From Simulation to Reality: Deployment of Reinforcement Learning-Based Neural Network Controllers Trained with Modelica Models

Joshua Brun[1] Thomas Sergi[1] Sylvan Mutter[1] Tim Arnold[1] Ulf Christian Müller[1]

[1]IME, Lucerne School of Engineering and Architecture, Switzerland,
{joshua.brun, ulfchristian.mueller}@hslu.ch

Abstract

To address limitations of traditional control methods in complex systems, reinforcement learning (RL) combined with simulation models provides an efficient approach for controller development. In this work, a complete toolchain for developing and deploying RL-based neural network controllers using Modelica system models is presented. Serving as a showcase, a real-world double-inverted pendulum is constructed. The system was modeled in Modelica by combining physics-based and data-driven modeling approaches for efficient development. Coupling simulation with an RL environment is achieved through Functional Mock-up Interface (FMI) standard. Successful training and sim-to-real transfer are demonstrated on a single-pendulum setup, validating the approach for extension to a double-inverted pendulum. This paper provides a reproducible and extensible framework which is well-suited for educational purposes and highlights strengths of Modelica in combination with machine learning approaches.

Keywords: data-driven system modeling, reinforcement learning, sim-to-real transfer, FMI, Modelica, double-inverted pendulum

1 Introduction

Combining reinforcement learning (RL) with system models is a promising approach, as it enables generation of training data using simulations of a system. In this work, a neural network control system is trained using RL coupled with a Modelica system model. After training, the controller is successfully applied to a real system, showcasing challenges and feasibility of this toolchain.

1.1 Motivation

Development of advanced controllers for complex systems requires deep expertise and can even become infeasible with traditional control system methods. This is where RL can fill a gap by learning a control strategy on its own, using a data-driven approach. However, real-world data is limited and costly, which is why generating required training data with a virtual system model is often the only possible path. By combining RL with physics-based simulation models, such as those typically built using Modelica, it becomes feasible to generate efficient training data while ensuring that learned controllers respect underlying system dynamics.

Methods combining RL with physics-based simulation models and sim-to-real transfer of neural network control systems have already been successfully applied in recent studies. Lukianykhin and Bogodorova (2019) introduced a toolbox for integration of Modelica-based models in RL via FMI ('Functional Mock-up Interface', 2014) standard. Heuermann et al. (2023) demonstrated that machine learning surrogate models can accelerate Modelica-based simulations by replacing nonlinear algebraic loops, thereby enabling more efficient training of RL controllers. Lee, Ju and Lee (2025) showcased the use of RL-based neural network control systems and sim-to-real transfer for transition control in a double-inverted pendulum on a cart-pole system, using a model based on Euler-Lagrange differential equations. These successful applications provide a strong foundation for further research that combines these approaches, as pursued in this work. The importance of such methods is expected to grow as systems get more complex in Industry 4.0 and beyond. While alternative approaches, such as *Model Predictive Control* (MPC), which rely on online simulation or optimization, also offer significant benefits, they are often less practical, especially for time-critical applications, due to their substantial computational demands (Morcego et al., 2023). A combination of RL and physics-based simulation models has already been addressed in research projects conducted in collaboration with industry. Among them is *IntelliBake* (INNOSUISSE, 2025), which investigates the application of RL-based neural network control systems for baking processes. Building upon these related efforts, this work contributes to this growing field by providing a demonstrator system and a reusable toolchain for RL-based controller development and deployment.

1.2 Goal

This work demonstrates the development of a neural-network control system trained entirely through RL on a Modelica model. Trained controllers are subsequently transferred to a real-world system, expected to perform a desired control task. This contribution establishes a working toolchain that serves both as proof of concept and as a foundation for future industrial and educational applications. In doing so, it aims to highlight typical challenges and potential solutions associated with following steps:

- System modeling with Modelica for state calculation during RL process.
- Training a neural network control system using RL and Modelica-based simulations.
- Sim-to-Real transfer of a trained neural network control system.

2 Showcase Selection and Setup

The chosen showcase system is a double-inverted pendulum. A motor drives the first pendulum, while the second pendulum is attached via a freely rotating plain bearing. The control task is to swing both pendulums from their initial downward-hanging position to an upright position. This involves two control phases: (1) swinging both pendulums upward by actuating the first joint and (2) stabilizing them in an upright position.

Figure 1. 3D model of the double-inverted pendulum.

This system was selected as a suitable showcase for following reasons:

- *Simple and compact construction:* This system is lightweight, portable, and space-efficient, making it suitable for use in various locations.
- *Demonstrative system for educational purposes:* A double-inverted pendulum allows for direct visual interpretation of control performance, making success and failure immediately apparent.
- *Benchmark character:* A double-inverted pendulum is a well-established benchmark in control theory, enabling direct comparison with related research.
- *Stepwise complexity:* A double-inverted pendulum can be easily simplified to a single pendulum, significantly reducing complexity. This allows initial focus on validation of the toolchain before addressing more advanced control challenges.
- *Challenging sim-to-real conditions:* The System's sensitivity to model inaccuracies (e.g., calibration errors) and a required high control frequency are creating challenges for a sim-to-real transfer, making it ideal for studying transferability.

2.1 System Construction

This mechanical setup features 3D-printed housing and pendulums, a metal flange on the motor mount, and a plain bearing between two pendulums. The assembly is clamped to a table. Physical parameters listed in Table 1 were derived from a computer-aided design (CAD) geometry.

Table 1. Physical parameters of the showcase system

Parameter	*Pendulum 1*	*Pendulum 2*
Mass	0.099 kg	0.061 kg
Length	0.19 m	0.19 m
COM distance from joint center	0.111 m	0.076 m
Moment of inertia	1.9e-3 kg·m^2	7.3e-4 kg·m^2

A brushless DC motor (GB54-2 Gimbal Type 3-6S UAV Drone Motor KV26 by T-Motors) serves as actuator and is controlled via a motor driver (SimpleFOCShield v2.0.4). A hollow motor shaft design allows for internal cable routing, and its permanent magnet rotor enables unrestricted rotation of pendulums. Two 14-bit absolute capacitive encoders (AMT222B by CUI Devices) are used to measure joint angles of both pendulums. These encoders provide direct high-resolution angular measurements for sensor feedback in a control loop. A neural network control system computes a motor voltage in real-time based on sensor inputs. These control signals are transmitted via serial communication to an Arduino Uno R3, which interfaces with electronic components. In this current configuration, the maximum achievable control loop frequency is approximately 250 Hz.

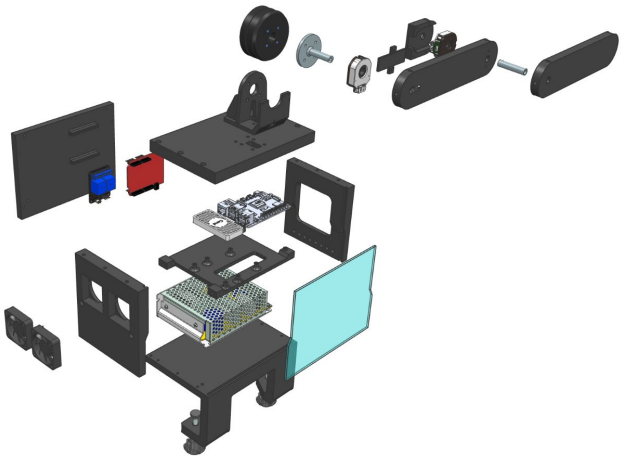

Figure 2. Exploded view of double-inverted pendulum.

DOI 10.3384/ecp218921

3 System Modeling: Combining Data-based and Physics-based Modeling in Modelica

Successful RL training of a neural-network controller requires an adequately accurate system model. This section outlines the modeling approach for a double-inverted pendulum. Due to the nonlinearity of its governing equations, a double-inverted pendulum exhibits chaotic behavior, making modeling and controlling particularly challenging when using analytical or traditional numerical methods.

This model primarily utilizes components from the `Mechanics` package within the `Modelica Standard Library` (Modelica Association, 2020). An exception is the brushless DC (BLDC) electric motor, which is modeled using a data-driven approach. This method will be described in section 3.1.

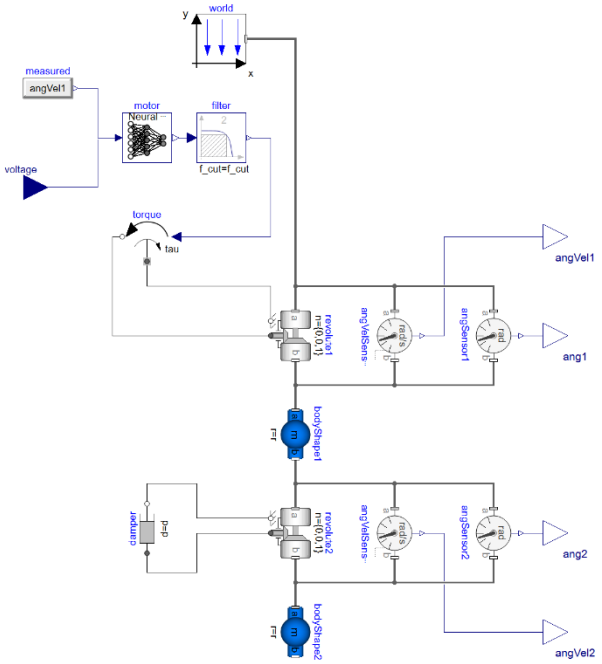

Figure 3. Graphical representation of the Modelica model for the double-inverted pendulum system.

Figure 3 shows a graphical representation of the system model used to train the neural network controller. The model uses two body shape components to precisely model unevenly distributed moments of inertia using a simple method. The pendulums are connected to each other by a joint with a single degree of freedom for rotational motion. Torque is calculated from a data-driven motor model and applied to the first pendulum. For both joints, bearing friction is considered. Friction torque as a function of angular velocity was derived through model calibration using experimental data and mathematical optimization with a Python library called Pygmo. (Biscani and Izzo, 2020). The optimization objective was to minimize normalized root mean squared error (NRMSE) between measurement data and simulation results.

This model has six boundary connectors to interface with an RL environment: two inputs (motor voltage and angular velocity feedback) and four outputs (joint angles and angular velocities of both pendulums). The voltage input is controlled by an RL agent, while angular velocity is required by the data-based motor model to compute torque (see following section). Output signals are used as observations and for reward calculation during training.

3.1 Data-based Methods for Efficient Modeling

A BLDC motor, including its field-oriented control (FOC), requires a specialized modeling approach. Due to its physical complexity, deriving an accurate model from first principles is both challenging and time-consuming. To effectively capture the motor's real-world behavior, a data-driven model was developed. It consists of a feedforward neural network (FNN) trained on measurement data using supervised learning. Available data points primarily cover the motor's static behavior, meaning that transient effects are not fully represented in the training set. To approximate dynamic effects, a Bessel function is included to capture transient torque responses. This approach was deliberately chosen to start with a simple surrogate model. This surrogate model can later be refined, for example by replacing an FNN with a recurrent neural network (RNN), which can leverage time-series information to improve predictions during acceleration and deceleration phases or sudden changes in setpoint speed.

Torque output of a BLDC motor primarily depends on electric current and angular velocity. Since the setup does not permit direct electric current measurement, the motor is controlled using pure voltage input.

For DC motors in general, torque M is proportional to current I, scaled by a torque constant k_m.

$$M = k_m \cdot I \tag{1}$$

Current I can be approximated as proportional to applied voltage U, assuming a constant internal resistance R.

$$I = \frac{U - EMF}{R} \quad \rightarrow \quad I \propto U \tag{2}$$

Therefore, the model assumes that torque M is proportional to input voltage U.

$$U \propto M \tag{3}$$

These assumptions hold for angular velocities near zero, where back- EMF is negligible. Given the motor's application in pendulum control, this simplification is appropriate.

Following on these assumptions, torque values as a function of velocity and input voltage were experimentally determined to develop a motor model. Measurement data was collected for several operational states of a specific BLDC motor. First, no-load operation was recorded across a range of input voltages to establish a zero-torque line in a velocity–voltage plane, with friction effects neglected. Second, the motor's maximum output torque was measured at various input voltages near zero angular velocity. Additionally, data points from runs involving a reversal of rotation direction were recorded along diagonal regions of the plane. Collected measurements are visualized in Figure 4, which displays the motor output torque as a function of angular velocity and input voltage. Measured data points are shown as black dots, while the 3D mesh surface represents the output of the trained feedforward neural network (FNN).

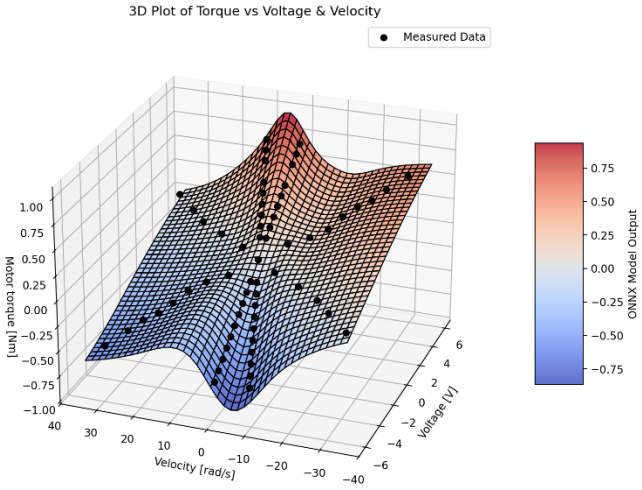

Figure 4. Output visualization of the data-driven motor model.

A supervised learning algorithm was applied to train an FNN. The high-level API `Keras` (Watson et al., 2024), built on top of `TensorFlow` (TensorFlow Developers, 2025), provides a flexible framework for deep learning and enables efficient training of this motor model.

The measured dataset was split into training and test subsets using an 80/20 ratio. This approach allows model performance to be validated on previously unseen data, which is essential for detecting potential overfitting. A Bayesian hyperparameter optimization strategy was employed to efficiently determine a suitable set of hyperparameters (listed in Table 2), after which the FNN was trained for 120 epochs. The resulting model achieved a mean absolute error (MAE) of less than 0.01 Nm on the test dataset. Corresponding mean squared error (MSE) was 0.000095 Nm^2, with a maximum absolute error of 0.038 Nm.

Table 2. Hyperparameters used to train the data-driven motor model.

Parameter	*Value*
Dense Layers	4
Neurons per Layer	80
Learning Rate	0.001
Activation Function	GELU
Optimizer	Adam
Loss	Mean Absolute Error

3.2 Integration of data-driven surrogate models in Modelica

To integrate a data-driven surrogate model of the motor into a Modelica model, the MoONNX (Sergi and Brun, 2024) library has been used. This self-developed library enables execution of Open Neural Network Exchange ('ONNX Project', 2024) models within Modelica environments and is available on GitHub. Currently, it supports Windows 10 (64-bit) and newer Windows operating systems. During simulation, the ONNX runtime library ('ONNX Runtime', 2021), which performs neural network inference, is called with an external function. Resulting outputs can then be used for further computations within the Modelica model.

4 Training Neural Network Controllers using RL and Modelica Simulations

When a system model is available, various methods can be used to develop an optimal control strategy. Neural networks are particularly well suited for implementing complex control policies while keeping real-time computational demands manageable. Discovery of an optimal control policy and training of a neural network are carried out iteratively through RL, involving a resource-intensive upstream training phase, resulting in a neural network controller that can then be executed very efficiently in deployment. When combined with a physics-based system model, it enables rapid and cost-effective generation of large volumes of training data, eliminating need for real-world interaction during training.

4.1 Reinforcement Learning Fundamentals

RL is a subfield of machine learning in which an agent learns to make decisions by interacting with an environment. This process can be modeled as a Markov Decision Process (MDP), defined by a set of *states*, *actions*, *transition dynamics*, and a *reward function*. The goal of an RL algorithm is to find an optimal strategy that maps states to actions in order to maximize cumulative long-term reward. This strategy is typically represented as

DOI 10.3384/ecp218921

a neural network, which can capture complex mappings between state and actions. This can even make high-dimensional, continuous, and partially observed state spaces controllable. Unlike other methods of machine learning, RL does not require pre-collected training data. Instead, an agent learns through trial-and-error interactions with an environment, enabling autonomous data collection and policy improvement.

A basic RL loop consists of an agent selecting an action based on its current policy, applying it to an environment, and receiving a new state and reward based on transition dynamics and reward function. The agent then updates its policy based on the observed outcome. In our setup, state transitions are computed within a Modelica model.

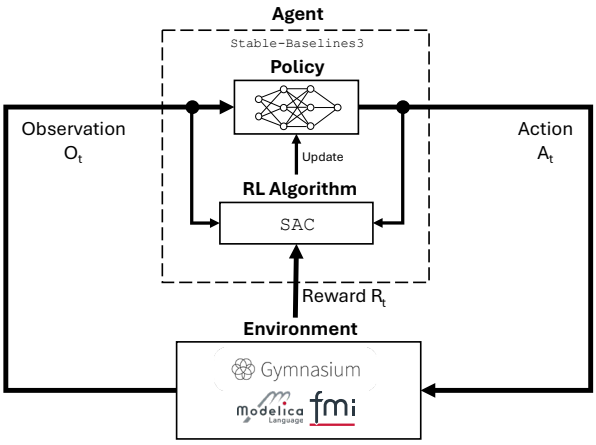

Figure 5. Block diagram of the RL loop, illustrating the interaction between components and the tools applied.

4.2 Coupling RL Environments with Modelica Simulations

Building an RL environment that incorporates physics-based simulations requires integration of multiple software frameworks and standardized interfaces. For this showcase, a training environment was constructed using an RL framework called `Gymnasium` (Mark Towers, Ariel Kwiatkowski et al., 2025), which enables the creation of customized environments via a Python API. `Gymnasium` follows a consistent structure tailored for RL tasks and is compatible with a wide range of existing algorithms, facilitating both development and testing.

The Modelica system model was then exported as a *Functional Mock-up Unit* (FMU), encapsulating relevant simulation equations and parameters according to FMI standard ('Functional Mock-up Interface', 2014). To perform physics-based state updates within an RL loop, the FMU was integrated using the `fmpy` library ('FMPy', 2024). During training, actions chosen by a RL agent are passed to a FMU via model inputs. In turn, FMU outputs are used as observations for an agent as well as values for reward computation. For this work, FMU version 2.0 in Co-simulation mode, which includes an internal solver to ensure stable and reliable simulations, is used. In comparison to Lukianykhin and Bogodorova (2019), the simulation is advanced in each RL step using the `doStep()` function provided by `fmpy`. This enables an RL loop that can be executed efficiently and more quickly, even when model complexity increases.

Listing 1. Illustrative Python code for integrating an FMU into a Gymnasium-based RL environment using FMPy library.

```
# called at beginning of the training
def __init__(…):
        fmu = fmpy.instantiate_fmu(fmu_directory,
        model_description, "CoSimulation")

# called at beginning of each training episode
def reset(…):
        fmu.reset()
        fmu.setupExperiment(startTime=0.0)
        fmu.enterInitializationMode()
        fmu.exitInitializationMode()

# called at each RL step
def step(…):
        fmu.setReal([parameter_names], [action_values])
        fmu.doStep(current_time, step_size)
        observations = fmu.getReal([parameter_names])
```

4.3 RL Agent Setup and Training Configuration

Integration of a RL agent is realized using `stable-baselines3` (Raffin et al., 2021). A library that is fully compatible with `Gymnasium` environments and provides a suite of reliable, state-of-the-art algorithms based on `PyTorch` (Jason Ansel et al., 2024). With only a few lines of code, a neural network is automatically initialized based on defined observation and action spaces and trained using a selected algorithm. Hyperparameters and training settings are freely adjustable to meet specific requirements of a control task.

Primary determinants of a controller's performance are *reward functions*, *observation and action spaces,* and an *environment* (including timestep length, episode termination criteria, initial state, etc.). These elements define an underlying mathematical optimization problem and thereby determine the global optimum. In contrast, the choice of *algorithms*, *hyperparameters*, and *neural network architecture* primarily affects training dynamics, convergence speed, and likelihood of reaching an optimal or near-optimal solution. In practice, especially for complex problems, training often converges only to a local optimum. As a result, these factors still have a significant impact on controllers' performance.

An action selected by an RL agent serves as control input to the system model and is derived from the current state and a probability distribution. In this case, the observable state consists of four state variables, $O_t = \{\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2\}$, representing angular positions and velocities of the

pendulums, respectively. θ_1 is measured relative to the downward vertical, and θ_2 is measured relative to the first pendulum. Thereby, $\dot{\theta}_1$ and $\dot{\theta}_2$ denote their respective angular velocities.

The action space includes the motor voltage, normalized to $-1 \leq a \leq 1$. This normalization improves learning efficiency, as many algorithms in `stable-baselines3` assume a Gaussian action distribution centered around zero.

The *reward function*, which evaluates the quality of the action-state pair at each timestep, is based on the work of Lee et al. (2025) for a double-inverted pendulum on a cartpole. It was adapted to match the system described in this paper and is composed of two components described in Eq. (4) and Eq. (5). A first component that focuses on pendulum position and a second one that focuses on pendulum velocity. As the objective is to stabilize the pendulum in an upright position with minimal angular velocity, an upright pendulum that is not moving yields the highest reward. This reward increases if the pendulum's height gets closer to the target height. Additionally, the reward increases with velocity if the pendulum's height is close to its initial position. However, if the pendulum is close to the target position, the reward increases when its velocity decreases. Unlike the approach taken by Lee et al. (2025), this reward function is specifically designed for a double-inverted pendulum. It ensures a reward path from the initial to the goal position that provides continuously increasing rewards when followed.

$$R_\theta = 0.5 * \sin(\theta_1) + 0.5 * \sin(\theta_1 + \theta_2) \quad (4)$$

$$R_{\dot{\theta}} = e^{\left(-0.02 * \sqrt{(\dot{\theta}_1)^2 + (\dot{\theta}_2)^2}\right)} \quad (5)$$

$$Reward = R_\theta \cdot R_{\dot{\theta}} \quad (6)$$

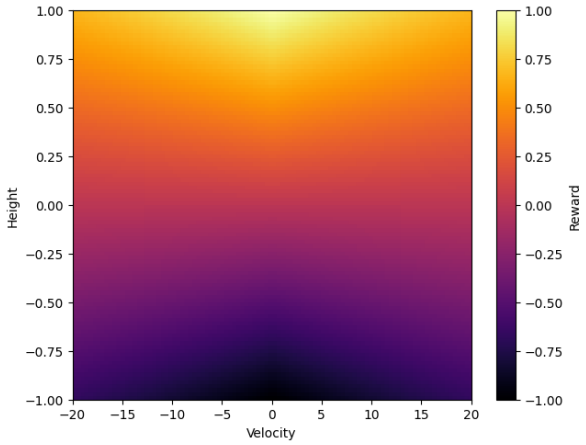

Figure 6. Reward function visualization for the double-inverted pendulum.

The environment is configured with an episode duration of 4 s, allowing the agent to interact with it up to 1'000 times per episode, given a reinforcement learning timestep of 4 ms. At the beginning of each episode, the pendulums are initialized in a downward position with zero angular velocity, corresponding to the natural starting state of a real-world pendulum system.

For training Soft Actor Critic (SAC) (Haarnoja et al., 2018), one of the most widely used modern off-policy RL algorithms, was used. It supports continuous action spaces and demonstrates strong performance in benchmark tests (Haarnoja et al., 2018; Huang et al., 2024). The hyperparameter configuration used in this work is summarized in Table 3.

Table 3. Hyperparameters used for this showcase with the SAC algorithm from stable-baselines3.

Hyperparameter	*Value*
Optimizer	Adam
Learning rate	0.0003
Discount factor (γ)	0.99
Replay buffer size	1e6
Number of hidden layers in critic networks	3
Number of neurons in hidden layers of critic network	512
Number of hidden layers of policy network	2
Number of neurons in 1st hidden layer of policy network	400
Number of neurons in 2nd hidden layer of policy network	300
Minibatch size	256
Nonlinearity	ReLU
Target smoothing coefficient (β)	0.005

Training was conducted on a Windows PC, parallelized across 12 cores of an Intel Xeon W-2275 3.3 GHz CPU. With this configuration, approximately 360 timesteps per second were processed during training.

4.4 Results and Discussion

As an initial step, RL training was conducted on a single-pendulum system. This simpler setup served as a validation of the toolchain by significantly reducing challenges associated with training and sim-to-real transfer. A stepwise increase in system complexity allows for targeted troubleshooting and a structured development approach. RL training for the single-pendulum system converged at a maximum episodic reward of 95%, using the reward function defined in Equation 7, where θ represents the pendulum angle relative to the downward vertical.

$$Reward = \cos(\theta) \quad (7)$$

DOI 10.3384/ecp218921

Convergence was reached after 500’000 timesteps, but training was extended to 1’100’000 timesteps to ensure model stability and robustness. The entire training process was completed in just 52 min. When evaluated with a system model, the trained neural network successfully stabilized the pendulum in an upright position.

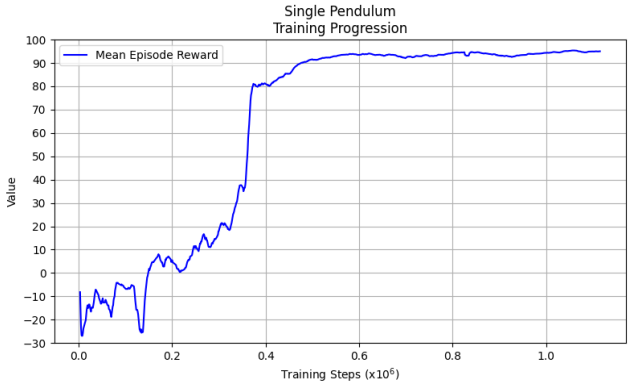

Figure 7. Mean episode reward during training of the single-pendulum system.

To deploy the trained neural network on a real-world system, it was converted into ONNX format, enabling interoperability and execution across different tools. The sim-to-real transfer introduced no additional issues, and the neural network successfully controlled the real-world system with performance comparable to the virtual system. The deployed controller for the single pendulum was also capable of handling external disturbances (e.g., applied forces) and inevitable model inaccuracies, demonstrating robust real-world performance.

Following the successful validation of the toolchain's functionality, it is applied to the double pendulum case. In that the toolchain was used to test various training configurations, each employing different training and pendulum parameters. These trainings have only been successful when system parameters have been chosen carefully. Figure 8 shows the mean reward progress for a representative successful training.

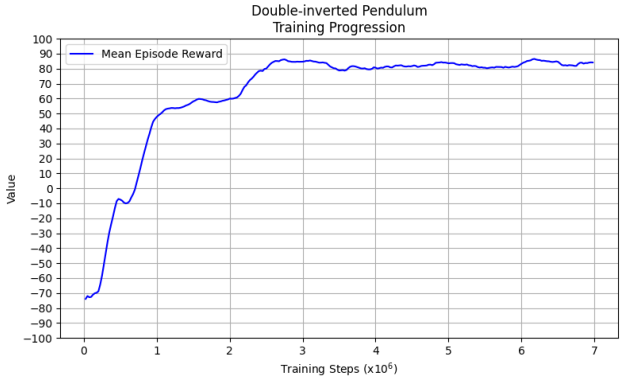

Figure 8. Mean episode reward during training of the double-inverted pendulum system.

In this example, training convergence was reached after 1’100’000 timesteps with a maximum mean episode reward of 86.5%. Despite some successful trainings, a sim-to-real transfer has not yet been achieved. In many cases, the controller could make the real-world double-inverted pendulum swing upwards but failed to balance it.

The conducted grid search indicates that stable control is much more likely when the proportions of the pendulums meet specific conditions. The first observed requirement is an appropriate ratio between the time constants of the motor and second pendulum that must be bigger than 28.5 to ensure stable control. (Eq. 8) The time constant of the second pendulum is determined solely by its geometry and friction, whereas the motor's time constant depends on the geometry of the first pendulum and a small control loop delay. Since this delay is minimal, it can be considered negligible. The second condition is related to the ratio between the damping coefficient D (i.e., the friction in the joint) and the second pendulum's moment of inertia. (Eq. 9) In the trainings, successful controllability could only be observed if this ratio was lower than 3.0.

$$\frac{\tau_{rod,2}}{\tau_{motor}} = \frac{\tau_{rod,2}(I_2, m_2, l_{cm}, D)}{\dot{\tau}_{motor}(\tau_{motor}, I_1)} > 28.5 \tag{8}$$

$$\frac{D}{2 * I_2} < 3.0 \tag{9}$$

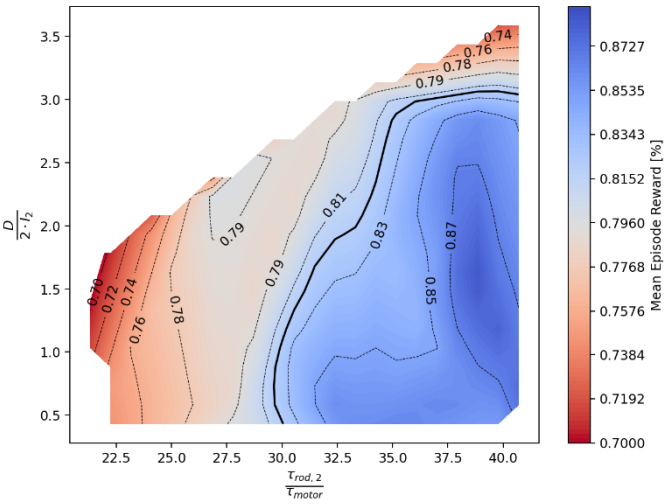

Figure 9. Controllability Analysis for double-inverted pendulum parameters.

Figure 9 visualizes the subset of our sweep that yielded mean rewards > 0.82. Within this empirical region controllability was consistently observed. This region can be accessed through a variety of parameter combinations. A straightforward approach would involve utilization of an electrical motor characterized by a low time constant. It is also plausible to utilize a secondary pendulum with an enhanced time constant. However, this requires a more detailed evaluation due to the dependence of the previously mentioned second condition on properties of the second pendulum.

5 Conclusion and Outlook

This work presents a comprehensive toolchain for training and deploying neural network control systems using reinforcement learning (RL) in combination with physics-based simulations developed in Modelica. A key achievement of this work is the successful transfer of a controller trained entirely in a virtual environment to a real-world system, demonstrating the feasibility of simulation-based training for physical control tasks. By offering a unified and structured workflow, this approach provides an accessible toolchain for both research and educational purposes. The use of well-established tools and frameworks ensures an efficient, reliable, and reproducible development process, laying the foundation for future applications of RL-based neural network controllers in complex, real-time dynamic systems across academic and industrial domains.

The RL training process poses a complex optimization challenge, where success relies heavily on the selected training setup, including algorithm choice, neural network architecture, and reward-shaping strategies. Numerous environment and training parameters, such as hyperparameters, observation augmentation, and initial state distributions, are highly system-specific and interact in intricate ways, adding further complexity to the optimization task. Identifying a configuration that allows a RL agent to reliably converge toward an optimal or near-optimal policy often requires repeated experimentation, with each training run being both time- and computation-intensive. Leveraging prior domain knowledge, heuristics, and insights from previous experiments can help reduce this search space, yet the process remains inherently challenging and iterative.

During the development of this showcase, it became evident that achieving good system controllability is a fundamental yet challenging prerequisite for successful RL training. The controllable region of this system is not intuitively defined, making it difficult to identify suitable configurations, and even a well-designed RL setup cannot compensate for a system that is only partially or poorly controllable. For this double-inverted pendulum case, ensuring controllability requires carefully selecting parameter ratios between the pendulum components, such as their respective moments of inertia, damping coefficients, and lengths, to guarantee that the system can, in principle, be well stabilized. Ongoing work focuses on improving these mechanical design aspects and thereby enabling a sim-to-real transfer of the learned policies. Completing these steps will be essential not only for finalizing the double-pendulum showcase but also for strengthening the reliability, scalability, and educational value of the entire toolchain as a practical teaching and research platform for reinforcement learning in real-world control applications.

5.1 Future Work

The next phase in the development of this showcase focuses on redesigning the physical system to enable a sim-to-real transfer for the double-inverted pendulum. This involves adjusting the system's proportions to align with the empirical guidelines identified in Chapter 4.4.

Optimization efforts will primarily target pendulum components. The first pendulum will be redesigned to reduce its moment of inertia, thereby improving the motor's time constant. In parallel, the second pendulum will be modified to increase its moment of inertia. Since replacing the motor would be significantly more complex, optimizing the pendulum design presents a more practical and effective approach to start with.

Other potential further steps will be focused on:

- *Controller implementation for embedded systems:* Investigating the use of trained neural network controllers on embedded platforms such as industrial PLCs (e.g., Beckhoff SPS)
- *Enhancing controller robustness through system randomization:* Applying randomization techniques during training to improve sim-to-real transfer, increasing resilience to virtual–physical discrepancies, sensor noise, and limitations in the control loop or actuators
- *Educational applications:* The developed showcase can be utilized in lectures to teach reinforcement learning (RL) in an engaging and interactive manner. A possible exercise would involve designing and comparing different reward functions for balancing the pendulums in an upright position. Using this provided toolchain, students can gain practical, hands-on experience with RL training. Furthermore, the most promising neural networks trained in simulation could be deployed on the physical system, enabling a real-world demonstration of RL.

Acknowledgements

The authors acknowledge Lucerne University of Applied Sciences and Arts (School of Engineering and Architecture) for supporting this study.

Language editing and writing style improvements were assisted by generative AI tools, including ChatGPT and DeepL. Scientific content, concepts, and results were independently developed by the authors.

References

Biscani, F. and Izzo, D. (2020) 'A parallel global multiobjective framework for optimization: pagmo', *Journal of Open Source Software*, 5(53), p. 2338. Available at: https://doi.org/10.21105/joss.02338.

'FMPy' (2024). Dassault Systems. Available at: https://github.com/CATIA-Systems/FMPy (Accessed: 25 April 2025).

'Functional Mock-up Interface' (2014). Modelica Association. Available at: https://fmi-standard.org/ (Accessed: 25 April 2025).

Haarnoja, T. et al. (2018) 'Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor'. arXiv. Available at: https://doi.org/10.48550/ARXIV.1801.01290.

Heuermann, A. et al. (2023) 'Accelerating the simulation of equation-based models by replacing non-linear algebraic loops with error-controlled machine learning surrogates', in. *15th International Modelica Conference 2023, Aachen, October 9-11*, pp. 275–284. Available at: https://doi.org/10.3384/ecp204275.

Huang, S. et al. (2024) 'Open RL Benchmark: Comprehensive Tracked Experiments for Reinforcement Learning'. arXiv. Available at: https://doi.org/10.48550/ARXIV.2402.03046.

INNOSUISSE (2025) *IntelliBake*, *ARAMIS*. Available at: https://www.aramis.admin.ch/Grunddaten/?ProjectID=54123 (Accessed: 23 April 2025).

Jason Ansel et al. (2024) 'PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation', in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. *ASPLOS '24: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, La Jolla CA USA: ACM, pp. 929–947. Available at: https://doi.org/10.1145/3620665.3640366.

Lee, T., Ju, D. and Lee, Y.S. (2025) 'Transition Control of a Double-Inverted Pendulum System Using Sim2Real Reinforcement Learning', *Machines*, 13(3), p. 186. Available at: https://doi.org/10.3390/machines13030186.

Lukianykhin, O. and Bogodorova, T. (2019) 'ModelicaGym: Applying Reinforcement Learning to Modelica Models'. Available at: https://doi.org/10.48550/ARXIV.1909.08604.

Mark Towers, Ariel Kwiatkowski et al. (2025) 'Gymnasium: A Standard Interface for Reinforcement Learning Environments'. Farama Foundation. Available at: https://gymnasium.farama.org/ (Accessed: 23 April 2025).

Modelica Association (2020) 'Modelica Standard Library'. Modelica Association.

Morcego, B. et al. (2023) 'Reinforcement Learning Versus Model Predictive Control on Greenhouse Climate Control'. arXiv. Available at: https://doi.org/10.48550/ARXIV.2303.06110.

'ONNX Project' (2024). ONNX Community. Available at: https://onnx.ai.

'ONNX Runtime' (2021). ONNX Runtime developers. Available at: https://onnxruntime.ai/ (Accessed: 25 April 2025).

Raffin, A. et al. (2021) 'Stable-Baselines3: Reliable Reinforcement Learning Implementations', *Journal of Machine Learning Research*, 22(268), pp. 1–8.

Sergi, T. and Brun, J. (2024) 'MoONNX'. Available at: https://github.com/sertho/modelica-ONNX.

TensorFlow Developers (2025) 'TensorFlow'. Zenodo. Available at: https://doi.org/10.5281/ZENODO.4724125.

Watson et al. (2024) 'KerasHub'. Available at: https://github.com/keras-team/keras-hub.