TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Mark Sisin 201628IASM

Practical Implementation of “Sim-to-Real” Deep Reinforcement Learning Control for Inverted Pendulum System

Master’s Thesis

Supervisors

Saleh Ragheb Saleh Alsaleh

Early Stage Researcher

Aleksei Tepljakov

Senior Researcher

TALLINN 2022

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Mark Sisin 201628IASM

Kinnitusega süvaõppe rakendamine pöördpendli süsteemi juhtimiseks

Magistritöö

Juhendajad

Saleh Ragheb Saleh Alsaleh

Nooremteadur

Aleksei Tepljakov

Vanemteadur

TALLINN 2022

Declaration of Originality

Declaration: I hereby declare that this thesis, my original investigation and achievement, submitted for the Master's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Deklareerin, et käesolev diplomitöö, mis on minu iseseisva töö tulemus, on esitatud Tallinna Tehnikaülikooli magistrikraadi taotlemiseks ja selle alusel ei ole varem taotletud akadeemilist kraadi.

Mark Sisin

Date: May 24, 2022

Signature: ....

Abstract

The following work describes how it was possible to use “sim-to-real” learning to design the deep reinforcement learning controller for the inverted pendulum. The “sim-to-real” learning is the branch of transfer learning that deals with transmitting knowledge from virtual environments to real-world applications. By following the “sim-to-real” methods, the controller for the inverted pendulum has been created from experience obtained from its digital twin. After optimizing the dynamics of the virtual model using the System Identification method, it was possible to reduce the training time for the physical twin by three times compared to the controller trained without prior experience. The paper also shows the suggested design of a deep reinforcement learning platform, a software library that aids in connecting the reinforcement learning training agent with control objects that lack a direct mechanism of communicating observations with the training agent. Throughout the research document, the reader will get familiar with all the technical steps required to achieve a successful “sim-to-real” experiment.

The thesis is in English and contains 86 pages of text, 8 chapters, 14 figures, 7 tables.

KEYWORDS: deep reinforcement learning; Simulink; “sim-to-real” learning; inverted pendulum;

Nomenclature

AI	Artificial intelligence
API	Application programming interface
DDPG	Deep deterministic policy gradient
DRL	Deep reinforcement learning
GAE	Generalized Advantage Estimate
GUI	Graphical user interface
MDP	Markov Decision Process
PID	Proportional-integral-derivative
PPO	Proximal policy optimization
PWM	Pulse-width modulation
RL	Reinforcement learning
RMSE	Root-mean-square error
SOTA	State-of-the-art
TCP	Transmission control protocol
TRPO	Trust Region Policy Optimization
UDP	User datagram protocol

Contents

List of Figures

List of Tables

1. Introduction

Nowadays, the revolutionary idea of digital transformation of the industry has shown a significant impact in real life [2]. The reduction of machine downtime, a decrease in maintenance costs, and the overall increase of productivity of technical professions are expected to come with the digitation of industrial processes. The described trend is named Industry 4.0, and many scientific fields are trying to adopt the benefits of this transformation. For instance, the domain of Systems and Controls has recently become an irreplaceable part of the design of the complex engineered systems due to its coupled usage with virtual environments [3]. In various control applications, scientists constantly utilize mathematical models, as they can provide the understanding behind the system's dynamics without having an actual physical system in first hands. But in present times, as the technology advances and the complexity of industrial processes increases, the problem of having the wide gap between physical and digital systems has become sharper than ever before. The same authors have classified this issue as data-driven modeling and control, which is currently one of the most significant innovation challenges in the Systems and Controls domain.

The challenge of data-driven modeling has already been addressed in different use cases. One of such novel concepts in the field of Systems and Controls is the digital twin, a virtual representation of the real-life entity or process at a certain fidelity [4]. For example, this article [5] shows the digital twin prototype of the multi-tank system in Extended Reality. The authors tuned the proportional-integral controller for the digital twin during the experiment. At the same time, the actual tank experienced the changes in water level according to the parameters set in the virtual environment. As a result of this experiment, it was possible to control the tank's liquid level by interfacing the real object through its digital twin. This allows to perform the laboratory training and controller tuning in Extended Reality without directly interacting with the physical system of interest.

Suppose a sufficiently accurate digital twin of the system exists. In that case, it can help address the problem of data-driven control, where the tuning of the controller can be done purely in a virtual environment. For instance, such tuning can be done with the help of reinforcement learning (RL), a branch of machine learning that uses the training agent to interact with a specific environment [6]. Based on the results of interaction with the environment, the RL learns a policy, the particular set of rules that indicate the best action for the training agent to take at the current state of the environment. The exciting achievements in the field of RL, such as the AlphaGo [7] algorithm for solving one of the most challenging classic games for artificial intelligence, increased the popularity of the deep reinforcement learning (DRL) in various fields, like games, robotics, and computer vision to name a few [8]. Referencing the same paper, deep reinforcement learning is a particular branch of reinforcement learning, where policy learning is linked with the deep learning principles. One of the main advantages of the DRL is model-free controller design, so the user does not need to know the full dynamics of the system of interest. Also, the utilization of neural networks allows the DRL to operate the complex nonlinear system, as the neural networks can adapt to particularly complicated dynamics. There are numerous examples of DRL usage in complex control applications [9, 10, 11]. The referenced papers use robotics simulators to train the controller in pendulum stabilization, mapless navigation, and automated liquid pouring. The model-based control approach for mentioned use cases would require understanding sophisticated robot dynamics, which actually made the model-free DRL controller an excellent choice for operation.

However, the main advantages of this control method carry along with the significant drawbacks that limit the use for real-life control applications [8]. The DRL is not sample efficient, and it requires a lot of training data to design the optimal controller. The same authors claim that it is often impossible to afford that for real-life control objects due to safety, time, or cost constraints. But this limitation can be resolved by using the transfer learning methods. The tuning of the controller on the physical device from experience perceived in a digital environment is called the "sim-to-real" learning [12]. The "sim-to-real" learning is the sub-field of transfer learning that identifies the methods that help to use the digital twins to control real equipment. Once training of the controller finishes on a sufficiently accurate virtual model, the "sim-to-real" methods allow this controller to be seamlessly attached to an actual object. As virtual environments represent an infinite source of experience for DRL algorithms, the issue of excessive use of training data could be resolved. Thus, the "sim-to-real" learning correlates with the data-driven control innovation challenge in the Systems and Controls domain. However, referring to the same authors, the "sim-to-real" learning in the DRL field contains many open problems currently under active research. The main problem is that for successful transfer learning, the digital twin of the control object has to be sufficiently accurate. But accuracy of the mathematical model depends on the use case. There are no uniform steps that would always guarantee the reduction of the gap between the physical and digital worlds. An additional problem, explicitly related to the DRL and control applications, highlights no common interface of interaction between the training controller and the control objects. Generally, the DRL training happens in Python, and it can not communicate with control objects without the custom integration script that creates a link between them.

In the current research, the DRL controller for a cart-pole-based inverted pendulum system has been designed with the help of "sim-to-real" learning methods. The created DRL model gained an experience from the digital twin of the pendulum in order to reduce the training time on the actual device. To enable the communication between the controller and digital and physical twins of the pendulum, the unique software bridge was created for sharing the data between the Python code and control objects. This bridge has been called the deep reinforcement learning platform, and it is available as the separate Python package [13].

1.1. Problem statement

The limitations of the DRL in the field of "sim-to-real" learning are conditioned by multiple open issues [12].

The main problem is that the knowledge transfer from a virtual environment to a physical world is bound by the misalignment in systems' dynamics. Generally, digital twins can represent the dynamics of a physical twin with a certain level of fidelity, but transferring the control from the digital world to the real object requires very accurate mathematical representations. Though the guideline methods for solving this issue are available, the solution to the "sim-to-real" problem is always specific to a use case.

Moreover, the recent state-of-the-art (SOTA) training algorithms for the DRL are only available in Python due to dominant computational backbone libraries, which are generally used for every machine learning problem. However, the control objects are usually not operated through Python. Their interface for operation varies depending on the manufacturer. For example, the control object can be operated in Simulink software, a programmable logic controller, or in any other form supported by the manufacturer. Thus, there is no direct integration between the Python code and interfaces for operating control objects. It means that DRL training for actual equipment requires custom integration scripts, which are also developed for a specific use case.

1.2. Contribution

The main contribution of this research is the creation of the DRL controller for the inverted pendulum that used the knowledge obtained from the digital twin using the “sim-to-real” learning. The virtual model of the pendulum was optimized by employing the System Identification method to achieve success in transfer learning. As a result of knowledge transfer, the training time for the physical system of interest has been reduced by three times, compared to the controller trained on the pendulum without applying the “sim-to-real” methods.

The training of the DRL controller has been performed in Python, though the interface for operating digital and physical pendulums was in Simulink software. The current work has proposed a special software bridge called the deep reinforcement learning platform to share the data between the training algorithm and the system of interest. The DRL platform enabled the data transfer between the controller, which was trained in Python, and the operating interface of the pendulum, available in the Simulink application. This platform is available as a separate Python package, and it can be applied to any control object whose operating interface supports networking protocols.

1.3. Thesis outline

The paper is organized in the following way. In Section 2, the literature review has been provided. The study describes the recent advancements of DRL in a domain of control and its status in a field of “sim-to-real” type of transfer learning. In Section 3, the most crucial paper methodology is given. In Section 4, the design for the DRL platform is proposed, and the architecture principles are described and explained by bringing along a sample example. In Section 5, the experimental setup of the system of interest with the DRL platform is described. In Section 6, the controllers for digital and physical twins of the system of interest are trained. Then, it explains how it was possible to optimize the virtual model and build a new type of controller to solve the “sim-to-real” problem. In Section 7, the discussion about the benefits of the DRL platform, its problems, and the next steps are present. Finally, in Section 8, the paper is concluded.

2. Prior work

2.1. Application of deep reinforcement learning in the control applications

The practical application of reinforcement learning in simple control use cases has been thoroughly investigated in [14]. According to the research, there were fundamentally two groups of methods when dealing with control theory: artificial intelligence (AI) and adaptive dynamic programming. Unfortunately, both can not guarantee an optimal solution and robust solution. However, a significant scientific effort has recently been spent in DRL to solve these issues.

The main benefit of reinforcement learning is that it does not require a complete understanding of the process dynamics. Hence, it is model-free compared to the most dynamic programming methods used to create the controllers. The same article [14] also states that nowadays, temporal-difference learning is the most popular way to design the RL control for discrete-time systems with finite state and action spaces. Temporal-difference learning, which is usually called Q-learning, has been showing good results at various control optimization objectives because of the reinforcement learning's ability to explore and exploit an unknown environment. In [15] the Q-learning was used to control the liquid level in a conical tank system. The authors have split the state and action space at finite-space metrics and conducted the model training on a tank simulation in a MATLAB environment. The trained model has been applied to a physical tank system, where the produced Q-map of the action-state reward table was optimized in a runtime. In the end, the Q-learning controller achieved the needed liquid level with accepted accuracy of 1cm below and above the setpoint.

However, the temporal-difference learning and conical tank system described above have specific problems. First of all, temporal-difference learning can not guarantee the stability requirement for the system control [14]. Secondly, more complex control objectives do not have the finite-space representation of action and state spaces. Thus, Q-learning can not be applied to every controller design, requiring a novel control approach.

The DRL is the branch of reinforcement learning that uses the neural network as function approximations to find the optimal policy for taking actions with the best long-awaited reward. One of the first DRL algorithms, namely Deep Q Network, was utilizing the neural network for solving the limitation of finite state-space constraints existing in temporal-difference learning. However, the limitation of finite action space has not been addressed. Of course it is possible to discretize the action space, but this approach leads to a curse of dimensionality and possible loss of action space dynamics. Paper [16] proposes a new deep reinforcement learning algorithm for continuous action systems control to deal with action and state-space limitations. The deep deterministic policy gradient (DDPG) algorithm was designed to deal with continuous state and action space representations. The DDPG algorithm uses batch normalization and actor-critic architecture to solve complex nonlinear systems. The main disadvantage of this method appeared to be the long training episodes, but in the end, the control performance can outstand the drawback. This is the novel control approach that allowed to address the problems the standard Q-learning couldn't solve.

To bring specific examples with DDPG use cases, the article [17] develops a self-organizing controller for tracking purposes. Two controllers have been designed to address the optimality problem, one to move the pendulum in a top-most position and the other to keep it stable at a required angle. The effectiveness of the system has been proven with simulation experiments. The paper [18] uses DRL with several training algorithms, including DDPG. The experiments have been performed within a virtual environment representing the robots' digital twins. What appeared is that the DDPG algorithm was able to control the robot gripper with two parallel fingers but was not able to operate with the robot that represents a human arm stably. The article's authors have used DDPG with the experience replay technique that utilizes a memory buffer during the training to address this issue. The DDPG with experience replay converged the cost function during the training and passed the validation with cube manipulation in a human arm robot. A similar approach of inducing a DDPG controller with experience replay is described in [19] for path tracking of the unmanned surface vehicle. This approach was not only trained virtually, but was also tested in a real marine environment. As a result, the DDPG controller with experience replay was able to resolve the stability issues during the validation in a real environment. The trained model was compared with a heuristically tuned proportional-integral-derivative (PID) controller, which assumes the user knows the system's dynamics. The resulting mean error between them and the required objective goal was optimal for the control objective and was not significantly different. It means that a controller without comprehension of system dynamics was performing with the same quality as a controller where the system dynamics have been identified, making DRL a better and faster approach for the design of the controller.

The DDPG is not the only DRL used in control domain use cases. Besides it, the PPO has been used to create the controller for the stabilization of the quadrotor [20]. The unmanned aerial vehicles are inherently unstable systems, and the PPO was able to create the control policy to hold the quadrotor in a stable position after 7.5 million training timesteps. The controller held the vehicle close to the setpoint in two scenarios: static setpoint and dynamic setpoint that moves along the specified trajectory. The biggest issue appeared to be a steady-state error between the target position and the stabilized position of the quadrotor. A new reward strategy was supposed to be applied in the next iteration of the controller to resolve this issue. The policy has been converged only in a virtual framework, and the experiment has not been conducted on actual equipment. However, the authors state that the simulation environment had complex dynamics engines.

Thus, the abovementioned examples prove the applicability of DRL in the control theory domain. As observed, some authors have based their experimental results purely on the results in a virtual environment. Other contributors were forcing the conclusions from the control outcomes on actual equipment. But there were some articles where the training was based on the virtual and physical environments. Such an interesting pipeline of training and inference is a particular type of transfer learning called "sim-to-real", which appears to be one of the research focuses in the domain of modern DRL [6].

2.2. Deep reinforcement learning in the domain of transfer learning

The paper [12] describes the domain of "sim-to-real" training and inference of DRL models. Based on the research results, the authors have stated that simulation environments have been constantly utilized for training, especially in the robotics domain. The paradigm of digital twin training provides a potentiality of infinite data sources and alleviation of safety constraints. However, nowadays, the "sim-to-real" approach contains the gaps that decrease the performance of inference of the trained model. The major problem appears to be the absence of standard transfer learning algorithms that need to be applied whenever the trained controller has to be used in an actual station, whose dynamics are different from the simulation environment. The more realistic the simulation is to real life, the better appear to be the inference results. For this reason, the most commonly identified techniques are systems identification, domain adaptation, and domain randomization, which help increase the realism of the virtual environment.

But there is also one problem that arises due to “sim-to-real” training. There is no standard environment for simulation execution for different use-cases, would it be a complicated robot-arm manipulation or a simpler control objective, such as a pendulum. All simulations have to be software compatible for training the DRL model. Some simulation frameworks, such as MuJoCo and PyBullet, represent the broader integration with RL environments [12]. However, it is unclear what an available set of training algorithms in these integration modules and how to use the most recent SOTA algorithms available in DRL libraries, such as Stable–Baselines, Tensorforce, or RLLib.

One of the commonly used simulation environments for control applications outside of the robotics domain is Matlab/Simulink. Simulink does not provide direct integration with DRL libraries containing SOTA training algorithms. This article [21] explains how the user can design a controller from the custom OpenAI Gym environment. But in the case of this article, the training procedure has been implemented within the internal Simulink RL toolbox. Currently, there is no available solution for using popular DRL libraries to train the model in the Simulink environment.

Thus, the following research gaps for transfer learning in the domain of the DRL were identified:

- There are no general rules for training once the system of interest is being run in an external application. People spend their time creating custom integration scripts that work only per a specific use case without considering any general approaches.
- The issues above underline the difficulty of the “sim-to-real” approach in the design of the DRL controller. Suppose there is no general standard for the DRL environment and integration with the external applications. In that case, it might not be possible to obtain a controller that can be seamlessly switched

between digital and physical twins.

It is essential to look at the problem of “sim-to-real” training from a generalized perspective. Despite the software where the control system is being executed, all of the use cases should be compatible with popular libraries in the RL domain. The digital-twin-based training should be done in an ad-hoc solution where you can use the DRL model with a specific algorithm despite the external application. With such design, the researchers can sustainably use popular libraries without creating custom integration scripts or training loops for particular algorithms.

Thus, to address those gaps, the special design of the DRL platform is suggested in this paper. This software abstraction creates a coupling between the RL environment and the external application where the system of interest runs. Using this platform, it was possible to develop the DRL controller for the digital twin of the cart pole-based inverted pendulum, which interface of interaction is given in Simulink. Then the platform was used in the design of the DRL controller for a real device from the same use case. Finally, the designed controllers allowed us to find the dynamics' misalignment in digital/physical twins. Thus, after optimization of the virtual model, it was possible to reduce the training time for the physical controller by three times due to the application of the “sim-to-real” knowledge transfer from the virtual controller.

3. Methodology

In this section, the main concepts, algorithms, and mathematical derivations used in the thesis are introduced to the reader. Firstly, the reinforcement learning methodology description is given along with the terminology in this domain. Secondly, the Systems Identification preliminaries are given, which take into account the concepts of Open-Loop identification and optimization technique, used to reduce the misalignment between digital and physical twins of the pendulum.

3.1. Reinforcement learning preliminaries

The classic RL problem lies in an interaction of the agent with an environment over time[6]. At each time step t, the training agent takes action a_t from the action space $\mathcal{A}$ based on the environment's observation s_t from the observation space $\mathcal{S}$. The illustration of this problem is given in Fig. 1 to understand the sequence of actions and state observations.

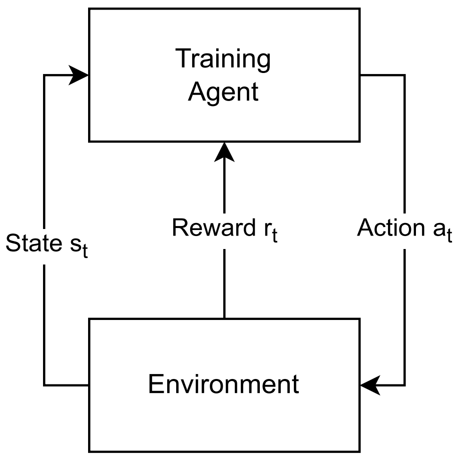


Figure 1. The schematic representation of the classic RL problem. Based on the Environment state and reward, the Training Agent decides the action to pick.

The selection of action by the training agent is performed through the control policy $\pi(a_t|s_t)$. The policy determines the behavior of the agent and the probability $\mathcal{P}(s_{t+1}|s_t, a_t)$ of taking action a_t given the current environment state s_t in order to transition to a next state s_{t+1}. Throughout the training episode, the agent receives the scalar reward r_t after performing the action-state transfer. The reward is the key metric that shows how well the controller can select the most suitable action given the current environment state. The main objective of the agent is to obtain the best control policy $\pi^*(a_t|s_t)$ that provides the highest episodic reward at time step t given as follows:

$$R_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

where γ is the discount factor for the future rewards, and k is the episode time step. Given the $k = 0$, we obtain the instantaneous reward R_{t+1} after performing an action a_t at the current time step t. Depending on the discount factor, which takes a range $\gamma \in [0, 1]$, we value either spontaneous or far-sighted rewards.

While the goal for all agents is to maximize the episode rewards, the ways of achieving that depend on the training algorithm.

3.1.1. Value function

The value function identifies the total possible reward$R$ for the environment state s if the training agent follows the control policy π. This metric is essential for optimizing the loss function in many training algorithms, such as Proximal Policy Optimization. Mathematically, the value function can be defined as follows:

$$V_\pi(s) = E_\pi(R_t \mid s_t) = E_\pi(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid s_t = s) \tag{2}$$

where E_π is the expected value of value function given the training agent is following the policy π at given time step t.

3.1.2. Generalized Advantage Estimate

Overall, the advantage function shows how good or bad was a taken action a_t compared to the state value function given in Eq. (2). Mathematically, it can be expressed in the following way:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{3}$$

and

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s, a_t = a\right] \tag{4}$$

where A_π is the advantage function and Q_π is the action-value function. The main disadvantage of the current representation of advantage function is that it is biased only to the next state value function after performing the action. Ideally, the user needs to have a possibility to determine how far the advantage function will look ahead during the optimization of the policy. This possibility is being provided by Generalized Advantage Estimate (GAE) given as follows:

$$\hat{A}_{GAE}(\gamma,\lambda) = \sum_{k=0}^{\infty}(\gamma\lambda)^k \delta_{t+k} \tag{5}$$

and

$$\delta_t = R_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \tag{6}$$

where δ_t is the temporal difference error, γ and λ are discount factor and lambda GAE coefficient respectively. The lambda coefficient allows the user to select how important the further steps of temporal difference errors are compared to the one-step temporal difference error.

3.1.3. Deep reinforcement learning

The DRL is used in a context of large state and action spaces, where it becomes unfeasible to store all possible state-action transitions in the lookup table. Neural networks are used for the approximation of policy $\pi_\theta(a \mid s)$, value function $\hat{v}_\theta(s)$ and the model (state transition and reward functions) [6]. The update of the network weights θ is performed through series of stochastic gradient descents.

3.1.4. Proximal Policy Optimization

The Proximal Policy Optimization (PPO) [22] is the training algorithm that utilizes the clipped surrogate loss function for an update of neural network weights θ:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right] \tag{7}$$

where $\hat{\mathbb{E}}_t$ is the empirical expectation over a finite batch of samples at timestep t, π_θ is the controller's policy given the weights θ, $c_{1,}c_2$ are coefficients, S denotes the entropy bonus for ensuring sufficient exploration, and $L_t^{VF}(\theta)$ is squared-error loss of the value function $\left(V_\theta\left(s_t\right) - V_{\theta_{old}}\left(s_t\right)\right)^2$ to ensure that GAE identifies the advantage increase from the action side and not changed value state function. The significant part in shaping the optimal weights for the controller policy is the clipped policy surrogate loss function $L_t^{CLIP}(\theta)$ given as:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(R_t(\theta)\hat{A}_t, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{8}$$

and

$$r_t(\theta) = \frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \tag{9}$$

where $\hat{A}_t$ is the GAE given in Eq. (5) and ϵ is the hyperparameter.

3.2. System Identification preliminaries

The System Identification field deals with mathematical representations of the real systems based on measured data. In this paper, the open-loop identification technique has been applied for digital and physical twins of the pendulum to obtain the error

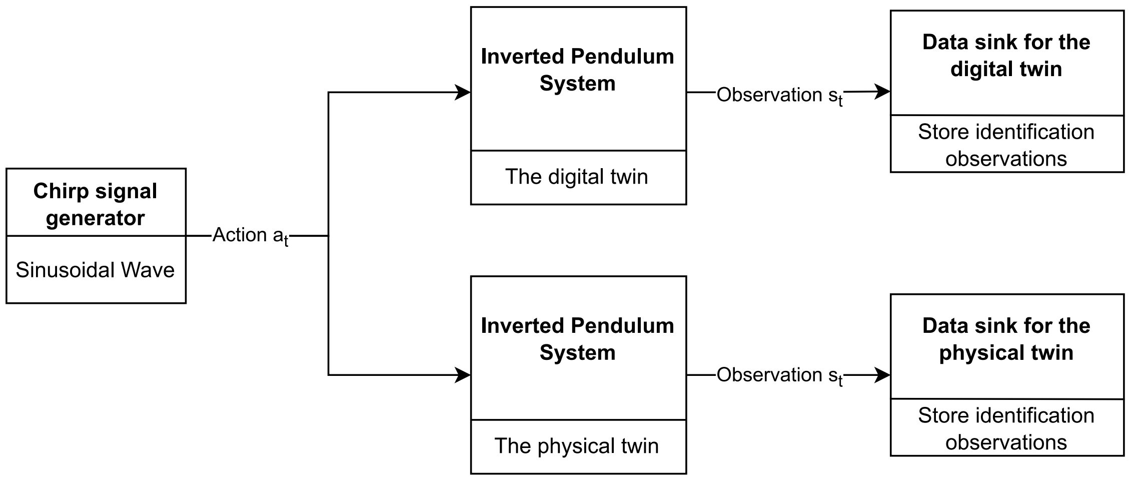


Figure 2. Open-Loop Identification of the inverted pendulum system. The Chirp signal generator provides the sinusoidal action input to both physical and digital twins of the model at the same time. As models are executed simultaneously, the time alignment of the output data is guaranteed. Each twin outputs the observation states to the data sinks, which allow storing identification data for the future optimization experiments.

in dynamics. Nextly, the obtained data was used during the optimization, which reduces the difference in dynamics between virtual and physical models.

3.2.1. Open-Loop Identification preliminaries

The Open-Loop Identification technique allows gathering output from multiple systems executed in parallel against the single action input. Usually, the action input signal is chosen so that it excites the dynamics of the systems of interest. In the case of this paper, the input chirp signal with a sinusoidal wave of varying frequency was chosen to excite the dynamics of digital and physical twins of the inverted pendulum. The detailed scheme of Open-Loop Identification is given in Fig. 2.

3.2.2. Optimization preliminaries

In this work, the problem of dynamics difference between the digital and physical twins of inverted pendulum has been addressed. To minimize the error in behavior for mathematical approximation of the system of interest, it was decided to use the Nelder-Mead optimization algorithm. It is a non-gradient method that optimizes the functions with unconstrained multiple input variables [23]. The algorithm creates the simplex, the convex geometrical figure with $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n+1}$ vertices that tries to find the optimal value of the cost function defined as follows:

$$\min F(x) \tag{10}$$

where $F : \mathbb{R}^n \to \mathbb{R}$ is the cost function having n input parameters. The main advantage of the Nelder-Mead algorithm is that it is a direct search method, which is "derivative-free" [24]. It does not construct the approximations of function F and does not require its derivative information. The Nelder-Mead algorithm finds the optimal set of n input parameters by iterating through the specific set of operations: order, reflection, expansion, contraction and shrinkage [25].

1. **Order**. The first $n+1$ vertices are ordered, so that $F(x_1) \leqslant F(x_2) \leqslant \cdots \leqslant F(x_{n+1})$. Because we solve the minimization problem, x_1 is referred as the best vertex, x_{n+1} is the worst vertex and x_n as the next-worst vertex.
2. **Reflection.** Compute the reflection point x_r

$$x_r = (1+\rho)\overline{\mathbf{x}} - \rho\mathbf{x}_{n+1} \tag{11}$$

   where $\overline{\mathbf{x}} = \sum_{i=1}^{n} \mathbf{x}_i/n$ is the centroid of n best vertices except the x_{n+1}, and ρ is the reflection coefficient. If $F(x_1) \leq F(x_r) < F(x_n)$, then accept the reflected point x_r and terminate the optimization iteration

3. **Expansion.** If the cost function at reflected point is lower that the cost function at the best vertex, expansion point is computed x_e

$$x_e = (1+\rho\chi)\overline{\mathbf{x}} - \rho\chi\mathbf{x}_{n+1} \tag{12}$$

   where χ is the expansion coefficient. If $F(x_e) < F(x_r)$, accept expansion point and terminate the optimization iteration. Otherwise, select the reflection point and terminate the iteration also.

4. **Contraction.** If $F(x_r) \geq F(x_n)$, perform the contraction between $\overline{\mathbf{x}}$ and the better of x_{n+1} and x_r.

   (a) **Outside Contraction.** If $F(x_n) \leq F(x_r) < F(x_{n+1})$, calculate the outside contraction point

$$x_c = (1+\rho\gamma)\overline{\mathbf{x}} - \rho\gamma\mathbf{x}_{n+1} \tag{13}$$

where γ is the contraction coefficient. If $F(x_c) \leq F(x_r)$,accept x_c and terminate optimization iteration. Otherwise, perform the shrinkage step.

(b) **Inside Contraction.** If $F(x_r) \geq F(x_{n+1})$, calculate the inside contraction point

$$x_{cc} = (1-\gamma)\overline{\mathbf{x}} + \gamma \mathbf{x}_{n+1} \tag{14}$$

If $F(x_{cc}) < F(x_{n+1})$, accept x_{cc} and terminate optimization iteration. Otherwise, perform the shrinkage step.

5. **Shrinkage.** Evaluate the cost function F at new set of vertices from

$$x_i = x_1 + \sigma\left(x_i - x_1\right), \quad i = 2, \ldots, n+1 \tag{15}$$

where σ is the shrinkage coefficient.

The reflection, expansion, contraction and shrinkage coefficients should satisfy

$$\rho > 0, \quad \chi > 1, \quad 0 < \gamma < 1, \quad 0 < \sigma < 1 \tag{16}$$

4. The proposed deep reinforcement learning platform

4.1. Implementation architecture

As described in previous paragraphs, the “sim-to-real” domain contains the problems with control objects running in applications that do not directly connect to the training loop. It means that some software platform needs to be created that can guarantee the seamless controller transfer between the physical and digital world. One can provide the digital twin in differential equations that can be converted into a script within a programming environment. Other digital twins can be received in the form of a black box with defined inputs and outputs. For example, MATLAB or Unreal Engine can encapsulate the digital twin in the platform's specific container so that only defined inputs and outputs are available through an environment. In addition, suppliers of complex industrial equipment can provide embedded simulators, usually digital replicas with inputs and outputs, and even an application programming interface (API) that is used to speak with a digital twin or a real industrial asset. For instance, Universal Robot provides the possibility to deploy the simulator in a virtualization environment [26]. The user can communicate with the simulator in the same manner as with physical robotic arm. This simulator supports the entire API used with a real robot and even the graphical user interface (GUI) for configuration and control.

Thus, especially considering the last two examples in the paragraph above, the implementation architecture of the DRL platform must somehow be injectable for variety of applications. Changing the digital twins to one specific format supported by the DRL platform would not be possible. Hence, the digital twins should never be altered to support the principle of the platform generality which tries to be achieved. However, the digital twins in various applications can not magically speak with the DRL platform without any modification. That's why, still, minor adaptation has to be done to an application to be able to speak with the DRL platform.

The special data hooks and data hubs were designed to address the issue of information propagation to the DRL platform. This way, the DRL platform does not change the behavior of the digital twin but will still be able to observe the state and action spaces of the object of interest. It is enough for reinforcement learning to observe the digital twin's environment with all necessary twin outputs to provide robust control with predefined inputs.

The other point the architecture has to support is the possibility of training the controller using various SOTA algorithms. Implementing different training algorithms from scratch would make the timespan of the following research too extensive. The DRL platform has to provide only the environment integration, not the training agent itself.

Generally, the proposed architecture demonstrated in Fig. 3 should be achieved. One of its main benefits is the injectability for variety of applications and DRL libraries.

For the platform to be able to speak with the control object, the control object interface should implement Data hub and Data hook for the information trespassing. These blocks should be able to accept the actions produced by the controller and output the observations for the controller to make a subsequent action. For example, the Data hub can be transmission control protocol/user datagram protocol (TCP/UDP) server accepting the inputs on a specific port, the Data hook can be TCP/UDP client that sends the observations to the DRL platform. To connect with Data hub/hook, the platform implements an Environment Server. This particular object holds the connection with an external application for propagating actions and forwarding observation for the Platform Environment.

The Platform Environment is the RL environment for training execution. The RL environment allows the DRL controller to observe the system's outputs, provide inputs for action, and get the reward per episode step. The DRL platform offers compatibility with OpenAI Gym. To understand why it was essential to use the solution provided by OpenAI, it is important to know why the Gym has been created in the first sense.

The OpenAI Gym environment is a software abstraction for the RL environments developed by the OpenAI. Initially, the OpenAI Gym was designed with an idea of public benchmarking [27]. Hence, for other users to see the performance of their model against the same problem, the OpenAI team implemented the interface that helps solve MDP. On the official webpage [28] OpenAI indicates the sudden urge in the trend of RL popularity recently. Thus, the second most prominent benefit of OpenAI Gym is the environment standardization. Subtle differences in problem definition, reward functions, and set of actions have created the problem of reproducing published research. That is why the standard environment from OpenAI can help in the study within the DRL domain. Pointing it out again, the OpenAI Gym is an excellent choice for proceeding with, not only because of the simplicity of the software abstraction but also due to its popularity [29] and easiness of use.

The Platform Environment is the main environment class within the DRL platform. It allows the user to overwrite the specific implementation of OpenAI Gym **step** and **reset** methods, which are required for the DRL controller. Also, it gives a possibility to inject the specific Environment Server during the runtime, making the Platform Environment class flexible during multi-agent training mode.

The DRL controller is an actual training agent library, such as TensorForce, Stable-Baselines or RLLib. The library is supposed to be selected by the platform user. As long as it supports OpenAI Gym, it should be compatible with the DRL platform.

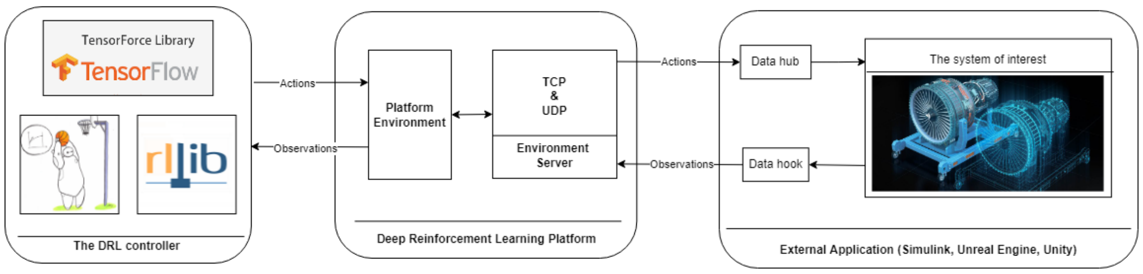


Figure 3. Proposed architecture of the DRL platform. The main architecture is located in a central block. The DRL controller is represented by a custom DRL library. The external application is an interface of communication with virtual or physical models.

To sum up, the provided architecture diagram represents the modular design of the DRL controller platform that will be applied at the INTECO inverted pendulum. The selected design supports the interaction of various modules in one execution, starting with the trainer agent and finishing with the system's external application.

4.2. Training sample controller for use case from external application

The DRL platform is available as a Python package [13] on PyPi webpage. The PyPi webpage contains a link to the official GitHub repository [30] of the project, which has a detailed description of the available API in the platform. There are also some notes regarding the creation of compatible custom environments to be used with the platform. As an example, this paper will show how it was possible to solve a straightforward synchronous custom RL environment using the DRL platform API.

Before starting, it is required to select the library for the training of a controller. The article [31] describes the current state of affairs within the domain of DRL. It provides an overview of multiple libraries used for reinforcement learning. As a result of this article, the authors have stated that the most suitable libraries appeared to be Stable-Baselines, Tensorforce, and RL_Coach as they are flexible for research purposes. As a result, the Stable-Baselines3 [32] library has been selected for training purposes. The training API of this library contains less initialization overhead, and there is no problem with injection of parameters, which are required from the software side of the platform.

The system of interest that has been selected for the testing purposes became the simple pendulum model provided by the Matlab RL Toolbox [33]. It is a digital model with a continuous action space that rotates the pole by applying the torque on the rotational joint. The main goal of this system is to keep the pole stable in an upright position.

Before proceeding with pendulum training, minor modifications to the Simulink model are required. These are the main steps to create the Data Input/Output hub/hooks which are used to connect to the DRL platform:

1. Open the simulation environment in an external application (Simulink, Unreal Engine, ...).
2. Create the block that initializes TCP or UDP server. This server needs to be configured such that it would be possible to connect to it from the DRL platform.
3. Create the block that initializes TCP or UDP clients. This client should be able to reach the DRL environment server used for the communication. A client should handle the connections and reconnections.

As shown in Fig. 4, the system consists of a Data Input hub, a Data Output hook, and the digital twin of the sample pendulum in a center. Both hub and hook have

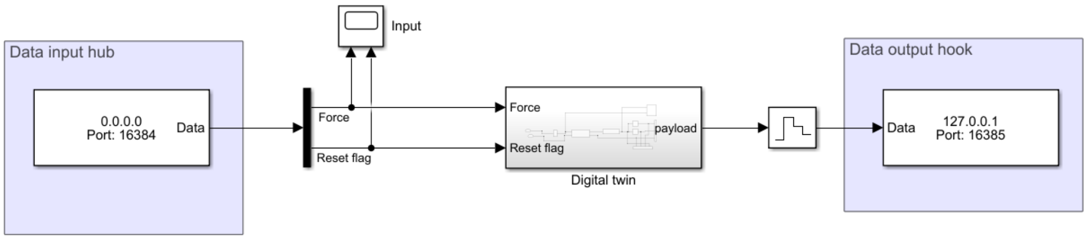


Figure 4. The sample pendulum environment in Simulink. The pendulum itself is located in a digital twin block, that accepts the action input from Data hub, and sends the observations output to the data hook.

the address with a port they use to communicate with the DRL platform. The DRL platform, in its turn, should take into account those values and use them during the initialization of environment server.

The next step to proceed is the creation of the platform environment itself. The sample code for the platform environment is given in Appendix B. This example shows how exactly the interface has been implemented. In the initialization of class, we identify the observation and action spaces with their limitations. In the case of the sample pendulum, the action space has been discrete.

The **reset** function uses the UDP environment server to send the input information to the Data input hub. As the environment is synchronous, the following received payload indicates that the Simulink has restored the dynamics in the Digital twin block.

The **step** function gets an action as an input, sends this action to the Simulink, and then receives the corresponding observation past the performed action. Based on obtained observation, we are calculating the reward per step. The greater the cumulative reward is, the better model appears. A mathematical explanation of the reward function in the sample environment is given as follows

$$R_t = -\left(\left|\frac{\alpha_t}{\pi}\right| - 1\right) \tag{17}$$

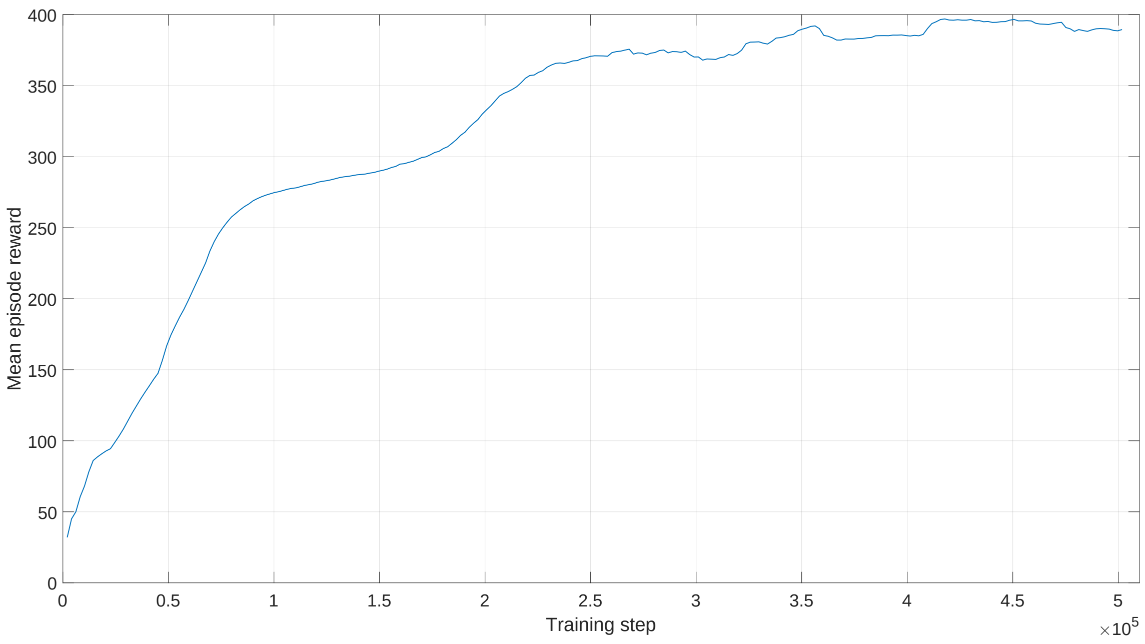
where α is the pendulum angle [rad].

The other two properties are needed to set and get the environment server during the runtime. One can launch the training loop if some training library has been installed, like Stable-Baselines3 in this code snippet. First of all, the Environment Server is initialized and connected to the running simulation. The launch of the server does not block the main thread. Once the connection is set up, it is possible to proceed to a training loop using the Platform environment.

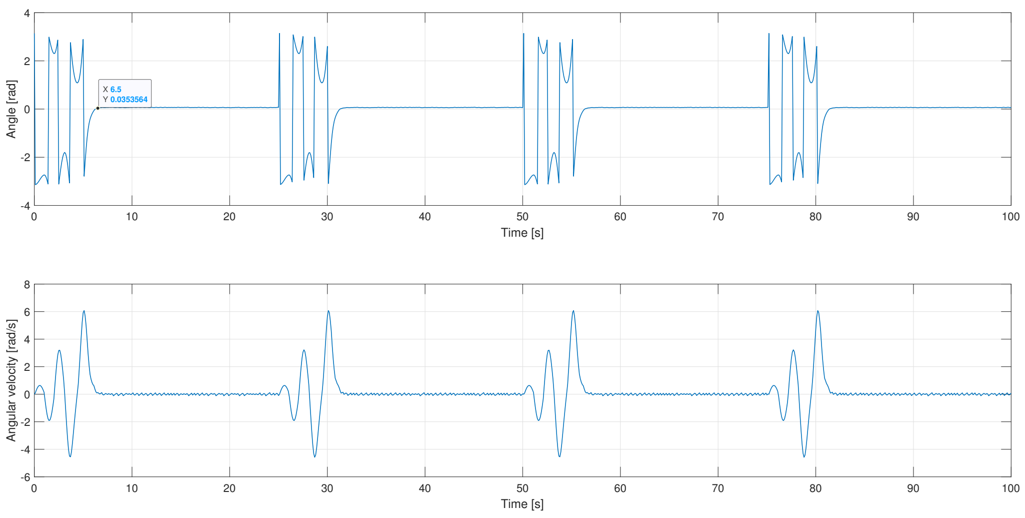
In the case of this sample, the RL model was able to achieve a stable high reward in 1 hour. The main view of the results of training and inference is given in Fig. 5

The training epoch for the sample pendulum took more time than within the Reinforcement Learning Toolbox provided by MathWorks. However, in this case, the network overhead could be an issue. And, of course, MathWorks has a significantly faster form of execution API when it comes to a trainer embedded in the Simulink. The trend for this system is given in Fig. 5a. The mean reward of $>$390 indicates the system's optimal level of control. During the inference of the model, it takes 6.5 seconds to stabilize the pendulum. The length of an entire episode is 25 seconds, 18.5 of which the pendulum is performing the control objective by staying at the upright position with zero output angle. The results of the model inference can be seen in Fig. 5b.

In this section, the basic functionality of the DRL platform was presented. The tool allowed solving a simple pendulum environment through the Stable-Baselines3. The provided solution is not very intrusive for external applications. It does not require a user to spend time choosing the way of environment representation and how it is supposed to be integrated with a trainer. By solving such a straightforward environment using platform API, it was decided to switch towards a more complex system, which became part of this paper's "sim-to-real" use case, the INTECO Inverted Pendulum.



(a) The resulting training trend of the sample pendulum use case. The training step is given hundred of thousands. The mean episode reward indicates how statistically well the model was trained each 500 steps.



(b) The result of model inference during 4 episodes. The mark in Angle graph represents the settling time for pendulum system. Throughout all episodes the angle was stabilized at 0 degrees, meaning that the pendulum was in an upright position.

Figure 5. Results of solving the sample pendulum environment. The graph 5a demonstrates the convergence of model to stable reward. The graph 5b shows how the converged model was able to control the sample system during the series of episodes.

5. Experimental Setup with the DRL platform

In this section, we will describe the INTECO Inverted Pendulum system, outline its mechanical design, and highlight the method of its control. Then, the conversion to the platform's compatible RL environment will be described, along with the operation schema.

The Stable-Baselines3 was selected as the main training library for this use case. Since it has shown promising results in a sample environment, no objective reasons highlighted the need for change in a library.

The source code with prepared platform compatible environment is located in Github under the project's use cases [34].

5.1. Description of the inverted pendulum system

The inverted pendulum is a system that contains the poles attached to the movable cart [35]. Its physical appearance is illustrated in Fig. 6. Four main components constitute this system.

The first of them is the frame. The frame consists of a pair of metallic supports that hold the rail. The rail is a long metallic bar on which the pendulum cart moves. The rail length is approximately two meters, and it contains the physical barriers to prevent the cart from hitting the system's drivers/motors.

The second part is the cart itself. There are four bearings attached to its bottom to make riding along the rail possible. Also, there is one optical encoder placed within the cart box. This encoder allows to determine the pendulum pole angle and return it as real-time feedback.

The third part is the driver, which controls the cart's position through the driving belt by applying the steering force at the end. The belt is moved by the direct current motor installed at the rail's end. The motor is being controlled through the PWM driver connected to a computer. Also, one of the rail ends contains the second optical encoder. This encoder allows transmitting the cart position feedback.

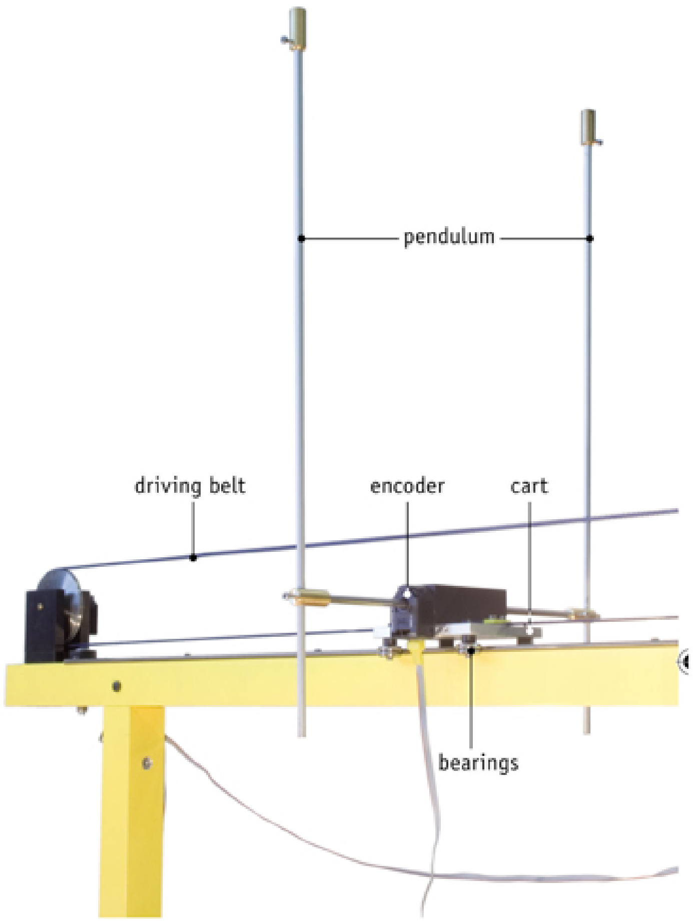


Figure 6. INTECO inverted pendulum system. Image source: http://www.inteco.com.pl/products/pendulum-cart-control-system/pendulum/

The last part is the actual pendulum. It is mounted in the cart and looks similar to the fork tower. As mentioned before, the optical encoder installed in a cart allows for determining the pendulum's angular position. The pole's towers can only move in a vertical plane, without any jitters in the horizontal direction.

Even though the pendulum contains all the necessary components for operation, it is impossible to move it without computer control. The control against physical system is performed programmatically through the Simulink. The manufacturer provided the initial version of the Simulink's model that includes the PWM driver necessary for system's movement. Also, the digital twin of the pendulum has been provided by the INTECO. The virtual model contains the similar interfaces for providing inputs and getting the observation outputs. The control is also performed within the Simulink environment.

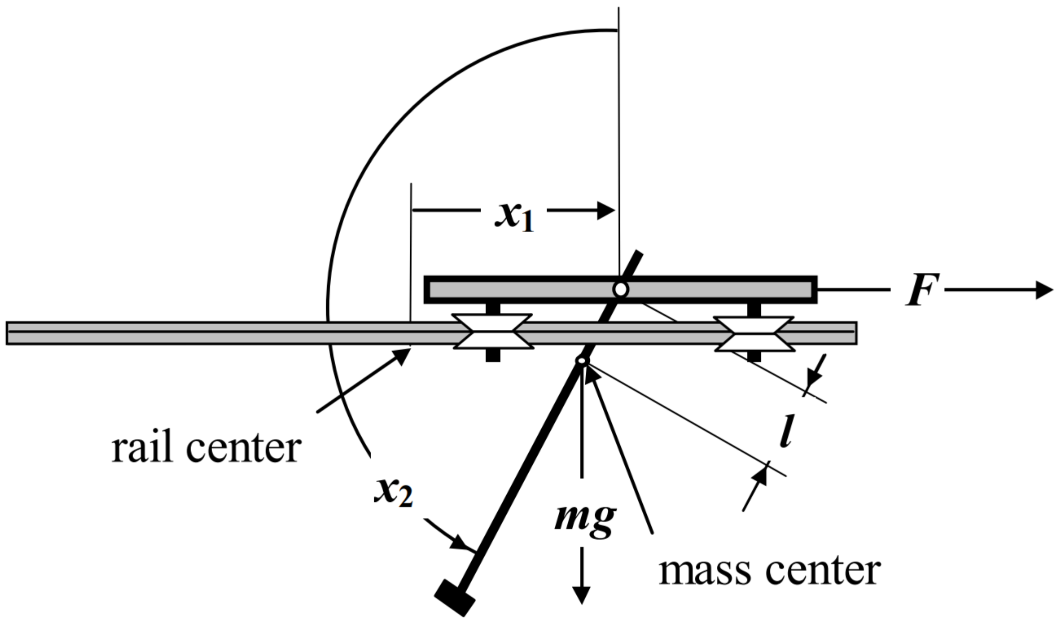


Figure 7. The schematic representation of pendulum system [1]. The pendulum rotates in a vertical plane on the axis located in cart. The cart moves horizontally on a rail due to the applied force parallel to the rail.

5.2. Mathematical model of the inverted pendulum system

The state of the system can be represented through the state vector [1] $X = [x_1, x_2, x_3, x_4]$, where $x_1$is the cart position, x_2 is the angle between the upward direction and the pendulum, measured counterclockwise, x_3 is the cart velocity, and x_4 is the pendulum angular velocity.

The control force F is applied to the cart in direction parallel to the rail, as illustrated in Fig. 7. The force is being produced by PWM voltage signal $u \in [-0.5, 0.5]$,so that $F = p_1 u + p_2 x_3$,where p_1 is the control signal to force ratio, and p_2 is the cart velocity to force ratio.

The state equations of the inverted pendulum's mathematical model are identified as follows:

$$\begin{aligned}
\dot{x}_1 &= x_3, \\
\dot{x}_2 &= x_4 \\
\dot{x}_3 &= \frac{a_1 w_1(x,u) + w_2(x)\cos x_2}{d(x)} \\
\dot{x}_4 &= \frac{w_1(x,u)\cos x_2 + a_2 w_2(x)}{d(x)}
\end{aligned} \tag{18}$$

where

$$w_1(X,u) = k_1 u - x_4^2 \sin x_2 - k_2 x_3 \tag{19}$$

$$w_2(X) = g \sin x_2 - k_3 x_4 \tag{20}$$

$$d(x) = b - \cos^2 x_2 \tag{21}$$

$$a_1 = \frac{J_p}{ml}, \quad a_2 = \frac{1}{l}, \quad b = a_1 a_2 = \frac{J_p}{ml^2} \tag{22}$$

$$k_1 = \frac{p_1}{ml}, \quad k_2 = \frac{f_c - p_2}{ml}, \quad k_3 = \frac{f_p}{ml} \tag{23}$$

The moment of inertia of pendulum related to the pendulum rotation axis is defined as:

$$Jp = \frac{1}{12} m_{pw} l_c^2 + \frac{1}{4} m_{pw} r_c^2 + m_{pw} l_{co}^2 + \frac{1}{12} m_{ps} l_p^2 + \frac{1}{4} m_{pw} r_p^2 + m_{ps} l_{po}^2 \tag{24}$$

The total moment of inertia related to the mass center of the system can be expressed through J_p as:

$$J = Jp - l^2 \left(m_c + m_p\right) \tag{25}$$

where

$$l = \frac{l_{po} m_{ps} + l_{pwo} m_{pw}}{m_c + m_{ps} + m_{pw}} \tag{26}$$

All parameters in the equations above are described in Table 1. Some of the parameters were configurable for the digital twin. These attributes are marked in a corresponding column.

5.3. Control objective of the inverted pendulum system

The main control objective of the inverted pendulum system is to put the pendulum in an upright position. Cart must be located close to the rail center as possible, keeping the pendulum in the same angular pose. The quality of the controller depends on how fast and reliable the cart upswings the pendulum.

The user has to apply the control in the pulse-width modulation (PWM) fraction to move the cart along the rail. Depending on the sign of the PWM fraction, the cart determines the direction of acceleration. Per outputs, the system provides the pendulum pole angle via one encoder and the cart position on the rail via the second encoder. The calculation of pole angular and cart velocities is performed within the Simulink environment for virtual and physical environments.

Hence, given the possibility of obtaining these specific system outputs, it is clear to state that the system's control problem follows Markov's property, which means that the future state does not depend on the past. And as the problem follows Markov's property, the control objective can be converted to the RL problem [6].

Without deep-diving into Markov Decision Process (MDP) it is possible to describe the scheme used to train the pendulum controller. Referring to Fig. 8, the general system demonstrates how the DRL controller is taught and what it experiences per a single step. The INTECO pendulum control problem can be described in this

Table 1. Parameters of the mathematical model of the inverted pendulum system.

Name	Unit	Description	Value	Is configurable
m	kg	Equivalent mass of cart and pendulum	0.6923	False
l	m	Distance from the axis of rotation to the center of mass of system	0.0196	True
f_c	$\frac{Ns}{m}$	Dynamic cart friction coefficient	0.5	True
f_s	N	Static cart friction coefficient	1.1976	True
f_p	$\frac{Nms}{rad}$	Rotational friction coefficient	$27.344 \cdot 10^{-5}$	True
J_p	kgm^2	Moment of inertia of pendulum with respect to axis of rotation	0.00292	False
g	$\frac{m}{s^2}$	Gravitational acceleration	9.81	True
m_c	kg	Equivalent cart mass	0.5723	True
m_{ps}	kg	Pole mass	0.12	True
R_l	m	Rail length	1.8	False
l_p	m	Length of pole	0.5	False
l_{po}	m	Distance between center of pole mass and rotation axis	0.107	False
l_c	m	Length of load	0.03	False
l_{pwo}	m	Distance between center of load mass and rotation axis	0.354	False
J	kgm^2	Total moment of inertia related to the mass center	0.00386	True

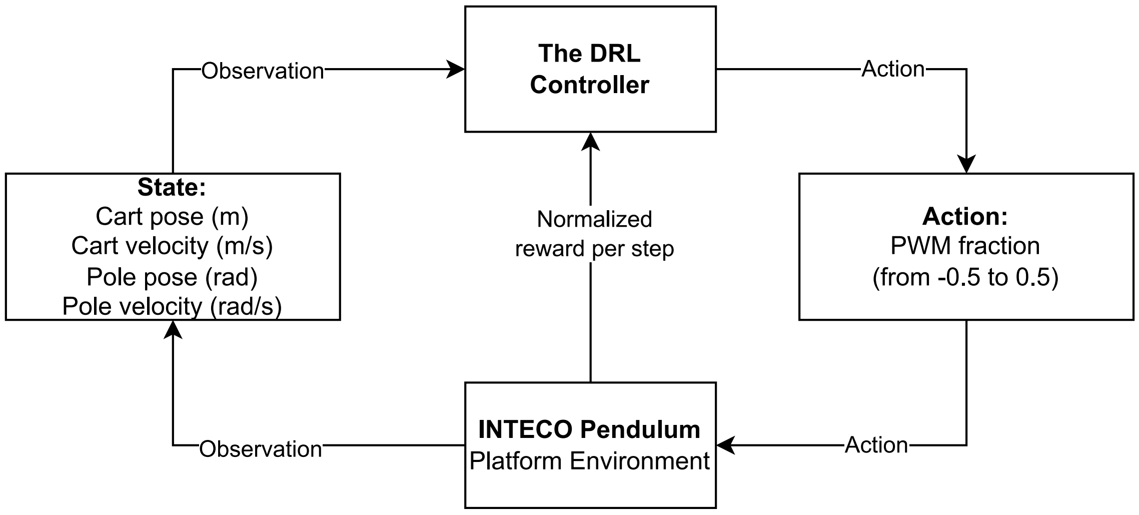


Figure 8. The summarized interaction of Platform Environment with the training agent. The controller sends the action to the INTECO Pendulum Platform Environment for each step. This environment sends an action to an actual pendulum and returns the latest observation with a calculated reward back to the training DRL controller.

specific sequence of points:

1. The training episode starts, and each episode has a fixed number of steps. Once the episode finishes, it calculates the cumulative reward. The cumulative reward indicates how successful is the current iteration of the controller.

2. Each step begins with the controller predicting the action based on some old observation. The actions space is cropped continuous input between -0.5 and 0.5.

3. After the action has been applied to the system, we wait until a new observation is generated. Once it is available, the Gym-compatible Environment calculates the normalized reward. Normalized reward means that its range falls between zero and one at each step. If the reward is equal to one, then the pendulum system, after the applied action, is in ideal state.

4. Once the reward is calculated, the pendulum state is propagated back to the controller trainer. Unless the specific outputs exceed the prohibited limits, such as the cart rail limit of pole velocity limit, the system performs the same action-reward-state transition sequence until the episode terminates. If limits have been exceeded, the episode ends abruptly with a much lower score than an average full-length episode.

The normalized reward function calculation was defined in the following way:

$$
\begin{aligned}
R_t = &- \left(\left| \frac{\alpha_t}{\pi} \right| - 1 \right) \cdot \frac{(\omega_{max} - |\omega_t|)}{\omega_{max}} \cdot \ldots \\
&\cdot \left(0.75 \cdot \frac{(x_{max} - |x_t|)}{x_{max}} + 0.25 \cdot \frac{(v_{max} - |v_t|)}{v_{max}} \right)
\end{aligned}
\tag{27}
$$

where α is the pole angle in radians, ω is the angular velocity in $\frac{rad}{s}$, x is the cart position in meters, and v is the cart velocity in $\frac{m}{s}$. The maximum values for velocities and poses have been determined empirically and do not depend on the dynamics of the actual pendulum. Specifically, in order not to damage equipment, limits were selected as $\omega_{max} = 15$, $v_{max} = 3$, and $x_{max} = 0.7$.

To sum up, the controller is being trained in favor of maximizing the cumulative normalized step reward. Specifically, each state of the system will have an approximated reward value if a specific action is taken. When the controller has been trained and launched in inference mode, it selects such action that brings the biggest reward. Since the controller is a detachable part of the reinforcement learning problem scheme, it is certainly possible to share the designed controllers between physical and virtual systems.

5.4. Creation of Platform Environment for the inverted pendulum

Two main points must be fulfilled to prepare a Platform Environment for the controller training. First is the injection of the Environment Server, which does not require any preliminary design for our specific use case. Thus, the user can select the particular address and ports for sending and receiving information with an external application, start the server in runtime, and inject it using the class attributes into the Platform Environment. Secondly, the OpenAI Gym step and reset methods have to be implemented. The development of those methods is specific to the inverted pendulum use case.

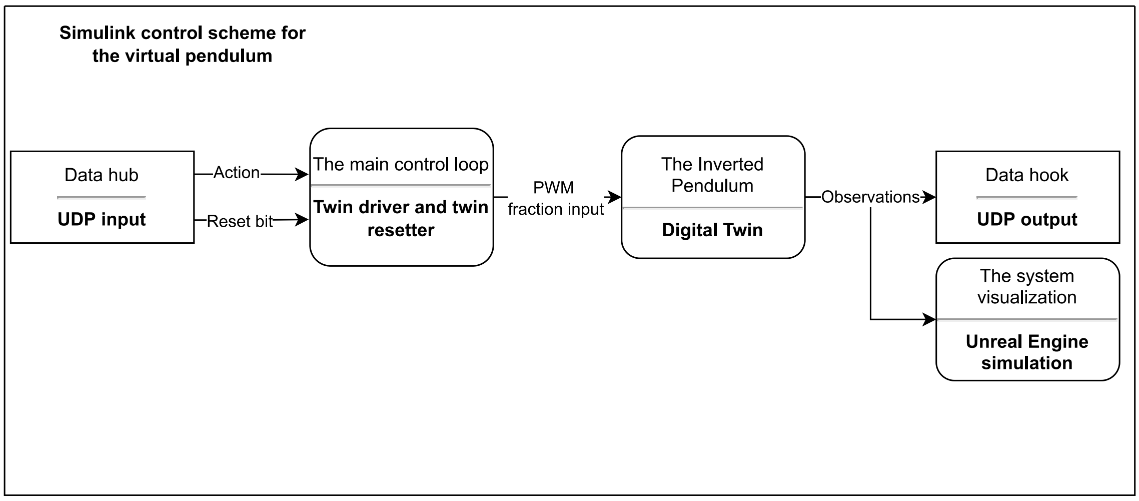
For the inverted pendulum use case, the implementation of the **step** method sends a socket datagram with a new action. Then the same function expects to receive the new observation from the Simulink as soon as the Environment Server receives the next package after a sent action. We calculate the normalized reward for the performed step based on the observed output. Nextly, before we return the reward with the latest observation from the method, it is essential to check whether the system went out of bounds for the expected observation space. If the pendulum rotates too quickly or the cart exceeds the rail limits, the episode instantly terminates, and the abrupt termination is penalized with a negative reward.

When designing the **reset** method, it was clear that it is impossible to make an actual reset of the system using purely software API from the DRL platform. The main reasons lie in the problem of not having full access to the dynamics of the system of interest, so it might not always be possible to restore the system's conditions for a new episode programmatically. Due to that fact, the actual reset of the environment has to be implemented in an external application itself. What the DRL platform will do in its turn is it will always append the reset flag at each action sent to the external application. Thus, as soon as the reset flag is experiencing a rising edge, the external application can start the procedure of the system's reset. And, if needed, find a way to state to the DRL platform if the operation has been finished[1].

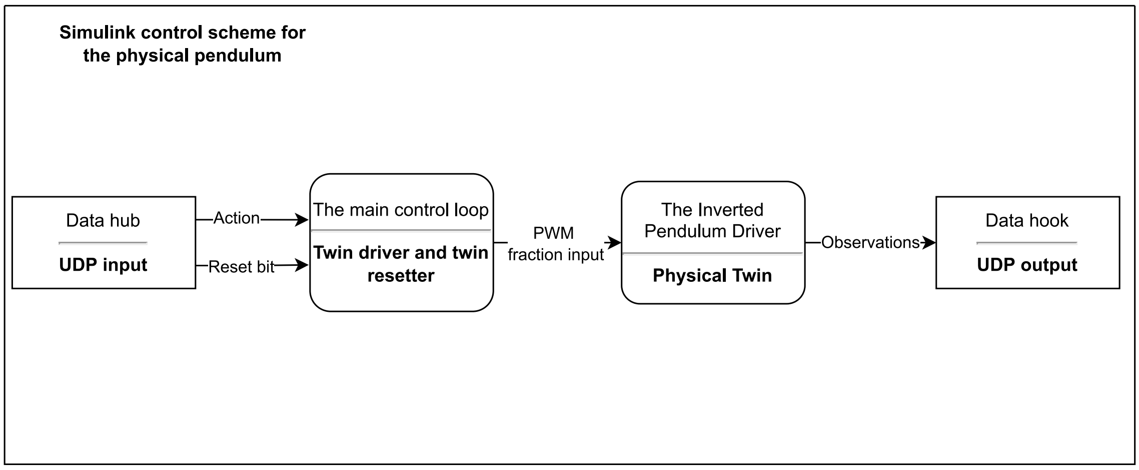
To be compatible with the OpenAI Gym, the platform environment should also have a specific number of steps and maximum reward per episode. Even though it is not prohibited to omit this configuration, its lack usually causes the training to overfit. For the DRL controller, it is essential to know the cumulative episode reward during the assignment of neural network weights. In the inverted pendulum problem, the maximum number of steps is 512. Since the reward per step is normalized, the total reward[2] per episode is also 512. Having the Platform Environment prepared, it is possible to proceed with a description of operation schema within the Simulink. The code for the Platform Environment, as well as action and observation spaces used for the trainer, are given in Appendix B.

[1]Important feature for real-time physical systems, which can not guarantee an instant reset. In case of the pendulum, the **reset** function looks at the latest state after the rising edge of the flag. As soon as states indicate the total reset of the episode, the trainer proceeds with the training.

[2]Though it is practically not possible to achieve such a reward during the training because the pendulum spends some timesteps for the swing up motion



(a) The Simulink control scheme for the digital twin of inverted pendulum



(b) The Simulink control scheme for the physical twin of inverted pendulum

Figure 9. The Simulink control schemes for digital/physical twins of the inverted pendulum. The Figure 9a shows the layout for the digital twin, and the next Figure 9b shows the layout for the interaction with a physical system.

5.5. Configuration of Simulink models for operation with the DRL platform

According to the platform architecture, the Simulink should contain the implementation of data trespassers in the form of hub and hook. The application allows to create the connection endpoints using the TCP/UDP communication, which was also demonstrated in a section that solved the sample pendulum environment. The control schemes for both virtual and physical INTECO pendulum systems are given in Fig. 9.

Both models are driven through the Data hub that acts as the UDP server on which the DRL platform sends the action and reset bit. The information from the hub is propagated to the control loop that handles the twins driving and the twins episode resets.

The main control loop drives the model and can reset the system to the initial state during the episode termination. The reset of the virtual environment was done through the rising edge of the reset bit and swing-down PID controller. This controller brings the cart to the center point of the rail with poles pointing downwards. Since the PID controller can not instantly reset the environment, the episode reset usually takes time, especially at the beginning of the training generation, when the model goes out of the observation space bounds too often.

In both virtual and physical use cases, the systems of interest have the same interface with interaction. Specifically, both receive the PWM fraction under the same limits, both of them output the observation in the form of the current pendulum's state. The only difference is that internally the digital twin contains the entire mathematical implementation of the model, whilst the physical twin holds only the driver that allows the user to speak with the PWM generator required to generate cart pulling force.

There there was no need to make external visualization for the physical twin. However, observation of the results for the digital twin needed some additional environment. It is possible to visualize the pendulum through the Simulink. Still, to not implement it from scratch, it was decided to take an existing virtual replica of the pendulum in Unreal Engine. The virtual model in Unreal Engine demonstrates the system's outputs through the designed game object. This simulation was a part of the Extended Reality project in TalTech's Centre for Intelligent Systems [5], and it helped a lot during the visual validation of the controller results.

Lastly, the observations for both physical/virtual twins are propagated to the DRL platform through the Data hook. The Data hook also uses the UDP protocol to act as a client connecting to the DRL platform Environment Server.

After the preparation of Platform Environment and the configuration of an external application, we can finally start with the training of DRL controller experiment, results of which are given in a next section.

6. Experimental Results with the DRL platform

6.1. Training section

The literature review has proven the applicability of gradient descent training algorithms, such as DDPG, to robotics and control equipment applications. Though the DDPG proves the possibility of designing optimal controllers, other gradient descent-based training algorithms could do much better at controlling actual equipment. One example of such an algorithm is the clipped PPO [22]. Developed by the OpenAI team, it has inherited the benefits of trust-region policy optimization (TRPO) methods, but with less heavy objective function implementation.

The main motivation of algorithm given in Eq. (7) is to take the minimum between clipped and unclipped objectives $\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t$ and $r_t(\theta)\hat{A}_t$ respectively. This prevents the destructively large policy updates during the series of stochastic gradient ascents. At the same time, the unclipped objective introduced in TRPO allows making the sample efficient gradient ascent for the policy network update. Thus, from a mathematical standpoint, the PPO is a reliable non-destructive algorithm that can be used for achieving the inverted pendulum control objective.

To show that PPO benefits over other training algorithms, the abovementioned paper brings empirical proof about algorithm's performance against "vanilla" gradient methods and generally the standard TRPO. As a result, the model training performance was better in various MuJoCo environments, making the PPO a better training algorithm for achieving the robust controller.

Considering these benefits, the models for the INTECO pendulum have been trained using the PPO method. The hyperparameters for training have been left similar for all environments (virtual, physical, "sim-to-real").

The selection of hyperparameters has been based on the Stable-Baselines Zoo, the repository which contains the collection of tuned models for the most popular OpenAI Gym environments [36]. The InvertedPendulumSwingupBulletEnv-v0 OpenAI Gym environment was selected for reference. The Table 2 describes the most critical hyperparameters used during the training through Stable-Baselines3 API.

Table 2. The PPO hyperparameters used for INTECO training

Parameter	**Description**	**Value**	**Stable-Baselines3 argument input**[3]
Number of steps	The number of steps for each Gym-environment under training per model update. Determines how many observations should be taken before the gradient descent.	2048	n_steps
Batch size	The size of the minibatch that is used during the model update. This size determines the amount of observation taken per descent updates.	64	batch_size
Number of epochs	Number of passes through the experience buffer during the optimization towards the decrement of a surrogate loss function.	20	n_epochs
Gamma discount factor	The discount factor shows how much the model will underline the importance of possible rewards in the future. The lower it is, the more valuable are the immediate rewards.	0.99	gamma
Lambda factor for GAE	The factor constructs the trade-off between bias and variance for the GAE. Higher lambda - higher is the bias estimator.	0.95	gae_lambda
Entropy coefficient	The entropy coefficient is used for the value loss function calculation. It prevents early convergence of policy to a specific action without proper exploration.	0.0	ent_coef
Clipping range	The range that prevents the drastic parameters update during the stochastic gradient ascent.	0.2	clip_range
Learning rate	Corresponds to the strength of the gradient descent update.	$2.5 \cdot 10^{-4}$	learning_rate
Number of hidden layers	The number of hidden layers in both actor and critic neural networks used during the training	2	policy
Number of neurons	The number of neurons in each hidden layer of the neural networks during the training	64	

Besides the model hyperparameters, each training epoch has a fixed number of steps it takes before the new generation starts. In the case of an inverted pendulum, there were no limitations on the number of epochs. The model was trained until it was visible that the control objective was fulfilled, and the training mean episode reward is not increasing anymore. Each generation took 500000 steps. The trainer saved the model every 5000 steps to analyze the model during the inference.

6.1.1. Training procedure for the virtual pendulum

The main goal was to implement a "sim-to-real" controller, so the controller design for the virtual pendulum was done first.

Before the training starts, a user must launch the Simulink environment execution. The pendulum is being executed through the Real-Time plugin to make it as close to the natural environment possible. This plugin allows the model to run in a parallel non-synchronous manner. Despite all of the dynamics and sample times fixed for the digital twin block, the Real-Time execution allowed assigning the specific sample times for the Data Input hub and Data Output hook. Different sample times brought different behavior of the controller.

As the training proceeded, it was essential to look at the behavior of the various sample times for the Data Input hub and Data Output hook. This parameter is crucial as it allows to:

1. Increase/decrease the training time;
2. Share more dynamics the virtual pendulum holds.

Since the model's training has been taking some time, only three different sample times were taken into account. The result is shown in the Table 3, which contains the information about the training with varying sample times. This test has been conducted to find out the best sample time for the future controller.

Based on the results of the conducted experiment, it was clearly visible that for the training against the virtual model, the best sample times were 0.03 and 0.01.

[3]The official API for the model training is described in Stable-Baselines3 documentation

Table 3. Results of the training per various sample time

Sample Time (seconds)	Training time for 100K steps (min)	Average episode reward for 100K steps
0.01	38	94.87
0.03	87	157.6
0.05	133	108.6

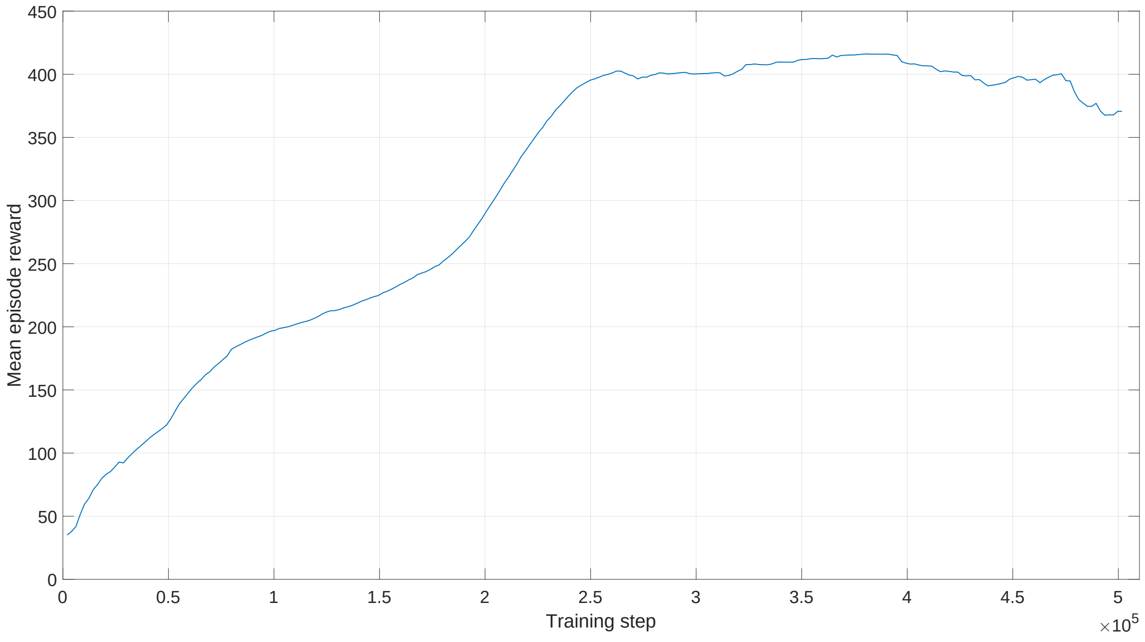


Figure 10. The controller's training trend for a virtual twin of INTECO pendulum. An overall tendency shows the sudden rise in mean episode reward. The controller has achieved a mean episode reward of 400, which is suitable for the control objective, after 280 000 training steps.

However, the sample time of 0.03 seconds has been selected to train the virtual pendulum. This decision has been based on the fact that high-frequency action input during the training/inference on an actual pendulum could have damaged equipment and created disturbances in the system due to the switching vibrations in the motor.

Overall, the training trend of the entire 500000 steps epoch for the model with a sample time of 0.03 seconds can be seen in Fig. 10. Based on the visual inspection of the training trend, it is possible to state that selected hyperparameters were able to train the model of the INTECO virtual pendulum. First of all, the mean reward of the model has been continuously increasing throughout the most considerable portion of the training epoch. Secondly, the model was able to converge to the mean reward per episode of 400, which allowed the controller to hold the pendulum in upright position.

The time it took to train the whole epoch was 6 hours and 8 minutes. The mean reward converged at step 280 000, which took 3 hours and 24 minutes. Because it is possible to save the training checkpoints before the epoch termination, it is not necessarily required to wait all 6 hours until the controller reaches the desired reward.

To conclude, the success of the virtual model training has proven that the DRL platform was able to generate a controller that can operate a complex nonlinear nonphysical object in Simulink software. But the training of the virtual model can not guarantee whether the platform can be used for the physical system or "sim-to-real" use case. Thus, the next step to solve the "sim-to-real" problem was the training in the real pendulum system.

6.1.2. Training procedure for the physical pendulum

Before starting with the training of the physical pendulum, one might be curious why it is not possible to dive straightly into the "sim-to-real" use case. Why can't we use the best checkpoint from the virtual controller or try the training from the same checkpoint? The answer - it has been tried, but during the model inference[4] the controller was not achieving the objective totally, not even trying to swing up the pendulum. These blockers are going to be described in a subsequent section, but still, they gave an idea of what could have been wrong:

1. Using the DRL platform with the real pendulum might not be possible. Primarily, the limitations could be related to long network delay, noise in the data, and the inapplicability of software architecture in the design of the controllers for such systems.
2. The dynamics of the virtual replica of the INTECO pendulum are completely unaligned with the dynamics of the real system.

To prove that it is not an architecture blocker, it was decided to train the new PPO controller with the same hyperparameters from scratch for the real pendulum. Compared with the training of the virtual controller, training of the physical RL

[4]Even continuing the training from a checkpoint did not help. The controller was hitting the physical barriers

model contained several problems related to new dynamics, which were not modeled within the digital replica.

1. Sometimes, sensors output incorrect data with significant deviation from the range. For instance, the pendulum encoder sent the negative value up to thousands resulting in a substantial negative reward. Such huge negative rewards were affecting the quality of the RL model negatively.
2. Cart observation space bounds used for the virtual model did not work for the actual model because the cart was constantly hitting the physical barriers of the rail due to the inertia. If the model's training is happening on the real equipment, it is vital to ensure that the procedure won't damage the equipment.

The created Gym-compatible DRL platform environment contained the noise detection that observes an instanteneous change of the observable states to mitigate the first issue. If an absolute value of instantaneous change exceeds the observation space limits, then the package with noisy outputs is discarded, and the platform waits for the next one to arrive. The described solution helped increase the training robustness and resolved the problem of sudden episode termination due to noisy data.

Unfortunately, to solve the second problem, it was required to change the cart position limit, after which the training episode terminates. The value has been determined experimentally by observing whether the cart hits the physical barriers during reset. As a final result, if the cart exceeds 70 cm from the rail center, the episode terminates, and the environment reset happens. These sudden environment terminations due to observation space limits also give the trainer a negative reward. This way the controller understands that equipment must be operated within certain boundaries.

Finally, it was possible to train the RL model on a real physical system after resolving those issues. The pendulum controller with the highest mean episode reward has shown the optimal control and robustness to disturbances. If one applies an external force on the poles of the pendulum, the system immediately applies the counterforce to keep the pendulum stable in an upright position. This behavior can be explained by the excellent inheritance of the system's dynamics. The result of the training can be observed in a Fig. 11.

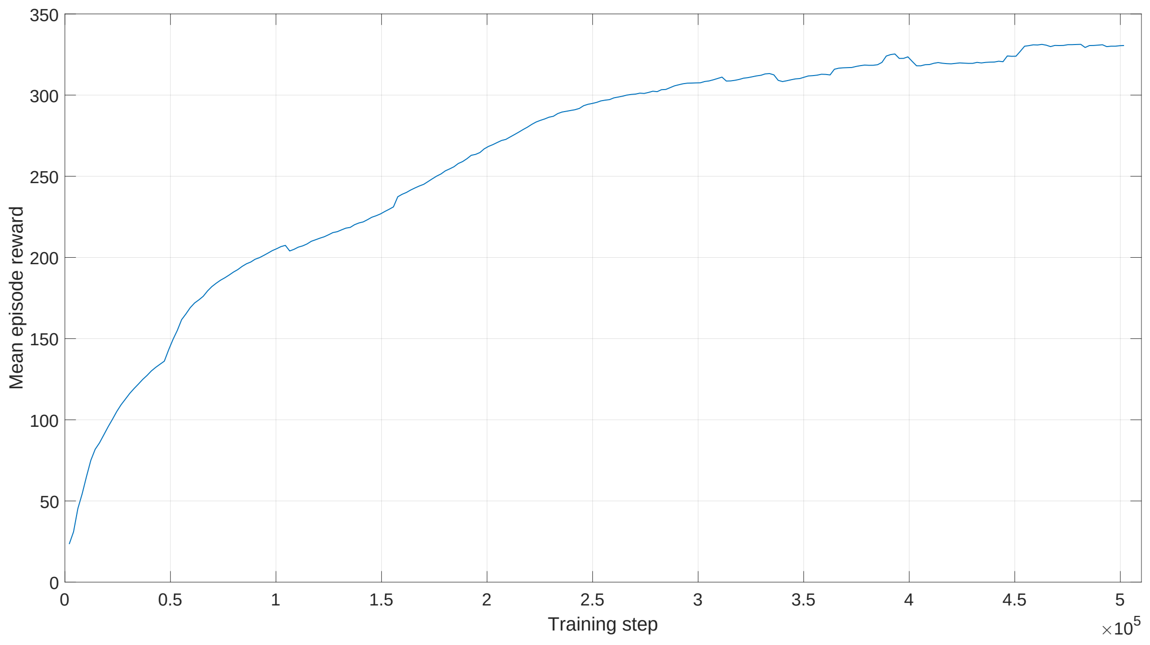


Figure 11. The controller's training trend for a physical twin of INTECO pendulum. The mean episode reward convergence is observed after 400 000 steps.

The sample time for the physical system was 0.03 seconds, as it was the best sample time for the virtual model controller. An overall reward trend has been increasing throughout the training generation. However, the terminal reward has not exceeded the range above 400. The converged reward has only achieved the plateau of 340, which was enough to perform the control objective, but the cart has been constantly drifting near the rail center.

This behavior identifies that the dynamics of the virtual system are different from the real system. The DRL controller on a physical pendulum takes more time to inherit the system's dynamics. The stable mean reward for the control was achieved at 400 000 steps, which took 6 hours and 26 minutes.

In conclusion, it was possible to state that the DRL platform could train the controller from scratch on a physical system. The speed of dynamics inheritance was different from the virtual model. Still, it didn't result in a complete failure of the training because the trend of increasing mean episode reward was also continued in the next epochs. Hence, the "sim-to-real" use case problem was not due to the incompatibility of platform architecture or network delays. But it was pointed out that the dynamics of the virtual model were significantly different from the physical system. This problem had to be solved before using the virtual model controller on actual equipment.

6.1.3. “Sim-to-real” training of the pendulum

Before proceeding with the “sim-to-real” use case, a pre-study was conducted on the most recent trends in this domain. The article [12] conducted the study on the most frequent methods that have been identified for solving “sim-to-real” problems. The Table 4 highlights each approach with its application domain description.

- The Zero-shot Transfer approach was tested as soon as the virtual controller was available. This test resulted in the wrong behavior of the pendulum. The cart constantly hit the rail barriers, and the poles were rotating with an unacceptable high angular velocity. Even the training from the checkpoint was far worse than training from scratch because the cart was hitting the barriers even with fixed observation space limits.
- The Domain Randomization, coupled with a correctly identified system, appears to be a promising approach. However, it was not entirely clear how to change the dynamics of the virtual system in Simulink runtime. To implement such a technique, one has to figure out the possibilities of internal Matlab/Simulink API.
- The Domain Adaptation seemed to not fit in a current use-case, as it was primarily used for vision-based problems. Also, it is not entirely clear how to get the observation data from two systems in parallel during the training. The main goal of the “sim-to-real” INTECO use case is to use as less training time on a physical pendulum as possible. If such an approach requires a simultaneous execution, it is better to train the controller from scratch.
- Learning with Disturbances could have helped if the virtual model’s dynamics were relatively close to the existing system. As the previous section has proven, the inference problem was most likely coming from the totally unaligned dynamics of virtual and physical models.
- The selection of different simulation environments is not a choice, as the problem lies in implementing the Simulink model subsystem, not the Simulink environment itself. This approach is much more suitable for complex robotics use cases, where the simulation environment like Gazebo and MuJoCo contain the implementation of the entire digital twin in their codebase.

Table 4. List of "sim-to-real" methods for DRL

Method	Description
Zero-shot Transfer	A straightforward approach where the trained controller is applied directly to the physical environment. The success of this approach depends on the virtual's model accuracy.
System Identification	This method aims to identify the precise mathematical model of the real system on which the controller has to be trained. Though it is possible to approximate the virtual environment for simple systems, identifying the precise model can become a big problem for complex equipment that has varying dependency of its dynamic on external conditions, such as temperature, humidity, etc.
Domain Randomization	This approach considers the limitation of the precision of the virtual models. Instead of adapting the model and trying to tune it for the best fitting parameters, it should be possible to change the model dynamics during the training dynamically. The solution has a powerful effect on sim-to-real experiences for robotics and other complex environments.
Domain Adaptation	The main goal of this method is to utilize the data from two sources of the environment: physical and virtual. This approach tries to unify the virtual and real observation spaces with almost identical reward functions and action spaces. For instance, one method can calculate the statistical difference between two domains and apply the difference depending on the environment. This approach is mainly conducted in vision-based RL problems.
Learning with Disturbances	The algorithm applies perturbations in a simulated environment to make the controller more robust in natural settings. The approach has found use in a multi-agent training environment, where multiple perturbations can be applied simultaneously during the one training generation.
Simulation Environment	The critical aspect of good transfer learning is an accurate execution environment with precise mathematical derivations. This approach can serve well for complex popular systems that have been transferred in simulation environments such as Gazebo, PyBullet or MuJoCo.

- The last approach that seemed the most suitable appeared to be System Identification. All of the other methods, in some sense, rely on the quality of the digital replica. However, if it is completely unaligned, these methods can not guarantee the training's success. If one could optimize the virtual model parameters to be relatively close to the physical system, then the training time on the physical system can be significantly reduced. It is possible to select a good checkpoint of the virtual controller and launch the training on the pendulum. As a result, it might be possible to observe faster convergence to the expected reward slice.

Thus, taking the points above into account, the "sim-to-real" use case used the Systems Identification method.

To understand the problem's root cause during zero-shot learning, it was necessary to find a common reference point from which it is possible to analyze the difference. As the controller for the physical system was trained, it was possible to get the data from the actual pendulum by running several data-collecting episodes. Since the interface of interactions between two Simulink models for physical and virtual systems were identical, it was possible to apply the same controller on the virtual model to take a look at observation differences. Fig. 12 compares two environments at all observation states.

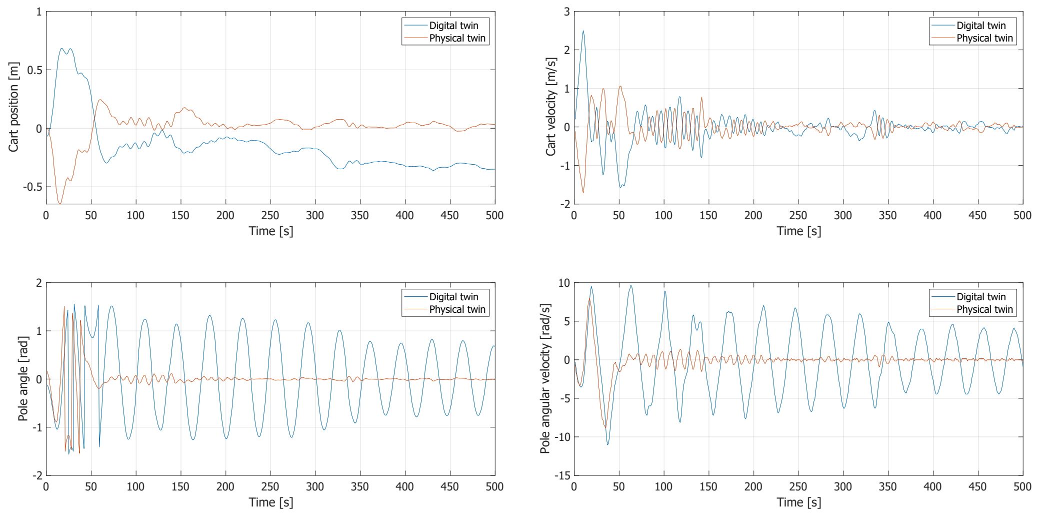


Figure 12. The observations from digital and physical twins under the same action input. The observations include cart position and its velocity, pole angle and its angular velocity. The results show the fundamental misalignment in dynamics between two systems.

Based on the graph, there are multiple problems indicating the significant dynamics difference in the virtual model.

- Both cart velocity and pose have a phase shift of 180 degrees compared to the existing system. It likely happened due to an inversion of action input in the PWM generator. However, this phase shift has not been compensated in a model for some reason.
- Outputs related to pole have a specific time delay, while generally, the value trend matches. This problem likely lies in incorrect inertia dynamics, which caused such delay.
- It is not seen, but in the beginning, the cart velocity has a higher rate of change than the actual model. It seems that the amplification of the input signal is wrong and should be optimized for the virtual model.

It was necessary to modify the model's internal structure to fix the phase shift. There was no direct way of inverting the phase without breaking the mathematical derivations used to calculate all other state variables besides cart pose and velocity. Eventually, it was decided to reverse the output's cart pose and velocity sign. The phase shift was not removed, but it was not a problem due to the symmetrical dynamics relative to the zero rail position. This approach did not break mathematical derivations, and it has also allowed having an output with an aligned phase.

To deal with other issues, it was necessary to properly excite the system using an input signal with varying frequencies. This approach is called open-loop identification of the system's dynamics. Through Simulink, it was possible to generate a Sine input signal with varying frequency for 100000 steps. The varying frequency signal allowed to explore the observation space for the actual pendulum and the virtual model simultaneously. Examined observations data had been saved in Matlab code for future analysis.

Once the data from all environments were available and aligned with a single timeframe, it was possible to calculate the root-mean-square error (RMSE) for both cart and pole positions. Of course, no direct input would help minimize the RMSE in favor of making the model closer to reality. However, the digital replica provided by INTECO has 12 configurable parameters. Suppose one can create a script that iteratively changes the configurable parameters. In that case, it is possible to calculate the new value of RMSE once the simulation runs again with an updated set of parameters. Thus, the optimization problem for the digital twin of inverted pendulum can be described as follows:

$$\min \ RMSE(X) \tag{28}$$

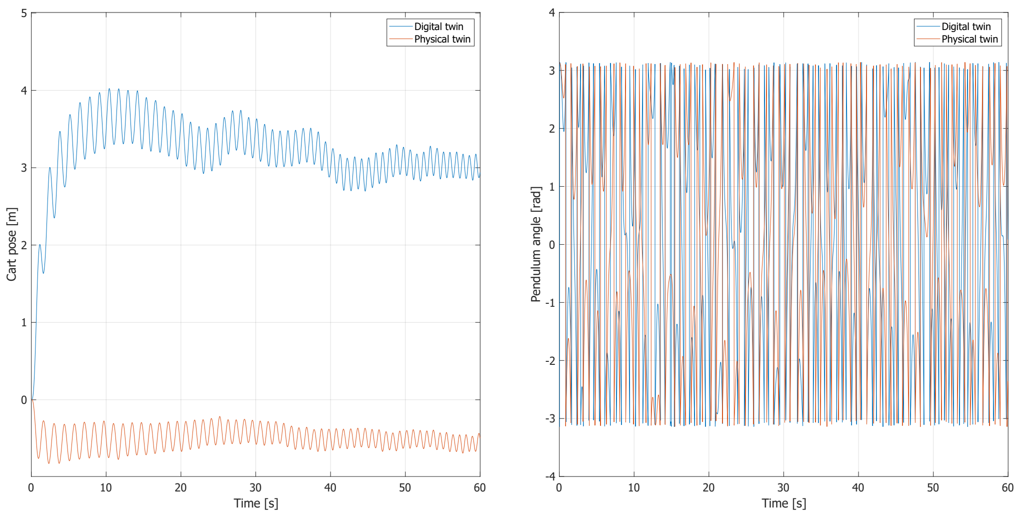
where $X$is the vector of configurable model parameters from the Table 1. In addition to 9 configurable parameters from mentioned table, the digital twin allows to control three additional variables, namely input force, cart velocity and pole velocity dead zones. The dead zones parameters allow to emulate the behavior of the real hardware, where in some signal ranges for input force, cart and pole velocities the system does not change its state. The RMSE equation from Eq. (28) is defined through the next equation:

$$\begin{aligned} RMSE(X) =&0.75 \cdot \sqrt{\frac{\sum_{t=0}^{N} \left(p_{physical_t} - p_{virtual_t}(X)\right)^2}{N}} + \dots \\ &0.25 \cdot \left(\sqrt{\frac{\sum_{t=0}^{N} \left(sin(\alpha)_{physical_t} - sin(\alpha)_{virtual_t}(X)\right)^2}{N}} + \dots \right. \\ &\left. + \sqrt{\frac{\sum_{t=0}^{N} \left(cos(\alpha)_{physical_t} - cos(\alpha)_{virtual_t}(X)\right)^2}{N}} \right) \end{aligned} \tag{29}$$

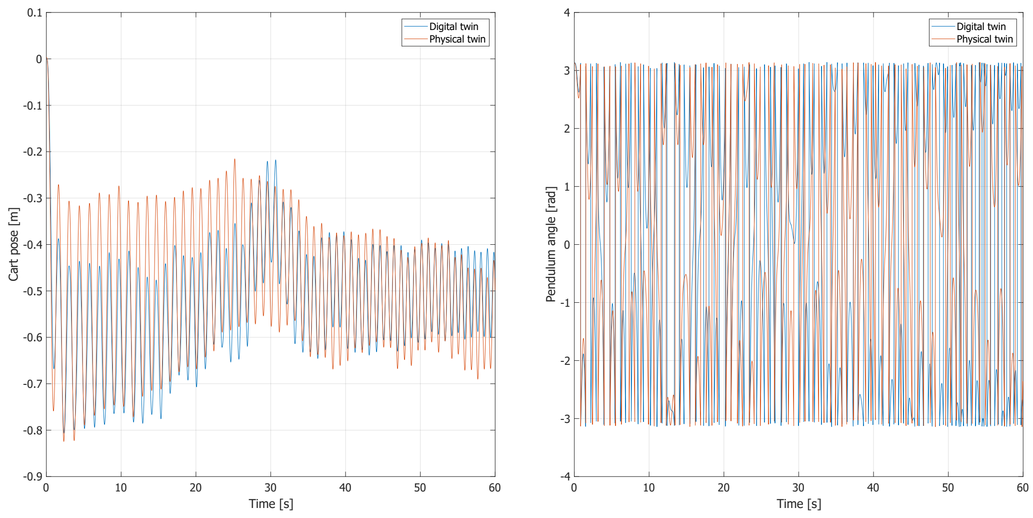
where N is the total number of time steps during the optimization iteration, p is the cart position, and α is the pole angle.

After the Nelder-Mead optimization epoch, the new set of parameters for the virtual model has been identified. The values of these updated parameters are given in Table 5. The model's optimization algorithm reduced the RMSE from 2.18 to 0.475 for the open loop identified data. The behavior of the optimized digital twin is illustrated in Fig. 13. The code for model optimization in Matlab is provided in Appendix B.

To solve the "sim-to-real" problem, it was required to train the controller on the updated digital model of the pendulum and test its inference on the real device. The



(a) Difference in dynamics between the unoptimized digital twin of pendulum and its physical twin.



(b) Difference in dynamics between the optimized digital twin of pendulum and its physical twin.

Figure 13. Results of the optimization of the digital twin of the pendulum. Both digital and physical twins were provided the single input in form of sinusoidal signal of varying frequency. To have identical dynamics, the trend for both physical and digital twins should coincide. The graph 13a shows how the behavior of unoptimized model has been different from actual pendulum. The graph 13b illustrates how dynamics became closer to the physical twin after the Nelder-Mead optimization.

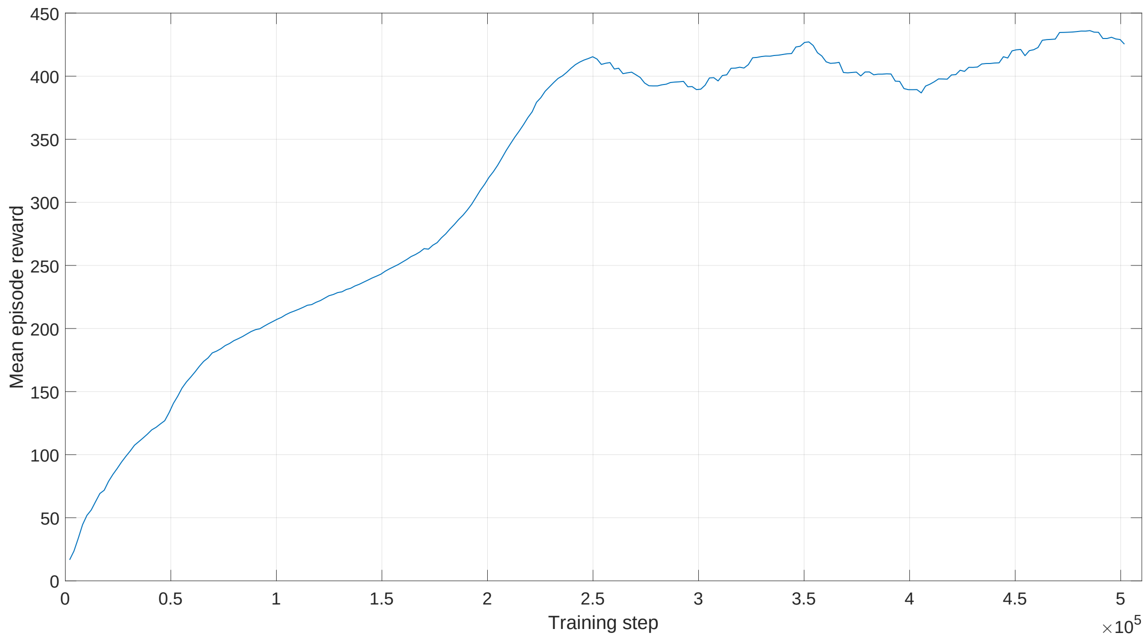
Table 5. Virtual model configurable parameters

Parameter	Unoptimized value	Optimized value
Mass of cart [kg]	0.5723	0.572066
Mass of pendulum [kg]	0.12	0.119648
Rotational friction coefficient [$\frac{Nms}{rad}$]	$27.344 \cdot 10^{-5}$	$27.6 \cdot 10^{-5}$
Static cart friction coefficient [N]	1.1975875	1.199442
Dynamic cart friction coefficient [$\frac{Ns}{m}$]	0.5	0.502926
Distance from axis of rotation to the center of mass [m]	0.01955717	0.019631
Moment of inertia related to mass centre [$kg \cdot m^2$]	0.0038583	0.003861
Gravity acceleration [$\frac{m}{s^2}$]	9.81	9.826466
Control signal to force ratio [N]	12.86	6.338787
Input force dead zone [N]	0.093125	0.227417
Cart velocity dead zone [$\frac{m}{s}$]	0.1	0.099952
Pole velocity dead zone [$\frac{rad}{s}$]	1.5	1.495618

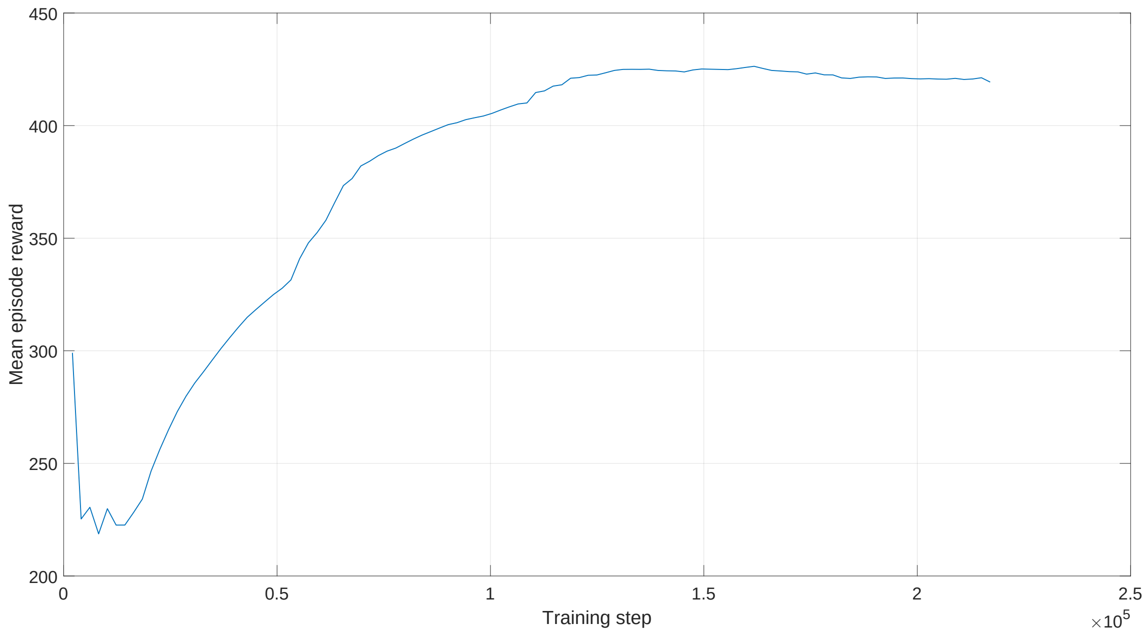
training trend for a new generation of the controller is given in Fig. 14a. By the 250 000 steps, a mean reward for the digital controller was 415, making it work ideally in a virtual environment and a good checkpoint to try an application on the real equipment.

After loading the mentioned checkpoint, the inference of the model on the actual system was better than the first zero-shot attempt. The cart was trying to swing up the pole, it was not exceeding the rail limits, but the rotational dynamics were not optimised very well. The pendulum was constantly rotating, trying to find a stabilization point. Actually, during some inference trials, the controller from a checkpoint could make a swing-up movement and hold the pendulum in the required position for a short period. It indicates a significant improvement in the virtual model accuracy, which could be used to gain a training advantage for the actual pendulum. Thus, instead of starting the controller training for the physical system from scratch, it was decided to start it from the abovementioned checkpoint. The training trend for such an approach is shown in Fig. 14b.

If one compares Fig. 11 with the newest one, it can be observed that the old controller



(a) Training trend of a new virtual controller against new model parameters. The required reward was achieved in 250 000 steps.



(b) Training trend for a real controller started from virtual controller's checkpoint. The mean episode reward has converged after 130 000 steps with a result that allowed to control a physical twin.

Figure 14. Trends of training of the controllers in "sim-to-real" use case. In 14a the controller is being trained on the optimized version of digital twin. In 14b we start the training from a checkpoint of the controller from the virtual model.

was not even able to reach such a level of mean reward in a span of the whole epoch.

The new trend has the following characteristics:

- First of all, it starts with already a very high reward. What has been observed during the start is that the controller, if the swing-up motion begins from the right side of the rail, was able to hold poles in an upright position. However, if the swing-up movement starts from the left side of the rail, it cannot stabilize it at all. This behavior created an impression that either rotational dynamics depended on rotation direction or the virtual controller had an overfit towards inaccurate dynamics.
- Secondly, the training trend shows a drastic quality drop in the first 15 000 steps. It most likely occurs due to the weights update of the overfitted neurons in neural network hidden layers.
- Thirdly, the rate of change of the reward suddenly boosts after 15 000 steps. It is achieving the stable episode reward of 400 in 1 hour. The trend in Fig. 11 did not reach such a good reward mean episode even in 7 hours of training. This behavior most likely shows time-dependent adaptability issues hidden within physical pendulum dynamics. Otherwise, the physical controller would obtain the mean reward of close to 400 in the first 250 000 steps, like in a virtual model trend, but it didn't happen.

Since the mean episode reward converged after 130 000 steps and did not experience any increasing trends in a span of 1 hour and 30 minutes, the training was finished before the epoch termination. It was possible to create a robust RL controller that took advantage of "sim-to-real" knowledge transfer. In 1 hour and 54 minutes, the new version of the controller converged to the mean reward above 400 hundred, that resulted in a very robust controller being able to hold the INTECO pendulum in an upright position.

6.2. Comparison of the controllers

The Table 6 is provided to explain the behavior of the DRL controller at different slices of mean episode reward. This data has been identified by observing the behavior of digital and physical twin controllers over the series of inference episodes. The description in the table fits both types of the inverted pendulum.

Three controllers have been designed as the result of the experiment that connected the DRL platform with the INTECO Inverted Pendulum system. The first controller is purely operational in the virtual environment. The second controller was trained from scratch on the physical twin of the pendulum. It did not manage to achieve the best mean average rewards but still performed the control with a drifting cart. After transferring the knowledge from the optimized virtual environment to a physical environment, the last controller appeared. The DRL model was trained from a specific checkpoint that allowed to reduce the training time on the equipment by a significant period. The overview of all controllers is given in Table 7 with a short description regarding their inference behavior.

Based on the results of this table, three controllers were designed to control the INTECO inverted pendulum. To control the virtual model of the pendulum provided by INTECO, one could use the DRL controller for the digital twin. When speaking about the control of the actual pendulum, it is possible to use the controller trained on a pendulum from scratch or the controller produced by solving the “sim-to-real” problem. The last option has the best quality that was achieved in the shortest time, making it the best DRL controller obtained during the experiment.

The main problem with controllers for the physical twin is adaptability. Controllers can not guarantee the same behavior during the inference after the long time break. This can be caused by the encoders reset, system dynamics changes, or training overfit during some hardware issues. For instance, the cart can start constantly moving in either direction without an explicit action input.

In the end, the DRL platform allowed to train controllers for digital and physical twins of the INTECO Inverted Pendulum. Also, with the help of system identification and optimization of the digital replica’s dynamics, it was possible to successfully build the controller for the physical twin that utilized the gained experience from a virtual model checkpoint.

Table 6. The behavior of RL model at different reward slices

Reward slice	Description
<50	The model stochastically moves on the rail. Usually, the system goes out of bounds and generates a negative reward. An episode usually terminates at this moment, and the cart moves to the center through a swing-down PID regulator.
50-100	The controller understands that it needs to keep the cart to the center of the rail and have the pendulum in an upright position for the maximum reward output. However, the cart still moves chaotically on a rail but usually does not go out of bounds. It is possible to observe the rotation of the poles on a rotational axis.
100-190	The controller rotates the pendulum with a car located near the center of the rail. The angular velocity of the rotation is decreased compared to the previous slice, but holding the pendulum in an upright position is still a problem.
190-300	The model understands that the biggest reward is obtained once the pendulum is located at an upright position with zero angular velocity. The cart is moving to hold the pendulum in the required angular pose for a short period. During this slice, the controller overfits towards instantaneous rewards from the pole. It is not necessarily trying to hold an upright position in the rail center.
300-400	The controller **starts** fulfilling the control criterion. It can hold the pole in an upright position. The biggest drawback at this stage is the behavior of the cart. The pendulum is stabilized, but the cart still drifts on the rail. Sometimes it can swing up the pole near the center of the rail, and sometimes, the pendulum is stabilized somewhere near the system's boundaries. Overall, the reward is high, and the pole never falls down, especially when reaching an approximate 380 average reward. However, the cart behavior needs to be optimized.
>400	If this slice is observed consecutively, it indicates that the training can be finished. After this mean episode reward, the controller can hold the pendulum upright in proximity to the rail center. Cart, in this case, does not experience sudden drifts in speed/position and performs a very stable and robust control of the movement once the pole has been swang up.

Table 7. Comparison of the controllers trained during the experiment. The results include the mean reward which was achieved at a specific step during the training. As well, training time to achieve that step is given.

Controller	Step	Mean reward	Training time	Description
The DRL controller for digital twin	280 000	400	3 hours and 24 minutes	This generation of controller has shown good trait at dynamic inference. The main problem of such controller is that it was trained on inaccurate digital twin. Thus, it is possible to use the DRL model only in virtual environment, inference of this controller in real environment can damage the equipment.
The DRL controller for physical twin, trained from scratch	400 000	340	6 hours and 26 minutes	The DRL controller for physical twin trained from scratch did not show an excellent reward convergence at high values. The long training time indicates the adaptability problem for the physical environment.
The DRL controller for physical twin trained from knowledge transfer	130 000	423	1 hour and 54 minutes	Training time has only took into account the training on an actual pendulum. The training time of the checkpoint was neglected because in "sim-to-real" problem it is a costless source of information. The controller itself shows the best training result in the shortest span of time. During the inference, the control of the pendulum was robust, meaning that cart was able to hold the poles in an upright position even during the application of external disturbances on the system.

7. Discussion

This section underlines the main points which could be concluded from the provided research, starting from the benefits/solved issue and finishing with the remaining problems and what could have been done better.

7.1. Main benefits

Out of all goals that have been achieved during the research, the listed below are bringing the most value:

1. Creation of the DRL platform in the form of a Python package. Any user can access it via the internet, refer to the Github wiki, and create a bridge for various applications, would it be Unreal Engine, Simulink, LabVIEW, Unity, or other software. The implementation of the platform environment, based on OpenAI gym, allows the platform to be used with multiple libraries for the DRL training.
2. The DRL platform helped create the controller that was able to pass knowledge from the digital environment to the actual system. The way the platform architecture provided an interface of interaction with both virtual and physical systems allowed to replace various controller versions and experiment with training algorithm hyperparameters relying purely on existing popular libraries, not some custom training loops.
3. While dealing with a "sim-to-real" use case, it was possible to optimize the digital replica of the INTECO pendulum to the next level of accuracy, which allowed the transfer learning to happen.
4. Even without considering "sim-to-real", it was possible to create a robust RL controller on a real physical system using only several lines of code for the training loop.

The most significant advantage is the reduction of training time for the physical controller by almost seven times compared to the controller trained from scratch.

Moreover, to design such a controller, no knowledge of the internal dynamics of the systems was required. And to additionally point out, the hyperparameters for training were not benchmarked and optimized, meaning that training time overall can be reduced. Any reduction in non-profitable machine utilization reduces the total cost of equipment. As it is a discussion section, let's introduce some imagined example to outline certain benefits of using the DRL platform with "sim-to-real" expertise.

7.2. Main problems

Besides the achievements, the research has encountered particular problems which are still not mitigated. Possibly, some points from the following subsection could have resolved some of the issues, but there is no explicit guarantee.

1. The controller for the physical pendulum has an adaptation problem. There is no guarantee that the freshly tuned controller will operate robustly after the day of operation, especially if the encoders are being reset.
2. The software design of the DRL platform remains simple and could be enhanced with an advanced integration applied to specific applications or training libraries.
3. The optimization of the INTECO virtual pendulum was not ideal. Though the cart dynamics seemed to have excellent accuracy, the rotational dynamics of the pole were not possible to configure through the Nelder-Mead algorithm. There are not enough parameters in digital replica to compensate for those dynamics.
4. Hyperparameters of the PPO training algorithm have been selected based on some custom benchmark pendulum environment. Tuning these parameters was not performed, so the training time is still relatively high.

These outlined problems could hide some other issues in addition. The biggest problems could be considered for the next iteration of the research related to "sim-to-real" transfer learning and the DRL platform.

7.3. Possible future steps

It is already possible to state what could have been tried out to achieve better account at each domain: for DRL platform, controller adaptability, training optimization, etc.

1. System Identification as a transfer learning method allowed to optimize the dynamics of virtual model. However, one can try to introduce the Domain Randomization method, which helps to change the internal dynamics of the digital replica of the pendulum during the training. This way the controller could also inherit the dynamics that cause the adaptation issue.

2. The Stable-Baselines3 Zoo or other methods for hyperparameter tuning could be used. Hyperparameter tuning can significantly decrease the training time for all three pendulum training use-cases: virtual, physical, and "sim-to-real".

3. There are other INTECO devices having their respective digital replicas provided by the manufacturer. The DRL platform can be applied to INTECO 3D Crane, INTECO Magnetic Levitation system, and other devices that could improve the overall quality of the DRL platform.

4. Creation of Input/Output Data hubs/hooks is done manually within the external application. It might be possible to design a script for deploying the needed components to the software automatically, making no need for the user to even take care of the platform configuration in an external environment. Of course, this solution would imply that automation scripts will be biased to the specific application.

5. Multi-agent environments had not been tested in the platform. Though architecture does not provide any problems for it, the trial of such an approach might increase the platform's user experience if some design points could have been done better.

Overall, taking the subsections above into account, the research provided a lot of specific benefits that could be utilized in future works or even used by other people. The number of particular problems and determined action points indicates a steady direction for improvement, which could bring the DRL platform to a higher level within the research domain.

8. Conclusions

The following research has proven that it is possible to design a deep reinforcement learning controller that can operate the real pendulum by partially inheriting the dynamics from its digital twin. With the help of the deep reinforcement learning platform, it was possible to create controllers for three different use-cases:

- Training the digital inverted pendulum system - creating the PPO controller that can operate the digital twin of an inverted pendulum. It took 3 hours and 24 minutes to achieve the model with very good quality.
- Training of the physical inverted pendulum system from scratch - creating the PPO controller that can operate the real equipment in a laboratory environment robustly. It took 6 hours and 26 minutes to achieve the model with good quality.
- Training of the physical inverted pendulum using "sim-to-real" learning - creating the PPO controller that could inherit some knowledge of virtual dynamics at inference for the actual pendulum. It has only taken 1 hour and 54 minutes to achieve higher quality than the two models above.

The System Identification method was used to transfer the virtual training experience to a real pendulum and solve the "sim-to-real" challenge during the controller design. Based on the quality increase of the last abovementioned controller, the approach has been successful. However, problems with rotational dynamics and system adaptation remain.

These issues could be resolved by Domain Randomization and PPO hyperparameters tuning, which were not considered in this research. Besides creating the "sim-to-real" controller, the accuracy of the INTECO digital replica allows it to be used for other research related to the identification of systems or control of the equipment.

The deep reinforcement learning platform can be downloaded as a Python package. Ease of interface for interaction enables users to focus more time on the research subject and not on the custom implementation of integration scripts or training algorithms compatible with Unreal Engine, Simulink, Unity, LabVIEW, or other applications.

To sum up, the conducted research provided a controller that was preliminary taught in the virtual version of the inverted pendulum. In addition, the solution to the "sim-to-real" problem gave a bridge for the comfortable use of popular training libraries with a variety of control objects running outside of the main training loop. Besides the "sim-to-real" use case for the INTECO inverted pendulum system, it is possible to proceed with other control equipment, such as the INTECO 3D crane or the INTECO magnetic levitation system.

References

[1] Inteco, "Pendulum-cart system: User's manual," a-lab.ee, Feb. 2013. [Online]. Available: https://a-lab.ee/man/Pendulum-user-manual.pdf

[2] P. Leitao, F. Pires, S. Karnouskos, and A. W. Colombo, "Quo vadis industry 4.0? position, trends, and challenges," *IEEE Open Journal of the Industrial Electronics Society*, vol. 1, pp. 298–310, 2020.

[3] F. Lamnabhi-Lagarrigue, A. Annaswamy, S. Engell, A. Isaksson, P. Khargonekar, R. M. Murray, H. Nijmeijer, T. Samad, D. Tilbury, and P. V. den Hof, "Systems & control for the future of humanity, research agenda: Current and future roles, impact and grand challenges," *Annual Reviews in Control*, vol. 43, pp. 1–64, 2017.

[4] D. T. Consortium, "Definition of a digital twin," digitaltwinconsortium.org, May 2022. [Online]. Available: https://www.digitaltwinconsortium.org/initiatives/the-definition-of-a-digital-twin.htm

[5] S. Jersov and A. Tepljakov, "Digital twins in extended reality for control system applications," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, jul 2020.

[6] Y. Li, "Deep reinforcement learning," 2018. [Online]. Available: https://arxiv.org/abs/1810.06339

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016.

[8] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[9] H. Moradi, M. T. Masouleh, and B. Moshiri, "Robots learn visual pouring task using deep reinforcement learning with minimal human effort," in *2021 9th RSI International Conference on Robotics and Mechatronics (ICRoM)*. IEEE, nov 2021.

[10] K. Zhu and T. Zhang, "Deep reinforcement learning based mobile robot navigation: A review," *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, oct 2021.

[11] N. Mellatshahi, S. Mozaffari, M. Saif, and S. Alirezaee, "Inverted pendulum control with a robotic arm using deep reinforcement learning," in *2021 International Symposium on Signals, Circuits and Systems (ISSCS)*. IEEE, jul 2021.

[12] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, dec 2020.

[13] M. Sisin, "The drl python package," PyPi, Apr. 2022. [Online]. Available: https://pypi.org/project/drl-platform/0.1/

[14] L. Buşoniu, T. de Bruin, D. Tolić, J. Kober, and I. Palunko, "Reinforcement learning for control: Performance, stability, and deep approximators," *Annual Reviews in Control*, vol. 46, pp. 8–28, 2018.

[15] P. Ramanathan, K. K. Mangla, and S. Satpathy, "Smart controller for conical tank system using reinforcement learning algorithm," *Measurement*, vol. 116, pp. 422–428, feb 2018.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv.org*, Sep. 2015.

[17] H. Iwasaki and A. Okuyama, "Development of a reference signal self-organizing control system based on deep reinforcement learning," in *2021 IEEE International Conference on Mechatronics (ICM)*. IEEE, mar 2021.

[18] M. Saeed, M. Nagdi, B. Rosman, and H. H. S. M. Ali, "Deep reinforcement learning for robotic hand manipulation," in *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*. IEEE, feb 2021.

[19] J. Woo, C. Yu, and N. Kim, "Deep reinforcement learning-based controller for path following of an unmanned surface vehicle," *Ocean Engineering*, vol. 183, pp. 155–166, jul 2019.

[20] G. C. Lopes, M. Ferreira, A. da Silva Simoes, and E. L. Colombini, "Intelligent control of a quadrotor with proximal policy optimization reinforcement learning,"

in *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)*. IEEE, nov 2018.

[21] P. Carvalho, "Solving openai gym environments with matlab rl toolbox," medium.com, Apr. 2020. [Online]. Available: https://medium.com/analytics-vidhya/solving-openai-gym-environments-with-matlab-rl-toolbox-fb9d9e06b593

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[23] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Comput. J.*, vol. 7, pp. 308–313, 1965.

[24] R. M. Lewis, V. Torczon, and M. W. Trosset, "Direct search methods: then and now," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 191–207, dec 2000.

[25] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 112–147, jan 1998.

[26] UniversalRobots, "Offline simulator - e-series - ur sim for non linux 5.9.4," universal-robots.com, Dec. 2021. [Online]. Available: https://www.universal-robots.com/download/software-e-series/simulator-non-linux/offline-simulator-e-series-ur-sim-for-non-linux-594/

[27] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[28] OpenAI, "Getting started with openai gym," gym.openai.com, 2022. [Online]. Available: https://gym.openai.com/docs/

[29] M. F. Argerich, "List of reinforcement learning environments," Medium.com, Apr. 2019. [Online]. Available: https://medium.com/@mauriciofadelargerich/reinforcement-learning-environments-cff767bc241f

[30] M. Sisin, "Deep reinforcement learning platform," github.com, Apr. 2022. [Online]. Available: https://github.com/Senerader/DeepReinforcementLearningPlatform

[31] V. Lyashenko and P. Januszewski, “The best tools for reinforcement learning in python you actually want to try,” Neptune.ai, Nov. 2021. [Online]. Available: https://neptune.ai/blog/the-best-tools-for-reinforcement-learning-in-python

[32] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[33] I. The MathWorks, “Train ddpg agent to swing up and balance pendulum,” mathworks.com. [Online]. Available: https://www.mathworks.com/help/reinforcement-learning/ug/train-ddpg-agent-to-swing-up-and-balance-pendulum.html

[34] M. Sisin, “Master’s thesis source code and templates,” github.com, Apr. 2022. [Online]. Available: https://github.com/Senerader/TalTechMasters2022

[35] INTECO, “Pendulum and cart control system,” inteco.com.pl. [Online]. Available: http://www.inteco.com.pl/products/pendulum-cart-control-system/

[36] A. Raffin, “Rl baselines3 zoo,” https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

A. Non-exclusive license

B. Software interfaces and classes

Interface for creation of Gym-compatible platform environment:

```
class PlatformEnvironment(gym.Env):

    def __init__(self) -> None:
        super(PlatformEnvironment, self).__init__()
        self._env_server: Optional[AbstractServer] = None

    def reset(self):
        raise NotImplementedError("Reset method should be implented
           specifically for each Gym environment")

    def step(self, _):
        raise NotImplementedError("Step method should be implented
           specifically for each Gym environment")

    @property
    def env_server(self) -> AbstractServer:
        """
        Interface for getting the environment server in Gym environment
        """
        assert self._env_server, "Environment server should be injected in
           runtime"
        return self._env_server

    @env_server.setter
    def env_server(self, server: AbstractServer):
        """
        Interface for setting the environment server in Gym environment at
           runtime
        """
        self._env_server = server
```

The Platform Environment implementation for the sample pendulum use case:

```
class SampleCartpole(PlatformEnvironment):
    """
    Custom Simulink env with non-realtime execution
```

```
"""
def __init__(self):
    super(SampleCartpole, self).__init__()
    self._logger = logging.getLogger(__name__)
    sin_max = 1
    cos_max = 1
    velocity_max = np.finfo(np.float32).max
    high = np.array(
        [
            sin_max,
            cos_max,
            velocity_max
        ],
        dtype=np.float32,
    )
    self.observation_space = Box(-high, high, dtype=np.float32) #type:
       ignore
    self.action_space = Discrete(3)
    self.angle_threshold = 12 * 2 * math.pi / 360
    self.vel_threshold = 10

def reset(self):
    self.done = False
    self._logger.warning("ENV RESET")
    self.env_server.send_payload([0, 1],">dd")
    payload = self.env_server.receive_payload(">ddd")
    sin, cos, vel = \
        payload[0], payload[1], payload[2]
    return np.asarray([sin, cos, vel])

def step(self, action):
    assert action in [0, 1, 2], action
    if action == 0:
        force = -2
    elif action == 1:
        force = 0
    elif action == 2:
        force = 2
    else:
        raise ValueError(f"Unknown action {action}")
        self.env_server.send_payload([force, 0], ">dd")
        payload = self.env_server.receive_payload(">ddd")
        sin, cos, vel = \
            payload[0], payload[1], payload[2]
        angle = atan2(sin, cos)
        self._logger.info(f"Angle: {angle}")
        self.done = bool(
            vel < -self.vel_threshold
            or vel > self.vel_threshold
        )
        #  Full range reward structure (normalized)
        self.reward = -1 * (abs(angle/math.pi) -1)
        self._logger.info(f"Reward: {self.reward}")
        return np.array([sin, cos, vel], dtype=np.float32), self.reward,
            self.done, {}

if __name__ == "__main__":
    try:
        from stable_baselines3.ppo.ppo import PPO #type: ignore
        from stable_baselines3.common.callbacks import CheckpointCallback #
            type: ignore
    except:
        print("To execute, this code must have Stable-Baselines3 installed")
        exit()
    env_server = UdpServer(
        "127.0.0.1",
        16385,
        "127.0.0.1",
        16384
    )
    env_server.start_server()
    gym.envs.register(
        id='SampleCartpole-v0',
        entry_point='sample_cartpole:SampleCartpole',
        max_episode_steps=500,
        reward_threshold=500,
    )
    env = gym.make('SampleCartpole-v0') #type: ignore
    env.env.env_server = env_server #type:ignore
    model = PPO('MlpPolicy', #type: ignore
                env,
                verbose=1,
                tensorboard_log=f"Logs/SamplePole/PPO",
                device="cpu"
                )
    try:
        checkpoint_callback = CheckpointCallback(save_freq=5000, save_path=f
            'Models/SamplePole/PPO') #type: ignore
        model.learn(total_timesteps=500_000, callback = checkpoint_callback,
             tb_log_name=f"PPO")
    finally:
        env_server.close_server()
```

The Platform Environment implementation for the INTECO Inverted Pendulum use case:

```
import logging
from typing import Optional
import gym
import math
import numpy as np
import time
from gym import spaces
from DrlPlatform import PlatformEnvironment

class InvertedPendulumRT(PlatformEnvironment):
    """Custom class that represents the Pendulum environment
    Limits were taken based on INTECO documentation of Fuzzy controller
    Observation:
        Type: Box(5)
        Num     Observation               Min                     Max
        0       Cart Position             -5                      5
        1       Cart Velocity             -inf                    inf
        2       Sine of Angle             -1                      1
        3       Cosine of Angle           -1                      1
        4       Pole Angular Velocity     -inf                    inf
    Actions:
        Type: Box(1)
        Num   Action                      Min                     Max
        0     PWM                        -0.5                   0.5
    """

    def __init__ (self):
        super(InvertedPendulumRT, self).__init__()
        self._logger = logging.getLogger(__name__)
        self.cart_pos_max = 1
        self.cart_vel_max = np.finfo(np.float32).max
        self.pole_sine_max = 1
        self.pole_cosine_max = 1
        self.pole_angular_vel_max = np.finfo(np.float32).max
        high = np.array(
            [
                self.cart_pos_max,
                self.cart_vel_max,
                self.pole_sine_max,
                self.pole_cosine_max,
                self.pole_angular_vel_max
            ],
            dtype=np.float32,
        )
        self.observation_space = spaces.Box(-high, high, dtype=np.float32) #
           type: ignore
        self.action_space = spaces.Box(
            low=-0.5, high=0.5, shape=(1,), dtype=np.float32
        )
        self.total_reward = 0.0
        self.done = False
        self.cart_vel_threshold = 3
        self.pole_vel_threshold = 15
        self.cart_pos_threshold = 0.7
        # Historical data
        self.previous_cart_pose = None
        self.previous_cart_velocity = None
        self.previous_pend_pose = None
        self.previous_pend_vel = None
        self.previous_action = None

    def _receive_payload(self):
        # Returns cart_pose, cart_vel, pend_
        payload = self.env_server.receive_payload(">dddd")
        formatted_payload = self.format_payload(payload)
        cart_pose, cart_vel, pendulum_pose, pendulum_vel = \
            formatted_payload[0], formatted_payload[1], formatted_payload
                [2], formatted_payload[3]
        if not (self.previous_cart_pose and self.previous_cart_velocity \
                and self.previous_pend_pose and self.previous_pend_vel):
                self.previous_cart_pose = cart_pose
                self.previous_cart_velocity = cart_vel
                self.previous_pend_pose = pendulum_pose
                self.previous_pend_vel = pendulum_vel
        else:
            # Noise detection
            while True:
                if abs(self.previous_cart_pose - cart_pose) > self.
                    cart_pos_threshold or \
                abs(self.previous_pend_vel - pendulum_vel) > self.
                    pole_vel_threshold or \
                abs(self.previous_cart_velocity - cart_vel) > self.
                    cart_vel_threshold:
                    self._logger.warning("Detected noise")
                    payload = self.env_server.receive_payload(">dddd")
                    formatted_payload = self.format_payload(payload)
                    cart_pose, cart_vel, pendulum_pose, pendulum_vel = \
                        formatted_payload[0], formatted_payload[1],
                            formatted_payload[2], formatted_payload[3]
                    continue
                else:
                    break
        self.previous_cart_pose = cart_pose
        self.previous_cart_velocity = cart_vel
        self.previous_pend_pose = pendulum_pose
        self.previous_pend_vel = pendulum_vel
        return cart_pose, cart_vel, pendulum_pose, pendulum_vel


    def step(self, action):
        # Negative reward for a step
        info = {}
        # Sending action through env server
```

```
self.previous_action = action
self.env_server.send_payload(
    payload=[action, 0],
    sending_mask=">dd"
)
time.sleep(0.031)
cart_pose, cart_vel, pendulum_pose, pendulum_vel = \
    self._receive_payload()
pendulum_sine = math.sin(pendulum_pose)
pendulum_cosine = math.cos(pendulum_pose)
self._logger.info(f"Pendulum pose: {pendulum_pose}\nPendulum vel: {
    pendulum_vel}\nCart pose: {cart_pose}\nCart vel: {cart_vel}")
# Cart reward calculation
if not self.done:
    self.done = bool(
        pendulum_vel < -self.pole_vel_threshold
        or pendulum_vel > self.pole_vel_threshold
    )
    if self.done:
        self.reward = -30
if not self.done:
    self.done = bool(
        cart_vel < -self.cart_vel_threshold
        or cart_vel > self.cart_vel_threshold
    )
    if self.done:
        self.reward = -30
if not self.done:
    self.done = bool(
        cart_pose < -self.cart_pos_threshold
        or cart_pose > self.cart_pos_threshold
    )
    if self.done:
        self.reward = -50
# self.reward = -((((pendulum_pose + np.pi) % (2 * np.pi)) - np.pi)
    ** 2 \
#              + 0.1 * pendulum_vel ** 2 + 0.01 * (cart_vel ** 2))
self.reward = self._pendulum_pose_reward(pendulum_pose) * \
              self._pendulum_velocity_reward(pendulum_vel) * \
              (self._cart_pose_reward(cart_pose) + \
                    self._cart_velocity_reward(cart_vel))
        self.total_reward += self.reward
        formatted_payload = np.array([cart_pose, cart_vel, pendulum_sine,
            pendulum_cosine, pendulum_vel], dtype=np.float32)
        # Return the result
        self._logger.info(f"Reward: {self.reward}")
        return formatted_payload, self.reward, self.done, info

    def _pendulum_pose_reward(self, angle: float):
        # return 1 at top pose
        return -1 * (abs(angle/math.pi) -1)

    def _pendulum_velocity_reward(self, velocity: float):
        # return 1 at zero velocity
        return (self.pole_vel_threshold - abs(velocity))/self.
            pole_vel_threshold

    def _cart_pose_reward(self, pose: float):
        # return 1 at zero pose
        return (self.cart_pos_max - abs(pose))/self.cart_pos_max * 0.75

    def _cart_velocity_reward(self, velocity):
        # return 1 at zero velocity
        return (self.cart_vel_threshold - abs(velocity))/self.
            cart_vel_threshold * 0.25


    def reset(self) -> np.ndarray:
        self.counter = 0
        self.total_reward = 0
        self.done = False
        self.previous_cart_pose = None
        self.previous_cart_velocity = None
        self.previous_pend_pose = None
        self.previous_pend_vel = None
        self._logger.warning("ENV RESET")
        if not self.previous_action:
            self.previous_action = 0
        self.env_server.send_payload([-self.previous_action, 1],
                        sending_mask=">dd")
        self.previous_action = None
        time.sleep(0.031)
        payload = self.env_server.receive_payload(">dddd")
        formatted_payload = self.format_payload(payload)
        while not self._is_env_reset(formatted_payload):
            self.env_server.send_payload([0, 1],
                        sending_mask=">dd")
            time.sleep(0.031)
            payload = self.env_server.receive_payload(">dddd")
            formatted_payload = self.format_payload(payload)
        self._logger.warning("ENV RESET DONE")
        cart_pose, cart_vel, pendulum_pose, pendulum_vel = \
            formatted_payload[0], formatted_payload[1], formatted_payload
                [2], formatted_payload[3]
        self._logger.info(f"Pendulum pose: {pendulum_pose}\nPendulum vel: {
            pendulum_vel}\nCart pose: {cart_pose}\nCart vel: {cart_vel}")
        pendulum_sine = math.sin(pendulum_pose)
        pendulum_cosine = math.cos(pendulum_pose)
        formatted_payload = np.array([cart_pose, cart_vel, pendulum_sine,
            pendulum_cosine, pendulum_vel], dtype=np.float32)
        return formatted_payload

    def format_payload(self, payload) -> np.ndarray:
        """
        Formatting payload which is going to be sent to training agent
        """
        formatted_payload = np.array(payload, dtype=np.float32)
        # Array poses:
        # 1. Pendulum position
        # 2. Pendulum velocity
        # 3. Cart position
        # 4. Cart velocity
        pendulum_pose, pendulum_velocity, cart_pose, cart_velocity = \
            formatted_payload[0], formatted_payload[1], formatted_payload
                [2], formatted_payload[3]
        return np.array([cart_pose, cart_velocity, pendulum_pose,
            pendulum_velocity], dtype=np.float32)

    def _is_env_reset(self, formatted_payload) -> bool:
        """
        Check if env has been reset
        """
        cart_pose, cart_velocity, pendulum_pose, pendulum_velocity = \
            formatted_payload[0], formatted_payload[1], formatted_payload
                [2], formatted_payload[3]
        counter = 0
        is_reset = True
        while counter <= 5:
            is_reset = is_reset * (abs(pendulum_pose) > 2.9) \
            and (abs(pendulum_velocity) < 1) \
            and ((abs(cart_velocity) < 0.2)) \
            and (abs(cart_pose) < 0.1)
            counter += 1
        return is_reset
```

MATLAB snippets for pendulum model optimization

```
% Anonymous function handler
optimizer_pend=@(x) function_optimizer_pendulum_all(x(1));

% Option to optimize
options = optimset('PlotFcns',@optimplotfval);
% mc, mp, fp, FS, FC, I, J, g, M, DZu, DZcv, DZcp
x0 = [
    0.5723
    0.12
    0.00027344
    1.1975875
    0.5
    0.019557
    0.003858
    9.81
    7.0
    0.093125
    0.1
    1.5]; % Initial solution
% mc, mp, fp, FS, FC, I, J, g, M, DZu, DZcv, DZcp
lb = [
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0];
% mc, mp, fp, FS, FC, I, J, g, M, DZu, DZcv, DZcp
ub = [
    1
    1
    0.01
    2
    1.5
    0.1
    0.01
    15
    13
    2
    1
    3];
options = optimset('Display','iter', 'TolX',1e-7);
x = fminsearchbnd(@function_optimizer_pendulum_all, x0, lb, ub, options)
```

The RMSE calculation function in MATLAB:

```
function [rmse] = function_optimizer_pendulum_all(x)
    mc = x(1);
    mp = x(2);
    fp = x(3);
    FS = x(4);
    FC = x(5);
    I = x(6);
    J = x(7);
    g = x(8);
    M = x(9);
    DZu = x(10);
    DZcv = x(11);
    DZcp = x(12);
    simopt=simset('solver','ode5','SrcWorkspace','base', 'FixedStep', 0.01);
    load('OLTestData');
    set_param('pidvirtual/Dynamics', 'm', sprintf('[%f, %f]', mc, mp), ...
                                     'fr', sprintf('[%f, %f, %f]', fp, FS,
                                        FC), ...
                                     'P4', sprintf('[%f, %f, %f]', I, J, g),
                                         ...
                                     'Up', sprintf('[%f, %f]', M, DZu), ...
                                     'DZ', sprintf('[%f, %f]', DZcv, DZcp));
    sim('IntecoVirtualOptimisation.slx',[0 120],simopt);
    StatesMatrix = new_states;
    optimized_sin = sin(StatesOptimization.Data(2:12001, 1));
    physical_sin = sin(StatesMatrix(2:12001, 2));
    optimized_cos = cos(StatesOptimization.Data(2:12001, 1));
    physical_cos = cos(StatesMatrix(2:12001, 2));
    optimized_cart = StatesOptimization.Data(2:12001, 3);
    physical_cart = StatesMatrix(2:12001, 4);
    rmse = 0.75*sqrt(sum((optimized_cart - physical_cart).^2)/ 12000) + ...
            0.25*(sqrt(sum((optimized_sin - physical_sin).^2)/ 12000) + ...
            sqrt(sum((optimized_cos - physical_cos).^2)/ 12000));
    clear StatesOptimization;
end
```