ranking them based on the additional performance index ℓ. We define the *automatic optimization problem* as follows:

$$j^* \in \arg\max_{j \in \{1,\dots,k\}} \ell(\mathfrak{A}_j, h_j^*),$$

where:

$$h_j^* \in \arg\max_{h \in \mathcal{H}_j} \ell_j(\mathfrak{A}_j, h), \quad j \in \{1, \dots, k\}.$$

When the stage corresponding to the automatic unit is executed, it reduces the automatic unit to a fixed one $\mathfrak{A}_{j^*}(h_{j^*}^*)$, and, subsequently, it generates the corresponding output.

In the AutoML community, the problem of jointly finding the best algorithm and its related hyper-parameter configuration is also called CASH (Combined Algorithm Selection and Hyper-parameter Optimization Problem, Thornton, Hutter, Hoos, & Leyton-Brown, 2013).

Intuitively, a fixed unit is a human hand-crafted unit in which an algorithm is selected, and the related hyper-parameters are specified. No automatic operations nor evaluations are performed here. Instead, in a tunable unit, the algorithm is specified, but the task of finding the best hyper-parameter configuration is left to the pipeline. Finally, in an automatic unit, both the choice of the best algorithm and the best hyper-parameter configuration are left to the pipeline.

4. AutoRL pipelines

In this section, we present the main methodological contribution of the paper, discussing the two AutoRL pipelines: *online* and *offline*. We focus on how to build these pipelines by describing the stages' interactions. The detailed description of each individual stage is reported in Section 5. A graphical representation of the pipelines is provided in Fig. 2.

Online pipeline. The *Online AutoRL Pipeline* (Fig. 2(a)) takes as input an environment $\mathcal{E}$ that is fed to the `Feature Engineering` stage, which modifies its state–action representations and the reward to facilitate the learning performed in the next stages. It outputs a transformed environment $\mathcal{E}'$, based on the features created in this stage. Subsequently, the environment $\mathcal{E}'$ is used to learn an estimate $\hat{\pi}^*$ of the optimal policy through the `Policy Generation`. Finally, the `Policy Evaluation` phase provides an estimate of the performance $\eta(\hat{\pi}^*)$, based on a performance index η.

Offline pipeline. In the *Offline AutoRL Pipeline* (Fig. 2(b)), differently from the online one, two additional preliminary stages are included: `Data Generation` and `Data Preparation`. If an environment $\mathcal{E}$ is provided as input, the `Data Generation` stage creates a dataset $\mathcal{D}$. This stage is omitted if a dataset $\mathcal{D}$ is already available, e.g., in the case it comes from a real process. In such a case, the environment $\mathcal{E}$ is employed for the evaluation of the policy performance only. The `Data Preparation` stage modifies the dataset $\mathcal{D}$, by applying corrections over the individual instances (i.e., the rows of the dataset) obtaining $\mathcal{D}'$. Then, the environment $\mathcal{E}$ and dataset $\mathcal{D}'$ pass through the `Feature Engineering` stage, which, similarly to its online counterpart, generates a dataset $\mathcal{D}''$ and an environment $\mathcal{E}'$ with transformed states, actions, and reward. After that, the dataset $\mathcal{D}''$ is used for learning an estimate of the optimal policy $\hat{\pi}^*$ through the `Policy Generation` stage. Differently from the online one, this stage uses the dataset $\mathcal{D}''$, while the environment $\mathcal{E}'$ is employed for estimating $\eta(\hat{\pi}^*)$ in the `Policy Evaluation` stage.

Before we start discussing the stages, we remark a notable difference between our framework compared to AutoML. By looking at Fig. 2 only, we may recognize similarities between our stages with the ones of an AutoML pipeline (e.g., `Feature Engineering`). While the *names* of the blocks are the same, the *implementation* of the blocks are different, as AutoML blocks are clearly non-compatible with an RL pipeline. For instance, feature engineering cannot be performed in RL as it is done for supervised learning or several learning algorithms consider samples that are i.i.d. in AutoML, while in AutoRL, we must consider that this assumption does not hold in general. Furthermore, the *interfaces* of our blocks (i.e., stage inputs and outputs) include the presence of an environment, which is an element not explicitly considered in AutoML pipelines.

5. Stages and units

We now provide examples of units for each of the stages, highlighting the differences between the online and offline pipelines. For each stage, we define its goal, performance index for tunable or automatic units, and implementation selected from the state-of-the-art approaches.

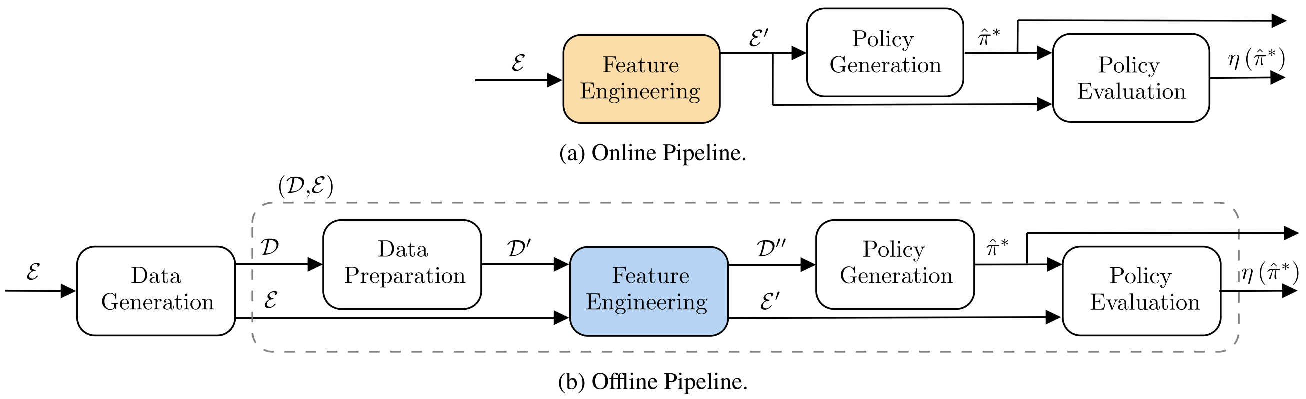


Fig. 2. The Online (a) and Offline (b) AutoRL Pipelines.

5.1. Data generation

The `Data Generation` stage takes as input an environment $\mathcal{E}$ and returns the unaltered environment $\mathcal{E}$ and a dataset $\mathcal{D}$ generated by interacting with the environment. The goal of this stage is to create a dataset that is retrieved by exploring the state space in the most effective way. Based on the type of environment, i.e., generative or forward model, the resulting dataset is made of transitions or trajectories.

In principle, this stage should output a dataset as "informative" as possible, i.e., that represents exhaustively the corresponding environment. As performance index for evaluating the quality of a `Data Generation` unit, we adopt the *entropy* of the state–action visitation distribution $d_\pi(s,a)$ generated by the policy $\pi(a|s)$, that is proportional to:[5]

$$-\int_{s\in\mathcal{S}}\int_{a\in\mathcal{A}} d_\pi(s,a)\log d_\pi(s,a)\,\mathrm{d}a\,\mathrm{d}s.$$

A straightforward implementation of `Data Generation` consists in collecting data with the random uniform policy. However, this approach is not guaranteed to explore the state space effectively (Endrawis, Leibovich, Jacob, Novik, & Tamar, 2021; Mutti, Pratissoli, & Restelli, 2021). In the pipeline, we consider the state-of-the-art solutions proposed by Pathak, Gandhi, and Gupta (2019), and Mutti et al. (2021). The former employs Proximal Policy Optimization (PPO, Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) using the estimated variance of the MDP dynamics as reward, as a proxy for the entropy. Instead, the latter provides a novel policy search algorithm maximizing a K-nearest neighbors-based estimate of the state distribution entropy.

5.2. Data preparation

This phase uses a dataset $\mathcal{D}$, coming either from a real-world environment or generated in the `Data Generation` stage, and returns a dataset $\mathcal{D}'$ with the same state–action features and reward, but with a possibly different number of entries. The goal of this phase is to optimize an existing dataset to be processed better in subsequent stages. `Data Preparation` includes data augmentation, data imputation, and data scaling. Moreover, it can embed further domain-specific sub-stages (e.g., for images, and audio data) and/or consistency checks (e.g., filling missing values).

No single automatic unit is deemed adoptable due to the difficulty of defining a general enough performance index for this stage. However, domain-specific performance indexes are available, e.g., for the data imputation sub-stages, we may rely on the indexes defined by Jadhav, Pramod, and Ramanathan (2019).

[5] In this stage, we rely on the Particle Based Entropy estimation developed by Singh, Misra, Hnizdo, Fedorowicz, and Demchuk (2003).

Possible implementations of this stage include the techniques for classical ML preprocessing, such as imputation from a dataset of trajectories via KNN imputation or Bayesian Multiple Imputation (Lizotte, Gunter, Laber, & Murphy, 2008). Moreover, for pixel-based observations (e.g., the Gym Atari environments) data augmentation techniques, e.g., cropping, reflection, and scaling, were employed in Ye, Khalifa, Bontrager, and Togelius (2020). Other approaches viable for feature-based representations are presented in Laskin et al. (2020), where experiments on the OpenAI Procgen Benchmark and on the MuJoCo environments are considered.

5.3. Feature engineering

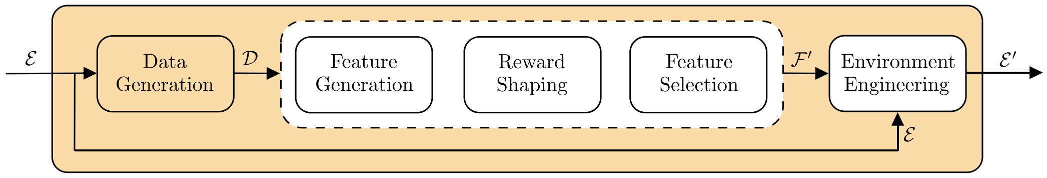
The `Feature Engineering` stage displays significant differences between online and offline pipelines (Fig. 3). Offline pipelines (Fig. 3(b)) take as input an environment $\mathcal{E}$ and a dataset $\mathcal{D}'$ and return a feature-adjusted environment $\mathcal{E}'$ and dataset $\mathcal{D}''$. Conversely, online pipelines (Fig. 3(a)) take as input an environment $\mathcal{E}$ and return a feature-adjusted environment $\mathcal{E}'$. In both cases, this stage requires an internal dataset for feature engineering that, for the online case, has to be generated.

The core task of this stage is to select and generate a set of features that properly model the state–action space of the problem and perform reward-shaping actions to facilitate the following learning phase. `Feature Engineering` stage includes one or more of the following *sub-stages*:

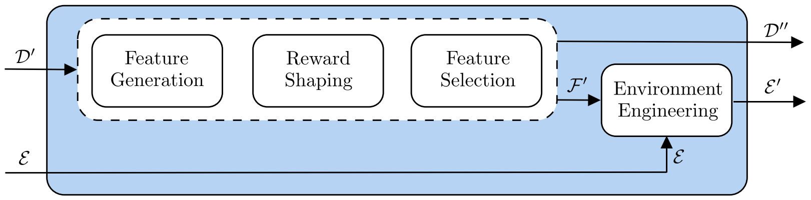
- `Feature Generation`, in charge of creating new features. This sub-stage makes use of techniques such as radial basis functions, tile coding, coarse coding (Sutton & Barto, 2018) or Nyström Map (Williams & Seeger, 2000).
- `Feature Selection`, aimed at selecting a meaningful subset of features, either to reduce the computation requirements or to regularize the following policy learning phase. Viable options are Mutual Information-based selection (Beraha, Metelli, Papini, Tirinzoni, & Restelli, 2019), correlation-based filtering methods, and tree-based variable selection (Castelletti, Galelli, Restelli, & Soncini-Sessa, 2011).
- `Reward Shaping`, performing specific transformations on the reward function, possibly preserving the optimal policy, to speed up the convergence of an RL algorithm (Ng, Harada, & Russell, 1999). For instance, in presence of sparse reward functions, reward shaping can be regarded as a form of *curriculum learning* (Portelas, Colas, Weng, Hofmann, & Oudeyer, 2020).

These sub-stages return a transformation that is applied to the environment through the `Environment Engineering` stage. In the offline case, the same transformation is applied to the dataset, while in the online case, the internal dataset is disregarded.

We consider as a performance index for the complete feature engineering stage the mutual information between the current state–action



(a) Feature Engineering stage in Online Pipeline.



(b) Feature Engineering stage in Offline Pipeline.

Fig. 3. The offline and online `Feature Engineering` stages.

pair (s, a) features and the next-state reward (s', r) features (Gao, Kannan, Oh, & Viswanath, 2017; Kraskov, Stögbauer, & Grassberger, 2004) regularized, e.g., by the number of selected features.[6]

5.4. Policy generation

The `Policy Generation` stage is in charge of the training phase of the RL learning algorithm. More specifically, it takes as input an environment $\mathcal{E}'$ or a dataset $\mathcal{D}''$, in the online and offline RL pipelines, respectively, to output an estimate $\hat{\pi}^*$ of the optimal policy.

Among the most common choices of performance indexes for this stage, we mention the *expected return*, i.e., the expected discounted sum of the rewards, the *average reward*, i.e., the long-term expected average reward, and the *total reward* i.e., expected cumulative sum of the rewards (in the case the environment is episodic, Puterman, 2014). For specific applications, e.g., risk-averse setting, one may adopt the mean–variance, mean-volatility, and CVaR (Bisi, Sabbioni, Vittori, Papini, & Restelli, 2021; Pratt, 1978).

Many works deal with hyper-parameter optimization for RL algorithms. In Franke, Köhler, Biedenkapp, and Hutter (2021) a framework based on Population-Based Training (PBT, Jaderberg et al., 2017) is proposed to tune off-policy RL algorithms. In Parker-Holder, Nguyen, Desai, and Roberts (2021), a new time-varying bandit algorithm was presented for tuning RL algorithms. Hyper-parameter tuning is a widely researched topic, and the techniques developed by ML algorithms can be used for RL algorithms as well. Nevertheless, the sample inefficiency of tuning techniques is a common problem, not unique to RL. Another issue is the sensitivity to hyper-parameters configurations, which increases the difficulty of benchmarking tuning algorithms due to the difficulty of obtaining reproducible results. Further methods were proposed by Falkner, Klein, and Hutter (2018), Lee, Laskin, Srinivas, and Abbeel (2021), Saphal, Ravindran, Mudigere, Avancha, and Kaul (2021), Team et al. (2021), Zhang et al. (2021).

The specific implementation of the `Policy Generation` stage depends on the selected RL algorithm. For offline pipelines, we mention, among the others, Least Squares Policy Iteration (LSPI, Lagoudakis & Parr, 2003), Fitted Q-Iteration (FQI, Ernst, Geurts, & Wehenkel, 2005). For online pipelines, a large surge of RL algorithms has been developed in recent years. We mention, among the most popular ones, Trust Region Policy Optimization (TRPO, Schulman, Levine, Abbeel, Jordan, & Moritz, 2015), Deep Q-Networks (DQN, Schaul, Quan, Antonoglou, & Silver, 2016), Deep Deterministic Policy Gradient (DDPG, Lillicrap et al., 2016), Proximal Policy Optimization (PPO, Schulman et al., 2017), and Soft Actor Critic (SAC, Haarnoja, Zhou, Abbeel, & Levine, 2018).

5.5. Policy evaluation

The `Policy Evaluation` stage takes as input the policy $\hat{\pi}^*$ produced by the `Policy Generation` phase and an environment $\mathcal{E}'$, and produces as output an estimation of a performance index $\eta(\hat{\pi}^*)$.

Regarding the performance index used in this stage, the options are the same as the ones we mentioned for `Policy Generation`. Notice that the performance index chosen in this stage may differ from the one of the `Policy Generation` one. For instance, it is a common practice to train RL algorithms using a discounted objective and evaluate the resulting policies using an undiscounted one (Duan, Chen, Houthooft, Schulman, & Abbeel, 2016). Notice that, due to the nature of the task, only fixed units are used in this stage.

6. Experimental results

In this section, we employ the Python implementation of ARLO on three RL problems. In addition to the presented stages, the library allows creating newly defined stages, if needed, and a set of analysis tools. The implementation of the framework is available at https://github.com/arlo-lib/ARLO. The implemented methods are reported in Appendix A. The `Policy Generation` stages have been integrated with the MushroomRL (D'Eramo, Tateo, Bonarini, Restelli, & Peters, 2021) library.[7] The optimization of the tunable units has been performed using a genetic algorithm as described in Appendix B.

In Sections 6.1 and 6.2, we present the results of our online pipelines whose `Policy Generation` stages contain tunable (Sections 6.1.1, 6.2) and automatic (Section 6.1.2) units to select the best hyper-parameters and algorithm over two simulated problems. In Section 6.3, we apply an offline pipeline including tunable `Feature Engineering` and fixed (Section 6.3.1) or automatic (Section 6.3.2) `Policy Generation` stages on a realistic dam control problem. The experimental details are reported in Appendix B.

6.1. Linear quadratic Gaussian regulator

In this experiment, we solve a Linear–Quadratic Gaussian Regulator (LQG, Dorato, Cerone, & Abdallah, 1994) with state dynamics evolving as $s_{t+1} = As_t + Ba_t + \sigma$, where s_t is the state at time t, a_t is the action at time t, A is the state dynamic matrix, B is the action

[6] For instance, one may use the ratio between the mutual information and the number of selected features.

[7] The ARLO library includes an easy procedure to integrate algorithms coming from other RL libraries.

Table 1
Results achieved tuning SAC hyper-parameters on an LQG environment (100 runs, mean ± std, higher is better).

Method	Default	Tuned
Van Dooren (1981)	−7.2 (4.9)	
1st Seed	−59.0 (24.0)	−8.6 (4.7)
2nd Seed	−67.4 (16.1)	−8.2 (5.1)
3rd Seed	−52.4 (12.5)	−8.7 (4.7)

Table 2
Comparison of the results achieved by different tuners in selecting the SAC hyper-parameters on an LQG environment (100 runs, mean ± std, higher is better).

Tuner	Reward
Optimum (Van Dooren, 1981)	−7.2 (4.9)
Genetic Algorithm	−8.5 (4.8)
Bayesian Optimization (TPE)	−8.6 (4.7)
Random Search	−9.5 (4.9)

dynamic matrix, and σ is Gaussian white noise. The reward function is $r_{t+1} = -s_t^T Q s_t - a_t^T R a_t$, where Q and R are the state and action cost weight matrices, respectively.[8] The discount factor is equal to $\gamma = 0.9$, and the time horizon is $T = 15$.

6.1.1. Tunable policy generation

In this experiment, we employ the *Soft-Actor Critic* (SAC, Haarnoja et al., 2018) algorithm. To tune its hyper-parameters, we create an online RL pipeline, using the expected return (Eq. (1)) as the performance index and a genetic algorithm (like in Sehgal, La, Louis, & Nguyen, 2019) as the tuning algorithm. Then, we compare the performance of different tuners, i.e., Bayesian Optimization, Random Search, and the genetic algorithm mentioned above, to assert which one performs better. The results are obtained after 50 generations of the genetic algorithm, each using a population of 20 agents.

Results. We compare the results provided by the ARLO framework with the optimal solution (Van Dooren, 1981). In Table 1, we report the estimated expected return, averaged over 100 episodes (with the standard deviation in brackets), for the default configuration and the corresponding tuned policy over three different seeds. Even if the performance of the tuned algorithms does not match the one of the optimal solution, the *default* hyper-parameter configuration of SAC is notably underperforming (≈ 5 times worse) compared to the tuned configuration (≈ 1.2 times worse). This result suggests that the proposed framework can generate solutions compatible with the optimal one without exploiting specific domain knowledge about the problem. On the other hand, Table 2 presents the performances in terms of estimated expected return for the genetic algorithm, Bayesian optimization, and random search tuner. It is worth noting how, in this scenario, we observe that the genetic tuner reaches the best performance, comparable with the one of the Bayesian optimizer, while, as expected, the random tuner performs the worst.

6.1.2. Automatic policy generation

In this experiment, differently from the previous one in which we used a single tunable unit for the `Policy Generation` stage, we run an automatic `Policy Generation` unit for the Linear Quadratic Gaussian Regulator, in which we tune both Soft Actor Critic (SAC, Haarnoja et al., 2018) and Proximal Policy Optimization (PPO, Schulman et al., 2017). To test the performance, we consider the Discounted Reward (Eq. (1)) in `Policy Evaluation` stage. Fig. 4 represents the pipeline and the topology of the units used in the experiment.

[8] The details about the hyper-parameters configuration space, the tuning procedure, and the compute requirements for the LQG experiment can be found in Appendix B.1.

Table 3
Performance obtained using the automatic `Policy Generation` unit on the Linear Quadratic Gaussian Regulator (100 runs, mean ± std, higher is better).

Method	Empirical expected return
Van Dooren (1981)	−7.2 (4.9)
Best SAC Tuned Configuration	−8.2 (5.1)
Best Automatic Configuration	−7.4 (5.0)

Table 4
Results achieved tuning DDPG hyper-parameters on HalfCheetah-v3 environment (100 runs, mean ± std, higher is better).

Method	Default	Tuned
Islam et al. (2017)	3725.3 (512.8)	
1st Seed	1157.7 (45.6)	3407.2 (952.1)
2nd Seed	850.8 (78.9)	4624.6 (110.9)
3rd Seed	956.2 (34.2)	3076.9 (77.9)

Results. Table 3 shows the results in terms of discounted reward. By tuning the hyper-parameters of two different `Policy Generation` tunable units, the pipeline further improved the results presented in Section 6.1.1, reaching a level of performance in line with the one of the optimal solution by Van Dooren (1981).

6.2. HalfCheetah

In this experiment, we apply the online RL pipeline to the MuJoCo HalfCheetah-v3 environment from OpenAI Gym (Zamora, Lopez, Vilches, & Cordero, 2016).[9] As a learning algorithm for the `Policy Generation`, we employ the *Deep Deterministic Policy Gradient* (DDPG, Lillicrap et al., 2016), whose hyper-parameter tuning is known to be a challenging task (Islam, Henderson, Gomrokchi, & Precup, 2017).[10] The hyper-parameters of DDPG have been tuned using the undiscounted cumulative reward as a performance index and, as a tuner, a genetic algorithm. Moreover, we employ a discount factor $\gamma = 1$ and the time horizon to $T = 1000$.

Results. In Table 4, we report the estimated total reward, averaged over 100 episodes, for the default and tuned configurations over three different seeds (the standard deviation is provided in brackets). The provided performances are in line with the literature ones (Islam et al., 2017) and show that the proposed pipeline provides an automatic way of achieving competitive performance.

In Fig. 5, we report the different hyper-parameters selected during the learning phase by individuals (agents) used in the genetic algorithm optimization procedure throughout the tuning procedure. These results show how some of the parameters have a strong influence on the reward obtained by the agents, i.e., the actor and critic learning rate and the steps per fit (Figs. 5(a), 5(b), and 5(c), respectively), which implies that the value of the parameter concentrates around the optimal value after a few generations of the genetic algorithm. Conversely, those which do not influence the outcome of the optimization procedure, i.e., the steps (Fig. 5(d)), continue to explore the available range until the end of the generations.

6.3. Dam

To showcase the capabilities of our framework, we propose an experiment with a more complex offline RL pipeline that includes

[9] https://gymnasium.farama.org/environments/mujoco/half_cheetah/
[10] The details about the hyper-parameters configuration space, the tuning procedure, and the compute requirements for the HalfCheetah experiment can be found in Appendix B.2.

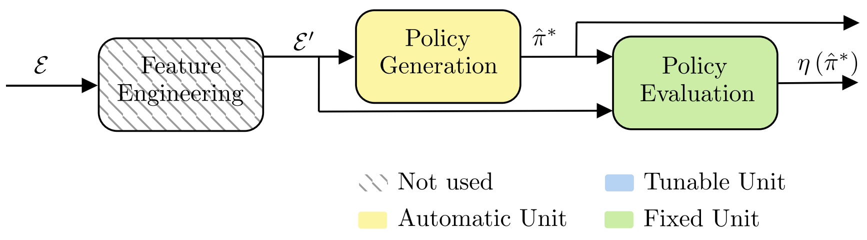


Fig. 4. Types of the units adopted in the online pipeline experiment.

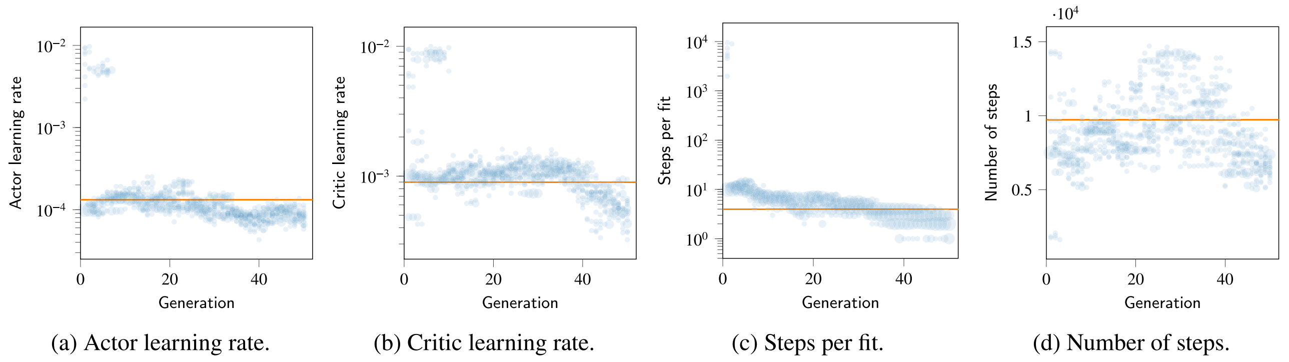


Fig. 5. Values of the hyper-parameters generated by the genetic optimization procedure over the 50 generations. The orange line corresponds to the best-found value.

Table 5
Results achieved tuning the hyper-parameters of a Feature Engineering stage (10 runs, mean ± std, higher is better).

Method	Discounted reward
Baseline	−1649.85 (112.88)
Tuned Configuration	−1224.67 (124.41)

Table 6
Results obtained for the additional experiment over the full offline pipeline (10 runs, mean ± std, higher is better).

Method	Empirical expected return
Pipeline of Section 6.3.1	−1224.67 (124.41)
New configuration	−1047.97 (213.37)

`Data Generation`, `Feature Engineering`, `Policy Generation`, and `Policy Evaluation` stages.[11] The selected environment consists of the control of a water reservoir (dam) that models the dynamics of a real alpine lake (Castelletti et al., 2011). The agent observes the current level of the lake and the sequence of the most recent 30 daily inflows. The actuation consists of the amount of daily water released. The goal of the agent is to trade off between avoiding floods and fulfilling the downstream water demand.

6.3.1. *Feature engineering*

In this experiment, the dataset is generated by a fixed `Data Generation` stage adopting a random uniform policy. The `Feature Engineering` stage performs forward feature selection via mutual information (as presented in Beraha et al., 2019) to identify a subset of the available inflows features. The `Policy Generation` stage is a fixed unit and uses the Fitted Q-Iteration (FQI, Ernst et al., 2005) algorithm. The hyper-parameters of FQI are fixed to a hand-tuned configuration as the one presented by Tirinzoni, Sessa, Pirotta, and Restelli (2018). The objective of this experiment is to show in a realistic environment that tuning the hyper-parameters of a `Feature Engineering` stage is beneficial for the final performance.

Results. In Table 5, we report the estimated expected return averaged over 10 episodes for the baseline configuration (standard deviation in brackets), in which all the features have been considered, and for the tuned configuration, in which only a subset of the features was selected automatically by the pipeline. We observe that the result achieved by the tuned agent significantly outperforms the baseline one, meaning that the feature selection techniques select only the most informative feature for the problem, with beneficial effects on the successive learning phase.

6.3.2. *Complete offline pipeline*

The objective of this experiment is to evaluate the capabilities of using the complete offline pipeline we defined in Section 4 in the Dam experiment described above.

The scheme of the used pipeline and the topology of the different stages is presented in Fig. 6. We use a fixed unit to generate the data, and a tunable one to perform the `Feature Engineering` stage, similarly to what has been done in Section 6.3.1. Subsequently, we use an automatic `Policy Generation` unit composed of two tunable units using different versions of Fitted-Q Iteration, i.e., one with XGBoost as regressor, and the other with Extremely Randomized Trees. The two hyper-parameters configurations spaces are those presented in Appendix B.

Results. In Table 6, we report the empirical expected return over 10 episodes (standard deviation in brackets). It is worth noting that by adding more automation in the pipeline we can increase the performances on the environment under analysis w.r.t. the results obtained in Section 6.3.1.

Even though we have improved over the previously obtained result by about 20%, we point out that obtaining a statistically significant result would require a huge computational effort. Indeed, the entire run took around 16 h. We leave to future experiments the test on a larger number of samples to assess the statistical significance of this result.

[11] The details about the hyper-parameters configuration space, the tuning procedure, and the compute requirements for the Dam experiment can be found in Appendix B.3.

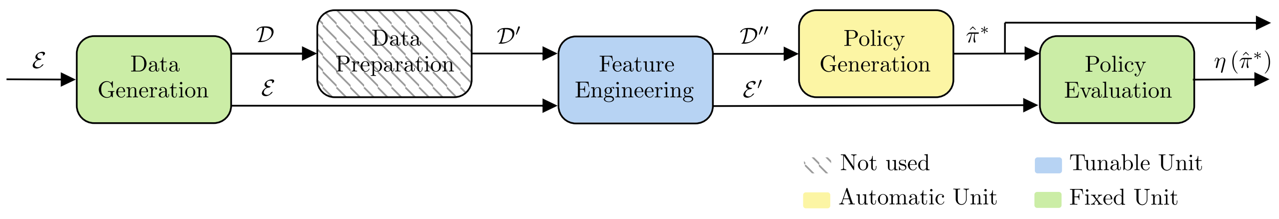


Fig. 6. Types of the units adopted in the offline pipeline experiment.

7. Conclusions and limitations

This paper introduced the ARLO framework for automating reinforcement learning by proposing two pipelines, one for the online setting and one for the offline setting. Moreover, we showcased the capabilities of such a framework by creating a Python library, and we tested its performance in both simulated and realistic settings.

While the proposed framework in its current formulation is flexible and allows adding customized stages, the complete democratization of RL is far from being achieved. First, the procedures to optimize the different stages were revealed to be computationally demanding. Thus, adding tools to predict and control the amount of computational time required by a pipeline is of paramount importance to obtain a flexible tool. Another interesting development, going in the opposite direction of what we have just mentioned, consists in including a "whole pipeline optimization" procedure, which *jointly* optimizes the entire learning process. This direction requires a preliminary development of less computationally demanding algorithms for each stage of the pipeline. Finally, we focused our attention on fully-observable, stationary, single-agent, single-objective settings. Developing a more general pipeline to relax some or all of the above assumptions would ease the application of RL algorithms in a wider spectrum of real-world problems.

Limitations. The goal of AutoRL is to bring RL closer to the non-expert user. This represents a source of opportunities and risks. On the one hand, making RL usable to a wide audience contributes to the *democratization* of the field, overcoming the need for specific education and opening it to the large public. On the other hand, such an abstract approach tends to compromise the transparency of the learning process and the traceability of the resulting model. Shadowing the underlying principles, AutoRL might pose the risk of misuse of RL approaches, leading to results that are not in line with expectations. Furthermore, AutoRL, even more than RL, requires huge amounts of data and computation that might represent a limit of the framework.

CRediT authorship contribution statement

Marco Mussi: Methodology, Formal analysis, Writing – original draft, Visualization. **Davide Lombarda:** Software, Data curation, Writing – original draft, Visualization. **Alberto Maria Metelli:** Conceptualization, Investigation, Formal analysis, Writing – review & editing. **Francesco Trovò:** Project administration, Investigation, Conceptualization, Writing – review & editing. **Marcello Restelli:** Conceptualization, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

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

Data availability

Data and Code available on GitHub.

Acknowledgments

This paper is supported by FAIR (Future Artificial Intelligence Research) project, funded by the NextGenerationEU program within the PNRR-PE-AI scheme (M4C2, Investment 1.3, Line on Artificial Intelligence).

Appendix A. Library

ARLO is a Python library implementing the framework described in this paper. It contains all the automation capabilities described in the main paper. It also provides the implementation of specific stages for each phase of the two pipelines we introduced in Section 4.

The RL algorithms present in the implementation are wrappers of those implemented in *MushroomRL* (D'Eramo et al., 2021). Moreover, we structured the library so that one has the option to implement wrappers for any other RL library, e.g., Stable Baselines, RLLib, and Tensorforce.

Supported units. In Table 7, we list the currently implemented units for each stage. As mentioned before, this is a non-exhaustive list of the possible methods that can be included in the proposed framework, but only those which we used for experimental purposes. See Section 5 for some suggestions about the methods which are appropriate for an extension for each stage.

Used libraries. ARLO requirements, in terms of libraries, are: catboost (v1.0.3), gym (v0.19.0), joblib (v1.1.0), matplotlib (v3.5.0), mushroom_rl (v1.7.0), numpy (v1.22.0), optuna (v2.10.0), plotly (v5.4.0), scikit_learn (v1.0.2), scipy (v1.7.3), torch (v1.10.1), xgboost (v1.5.1).

Appendix B. Details on the experiments

All the experiments were run on a Linux-based server with an *AMD Ryzen* 9 5950X 16-Core Processor with 128 GB DDR4 RAM running *Python* 3.8.8 on *CentOS* 8.5.2111.

Hyper-parameter tuning. The pseudo-code of the genetic tuner is detailed in Algorithm 1. The hyper-parameter tuning of the genetic algorithms are run for 50 generations, each one including 20 agents. Throughout each generation, *elitism* is performed, i.e., the best agent of the generation is preserved, and the new generation is created via tournament selection. More specifically, we take the best agent, out of a subset of 3 agents of the previous generation, and we repeat such an operation until 19 agents are selected (as the remaining spot is reserved for the best-performing agent in the previous generation).

Each hyper-parameter is mutated with probability 0.5, and two different types of mutation can take place:

- for *categorical* hyper-parameters and for the ones having discrete support, we sample from a uniform distribution over the possible values;
- for *numerical*, i.e., continuous domains, we sample hyper-parameters from a uniform distribution over 0.8 and 1.2 times the current value of the hyper-parameter.

Table 7
Supported units in the current ARLO implementation.

Stage	Implementations
`Data Generation`	Random Uniform Policy MEPOL (Mutti et al., 2021)
`Data Preparation`	Mean Imputation 1-NN Imputation
`Feature Engineering`	Recursive Feature Selection Forward Feature Selection via Mutual Information (Beraha et al., 2019) Nyström Map Feature Generation (Williams & Seeger, 2000)
`Policy Generation`	Fitted-Q Iteration (FQI, Ernst et al., 2005) Double Fitted-Q Iteration (DoubleFQI, D'Eramo, Nuara, Pirotta, & Restelli, 2017) Least Squares Policy Iteration (LSPI, Lagoudakis & Parr, 2003) Deep Q-Network (DQN, Mnih et al., 2015) Proximal Policy Optimization (PPO, Schulman et al., 2017) Deep Deterministic Policy Gradient (DDPG, Lillicrap et al., 2016) Soft Actor Critic (SAC, Haarnoja et al., 2018) GPOMDP (Baxter & Bartlett, 2001)

Algorithm 1 Genetic Tuner

```
1: Randomly initialize first generation
2: for $i \in [0, \dots, n\_generations)$ do
3:   Fit and evaluate each agent in the generation $i$
4:   Select the best agent and add it to the new generation $i+1$
5:   for $j \in [0, \dots, n\_agents - 1)$ do
6:     Add best agent, out of a random subset of 3, to the new generation $i+1$
7:   end for
8:   Mutate the new generation $i+1$
9: end for
```

Table 8
Supported environments in the current ARLO implementation.

Source	Type	Environment
Gym	Classic Control	Grid World Mountain Car Cart Pole
	MuJoCo	Inverted Pendulum Walker2d HalfCheetah Ant Hopper Humanoid Swimmer
Other	Controller	LQG

ARLO implements the Genetic Algorithm presented above as well as the hyper-parameter tuning solutions from Optuna (Akiba, Sano, Yanase, Ohta, & Koyama, 2019).

In all the experiments, we chose reasonable hyper-parameters configuration spaces so that they were neither too small, to avoid exploring a space that was too little and thus finding solutions quite far off, nor too large, to avoid increasing the total computational time (as some hyper-parameters have a great impact on the training time of the `Policy Generation` units).

Environment. Whenever an environment is used for the training of an RL algorithm, a deep copy of such an environment is provided to each agent in the generation, while in the case a dataset is used for the training of an RL algorithm, we provided each agent with a bootstrapped dataset coming from the original one. The environments currently available are presented in Table 8.

Loss function. As a loss function for guiding the tuning procedure, we used the empirical expected return defined in Eq. (1). The specific loss functions used in the different experiments for policy evaluation are detailed in the following sections.

Table 9
Hyper-parameters configuration space for the Linear Quadratic Gaussian Regulator experiment.

Hyper-parameter	Search space
Actor Learning Rate	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$
Actor Network	One layer with 16 neurons and ReLU activation
Critic Learning Rate	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$
Critic Loss	MSE
Critic Optimizer	Adam
Critic Network	One layer with 16 neurons and ReLU activation
Batch Size	{8, 16, 32, 64, 128}
Initial Replay Size	{10, 100, 300, 500, 1000, 5000}
Max Replay Size	{3000, 10000, 30000, 100000}
Warmup Transitions	{50, 100, 500}
Tau	0.005
Alpha Learning Rate	$\{10^{-5}, 10^{-4}, 10^{-3}\}$
Log Std Min	−20
Log Std Max	3
N Epochs	[1, 30]
N Episodes	[1, 1600]
N Episodes Per Fit	[1, 500]

B.1. Linear quadratic Gaussian regulator

In this experiment, presented in Section 6.1, we consider a Linear Quadratic Gaussian (LQG) Regulator characterized as follows:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$Q = 0.7 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad R = 0.3 \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Along each dimension, and for each time step t, the action a_t can take values in $[-3.5, 3.5]$, while the discount factor and the time horizon were set to $\gamma = 0.9$, and $T = 15$, respectively. We used a noise standard deviation as follows:

$$\sigma = 0.1 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

In Section 6.1.1, we perform three experiments using three different seeds: 2, 42, 2022. We tune the hyper-parameters of SAC through the genetic algorithm described above, using as a metric the empirical expected return. For all the seeds we consider, we used the hyper-parameters configuration space reported in Table 9. Notice that if only a single value is specified for its domain, it means that the hyper-parameter is considered fixed. The three runs performed (one for each seed) took on average 44 (±17.2) h each.

In Fig. 7, we report the value of the performance over time of the best agent of each generation for this experiment. It shows how the performances are almost constant in the last ≈ 25 generations, meaning

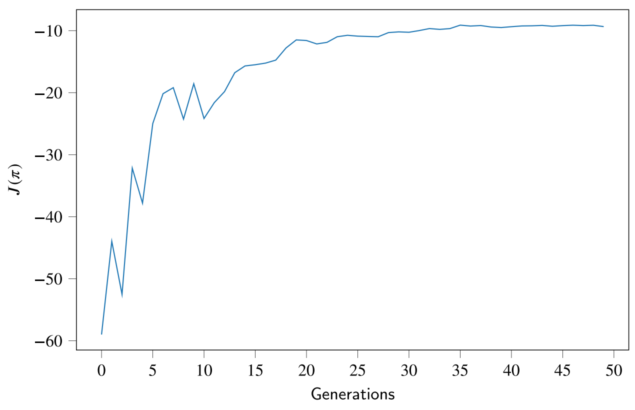


Fig. 7. SAC best agent performance for each generation.

Table 10
Hyper-parameters configuration space for the HalfCheetah experiment.

Hyper-parameter	Search space
Actor Learning Rate	$[10^{-5}, 10^{-2}]$
Actor Network	Two layers with 128 neurons and ReLU activations
Critic Learning Rate	$[10^{-5}, 10^{-2}]$
Critic Loss	MSE
Critic Optimizer	Adam
Critic Network	Two layers with 128 neurons and ReLU activations
Batch Size	[8, 256]
Initial Replay Size	[1000, 20000]
Max Replay Size	[10000, 1500000]
Tau	0.001
Policy delay	1
Policy	OrnsteinUhlenbeckPolicy($\sigma = 0.2$, $\theta = 0.15$, dt = 10^{-2})
N Epochs	[1, 50]
N Steps	[1000, 15000]
N Steps Per Fit	[1, 10000]

Table 11
Hyper-parameters configuration space for the `Feature Engineering` stage of the Dam experiment.

Hyper-parameter	Search space
K	{1, 2, 3, 4, 5, 10, 20, 50}
N Features	{1, 2, …, 31}

Table 12
Hyper-parameters used in the `Policy Generation` stage of the Dam experiment (Section 6.3.1 and Section 6.3.2).

Hyper-parameter	Value
N Iterations	60
N Estimators	100
Criterion	MSE
Min Samples Split	10

Table 13
Hyper-parameters configuration space of XGBoost (Section 6.3.2).

Hyper-parameter	Search space
N Iterations	[2, 60]
N Estimators	[5, 250]
Min Child Weight	[1, 100]
Subsample	[0.5, 1]
Learning Rate	[10^{-3}, 0.4]
Max Depth	[4, 15]

Table 14
Hyper-parameters configuration space of Extremely Randomized Trees (Section 6.3.2).

Hyper-parameter	Search space
N Iterations	[2, 60]
N Estimators	[5, 250]
Criterion	MSE
Min Samples Split	[1, 50]

that the optimization procedure converged to a solution near to a local minimum point. The scripts needed to run these three experiments (each corresponding to a different seed) are available at https://github.com/arlo-lib/ARLO/tree/main/experiments/LQG.

B.2. HalfCheetah

In this second experiment, presented in Section 6.2, we used the simulated environment of HalfCheetah to run an experiment on the model generation stage. The MDP corresponding to this environment is assumed to have a discount factor and a time horizon of $\gamma = 1$, and $T = 1000$, respectively. We perform three experiments using three different seeds: 2, 42, and 2022. We tune the hyper-parameters of DDPG using the genetic algorithm described above, considering as a metric the Average Reward.

For all three seeds, we consider the hyper-parameters configuration space reported in Table 10. The three runs performed (one for each seed), took on average 124.7 (±8.8) h each. The scripts needed to run these three experiments (each corresponding to a different seed) are available at https://github.com/arlo-lib/ARLO/tree/main/experiments/HalfCheetah-v3.

B.3. Dam

In this experiment, presented in Section 6.3, we consider the control of a water reservoir (dam) that models the dynamics of a real alpine lake, as described by Castelletti et al. (2011). The observation space is a continuous space with 31 dimensions, each of which takes values in $\mathbb{R}^+$. This state space features represent the inflow values for the previous month. The action space is sampled to get a discrete space with 8 actions, each one corresponding to a different amount of water released in a day. The full description of the environment is provided in Castelletti et al. (2011). The discount factor and the time horizon have been set to $\gamma = 0.999$ and $T = 360$, respectively. The dataset employed to run the experiments is constituted of 30 episodes, for an overall amount of 10800 samples.

B.3.1. Feature engineering

In this experiment, presented in Section 6.3.1, once we extract the dataset, we perform forward feature selection via mutual information, as described by Beraha et al. (2019). Hyper-parameters search space for the tested tunable feature selection unit is reported in Table 11.

Once feature selection is performed, we fit a `Policy Generation` unit, i.e., FQI, using an Extremely Randomized Trees Regressor (Geurts, Ernst, & Wehenkel, 2006) with the hyper-parameters present in Table 12. The entire run took around 2 h. The script needed to run this experiment is available at https://github.com/arlo-lib/ARLO/tree/main/experiments/Dam/dam.py.

B.3.2. Complete offline pipeline

In this experiment, presented in Section 6.3.2, we consider the same hyper-parameters search space for `Feature Engineering` tunable unit, and those presented in Tables 13 and 14 for the `Policy Generation` automatic unit for FQI with XGBoost and Extremely Randomized Trees, respectively.

The script needed to run this experiment is available at https://github.com/arlo-lib/ARLO/blob/main/experiments/Dam/hp_tuning_fqi_dam.py.

References

Afshar, R. R., Zhang, Y., Vanschoren, J., & Kaymak, U. (2022). Automated reinforcement learning: An overview. CoRR, abs/2201.05000.

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the ACM international conference on knowledge discovery and data mining*.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. CoRR, abs/1708.05866.

Baxter, J., & Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, *15*, 319–350.

Beraha, M., Metelli, A. M., Papini, M., Tirinzoni, A., & Restelli, M. (2019). Feature selection via mutual information: New theoretical insights. In *Proceedings of the international joint conference on neural networks* (pp. 1–9). IEEE.

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning, vol. 4*. Springer.

Bisi, L., Sabbioni, L., Vittori, E., Papini, M., & Restelli, M. (2021). Risk-averse trust region optimization for reward-volatility reduction. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence* (pp. 4583–4589).

Buşoniu, L., de Bruin, T., Tolić, D., Kober, J., & Palunko, I. (2018). Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, *46*, 8–28.

Castelletti, A., Galelli, S., Restelli, M., & Soncini-Sessa, R. (2011). Tree-based variable selection for dimensionality reduction of large-scale control systems. In *Proceedings of the IEEE symposium on adaptive dynamic programming and reinforcement learning* (pp. 62–69).

D'Eramo, C., Nuara, A., Pirotta, M., & Restelli, M. (2017). Estimating the maximum expected value in continuous reinforcement learning problems. In *Proceedings of the conference on artificial intelligence, vol. 31*.

D'Eramo, C., Tateo, D., Bonarini, A., Restelli, M., & Peters, J. (2021). MushroomRL: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, *22*, 131:1–5.

Dorato, P., Cerone, V., & Abdallah, C. (1994). *Linear-quadratic control: An introduction*. Simon & Schuster, Inc.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the international conference on machine learning* (pp. 1329–1338).

Endrawis, S., Leibovich, G., Jacob, G., Novik, G., & Tamar, A. (2021). Efficient self-supervised data collection for offline robot learning. In *Proceedings of the IEEE international conference on robotics and automation* (pp. 4650–4656).

Ernst, D., Geurts, P., & Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, *6*, 503–556.

Falkner, S., Klein, A., & Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the international conference on machine learning, vol. 80* (pp. 1436–1445).

Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., & Hutter, F. (2020). Auto-sklearn 2.0: Hands-free automl via meta-learning. CoRR, abs/2007.04074.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J. T., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. In *Proceedings of the neural information processing systems* (pp. 2962–2970).

Franke, J. K. H., Köhler, G., Biedenkapp, A., & Hutter, F. (2021). Sample-efficient automated deep reinforcement learning. In *Proceedings of the international conference on learning representations* (pp. 1–23).

Gao, W., Kannan, S., Oh, S., & Viswanath, P. (2017). Estimating mutual information for discrete-continuous mixtures. In *Proceedings of the neural information processing systems, vol. 30*.

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, *63*(1), 3–42.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the international conference on machine learning, vol. 80* (pp. 1856–1865).

He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, *212*, Article 106622.

Islam, R., Henderson, P., Gomrokchi, M., & Precup, D. (2017). Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. CoRR, abs/1708.04133.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., et al. (2017). Population based training of neural networks. CoRR, abs/1711.09846.

Jadhav, A., Pramod, D., & Ramanathan, K. (2019). Comparison of performance of data imputation methods for numeric dataset. *Applied Artificial Intelligence*, *33*(10), 913–933.

Kraskov, A., Stögbauer, H., & Grassberger, P. (2004). Estimating mutual information. *Physical Review E*, *69*, Article 066138.

Lagoudakis, M. G., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*, 1107–1149.

Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., & Srinivas, A. (2020). Reinforcement learning with augmented data. In *Proceedings of the neural information processing systems*.

LeDell, E., & Poirier, S. (2020). H2O AutoML: Scalable automatic machine learning. In *Proceedings of the ICML workshop on automated machine learning*.

Lee, K., Laskin, M., Srinivas, A., & Abbeel, P. (2021). SUNRISE: A simple unified framework for ensemble learning in deep reinforcement learning. In *Proceedings of the international conference on machine learning, vol. 139* (pp. 6131–6141).

Levine, S., Kumar, A., Tucker, G., & Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. CoRR, abs/2005.01643.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., et al. (2016). Continuous control with deep reinforcement learning. In *Proceedings of the international conference on learning representations*.

Lizotte, D. J., Gunter, L., Laber, E., & Murphy, S. A. (2008). Missing data and uncertainty in batch reinforcement learning. In *Proceedings of the neural information processing systems*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Mutti, M., Pratissoli, L., & Restelli, M. (2021). Task-agnostic exploration via policy gradient of a non-parametric state entropy estimate. In *Proceedings of the conference on artificial intelligence* (pp. 9028–9036).

Ng, A. Y., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the international conference on machine learning, vol. 99* (pp. 278–287).

Nguyen, H., & La, H. M. (2019). Review of deep reinforcement learning for robot manipulation. In *Proceedings of the IEEE international conference on robotic computing* (pp. 590–595).

Olson, R. S., Bartley, N., Urbanowicz, R. J., & Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the ACM genetic and evolutionary computation conference* (pp. 485–492).

Parker-Holder, J., Nguyen, V., Desai, S., & Roberts, S. J. (2021). Tuning mixed input hyperparameters on the fly for efficient population based AutoRL. CoRR, abs/2106.15883.

Parker-Holder, J., Rajan, R., Song, X., Biedenkapp, A., Miao, Y., Eimer, T., et al. (2022). Automated reinforcement learning (AutoRL): A survey and open problems. *Journal of Artificial Intelligence Research*, *74*, 517–568.

Pathak, D., Gandhi, D., & Gupta, A. (2019). Self-supervised exploration via disagreement. In *Proceedings of the international conference on machine learning* (pp. 5062–5071).

Portelas, R., Colas, C., Weng, L., Hofmann, K., & Oudeyer, P. (2020). Automatic curriculum learning for deep RL: A short survey. In *Proceedings of the twenty-ninth international joint conference on artificial intelligence* (pp. 4819–4825). ijcai.org.

Pratt, J. W. (1978). Risk aversion in the small and in the large. In *Uncertainty in economics* (pp. 59–79). Elsevier.

Puterman, M. L. (2014). *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons.

Saphal, R., Ravindran, B., Mudigere, D., Avancha, S., & Kaul, B. (2021). SEERL: Sample efficient ensemble reinforcement learning. In *Proceedings of the ACM international conference on autonomous agents and multiagent systems* (pp. 1100–1108).

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay. In *Proceedings of the international conference on learning representations*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., & Moritz, P. (2015). Trust region policy optimization. In *Proceedings of the international conference on machine learning, vol. 37* (pp. 1889–1897).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. CoRR, abs/1707.06347.

Sehgal, A., La, H., Louis, S., & Nguyen, H. (2019). Deep reinforcement learning using genetic algorithm for parameter optimization. In *Proceedings of the IEEE international conference on robotic computing* (pp. 596–601).

Singh, H., Misra, N., Hnizdo, V., Fedorowicz, A., & Demchuk, E. (2003). Nearest neighbor estimates of entropy. *American Journal of Mathematical and Management Sciences*, *23*(3–4), 301–321.

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

Team, O. E. L., Stooke, A., Mahajan, A., Barros, C., Deck, C., Bauer, J., et al. (2021). Open-ended learning leads to generally capable agents. CoRR, abs/2107.12808.

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 847–855).

Tirinzoni, A., Sessa, A., Pirotta, M., & Restelli, M. (2018). Importance weighted transfer of samples in reinforcement learning. In *Proceedings of the international conference on machine learning* (pp. 4936–4945).

Van Dooren, P. (1981). A generalized eigenvalue approach for solving Riccati equations. *SIAM Journal on Scientific and Statistical Computing*, *2*(2), 121–135.

Wang, S., Jia, D., & Weng, X. (2018). Deep reinforcement learning for autonomous driving. CoRR, abs/1811.11329.

Williams, C. K. I., & Seeger, M. W. (2000). Using the Nyström method to speed up kernel machines. In *Advances in neural information processing systems* (pp. 682–688). MIT Press.

Ye, C., Khalifa, A., Bontrager, P., & Togelius, J. (2020). Rotation, translation, and cropping for zero-shot generalization. In *Proceedings of the IEEE conference on games* (pp. 57–64).

Zamora, I., Lopez, N. G., Vilches, V. M., & Cordero, A. H. (2016). Extending the OpenAI gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. CoRR, abs/1608.05742.

Zhang, B., Rajan, R., Pineda, L., Lambert, N. O., Biedenkapp, A., Chua, K., et al. (2021). On the importance of hyperparameter optimization for model-based reinforcement learning. In *Proceedings of the international conference on artificial intelligence and statistics, vol. 130* (pp. 4015–4023).

Zhang, Z., Zohren, S., & Roberts, S. (2020). Deep reinforcement learning for trading. *The Journal of Financial Data Science*, *2*(2), 25–40.