Applied Intelligence
https://doi.org/10.1007/s10489-024-05573-0

LExCI: A framework for reinforcement learning with embedded systems

Kevin Badalian[1] · Lucas Koch[1] · Tobias Brinkmann[1] · Mario Picerno[1] · Marius Wegener[2] · Sung-Yong Lee[1] · Jakob Andert[1]

Accepted: 27 May 2024

Abstract
Advances in artificial intelligence (AI) have led to its application in many areas of everyday life. In the context of control engineering, reinforcement learning (RL) represents a particularly promising approach as it is centred around the idea of allowing an agent to freely interact with its environment to find an optimal strategy. One of the challenges professionals face when training and deploying RL agents is that the latter often have to run on dedicated embedded devices. This could be to integrate them into an existing toolchain or to satisfy certain performance criteria like real-time constraints. Conventional RL libraries, however, cannot be easily utilised in conjunction with that kind of hardware. In this paper, we present a framework named LExCI, the *Learning and Experiencing Cycle Interface*, which bridges this gap and provides end-users with a free and open-source tool for training agents on embedded systems using the open-source library RLlib. Its operability is demonstrated with two state-of-the-art RL-algorithms and a rapid control prototyping system.

Keywords Reinforcement learning · embedded systems · automation · control engineering

1 Introduction

1.1 RL, Control Tasks, and Embedded Systems

In recent years, artificial intelligence (AI) has evolved into a scientific discipline with tangible effects on the lives of ordinary people. Not only does it allow for convenience features such as speech recognition or auto-completion when writing [1], but it is also increasingly being utilised to control complex devices and even safety-critical systems [2, 3]. Modern advanced driver-assistance systems (ADAS), not to mention autonomous driving, would be unimaginable without it [4].

Reinforcement learning (RL) is an especially useful area of AI when it comes to control tasks. Since it is based on agents that learn through their own interactions with the environment (i.e. they generate their own training data), RL has the potential to find optimal solutions to non-trivial problems with minimal input from experts. One problem engineers have to address, though, is the integration of the RL agent into the system it shall control. Industrial applications often come with a long list of strict requirements regarding their information technology (IT) ecosystems: physical space, power, and cooling capacity are usually limited [5]. At the same time, devices need to be rugged enough to withstand vibrations or extreme fluctuations in temperature. Beyond such hardware-related matters, a great number of use-cases necessitate a real-time operating system (OS) which guarantees that computations are performed within a fixed time window [5]. Then, there is the cost factor. High-performance components needlessly drive up the prices of commercial products if their potential is not fully harnessed. A cheaper device is, therefore, more favourable so long as it is adequate for the task [6].

As a consequence of these boundary conditions, traditional personal computers (PCs) are not suitable for a wide range of applications. Professionals choose embedded systems instead: dedicated computers that are integrated into a larger system for the purpose of controlling the same [6]. Embedded systems are designed from the ground up to meet the requirements outlined above. Nonetheless, they can be incapable of running programs intended for conventional computers due to their inherent limitations. Established RL

https://github.com/mechatronics-RWTH/lexci-2

Extended author information available on the last page of the article

Published online: 26 June 2024

libraries like Ray/RLlib[1] [7, 8] or Stable-Baselines3[2] [9] further rely on third-party software (e.g. Python) which might take up too much data storage space or simply not be available on the target platform/OS. Part of that list of dependencies are libraries for machine learning (ML) models, i.e. the mathematical structures (most notably neural networks (NNs)) which, among other things, represent the behaviour of the agent. Prominent exemplars — for instance, TensorFlow (TF)[3] [10] or PyTorch[4] [11] — suffer from the same problems, meaning that merely executing a trained agent on an embedded system may not be a straightforward endeavour [12].

1.2 Model Execution

Even if a ML library cannot be installed on an embedded device, there are still ways to put its agents to use. The simplest is to run them on external machines that are then contacted by embedded devices in order to retrieve actions for their observations. Due to the latency associated with this option, it is likely to be sub-optimal. Another detracting factor is that the agents are not executed *on* the actual controllers. A more fitting solution is to convert the models to a format that is suitable for the target hardware, possibly by translating them into a generic representation like Open Neural Network Exchange (ONNX)[5] or some other intermediate format first.

TensorFlow Lite Micro (TFLM)[6] [13], for example, condenses TF to its core functionality, optimises its code for micro-controllers [12], and reduces the number of third-party dependencies. TFLM can be thought of as a subset of TensorFlow Lite (TF Lite)[7], a lean version of the full library geared towards mobile and edge devices. It is hence capable of reading TF Lite models as long as they are comprised of common operations. Conveniently, TF can natively convert full models to TF Lite.

cONNXr [14], on the other hand, is agnostic to the model's original framework as it is written to work with models defined in the ONNX format. Other libraries take a more puristic approach and implement their own model formats in C or C++ using either nothing but the respective standard library or just a handful of header-only libraries. Projects in that category are Genann [15], KANN [16], tiny-dnn [17], or MiniDNN [18]. End-users have to manually re-write and configure their models with those solutions, though, because they typically lack converters.

Besides the above, there are solutions that transpile existing model formats to pure C/C++ code which is then compiled for the target hardware [5]. frugally-deep [19], keras2cpp [20], or onnx2c [21] follow that philosophy. Likewise, MATLAB[8] is capable of generating code from imported ONNX models when using its Reinforcement Learning Toolbox [22].

1.3 Training the Model

Training — that is the act of updating an agent's model — is performed using RL libraries like the aforementioned Ray/RLlib. Given the limitations of most embedded systems, this step is usually outsourced to a powerful workstation or a cluster so as to merely deploy the agent on the target hardware [5]. If not automated, this TinyML [23] strategy becomes tedious when learning with on-policy RL algorithms (cf. Section 2.1) due to the fact that the deployment process must be repeated after each and every modification of the agent. To make matters worse, the generated training data usually cannot be passed directly to the algorithm either and requires post-processing.

1.4 Proposed Solution

Motivated by the shortcomings of RL software in this area, we developed the *Learning and Experiencing Cycle Interface* or LExCI for short. This general-purpose framework allows experts to easily train RL agents with Ray/RLlib when model execution happens on an embedded system and training takes place on another, conventional machine. All models are implemented in TFLM/TF Lite and TF, respectively. LExCI is open-source and freely available to the public through its official GitHub repository (https://github.com/mechatronics-RWTH/lexci-2). Our contributions are:

- a free, open-source, general-purpose RL framework based on established libraries
- training with embedded systems
- out-of-the-box support for elaborate NN architectures such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs) that yield better results than merely fully-connected ones for certain RL problems [24]
- compatibility with different model-free RL algorithms, both on- and off-policy (see Section 2.1)
- helper classes for automating various pieces of control software
- an architecture that lends itself to parallelisation

[1] https://docs.ray.io/en/latest/rllib/index.html

[2] https://stable-baselines3.readthedocs.io/en/master/

[3] https://www.tensorflow.org/

[4] https://pytorch.org/

[5] https://onnx.ai/

[6] https://www.tensorflow.org/lite/microcontrollers

[7] https://www.tensorflow.org/lite

[8] https://www.mathworks.com/products/matlab.html

Earlier versions of the software have already proven themselves in academic research. In [25] and [26], agents were trained to control the high-pressure exhaust gas recirculation (EGR) valve of a Euro 6d Diesel engine on different X-in-the-loop (XiL) virtualisation levels, in part by utilising LExCI's transfer learning (TL) capabilities. The resulting strategies led to lower NO_x and soot emissions while maintaining the same performance as a virtual (-5% NO_x, -10% soot) and real engine control unit (ECU) (-0.1% NO_x, -5.8% soot). Similarly, [27] applied the framework to learn a control strategy for the variable-geometry turbocharger (VGT) in the same setup which reduced NO_x emissions by 4% and soot emissions by 10%. In [28], LExCI was embedded into a cloud-based service in order to train an agent to control the longitudinal acceleration of an electric vehicle in a a simulated medium-sized German city. The trained agent learned to factor in any preceding vehicle, traffic lights, the speed limit, road curvature, and slope such that the journey is safe, comfortable and efficient.

Concerning the state of the art, a related solution has been proposed in [29] where the authors present a conceptually similar toolchain that employs a modified version of keras-rl's [30] Deep Deterministic Policy Gradient (DDPG) implementation to train an agent that is executed on a rapid control prototyping (RCP) system. Their program is designed such that it could interface various algorithm implementations from different libraries and it requires the third-party tool ControlDesk[9] to access the embedded system. The NNs on the embedded side were hand-coded by the authors in MATLAB/Simulink[10] and are limited to fully-connected feed-forward networks. In comparison, LExCI offers more flexibility regarding the control software, design of NNs, and the choice of RL algorithms. Another related piece of work is [31] where a *classic* Q-learning agent is first trained on a powerful computer and its Q-table then transferred to the embedded device. The solution that the authors offer is limited to that one algorithm and does not support NNs, i.e. it does not allow for deep RL [32]. Furthermore, it does not come with a mechanism to automatically and repeatedly replace the agent on the target system.

The remainder of this paper is structured as follows: First, the foundations of RL and two state-of-the-art RL algorithms are expounded in Section 2. After describing LExCI and its inner workings in Section 3, Section 4 summarises the experiments that were conducted to showcase the viability of the framework and discusses the results. Finally, Section 5 recapitulates LExCI's performance, its strengths, and how it can be extended in the future.

[9] https://www.dspace.com/de/gmb/home/products/sw/experimentandvisualization/controldesk.cfm

[10] https://www.mathworks.com/products/simulink.html

2 Theoretical Background

In order to understand the manner in which LExCI operates, it is crucial to cover the theory behind RL. Along with the general concepts, this section delineates two state-of-the-art algorithms and their distinct requirements regarding the framework.

2.1 Reinforcement Learning

RL is a ML paradigm based on the concept of training an agent by letting it freely interact with its environment. The experiences that are generated in the process are collected and utilised to update the agent's policy such that the cumulated reward it receives for its behaviour is maximised. [32]

The mathematical foundation of the environment is a time-discrete Markov decision process (MDP)[11] defined by the four-tuple (S, A, P, R), that is

- the set of all possible states S,
- the action space A,
- the transition probability function $P : S \times A \times S \to [0, 1]$, and
- the reward function $R : S \times A \times S \to \mathbb{R}$.

During an interaction, the agent observes the current state $s_t \in S$ and chooses an action $A \ni a_t \sim \pi_\theta(\cdot|s_t)$. This causes the environment to transition into the next state $s'_t = s_{t+1} \in S$ with a probability of $P(s'_t|s_t, a_t)$ and the reward $r_t = R(s_t, a_t, s'_t)$ is given. The flag d indicates whether s' is a terminal state ($d = 1$) or not ($d = 0$). The action distribution $\pi_\theta : S \times A \to [0, 1]$ with configurable parameters θ is the agent's policy and determines its strategy. An episode or trajectory is a sequence $\tau = (\chi_0, \chi_1, \ldots, \chi_T)$ of experiences $\chi_t = (s_t, a_t, s'_t, r_t, d_t)$. The goal of RL is to tweak θ in order to maximise the discounted return with discount factor γ (1) or the expected return (2).

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t, \quad \gamma \in (0, 1] \tag{1}$$

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2}$$

One prominent optimisation method is gradient ascent which performs iterative update steps

$$\theta_{i+1} = \theta_i + \eta \nabla_\theta J(\pi_{\theta_i}) \tag{3}$$

with a learning rate $\eta \in \mathbb{R}$. The policy is typically implemented as an NN in which case the parameter set θ consists of its weights and biases. [32]

[11] Discrete-time MDPs particularly lend themselves to embedded problems because the latter usually sample signals in fixed, discrete time steps.

There are three key metrics to quantify how well an agent fares in a certain situation: The value function (VF) V_{π_θ} (4) estimates the return at a state $s \in S$ when acting on-policy (i.e. when choosing actions according to the current policy) from there on. Similarly, the action-value function or Q-function Q_{π_θ} (5) estimates the return when taking an action $a \in A$ at a state $s \in S$ on the assumption that all following actions are on-policy. The advantage function A_{π_θ} (6) is the difference of the two and measures how much better it is to take an action compared to what the policy would do. [32]

$$V_{\pi_\theta}(s) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s] \tag{4}$$

$$Q_{\pi_\theta}(s,a) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)|s_0 = s, a_0 = a] = r + \gamma V_{\pi_\theta}(s') \tag{5}$$

$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \tag{6}$$

Approximations of the above are denoted as $\hat{V}_{\pi_\theta}$, $\hat{Q}_{\pi_\theta}$, and $\hat{A}_{\pi_\theta}$, respectively.

An important property that distinguishes RL algorithms is whether they insist that the actions in their training data be sampled using the current policy. Those that do are called *on-policy*, the rest *off-policy*. Furthermore, if the algorithm has access to a model of the environment or learns one for the purpose of predicting the outcome of actions, it is called *model-based*, otherwise *model-free*. It has to be noted that this model is distinct from the agent's behaviour model or any of its value function approximators. [32, 33]

2.2 Algorithms

2.2.1 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a state-of-the-art model-free, on-policy RL algorithm for discrete and continuous action spaces. It features a surrogate loss function whose scaled advantages are clipped to avoid excessively large update steps that could destabilise the training. To that end, PPO trains a VF approximator in addition to the policy. [34, 35]

The algorithm first garners a training batch, i.e. a defined number of experiences, using its current parameters θ. When updating the agent, subsets known as mini-batches are drawn therefrom to perform multiple steps of stochastic gradient descent (SGD) to minimise the following loss function:

$$L_\theta(\chi,\xi) = -\left(L_\theta^{\text{clip}}(\chi,\xi) + L_\theta^{\text{KL}}(\chi,\xi) - c_{\text{VF}} L_\theta^{\text{VF}}(\chi) + c_{\text{S}} S(\chi,\xi)\right) \tag{7}$$

ξ denotes the policy's parameter set after a SGD step. The individual components of (7) are the clipped surrogate objective

$$L_\theta^{\text{clip}}(\chi,\xi) = \min\left(\max\left(\min\left(\frac{\pi_\xi(a|s)}{\pi_\theta(a|s)}, 1+\epsilon\right), 1-\epsilon\right)\hat{A}_{\pi_\theta}(s,a), \frac{\pi_\xi(a|s)}{\pi_\theta(a|s)}\hat{A}_{\pi_\theta}(s,a)\right) \tag{8}$$

for a clip parameter $\epsilon \in \mathbb{R}$, the Kullback-Leibler (KL) divergence penalty

$$L_\theta^{\text{KL}}(\chi,\xi) = \frac{\pi_\xi(a|s)}{\pi_\theta(a|s)}\hat{A}_{\pi_\theta}(s,a) - \beta \cdot \text{KL}(\pi_\theta(\cdot|s), \pi_\xi(\cdot|s)) \tag{9}$$

with an adaptive coefficient $\beta \in \mathbb{R}$, the squared error of the VF approximator

$$L_\theta^{\text{VF}}(\chi) = \left(\hat{V}_{\pi_\theta}(s) - V_{\pi_\theta}(s)\right)^2 \tag{10}$$

and its coefficient $c_{\text{VF}} \in \mathbb{R}$, and an optional entropy bonus $S(\chi,\xi)$ with its coefficient $c_{\text{S}} \in \mathbb{R}$ to encourage exploration. [34, 35]

2.2.2 Deep Deterministic Policy Gradient

The DDPG algorithm is a modern model-free, off-policy RL method that extends the idea of the Deep Q-Network (DQN) algorithm to continuous action spaces. Since its policy is deterministic, exploration is achieved by adding random noise, e.g. from a Gaussian distribution or an Ornstein-Uhlenbeck (OU) process, to its output. [36, 37]

DDPG trains a NN with parameters θ_Q as an approximation $\hat{Q}_{\theta_Q}$ of the Q-function when acting greedily and another NN with parameters θ_μ for the deterministic policy μ_{θ_μ} that seeks to maximise $\hat{Q}_{\theta_Q}$. To stabilise training, target networks $\hat{Q}_{\theta'_Q}$ and $\mu_{\theta'_\mu}$ with parameters θ'_Q and θ'_μ are employed. The Q-network is trained by minimising

$$L_{\theta_Q,\theta'_Q,\theta'_\mu}(\chi) = \left(\hat{Q}_{\theta_Q}(s,a) - \left(r + \gamma(1-d)\hat{Q}_{\theta'_Q}(s', \mu_{\theta'_\mu}(s'))\right)\right)^2 \tag{11}$$

and the policy is updated by performing gradient ascent using $\nabla_{\theta_\mu}\hat{Q}_{\theta_Q}(s, \mu_{\theta_\mu}(s))$. [36, 37]

The target networks are updated via polyak averaging, i.e.

$$\theta'_{Q,i+1} = \rho\theta_{Q,i} + (1-\rho)\theta'_{Q,i} \tag{12}$$

$$\theta'_{\mu,i+1} = \rho\theta_{\mu,i} + (1-\rho)\theta'_{\mu,i} \tag{13}$$

for $\rho \ll 1$. Also, batches are sampled from a replay memory buffer which can be supplemented with off-policy experiences. [36, 37]

3 Software

This section describes LExCI's components, how it operates, and the steps one has to take in order to set it up for a new RL problem. Furthermore, the *RL Block*, a plug-and-play Simulink model that encapsulates all necessary parts to execute an agent's policy model and to store experiences, is presented.

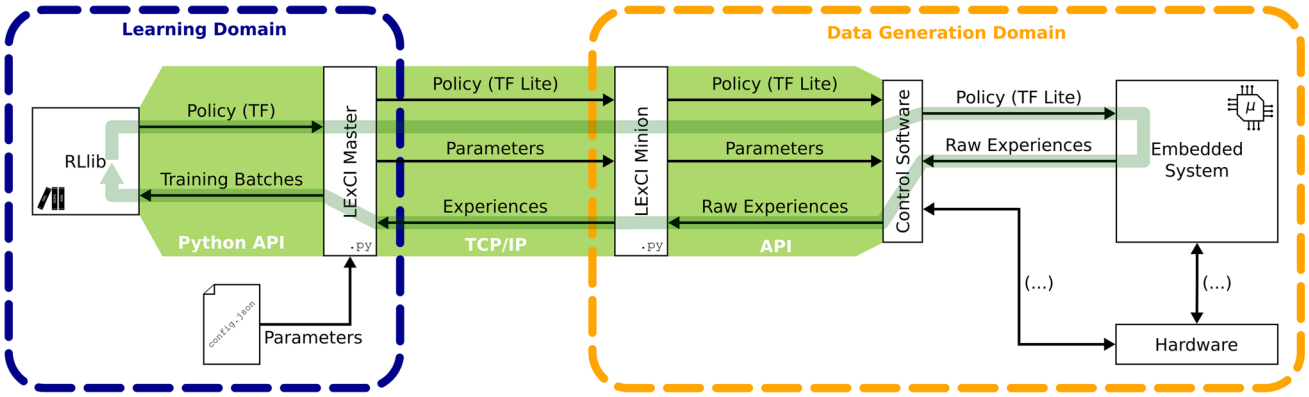


Fig. 1 Software architecture of the LExCI framework with the eponymous cycle as a light green arrow. There are multiple independent instances of the data generation domain when the process is parallelised

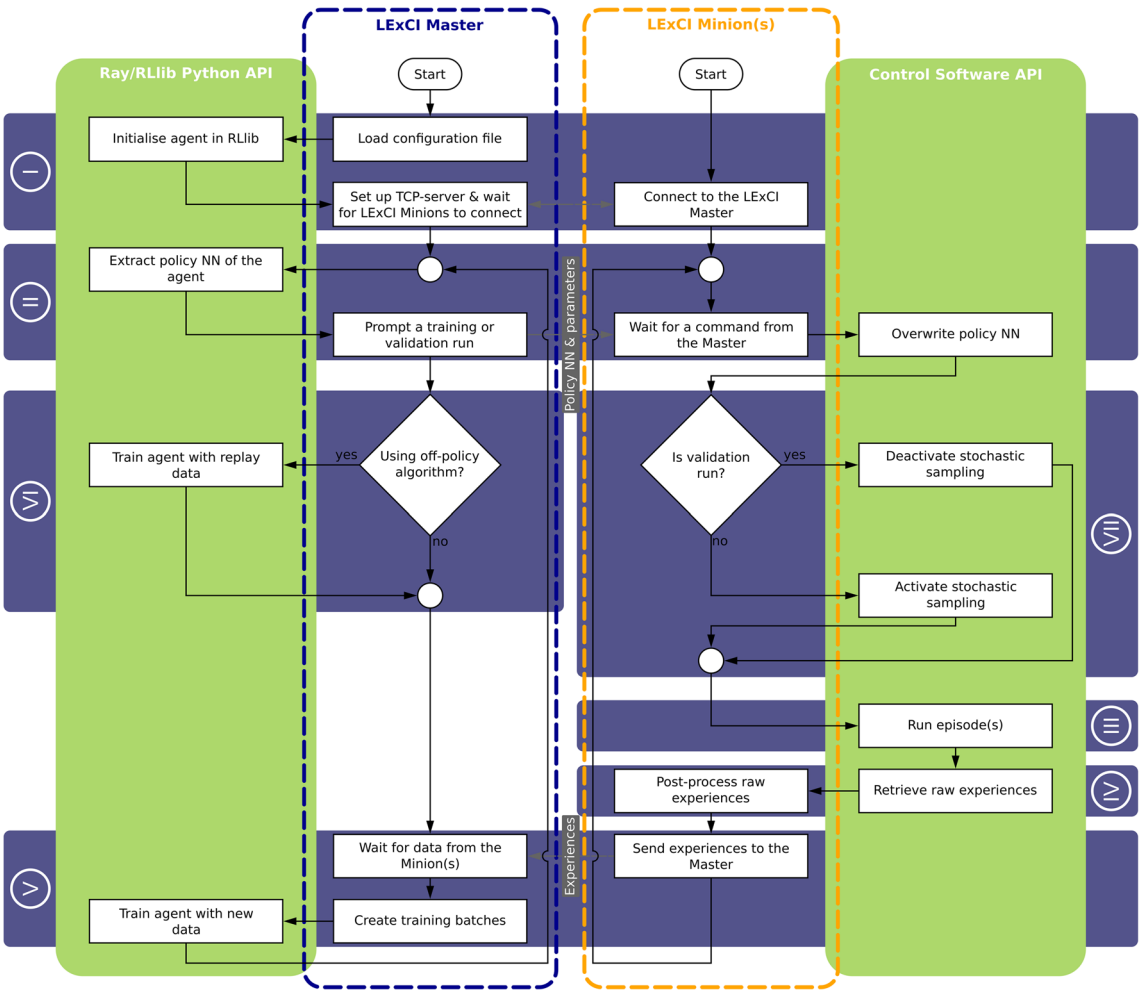


Fig. 2 Simplified flowchart of the LExCI framework. The grey, dashed arrows indicate communication/data exchange between the Minion and the Master. The blue areas tagged with Roman numerals serve as references for the textual description of the figure

3.1 Architecture and General Workflow

LExCI is logically divided into two domains as illustrated in Fig. 1: The first is the learning side of the framework with the LExCI Master at its head. Its counterpart is the data generation side where the LExCI Minion is located. To understand their roles and how they work together, it is best to have a look at the framework's *modus operandi*. As an aid, Fig. 2 complements Fig. 1 with the chronological order of the steps. The sections highlighted there shall be used as a guide.

Section I The LExCI Master makes use of a slightly modified version of Ray/RLlib 1.13.0[12] via the library's Python application programming interface (API). At program startup, it loads a JSON-formatted configuration file containing the parameters of the training. These include the characteristics of the problem (the dimensions of the observation and action space, whether actions are continuous or discrete, etc.), general settings (networking details, where to store logs and results, and the like) as well as the algorithm's hyperparameters (the architecture of the agent's NN(s), the learning rate LR/schedule, or batch sizes to name a few). The Master initialises the agent based on the settings above before proceeding to its main loop for training. In addition to being the gateway to the RL library, the Master acts as a server and listens for incoming TCP/IP connections from LExCI Minions. Established connections are constantly monitored for their status and closed if the opposite side stops sending heartbeats (e.g. after a program crash) or takes too long to finish its task. Thus, the system is able to cope with unforeseen events.

Sections II & III Training is carried out by completing so called cycles. At the beginning of a cycle, the LExCI Master retrieves the agent's current policy from RLlib and converts it from its original TF format to TF Lite. This model, along with all relevant training parameters (e.g. the number of experiences to generate), is broadcast to the connected LExCI Minions using a custom JSON-based protocol. Upon receipt, each Minion utilises the API of its control software to overwrite the policy on the embedded device which is then prompted to generate experiences. Additional pieces of hardware can be part of the data generation domain and interact with the embedded device. Besides the closed-loop control system, those include physical actuators or sensors.

Section IV Once enough data has been collected, the Minion uses the control software again to get the raw experiences and post-processes them. For instance, a domain expert could define auxiliary penalties that are added to the reward in situations where the agent's actions were clearly nonsensical.

Section V The experiences are sent to the LExCI Master and arranged into training batches, i.e. the data format RLlib expects for training. During that process, experiences are supplemented with additional information if the algorithm calls for it. For example, PPO requires the predicted value of the VF approximator (see (4)), the action distribution, and the probability of the action on top of the standard quantities. After the training batch has been assembled, it is given to RLlib for training the agent and the cycle starts anew.

Section VI When learning with off-policy algorithms, the LExCI Master does not remain idle while the Minions are doing their part. Instead, the Master continues training with experiences drawn from its replay memory buffer. The size of the buffer, the number of replay training steps per cycle, and the extent to which the buffer must be filled before replay training starts are set in the configuration file.

Section VII Apart from training runs, LExCI can be configured to conduct validation episodes with a defined frequency. They differ in that actions are always set to the mean of the action distribution and are hence deterministic during validations rather than being sampled stochastically. Thus, the results are more comparable and lend themselves better to assessing the agent's performance. Further, validations are conducted by a single Minion.

The master-minion architecture has the added benefit that it enables easy parallelisation. In light of the fact that embedded devices usually operate in real-time, this feature can speed up the data generation process dramatically. When there are multiple LExCI Minions available, the Master splits the workload between them so each only has to generate a fraction of the required number of experiences.

3.2 Setup

When employing the framework for a new use-case, the Master and the Minion must first be set up. LExCI is shipped with what is called a *universal Master* for each RL algorithm. Those are ready-to-use Python programs that function as described in Section 3.1, so one merely has to select the right algorithm and adjust the parameters in the configuration file. Alternatively, users can write their own custom Master programs which create an instance of the `Master` class and call its main loop. The Minion is always tailor-made for the problem by writing a program that instantiates the `Minion` class and invokes its main loop. There, the logic for preparing the embedded system, overwriting the agent's model, running episodes, post-processing experiences, etc. is programmed. The class expects callback functions for generating training and validation data. To this effect, LExCI offers helper classes that facilitate interacting with the embedded system via a control software. At the time of writing, there are helpers for ControlDesk, MATLAB/Simulink, and ECU-TEST[13].

[12] The modified version allows DDPG agents to choose between on-policy/off-policy training and is part of the LExCI repository.

[13] https://www.tracetronic.com/products/ecu-test/

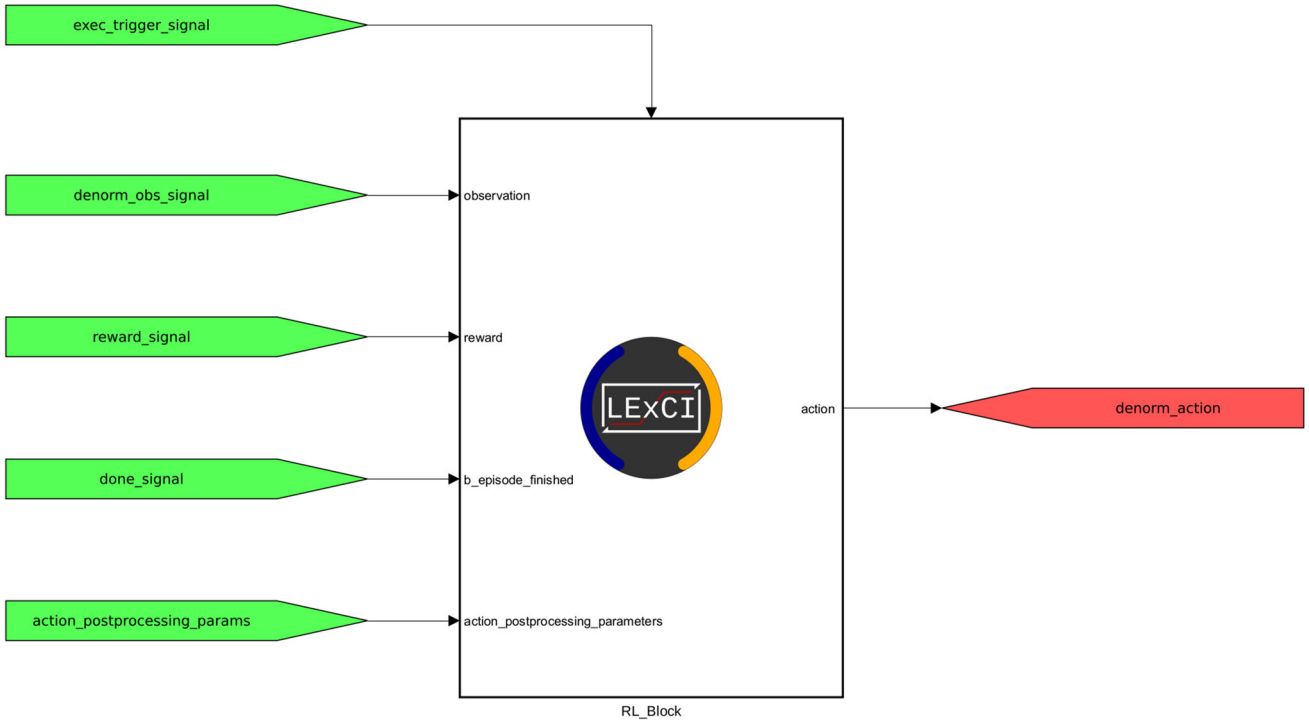


Fig. 3 LExCI's RL Block in Simulink. The ports `observation` and `action` are in the denormalised space of the environment

Another significant facet of the setup process involves the software that shall be running on the embedded system itself. After all, it is responsible for executing the policy NN of the agent. Users are free to implement the inference of actions in whatever way they deem fit. Having said that, it is of paramount importance that they distinguish between what are called *normalised* and *denormalised* spaces. Normalised observations and actions are the raw quantities passed to and received from NNs. Denormalised quantities, on the other hand, are the ones that the environment provides or expects. It is standard practice to, for example, min-max normalise observations (from the environment) to the range $[-1, +1]$ (which would then be the agent's normalised observation space) to stabilise and expedite training [32]. By the same token, the normalised actions of the agent must be mapped to the allowed (denormalised) range in the environment, e.g. via a hyperbolic tangent and scaling or simply by clipping. The data that the Minion retrieves from the embedded system must always be normalised. To aid users, LExCI comes with software modules that can be used to execute the agent (`neural_network_module`) and to transform quantities between the spaces.

Considering how widely used MATLAB/Simulink are in the engineering domain, especially in control prototyping, LExCI's *RL Block* (Fig. 3) plays a prominent role in that regard. It is a ready-to-use Simulink subsystem that houses the RL-based controller such that employing it becomes as simple as copying it into the plant model, connecting its ports, and setting some basic parameters. Inside, the RL Block min-max normalises observations, feeds them to the policy NN of the agent, samples an action from the inferred action distribution, and denormalises the same before returning it. Its centrepieces are the S-Function containing the C++-code to execute the agent using TFLM and the internal experience buffer which can be accessed via the control software. The RL Block is externally triggered so that the agent can be executed at a different (i.e. slower) sample rate than the surrounding model.

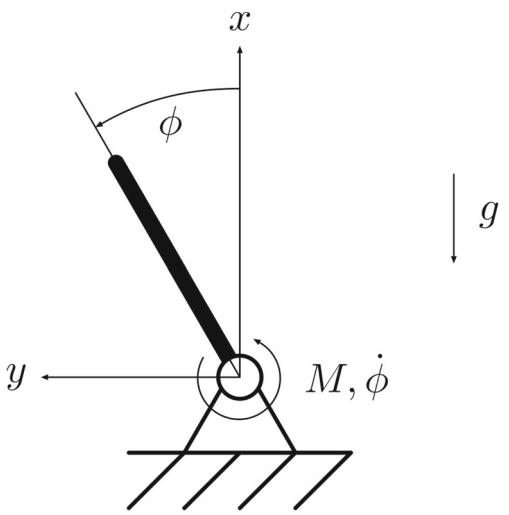


Fig. 4 The pendulum swing-up problem according to [39]

Table 1 The observation and action space of the pendulum swing-up problem

Number	Observation	Minimum	Maximum	Unit
1	x	-1	$+1$	m
2	y	-1	$+1$	m
3	$\dot{\phi}$	-8	$+8$	rad s^{-1}
Number	Action	Minimum	Maximum	Unit
1	M	-2	$+2$	N m

4 Experiments

For this paper, LExCI was applied to the inverted pendulum swing-up problem which is a standard benchmark for continuous control. To highlight its versatility, multiple trainings were performed with the framework, each with a different RL algorithm and target system.

4.1 Pendulum Environment and Setup

In the inverted pendulum swing-up environment, a rod of length $l = 1\text{m}$ and mass[14] $m = 1\text{kg}$ is mounted to a wall on one end with a single rotational degree of freedom (cf. Fig. 4). The objective is to apply a torque M at the pivot point in every time step such that it stands upright, i.e. the angle $\phi \in (-\pi, +\pi]$ between the rod and the vertical axis as well as its angular velocity $\dot{\phi}$ become 0. The time step length is $\Delta t = 0.05\text{s}$. Using $x = l \cdot \cos(\phi)$ and $y = l \cdot \sin(\phi)$, Table 1 summarises the environment's observation and action space while (14) describes its reward function. Episodes are 200 time steps long and start at a random position $\phi_0 \in (-\pi, +\pi]$ and with a random angular velocity $\dot{\phi}_0 \in [-1\text{rad s}^{-1}, +1\text{rad s}^{-1}]$. [38–40]

$$R(\phi, \dot{\phi}, M) = -\phi^2 - 0.1 \cdot \dot{\phi}^2 - 0.001 \cdot M^2 \tag{14}$$

The pendulum problem was tackled three times with LExCI:

Python First, purely in Python using the gym implementation of the environment [38] and LExCI's `neural_network_modules` (cf. Section 3.2) to execute the agent's policy.

Simulink Second, with the pendulum environment running in Simulink using the RL Block (see Section 3.2) and a custom model that is identical in behaviour to gym's implementation.

[14] We chose $g = 9.81\text{m s}^{-2}$ instead of the default value of 10 in [38].

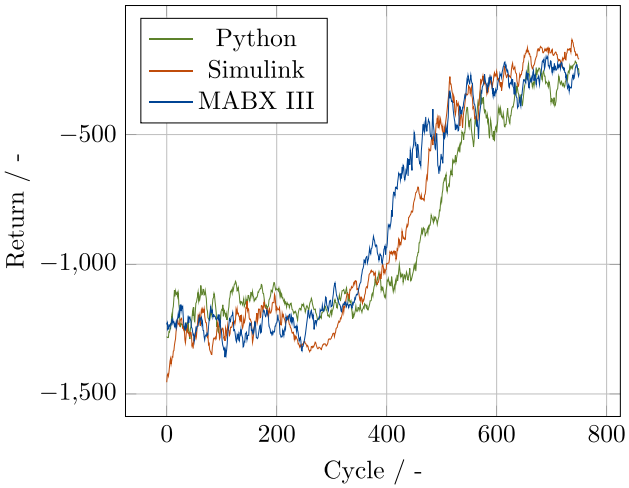


Fig. 5 Average LExCI PPO training returns with three episodes per cycle. The data has been smoothed with a moving average filter of size 11

MABX III Third, with the environment running on a dSPACE MicroAutoBox (MABX) III[15], a RCP system commonly used for embedded control by the automotive industry. This run, too, used a custom model of the pendulum environment and the RL Block (cf. Section 3.2).

With each target system, one agent was trained with PPO and one with DDPG for a total of six training runs. The choice of algorithms was motivated by their widespread use in engineering and the fact that one is on-policy while the other is not. Observations were min-max normalised and the real-valued actions were mapped via a scaled hyperbolic tangent to the boundaries of the environment (see Section 3.2). The hyperparameters were chosen based on RLlib's pre-tuned configurations for the respective algorithms and extended by LExCI's custom ones. Appendix A lists the most important parameters.

Validations were performed every five cycles so that the agent's performance was tested frequently enough without creating too much overhead. For that purpose, the pendulum environment was initialised with $\phi_0 = \pi$ and $\dot{\phi}_0 = 0$ rad s^{-1}, i.e. with the rod hanging still at the six o'clock position.

4.2 Results

Given the definition of the pendulum environment and the hyperparameters that were chosen, three episodes were generated in every cycle. Figures 5 and 6 plot their smoothed average returns over the cycle number while the unfiltered quantities can be found in Figs. 11 and 12 of Appendix B. The

[15] https://www.dspace.com/en/pub/home/products/hw/micautob/microautobox3.cfm

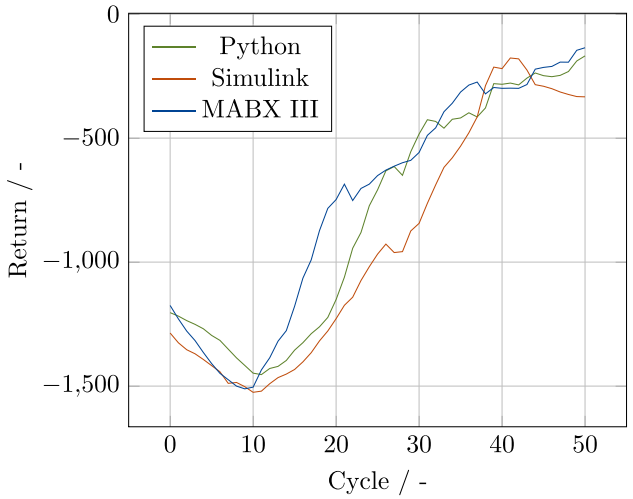


Fig. 6 Average LExCI DDPG training returns with three episodes per cycle. The data has been smoothed with a moving average filter of size 11

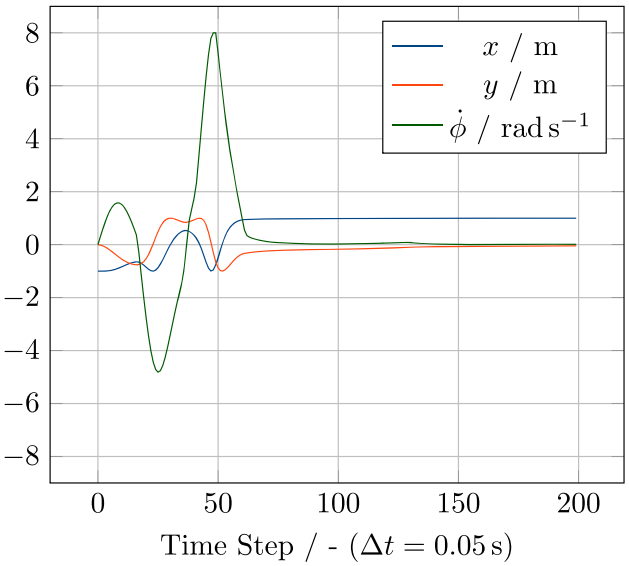


Fig. 8 Best validation at cycle 45 of the LExCI DDPG training with the MABX III. The return of the episode was −367.32

plots show some noteworthy characteristics of the trainings: First, every combination of RL algorithm and target system converged towards the optimum where the agent exhibits good performance. To substantiate this claim, Figure 8 shows the best validation run of the DDPG-training on the MABX III where the agent swings the pendulum to the 12 o'clock position ($x = 1$m and $y = 0$m) within the first 50 time steps (i.e. in just 2.5s) and holds it there for the remainder of the episode ($\dot{\phi} = 0$rad s^{-1}). The same is true for the best PPO validation on that platform (cf. Fig. 7). Other combinations performed analogously once the training had converged (see Figs. 13, 14, 15, and 16 in Appendix B). Please note that the variations in maximum return are merely a result of the stochastic nature of exploration paired with the random initialisation of the environment at the beginning of every episode. They do not mean that one target system performs better than the others. This randomness also entails that each agent finds a slightly different optimum. For example, the angular velocity has higher amplitudes in Fig. 8 than in Fig. 7. Second, all target systems display a similar course of

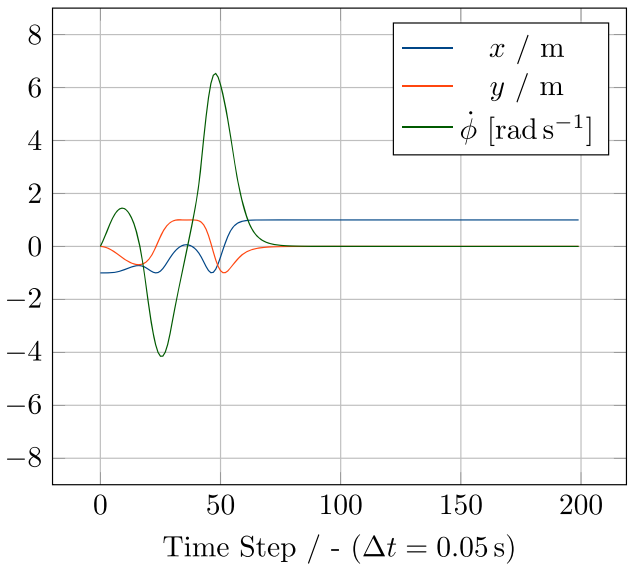


Fig. 7 Best validation at cycle 750 of the LExCI PPO training with the MABX III. The return of the episode was −378.22

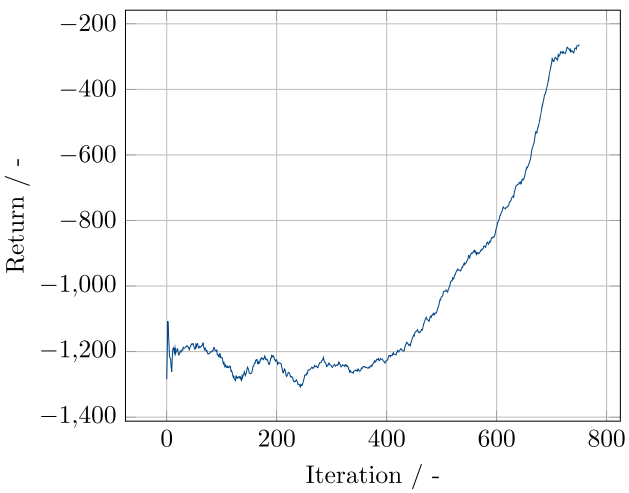


Fig. 9 Smoothed average PPO training returns in native Ray/RLlib

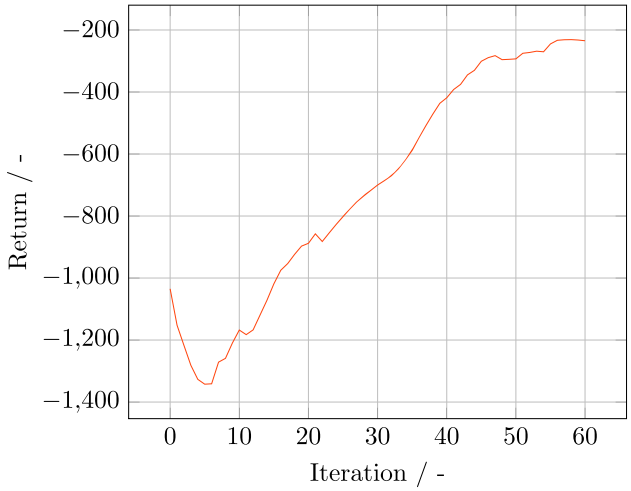


Fig. 10 Smoothed average DDPG training returns in native Ray/RLlib

training progression for each algorithm which proves that i) our Simulink and MABX III models of the pendulum environment are equal to gym's implementation in terms of behaviour and ii) that LExCI is able to train well on various platforms. Furthermore, Figures 5 and 6 are in accord with the results of [41] though the author used a different set of hyperparameters. Third, all agents remain stable after convergence. The oscillations in the average cycle returns are mainly caused by the random initialisation of the pendulum which sometimes starts in more and sometimes in less advantageous states.

To further validate the results, the pendulum environment was also trained without LExCI, i.e. using Ray/RLlib only. These trainings shall be referred to as *native*. For the sake of comparability, the environment was configured such that observations are min-max normalised and actions are mapped with a scaled hyperbolic tangent. When analysing the results in Figs. 9 and 10 and comparing them to the ones above, one has to consider two things: 1) Ray/RLlib's iterations do not directly correspond to LExCI's cycles. Because of that, the hyperparameters from Appendix A had to be slightly varied so as to best replicate LExCI's behaviour. This mainly affected the DDPG settings that govern how many samples are generated and how often replay data is used for training. 2) Native Ray/RLlib utilises an OU process for exploration and not Gaussian noise for DDPG. With that in mind, the average training returns display the same general progress and — more important — have the same minima and maxima: Considering all (smoothed) results, the maximum return averages at −203.5 with a standard deviation of 53.2 for PPO and at −178 with a standard deviation of 39.6 for DDPG. Moreover, the mean standard deviation over the course of the whole training is 130.6 for PPO and 122.4 for 50 cycles of DDPG. These numbers demonstrate that, despite exploration and environment initialisation being stochastic, the trainings closely resemble one another, regardless of the platform. This proves that LExCI interfaces RLlib correctly and that the framework is able to train agents to the same level of quality as the original library setup.

5 Conclusion

This paper explained the importance of RL for developing today's and tomorrow's control functions and highlighted the difficulties engineers face during training and deployment of RL agents with/on embedded devices. The LExCI framework was presented as an open-source solution and its performance has been demonstrated across various target systems, including a state-of-the-art RCP system, for a classic control task. Not only did LExCI succeed in integrating those platforms into the process, the results were also on a par with what the underlying RL library can produce natively on a conventional PC.

The framework enables users to apply RL to real-world engineering problems on professional hardware as has been shown in prior works. Considering that one had to resort to specialised solutions to do so in the past, LExCI facilitates the process many times over because of its generic interface to embedded devices. Moreover, the fact that it relies on free, established libraries means that end-users are neither forced to content themselves with proprietary implementations nor do they have to go through the ordeal of writing and testing the algorithms or data structures themselves. Instead, they can leverage the full expertise of the open-source communities behind said libraries and thus obtain better results. With all that said, it is important to acknowledge that there is some upfront work in the form of writing a Minion that one has to do in order to apply the framework to a new problem. In addition to that, the master-minion architecture introduces overhead and complexity to the training process. While these costs are acceptable in LExCI's intended setting, they may not be in situations where the full feature set of the framework is not required.

In the future, LExCI will be updated to the latest RLlib release as the latter has since undergone a major version change. Additionally, support for more algorithms will be implemented as well as features that aid in exercising advanced techniques. For instance, the framework shall have a more extensive repertoire of TL functionalities.

Appendix A Hyperparameters

Table 2 PPO-hyperparameters used for training the agents in Section 4. All NNs were fully-connected and feed-forward. Values that differ from RLlib's pendulum hyperparameters are printed in bold

Policy NN	**3** × **64** × **64** × **2**, tanh-activated
VF NN	**3** × **64** × **64** × **1**, tanh-activated
Train batch size	512
SGD mini-batch size	64
SGD iterations per batch	6
γ	0.95
λ	0.1
ϵ	0.3
VF clip parameter	**10000**
LR	0.0003
KL target	0.01

Table 3 DDPG-hyperparameters used for training the agents in Section 4. All NNs were fully-connected and feed-forward. LExCI-specific parameters are marked with an asterisk

Policy NN	3 × 64 × 64 × 1, ReLU-activated
Q-function NN	3 × 64 × 64 × 1, ReLU-activated
Replay buffer size	10000
Experiences per cycle*	600
Experiences before replay training*	2400
Percentage of buffer used for replay training*	0.25
Train batch size	64
γ	0.99
LR (policy)	0.001
LR (Q-function)	0.001
Huber threshold	1
ρ	0.001

The hyperparameters used for training the agents presented in this paper are based on RLlib's pre-tuned configurations for PPO[16] and DDPG[17] in the pendulum environment. In the case of PPO, the NN was reduced from the template's original 256 nodes per hidden layer to 64 lest the agent overfit. This is the same value used in the DDPG policy network. When doing so, it proved beneficial to raise the clipping threshold of the VF to match the scale of the rewards[18]. Parameters not specified in Tables 2 and 3 were set to their default values.

Appendix B Additional Training Data

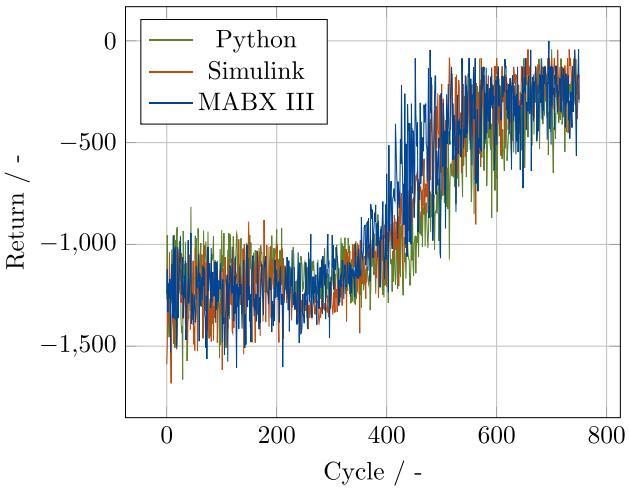


Fig. 11 Unfiltered average LExCI PPO training returns with three episodes per cycle

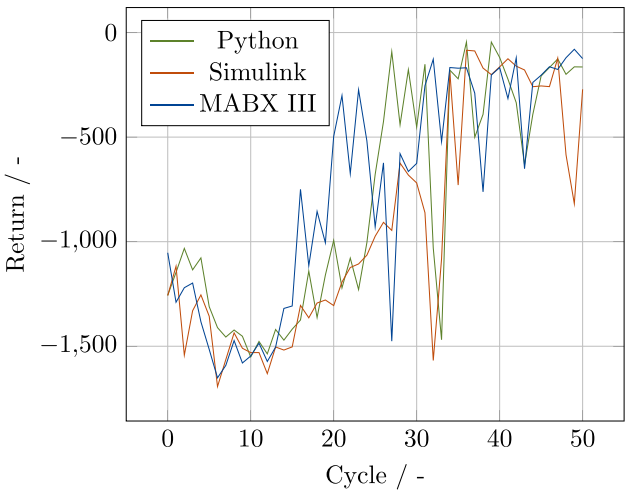


Fig. 12 Unfiltered average LExCI DDPG training returns with three episodes per cycle

[16] https://github.com/ray-project/ray/blob/ray-1.13.0/rllib/tuned_examples/ppo/pendulum-ppo.yaml

[17] https://github.com/ray-project/ray/blob/ray-1.13.0/rllib/tuned_examples/ddpg/pendulum-ddpg.yaml

[18] This optimisation is explained here: https://github.com/ray-project/ray/blob/ray-1.13.0/rllib/agents/ppo/ppo.py

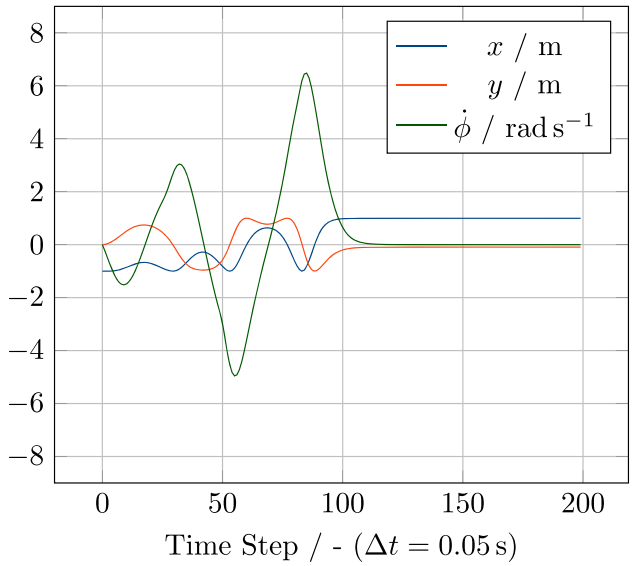


Fig. 13 Best validation at cycle 745 of the LExCI PPO training with Python. The return of the episode was −560.49

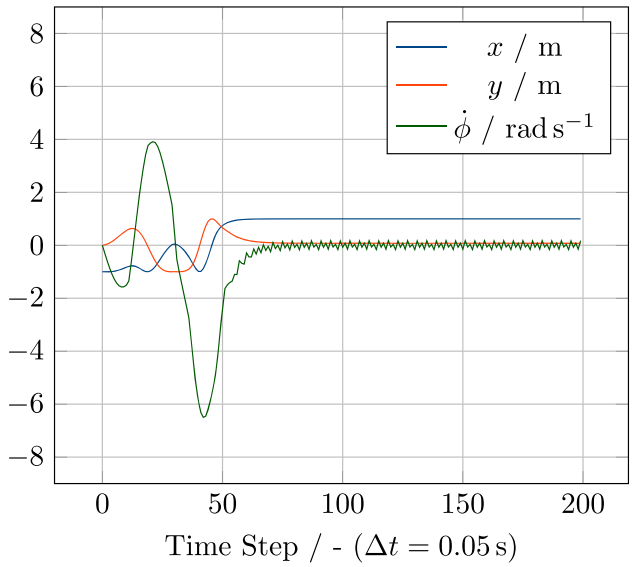


Fig. 14 Best validation at cycle 45 of the LExCI DDPG training with Python. The return of the episode was −348.05

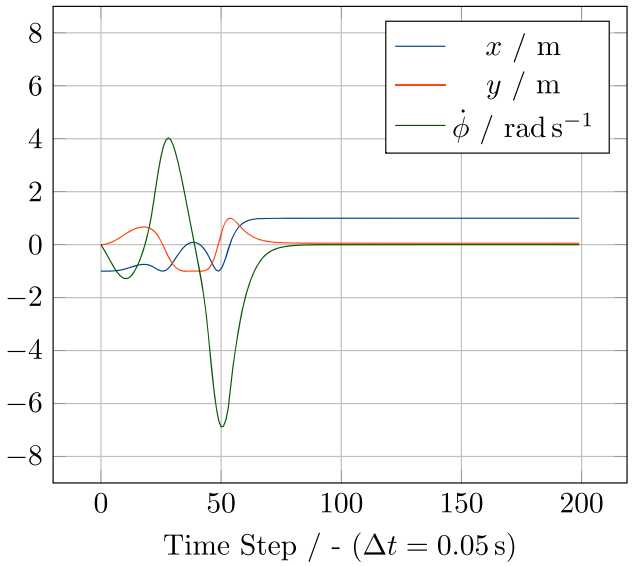


Fig. 15 Best validation at cycle 710 of the LExCI PPO training with Simulink. The return of the episode was −397.86

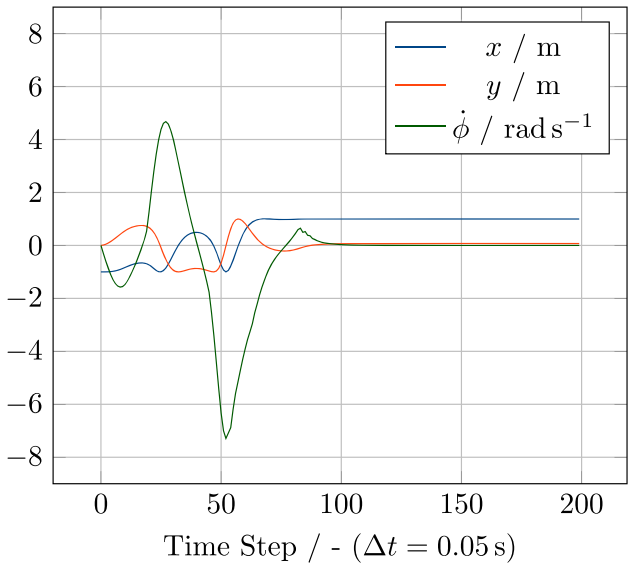


Fig. 16 Best validation at cycle 45 of the LExCI DDPG training with Simulink. The return of the episode was −382.50

Acknowledgements We would like to thank TraceTronic GmbH and our student assistant Abdus Hashmy for their valuable support.

Author Contributions **Conceptualisation:** Kevin Badalian, Lucas Koch, Marius Wegener; **Methodology:** Kevin Badalian, Lucas Koch, Tobias Brinkmann, Mario Picerno; **Software:** Kevin Badalian, Lucas Koch, Tobias Brinkmann; **Validation:** Kevin Badalian, Lucas Koch, Mario Picerno, Tobias Brinkmann; **Formal analysis:** Kevin Badalian, Lucas Koch, Mario Picerno; **Investigation:** Kevin Badalian, Lucas Koch; **Resources:** Jakob Andert; **Data curation:** Kevin Badalian, Lucas Koch, Mario Picerno; **Writing – original draft:** Kevin Badalian, Lucas Koch; **Writing – review & editing:** Lucas Koch, Tobias Brinkmann, Mario Picerno, Marius Wegener, Sung-Yong Lee, Jakob Andert; **Visualisation:** Kevin Badalian; **Supervision:** Jakob Andert; **Project administration:** Jakob Andert; **Funding acquisition:** Jakob Andert

Funding Open Access funding enabled and organised by Projekt DEAL. This work and the scientific research behind it have been funded by the *Hy-Nets4all* project (grant no. EFRE-0801698) of the European Regional Development Fund (ERDF), the Federal Ministry for Economic Affairs and Climate Action (BMWK) through the German Federation of Industrial Research Associations (AiF, IGF no. 21407 N) and assigned by the Research Association FVV as project *Heuristic Search and Deep Learning*, and the *VISION* project (grant no. KK5371001ZG1) of the BMWK on the basis of a decision by the German Bundestag. Work was performed at the Center for Mobile Propulsion (CMP) funded by the German Research Foundation (DFG) and the German Science and Humanities Council (WR).

Availability of Data and Materials See *Code Availability*.

Code Availability LExCI's source code is available in its official GitHub repository:
https://github.com/mechatronics-RWTH/lexci-2
The version used for this paper (including the experiment code, models, and data) can be found in the `lexci_paper` branch.

Declarations

Competing Interests The following could be considered a potential financial interest: Lucas Koch, Mario Picerno, Kevin Badalian, Sung-Yong Lee, and Jakob Andert have a patent pending for *Automatisierte Funktionskalibrierung/Automated Feature Calibration* (patent no. DE102022104648A1/EP4235319A1).

Ethics Approval Not applicable.

Consent to Participate Not applicable.

Consent for Publication Not applicable.

References

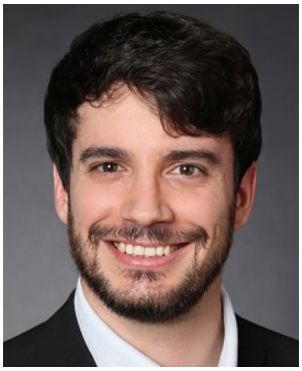
1. Howard J (2019) Artificial intelligence: Implications for the future of work. American Journal of Industrial Medicine. 62(11):917–926
2. Laplante P, Milojicic D, Serebryakov S, Bennett D (2020) Artificial Intelligence and Critical Systems: From Hype to Reality. Computer. 53(11):45–52. https://doi.org/10.1109/MC.2020.3006177
3. Eurostat (2022) Use of artificial intelligence in enterprises. Accessed: 2023-05-15. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Use_of_artificial_intelligence_in_enterprises
4. Grigorescu S, Trasnea B, Cocias T, Macesanu G (2020) A Survey of Deep Learning Techniques for Autonomous Driving. Journal of Field Robotics. 37(3):362–386. https://doi.org/10.1002/rob.21918
5. Branco S, Ferreira AG, Cabral J (2019) Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey. Electronics. **8**(11) https://doi.org/10.3390/electronics8111289
6. Barkalov A, Titarenko L, Mazurkiewicz M (2019) Foundations of Embedded Systems, 1st edn. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-11961-4
7. Moritz P, Nishihara R, Wang S, Tumanov A, Liaw R, Liang E, Elibol M, Yang Z, Paul W, Jordan MI et al (2018) Ray: A distributed framework for emerging {AI} applications. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 561–577
8. Liang E, Liaw R, Nishihara R, Moritz P, Fox R, Goldberg K, Gonzalez J, Jordan M, Stoica I (2018) RLlib: Abstractions for distributed reinforcement learning. In: International Conference on Machine Learning, pp. 3053–3062. PMLR
9. Raffin A, Hill A, Gleave A, Kanervisto A, Ernestus M, Dormann N (2021) Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research. 22(268):1–8
10. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org. https://www.tensorflow.org/
11. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox E, Garnett R (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc., Vancouver, Canada. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
12. Chen Y, Zheng B, Zhang Z, Wang Q, Shen C, Zhang Q (2020) Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions. ACM Comput Surv 53(4). https://doi.org/10.1145/3398209
13. David R, Duke J, Jain A, Reddi VJ, Jeffries N, Li J, Kreeger N, Nappier I, Natraj M, Regev S, Rhodes R, Wang T, Warden P (2020) Tensorflow lite micro: Embedded machine learning on tinyml systems. CoRR. arXiv:2010.08678
14. cONNXr (software) (2019) GitHub. Accessed: 2023-06-30. https://github.com/alrevuelta/cONNXr
15. Genann v1.0.0 (software) (2016) GitHub. Accessed: 2023-06-30. https://github.com/codeplea/genann

16. KANN (software) (2016) GitHub. Accessed: 2023-06-30 . https://github.com/attractivechaos/kann
17. tiny-dnn v1.0.0 (software) (2012) GitHub. Accessed: 2023-06-30. https://github.com/tiny-dnn/tiny-dnn/
18. MiniDNN (software) (2017) GitHub. Accessed: 2023-06-30. https://github.com/yixuan/MiniDNN
19. frugally-deep v0.15.20-p0 (software) (2016) GitHub. Accessed: 2023-06-30. https://github.com/Dobiasd/frugally-deep
20. keras2cpp (software) (2016) GitHub. Accessed: 2023-06-30. https://github.com/pplonski/keras2cpp
21. onnx2c (software) (2020) GitHub. Accessed: 2023-06-30. https://github.com/kraiskil/onnx2c
22. MathWorks (2021) Reinforcement Learning Toolbox (software). Accessed: 2023-06-30. https://www.mathworks.com/products/reinforcement-learning.html
23. Han H, Siebert J (2022) TinyML: A Systematic Review and Synthesis of Existing Research. In: 2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), pp. 269–274. IEEE
24. Hausknecht M, Stone P (2015) Deep Recurrent Q-Learning for Partially Observable MDPs. In: 2015 AAAI Fall Symposium Series
25. Koch L, Picerno M, Badalian K, Lee S-Y, Andert J (2023) Automated function development for emission control with deep reinforcement learning. Eng Appl Artif Intell 117:105477. https://doi.org/10.1016/j.engappai.2022.105477
26. Picerno M, Koch L, Badalian K, Wegener M, Schaub J, Koch CR, Andert J (2023) Transfer of Reinforcement Learning-Based Controllers from Model-to Hardware-in-the-Loop. arXiv:2310.17671
27. Picerno M, Koch L, Badalian K, Lee S-Y, Andert J (2023) Turbocharger control for emission reduction based on deep reinforcement learning. IFAC-PapersOnLine. 56(2):8266–8271. https://doi.org/10.1016/j.ifacol.2023.10.1012. 22nd IFAC World Congress
28. Koch L, Roeser D, Badalian K, Lieb A, Andert J (2023) Cloud-Based Reinforcement Learning in Automotive Control Function Development. Vehicles. 5(3):914–930. https://doi.org/10.3390/vehicles5030050
29. Book G, Traue A, Balakrishna P, Brosch A, Schenke M, Hanke S, Kirchgässner W, Wallscheid O (2021) Transferring Online Reinforcement Learning for Electric Motor Control From Simulation to Real-World Experiments. IEEE Open Journal of Power Electronics. 2:187–201. https://doi.org/10.1109/OJPEL.2021.3065877
30. Plappert M (2016) keras-rl (software). GitHub. https://github.com/keras-rl/keras-rl
31. Szydlo T, Jayaraman PP, Li Y, Morgan G, Ranjan R (2022) TinyRL: Towards Reinforcement Learning on Tiny Embedded Devices. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management, pp. 4985–4988
32. Sutton RS, Barto AG (2018) Reinforcement Learning: An Introduction, 2nd edn. The MIT Press, Cambridge, Massachusetts, USA. http://incompleteideas.net/book/RLbook2020.pdf
33. OpenAI (2018) Spinning Up: Introduction to RL - Part 2: Kinds of RL Algorithms. Accessed: 2023-05-03. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html
34. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal Policy Optimization Algorithms. CoRR. https://arxiv.org/abs/1707.06347
35. OpenAI (2018) Spinning Up: Algorithms Docs: Proximal Policy Optimization. Accessed: 2023-05-03. https://spinningup.openai.com/en/latest/algorithms/ppo.html
36. Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D (2016) Continuous control with deep reinforcement learning. In: Bengio Y, LeCun Y (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. http://arxiv.org/abs/1509.02971
37. OpenAI (2018) Spinning Up: Algorithms Docs: Deep Deterministic Policy Gradient. Accessed: 2023-05-03. https://spinningup.openai.com/en/latest/algorithms/ddpg.html
38. OpenAI (2016) implementation of the inverted pendulum swing-up problem (code). Accessed: 2023-10-12 . https://github.com/openai/gym/blob/v0.21.0/gym/envs/classic_control/pendulum.py
39. The Farama Foundation (2022) Pendulum. Accessed: 2023-07-13. https://gymnasium.farama.org/environments/classic_control/pendulum/
40. Bi Y, Chen X, Xiao C (2021) A Deep Reinforcement Learning Approach towards Pendulum Swing-up Problem based on TF-Agents. arXiv preprint arXiv:2106.09556
41. Kumar S (2021) Controlling an Inverted Pendulum with Policy Gradient Methods - A Tutorial. arXiv preprint arXiv:2105.07998

Kevin Badalian received his B.Sc. degree in computer science from RWTH Aachen University, Germany, in 2020. In the same year, he started working as a programmer at the Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University, where he focuses on simulations and the application of reinforcement learning in automotive engineering.

Lucas Koch received his M.Sc. degree in automotive engineering from RWTH Aachen University, Germany, in 2019. Since 2020, he is a research associate (Ph.D.) with the Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University. His current research interest focuses on the utilisation of reinforcement learning methods to derive sustainable control functions for various applications in the field of automotive engineering.

Tobias Brinkmann received his M.Sc. degree in automotive engineering from RWTH Aachen University, Germany, in 2020. Since 2021, he is a research associate (Ph.D.) with the Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University. His current research interest focuses on the application of machine learning methods in the field of automotive engineering and unsupervised methods for anomaly detection in powertrain systems.



Mario Picerno received his M.Sc. degree in mechanical engineering from the Polytechnic University of Turin, Italy, in 2019. In the same year, he started as a research associate (Ph.D.) with the Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University. His current research interest focuses on Hardware-in-the-Loop simulations and the application of reinforcement learning methods to control development in the field of automotive engineering.

Marius Wegener received his M.Sc. and Dr.-Ing. degree from RWTH Aachen University, Germany, in 2017 and 2022, respectively. His dissertation focused on automated eco-driving in urban scenarios using reinforcement learning. Since 2021, he is a team leader of function development for e-mobility systems with FEV Europe GmbH.

Sung-Yong Lee received his Dr.-Ing. degree from RWTH Aachen University, Germany, in 2023. His research focuses on the Hardware-in-the-Loop based development for mobile propulsion systems. Currently, he serves as the managing chief engineer at the Teaching and Research Area of Mechatronics in Mobile Propulsion (MMP) at RWTH Aachen University.

Jakob Andert received his Dipl.-Ing. and Dr.-Ing. degree from RWTH Aachen University, Germany, in 2007 and 2012, respectively. His dissertation focused on a real-time cycle-to-cycle control of homogeneous charge compression ignition engines. From 2014 to 2021, he was a junior professor of mechatronic systems for combustion engines with RWTH Aachen University. In 2021, he was appointed as a full professor of mechatronics in mobile propulsion. His teaching and research area's portfolio ranges from electrified vehicle drives and predictive powertrain control to the application of artificial intelligence methods, data science, and systems engineering in powertrain development.

Authors and Affiliations

Kevin Badalian[1] · Lucas Koch[1] · Tobias Brinkmann[1] · Mario Picerno[1] · Marius Wegener[2] · Sung-Yong Lee[1] · Jakob Andert[1]

✉ Kevin Badalian
badalian_k@mmp.rwth-aachen.de

Lucas Koch
koch_luc@mmp.rwth-aachen.de

Tobias Brinkmann
brinkmann@mmp.rwth-aachen.de

Mario Picerno
picerno_ma@mmp.rwth-aachen.de

Marius Wegener
wegener@fev.com

Sung-Yong Lee
lee_sun@mmp.rwth-aachen.de

Jakob Andert
andert@mmp.rwth-aachen.de

[1] Teaching and Research Area Mechatronics in Mobile Propulsion, RWTH Aachen University, Forckenbeckstraße 4, Aachen 52074, NRW, Germany

[2] FEV Europe GmbH, Neuenhofstraße 181, Aachen 52078, NRW, Germany