Model-Structured Neural Networks for Dynamics Learning of Physical Systems: A Compositional Modeling Approach with nnodely

ARTICLE INFO

Keywords:
Model-Structured Neural Networks
Physics-encoded machine learning
Scientific machine learning
Dynamics learning
Compositional modeling

ABSTRACT

Accurate and interpretable models of physical-system dynamics are fundamental to simulation, prediction, and control. First-principles models provide physical consistency and interpretability but often rely on simplifying assumptions, whereas purely data-driven neural networks offer greater expressive power at the expense of data efficiency. This paper introduces Model-Structured Neural Networks (MSNNs), a methodology for learning physical-system dynamics by embedding physics priors directly into neural architectures. Rather than enforcing physics through loss-function constraints, MSNNs encode prior knowledge through Neural Compositional Modules (NCMs), modular building blocks representing physically meaningful dynamic effects. We formalize a compositional framework in which NCMs are combined through algebraic operations and equipped with physics-encoded input-output transformations, operating-regime-dependent activations, residual components, and temporal memory, enabling the representation of complex dynamics while preserving interpretability and modularity. A further contribution is `nnodely`, a new open-source library providing an end-to-end workflow for MSNN definition, training, validation, inference, and export. The methodology is evaluated on four simulated and real-world engineering case studies involving single and double inverted pendula, friction learning for a Unitree Z1 manipulator, and lateral vehicle dynamics learned from public Ferrari 250 LM telemetry. Across these applications, MSNNs preserve physically interpretable structure while achieving competitive or superior predictive accuracy relative to black-box neural networks and purely model-based approaches. Experiments with progressively reduced training sets and recurrent rollouts further demonstrate improved data efficiency and robust long-horizon prediction. The software, data-generation scripts, and case-study implementations are openly available in the `https://github.com/tonegas/nnodely` and `https://github.com/tonegas/nnodely-applications` repositories.

1. Introduction

Learning the dynamics of physical systems is a foundational task for control, planning, and estimation. Classical model-based approaches based on physics principles, ranging from analytical state-space formulations to input-output representations, provide interpretability and strong physical grounding, but often rely on simplifying assumptions that limit their accuracy in the presence of complex, nonlinear, or poorly characterized dynamics. Conversely, purely data-driven neural networks offer high expressive power, yet their black-box nature, data requirements, and limited interpretability hinder their adoption in safety-critical real-world applications, where physical consistency, transparency, and sample efficiency are often required.

To combine the strengths of physics-based and data-driven approaches, recent research has explored the integration of physics priors into neural networks [1, 2, 3, 4, 5]. These approaches can be distinguished according to where such physical knowledge enters the learning pipeline [4]. *Physics-informed* methods introduce governing equations, conservation laws, or physical constraints through the training objective; *physics-guided* methods exploit physical models to construct inputs, representations, labels, or training data; and *physics-encoded* methods embed physical structure directly into the architecture, parameterization, or forward computation of the learned model.

While related structured neural architectures have been explored in specific domains, such as vehicle dynamics [6, 7, 8, 9] and robotics [10, 11], the definition of such models remains largely application-dependent. Encoding effective physics priors in neural architectures requires simultaneous expertise in physical system modeling and neural networks, and the mapping from available prior knowledge to a suitable computational graph is often performed on a case-by-case basis. Existing software tools also tend to address individual aspects of the problem (Section 2), such as constrained learning, equation discovery, or differentiable simulation, while providing limited support for systematically defining, composing, validating, and reusing structured neural representations of physical dynamical systems. As a result, physics-encoded neural architectures remain hard to design and implement, and their integration into broader computational modeling and simulation workflows is challenging.

We address these limitations from both a methodological and an implementation perspective. Within the broader class of physics-encoded methods, we introduce ***Model-Structured Neural Networks*** (MSNNs) as a new compositional methodology for dynamics learning of physical systems, and provide a systematic construction based on reusable and interpretable Neural Compositional Modules (NCMs). In our methodology, NCMs are specialized according to the available prior knowledge, and composed through physically meaningful operations to define a complete MSNN. In this way, physical relations such as force decomposition, operating-regime dependencies, nonlinear input transformations, and finite-memory effects are translated explicitly into the MSNN architecture. The contribution of this work is therefore not the first use of structured neural architectures per se, but their formalization into a unified MSNN methodology and a reusable NCM-based construction principle.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

We implement the proposed methodology in the new **open-source software framework** `nnodely`, which provides a computational environment for defining, composing, training, and evaluating MSNNs and structured neural models. By abstracting low-level implementation details while preserving access to physically interpretable components, the framework supports the development of MSNNs within a coherent modeling and simulation workflow. More generally, the proposed formulation supports the progressive construction of increasingly complex architectures through reusable structured components. In this way, complex MSNNs of physical dynamical systems can be constructed hierarchically from simpler interpretable components.

The main contributions of this work are:

- **Compositional methodology for MSNNs** (Section 3): we formalize MSNNs through Neural Compositional Modules (NCMs), providing reusable abstractions for encoding physical relations, temporal dependencies, operating regimes, and residual dynamics directly into neural architectures.
- **Computational implementation in** `nnodely` (Section 4): we provide a framework for the systematic construction, training, validation, and composition of structured neural models based on the proposed methodology.
- **Validation on representative physical systems** (Section 5): we assess the proposed approach on simulated and experimental dynamics learning case studies, including rigid-body robot systems and vehicles. We evaluate interpretability, data efficiency, predictive accuracy, and robustness over long prediction horizons.

More broadly, this work contributes to the development of structured learning methodologies for computational modeling and simulation. By introducing a common formulation based on NCMs and a corresponding software implementation, the proposed approach aims to make the design of physically grounded neural models more systematic, reusable, and accessible not only to machine-learning practitioners, but also to scientists and engineers with limited expertise in neural network design.

2. Related Work

Following the classification in [4], existing approaches are commonly grouped into *physics-informed*, *physics-guided*, and *physics-encoded* methods. Since the present work focuses on architectural incorporation of prior knowledge, the following discussion emphasizes physics-encoded approaches, including energy-based Lagrangian and Hamiltonian models, variational integrator networks, hybrid physics–neural models, physics-encoded topology learning, and MSNNs.

Physics-informed methods impose governing equations or admissibility conditions through the training objective. Physics-informed neural networks (PINNs) are the canonical example [1]: differential-equation residuals are penalized together with data, initial-condition, and boundary-condition errors. In robotics, this principle has been used, for example, to regularize quadrotor dynamics with Newton–Euler equations [12] and to identify quadrotor thrust and drag parameters from limited flight data [13]. Physics-guided methods instead retain a largely generic learning architecture and use physical knowledge to construct inputs, features, representations, or training data. Examples include analytical slider–crank states supplied to a neural acceleration model [14] and nominal-model residuals used as inputs to a recurrent quadrotor model [15].

Physics-encoded methods embed physical knowledge directly in the model equations, architecture, or parameterization, so the constraint is part of the forward pass itself and therefore remains active both during training and at inference time. In other words, physics is not only encouraged by a loss term or used to build features: it defines which input–output mappings are structurally admissible (or strongly preferred) by construction. Among these methods, energy-based architectures parameterize physically meaningful scalar functions and recover the dynamics through analytical mechanics relations. Deep Lagrangian Networks (DeLaNs) learn the inertia matrix and potential energy and use the Euler–Lagrange equations to compute forward or inverse dynamics [10]; positive definiteness of the inertia matrix can be enforced by construction through a Cholesky parameterization. Lagrangian Neural Networks instead learn the complete Lagrangian as a scalar function [16]. Hamiltonian Neural Networks learn the Hamiltonian and obtain the state evolution through Hamilton's equations [17], while structured and port-Hamiltonian extensions explicitly represent actuation, dissipation, energy exchange, and interconnection [18]. These formulations provide strong inductive biases, but they are tied to a specific mechanics formalism.

Another physics-encoded family is based on variational integrators. Neural ODEs embed a numerical ODE solver in the computational graph, but a generic solver does not necessarily preserve the geometric properties of mechanical systems. Variational integrators instead discretize Hamilton's principle and yield structure-preserving time-stepping schemes. Variational Integrator Networks encode the corresponding discrete equations of motion directly in the learned architecture, promoting symplecticity and accurate long-horizon behavior [19]. Forced variants include control inputs and damping [20], whereas Lie-group formulations extend the approach to rigid-body configuration spaces [21].

Hybrid models retain a known physics-based backbone and use learned components for selected uncertain or unresolved terms. Learned mappings can represent tire forces within a vehicle model [22], while residual networks can correct a nominal model and support tire-parameter identification [23]. Universal differential equations generalize this principle by embedding trainable functions within differential equations that retain known physical terms [24]. Thus,

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

the mechanistic structure remains explicit, while learning is confined to specific unresolved components.

A complementary family uses physical knowledge to select not only model parameters but also candidate operators and connections. First-Order-Principle networks assemble Newton–Euler-inspired operators while learning the topology required for inverse dynamics [25]. Sparse Identification of Nonlinear Dynamics (SINDy) selects governing terms from a candidate library [26]; physics can be encoded by constructing that library from kinematic, Newtonian, hydrodynamic, or Euler–Lagrange terms, as demonstrated for soft robots and manipulators [27, 28]. Equation-learning networks similarly combine predefined algebraic or nonlinear operators to obtain compact, interpretable relations [11]. These approaches encode prior knowledge through the choice of the admissible operator library and the rules used to select or connect its elements.

MSNNs constitute the class of physics-encoded architectures presented in this work. In contrast to Lagrangian, Hamiltonian, or variational architectures, MSNNs do not prescribe a universal mechanics formalism; they translate the available system-specific knowledge into the neural topology.

Some existing works may be classified as MSNNs, although they were not originally presented in the general methodological framework proposed in this paper. For example, in the field of vehicle dynamics, the authors of [29, 30] developed structured neural models that decompose the longitudinal vehicle acceleration into aerodynamic, traction, braking, and gear-dependent contributions, combining force superposition, physically motivated nonlinearities, local models, and finite input histories. This structure was extended with neuro-fuzzy finite-impulse-response components to represent road-dependent dynamics and tire–road friction [7]. MSNN-like models have also been used for longitudinal-lateral vehicle dynamics learning [31, 32] and for automated parking, combining a neural polynomial representation of nonlinear steering characteristics with a finite-impulse-response representation of transient steering dynamics [33]. Moreover, an MSNN-like example was used for human-motion prediction in real-time robotics applications [34]. Structured models have also encoded spring–damper decompositions in vehicle suspensions [35] and inertial, stiffness, and damping terms in collaborative-robot joints [36]. Across these applications, the benefit is not only a physical constraint but an architecture whose internal variables and modules retain an engineering interpretation.

The approach presented in this paper systematizes the construction of MSNNs by abstracting modeling structures into reusable NCMs. These NCMs provide a common representation through which system-specific physical knowledge can be translated into neural architectures and assembled according to the relationships governing the target dynamics. The contribution is therefore a general and reusable methodology for designing, training, validating, and composing MSNNs, rather than a new physics formalism tailored to a particular class of systems.

From an implementation perspective, the methodological distinctions discussed above are reflected in a heterogeneous software ecosystem, with different frameworks targeting specific mechanisms for incorporating physical knowledge into learned models. For physics-informed learning, `DeepXDE` [37], `NeuralPDE.jl` [38], `PhysicsNeMo` [39, 40], and `NeuroDiffEq` [41] provide tools for constructing and differentiating equation-residual losses. Differentiable integration is available through `torchdiffeq` [42], `torchdyn` [43], and `DiffEqFlux.jl` [44], the latter also supporting universal differential equations and mechanistic–learned coupling. Differentiable simulators such as `Brax` [45], `Nimble Physics` [46], the Tiny Differentiable Simulator [47], and `Warp` [48] propagate gradients through rigid-body dynamics, contact, and simulation rollouts. For topology and equation discovery, `PySINDy` [49] implements sparse identification, while `deepSI` [50], `SysIdentPy` [51], and `deeptime` [52] support state-space, NARMAX, Koopman, and time-series identification workflows. `NeuroMANCER` [53] further supports constrained system identification, differentiable optimization, and model-based control.

Despite this broad ecosystem, support for physics-encoded neural models remains fragmented and requires substantial expertise to map domain knowledge onto general-purpose deep learning frameworks. Existing tools do not provide a unified abstraction for composing custom physics-based layers, explicit temporal windows, reusable structured modules, hierarchical model compositions, and autoregressive connections.

Our open-source software `nnodely` [54] addresses this gap by providing a unified environment for constructing and composing physics-encoded neural models. It implements the MSNN methodology through custom parametric and learned relations, temporal-window operations, modular composition, autoregressive connections, and differentiable numerical integration. The same workflow also supports topology and equation learning, residual components, structured loss functions, and physics-guided input shaping, although the latter two are not considered in the examples presented here.

3. A Compositional Modeling Approach for Model-Structured Neural Networks

This section introduces a *compositional modeling* approach to construct MSNNs for dynamics learning of physical systems. The core idea is to compose the MSNN architecture from tailored building blocks, named ***Neural Compositional Modules*** (NCMs) and denoted by Ψ (Fig. 1). Starting from a general, physically motivated design, we specialize the internal structure of each NCM to learn specific aspects of the system dynamics. For example, we employ NCMs to represent individual physical contributions to the dynamics, such as force components, dissipation terms, or stiffness effects. Alternatively, NCMs may represent frequency-dependent

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

effects, such as steady-state or transient responses, or unmodeled dynamic phenomena. We design MSNNs by composing multiple NCMs, each representing a specific contribution to the overall system behavior. The composition follows the physical, temporal, or functional structure of the system. This allows the resulting MSNN to inherit the interpretability of the individual NCMs, while capturing the overall system behavior through their structured combination.

Instead of using a monolithic neural network and imposing physics-informed knowledge only through the loss function [1], our MSNNs encode physics priors through physics-based architectural layers, internal connections, temporal operators, and architectural constraints. In the following, we present the overall composition principles (Section 3.1), the internal structure of the NCMs (Section 3.2), and the autoregressive mechanisms used to connect the NCM outputs to the temporal evolution of the system (Section 3.3).

The proposed methodology is implemented in the `nnodely` framework, which provides the computational tools to define, compose, train, and deploy structured neural models. In the following, we connect the main functional elements of the methodology, such as NCMs, local models, parametric functions, temporal operators, residual blocks, and autoregressive mechanisms, to the corresponding coding blocks available in `nnodely`. This link between the mathematical formulation and the software implementation highlights how MSNN architectures can be constructed in practice while preserving the same compositional modeling principles.

Notation: Throughout this section, boldface and overbars identify two distinct dimensions of a quantity. Boldface denotes a multicomponent variable at a single time instant: for example, $v[k] \in \mathbb{R}$ is a scalar sample, whereas $\boldsymbol{v}[k] = [v_1[k], \dots, v_{n_v}[k]] \in \mathbb{R}^{n_v}$ collects n_v variables at time k. An overbar instead denotes a finite temporal representation; thus, $\bar{v}_k = [v[k-P], \dots, v[k]]^\top \in \mathbb{R}^{P+1}$ collects the current value of a scalar signal and its P previous samples. When both conventions are used, $\bar{\boldsymbol{v}}_k = [\boldsymbol{v}[k-P], \dots, \boldsymbol{v}[k]]^\top = [\bar{v}_1[k], \dots, \bar{v}_{n_v}[k]] \in \mathbb{R}^{(P+1)\times n_v}$ represents a vector-valued variable over the same temporal window, with rows corresponding to time samples and columns to variable components. The same boldface convention is used for multidimensional parameters, such as the matrix $\boldsymbol{w} \in \mathbb{R}^{A\times B}$. Finally, the semicolon separates input arguments from learnable parameters, as in $f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f)$, and $\odot$ denotes element-wise multiplication.

3.1. Model-Structured Architectures using Neural Compositional Modules

In this section, we formalize the MSNN architecture from a finite-memory, sampled-time perspective.

3.1.1. Input-Output Representation

As depicted in Fig. 1, we formulate an MSNN as:

$$y_k = \mathcal{A}\left(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k\right), \tag{1}$$

where $\mathcal{A}$ is the MSNN operator, and k denotes the discrete-time instant. The output $y_k \in \mathbb{R}$ is a scalar that represents the target physical quantity to be predicted[1]. Our MSNN operates on two kinds of variables: the inputs $\bar{\boldsymbol{v}}_k$, which collect the quantities available for prediction, and the scheduling variables $\bar{\boldsymbol{z}}_k$, which describe the operating condition of the system. We select the inputs $\bar{\boldsymbol{v}}_k$ using domain knowledge: $\bar{\boldsymbol{v}}_k$ may include measured states, controls, estimated quantities, intermediate network variables, or autoregressive inputs. The scheduling variables $\bar{\boldsymbol{z}}_k$ collect the quantities used to describe operating regimes or local behaviors. $\bar{\boldsymbol{z}}_k$ may be selected from $\bar{\boldsymbol{v}}_k$, built from available signals, encoded as categorical regimes, or provided as additional measurements. Inputs and scheduling variables are defined over a finite past history window of length P, which is selected according to the relevant time scales of the system:

$$\begin{aligned}
\bar{\boldsymbol{v}}_k &= \begin{bmatrix} v_1[k-P] & \dots & v_{n_v}[k-P] \\ \vdots & \vdots & \vdots \\ v_1[k-1] & \dots & v_{n_v}[k-1] \\ v_1[k] & \dots & v_{n_v}[k] \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}[k-P] \\ \vdots \\ \boldsymbol{v}[k-1] \\ \boldsymbol{v}[k] \end{bmatrix}, \\
\bar{\boldsymbol{z}}_k &= \begin{bmatrix} \boldsymbol{z}[k-P] \\ \vdots \\ \boldsymbol{z}[k-1] \\ \boldsymbol{z}[k] \end{bmatrix},
\end{aligned} \tag{2}$$

where $\bar{\boldsymbol{v}}_k \in \mathbb{R}^{(P+1)\times n_v}$ and $\bar{\boldsymbol{z}}_k \in \mathbb{R}^{(P+1)\times n_z}$ are the input and scheduling matrices, respectively. Inside the matrices $\bar{\boldsymbol{v}}_k$ and $\bar{\boldsymbol{z}}_k$, the rows represent the temporal samples, while the columns represent different physical variables. The instantaneous vectors $\boldsymbol{v}[k] \in \mathbb{R}^{n_v}$ and $\boldsymbol{z}[k] \in \mathbb{R}^{n_z}$ (at the current time instant k) are the last rows of these matrices, representing the current values of the input and scheduling variables.

Interpretation and Selection of the Input & Scheduling Variables: The finite-memory variables introduced above can be interpreted as a sampled-time approximation of a more general non-Markovian dynamical relation. In its most general form, the output at time k of a non-Markovian dynamical system depends on the entire past history of states $\{\boldsymbol{x}[k-i]\}_{i=0}^{\infty}$ and controls $\{\boldsymbol{u}[k-i]\}_{i=0}^{\infty}$:

$$y[k] = \mathcal{F}\left(\{\boldsymbol{x}[k-i]\}_{i=0}^{\infty}, \{\boldsymbol{u}[k-i]\}_{i=0}^{\infty}\right). \tag{3}$$

In practical use cases, we replace the infinite history of $\boldsymbol{x}$ and $\boldsymbol{u}$ by the finite-memory temporal variables $\bar{\boldsymbol{v}}_k$ and $\bar{\boldsymbol{z}}_k$ in Eq. (2). Their roles are different: $\bar{\boldsymbol{v}}_k$ collects the predictive information (controls $\boldsymbol{u}$ and possibly measured states $\boldsymbol{x}$) used to estimate the target quantity, while $\bar{\boldsymbol{z}}_k$ describes the operating regime of the system. The resulting MSNN formulation in Eq. (1) is therefore a structured finite-memory representation of the more general map in Eq. (3).

Different choices of the inputs $\bar{\boldsymbol{v}}_k$, the scheduling variables $\bar{\boldsymbol{z}}_k$, and their memory length P recover different

[1] To ease the notation, we present the formulation for a scalar output. Vector-valued outputs can be obtained by applying the same MSNN construction component-wise to each output component.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

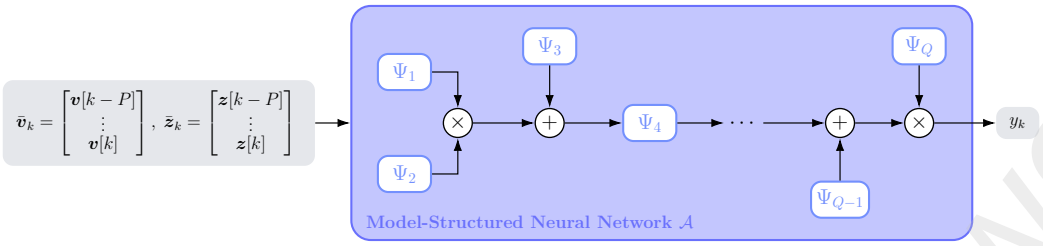


Figure 1: Model-Structured Neural Network (MSNN) architecture obtained by composing Neural Compositional Modules (NCMs) $\Psi_1, \ldots, \Psi_Q$. At each time step k, the MSNN receives the inputs $\bar{\boldsymbol{v}}_k$, built from physical variables $\boldsymbol{v}[k]$ over a memory window of length P, and the scheduling variables $\bar{\boldsymbol{z}}_k$. Each NCM learns specific aspects of the system dynamics, such as physical forces or frequency-dependent dynamic effects. The internal architecture of each NCM follows our proposed template, and is specialized according to the available prior knowledge, modeling assumptions, and desired level of interpretability. The MSNN is formed by combining NCMs through sums, products, and cascaded compositions, producing a scalar output y_k.

sampled-time representations. In particular, the predictive variables $\bar{\boldsymbol{v}}_k$ may contain the current value and past histories of controls $\boldsymbol{u}$ and states $\boldsymbol{x}$, depending on the available measurements and on the intended model structure. This allows the same formulation to cover Markovian and non-Markovian state-space models, partially observed systems, delayed systems, and finite-memory input-output models. The scheduling variables $\bar{\boldsymbol{z}}_k$ can be selected independently of the predictive variables $\bar{\boldsymbol{v}}_k$, so that operating regimes, categorical modes, or hysteretic effects can modulate the learned dynamics without changing the predictive input description. Additional representative choices of $\bar{\boldsymbol{v}}_k$ and P are reported in Appendix A, with examples of how the same finite-memory formulation can be specialized to different modeling assumptions and available measurements.

This sampled-time viewpoint provides the common representation used throughout the rest of the methodology. We next specify how the operator $\mathcal{A}$ in Eq. (1) is structured by composing multiple NCMs.

3.1.2. Compositional Architecture of an MSNN

As anticipated at the beginning of the current Sec. 3, we build MSNNs by composing multiple NCMs Ψ, each representing a specific contribution to the overall system behavior (Fig. 1). Mathematically, we expand the MSNN operator $\mathcal{A}$ in Eq. (1) as:

$$y_k = \mathcal{A}\big(\Psi_1(\bar{\boldsymbol{v}}_{1,k}, \bar{\boldsymbol{z}}_{1,k}; \boldsymbol{w}_1), \ldots, \Psi_Q(\bar{\boldsymbol{v}}_{Q,k}, \bar{\boldsymbol{z}}_{Q,k}; \boldsymbol{w}_Q)\big), \quad (4)$$

where $\Psi_i(\cdot)$, $i \in \{1, \ldots, Q\}$, are the individual NCMs and Q is their total number. The operator $\mathcal{A} \in \mathcal{C}(\{+, \times, \circ\})$ defines an architectural composition rule, where $\mathcal{C}(\{+, \times, \circ\})$ is the class of architectures generated by additions, multiplications, and sequential (*i.e.*, cascaded) compositions of NCMs. Additions encode the superposition of physical contributions, such as forces, moments, dissipation terms, or independent actuation channels. Multiplications encode modulations, scheduled gains, and known product relations. Sequential compositions encode cascaded effects, where one module processes the output of another, for instance to combine steady-state and transient components. The interpretability of the resulting MSNN therefore comes from two complementary levels: the internal structure of each NCM and the physically meaningful composition of multiple NCMs.

Each NCM $\Psi_i(\cdot)$ in Eq. (4) is denoted as:

$$\Psi_i = \Psi_i\left(\bar{\boldsymbol{v}}_{i,k}, \bar{\boldsymbol{z}}_{i,k}; \boldsymbol{w}_i\right), \qquad i = 1, \ldots, Q, \quad (5)$$

and uses selected subsets of the available inputs $\bar{\boldsymbol{v}}_k$ and scheduling variables $\bar{\boldsymbol{z}}_k$. We denote these subsets as $\bar{\boldsymbol{v}}_{i,k} \subseteq \bar{\boldsymbol{v}}_k$ and $\bar{\boldsymbol{z}}_{i,k} \subseteq \bar{\boldsymbol{z}}_k$, respectively. The selection of these subsets is guided by physical knowledge, modeling assumptions, or prior information about the system. For example, a module representing a specific force component may only require the relevant state variables and inputs that influence that force, while a module representing a regime-dependent effect may only require the scheduling variables that determine the active regime. Finally, the learnable parameters of the i-th NCM are denoted as $\boldsymbol{w}_i$.

3.2. Internal Structure of a Neural Compositional Module

As highlighted in the previous section, NCMs are the building blocks of MSNNs. We design each NCM Ψ in Eq. (4)-(5) starting from the proposed template architecture in Fig. 2, expressed as:

$$\Psi(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k; \boldsymbol{w}) = \sum_{s=1}^{S} \overbrace{g\Big(\underbrace{\phi_s(\bar{\boldsymbol{z}}_k)}_{\text{activation}} \odot \underbrace{f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{f,s})}_{\text{input function}}; \boldsymbol{w}_{g,s}\Big)}^{\text{output function } g(\cdot)} + \overbrace{\mathcal{R}\left(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{\mathcal{R}}\right)}^{\text{residual term}}. \quad (6)$$

with inputs $\bar{\boldsymbol{v}}_k$ and scheduling variables $\bar{\boldsymbol{z}}_k$ defined in Eq. (2), learnable parameters $\boldsymbol{w}$, and $\odot$ denoting element-wise vector multiplication. The NCM consists of the following basic components (Fig. 2):

- Regime-dependent **activation functions** $\phi_s(\bar{\boldsymbol{z}}_k)$. They map $\mathbb{R}^{(P+1)\times n_z}$ to $\mathbb{R}^{P+1}$ and transform the scheduling variables $\bar{\boldsymbol{z}}_k$ into a set of S local activations, each

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

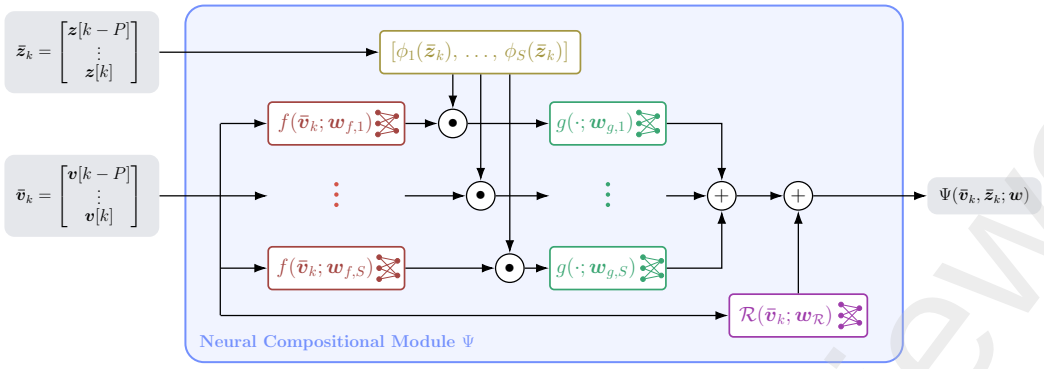


Figure 2: Internal structure of a Neural Compositional Module (NCM) Ψ. The module receives the finite temporal input $\bar{\boldsymbol{v}}_k$, constructed from the available model variables over a memory window of length P, and the scheduling variable $\bar{\boldsymbol{z}}_k$, constructed from scheduling variables selected from $\bar{\boldsymbol{v}}_k$, provided as external signals, or obtained as functions of available measurements. The scheduling variable is mapped into the activation functions $\phi_s(\bar{\boldsymbol{z}}_k)$, $s = 1, \ldots, S$, where S is the number of local models or operating regimes. In each branch, the nonlinear input function $f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{f,s})$ generates local features, which are weighted by the corresponding activation coefficient and then processed by the output temporal operator $g(\cdot; \boldsymbol{w}_{g,s})$, for instance a finite-memory or FIR operator. The functions f and g share the same functional structure across all branches, while their learnable parameters $\boldsymbol{w}_{f,s}$ and $\boldsymbol{w}_{g,s}$ are regime-dependent. The branch outputs are summed and optionally complemented by a black-box residual network $\mathcal{R}(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{\mathcal{R}})$, which accounts for unmodeled effects or approximation errors.

creating a *local model*. The function is not typeset in bold because it is defined as a scalar function and repeated across the entire temporal dimension $P + 1$. Each activation function is associated with a specific local operation regime of the physical system, and the corresponding activation $\phi_s(\bar{\boldsymbol{z}}_k)$, $s = \{1, \ldots, S\}$, is used to weigh the contribution of the s-th local model.

- Learnable **input functions** $f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{f,s})$. They map $\mathbb{R}^{(P+1)\times n_v}$ to $\mathbb{R}^{P+1}$ and process the inputs $\bar{\boldsymbol{v}}_k$ using custom physics-based transformations, equation-learning operators, or generic neural mappings, depending on the available prior knowledge. The function is not typeset in bold because it is defined as a scalar function and repeated across the entire temporal dimension $P + 1$. The learnable parameters $\boldsymbol{w}_{f,s}$ are specific to the s-th local regime, activated through the corresponding $\phi_s(\bar{\boldsymbol{z}}_k)$.

- Learnable **output functions** $g(\cdot; \boldsymbol{w}_{g,s})$. They map $\mathbb{R}^{P+1}$ to $\mathbb{R}$ and produce a scalar output from the locally-activated models. The learnable parameters $\boldsymbol{w}_{g,s}$ are specific to the s-th local regime, activated through the corresponding $\phi_s(\bar{\boldsymbol{z}}_k)$.

- Learnable **residual function** $\mathcal{R}(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{\mathcal{R}})$. It maps $\mathbb{R}^{(P+1)\times n_v}$ to $\mathbb{R}$ and accounts for unmodeled effects, higher-order terms, neglected dynamics, or approximation errors that are not represented by the structured part of the NCM. Its learnable parameters are $\boldsymbol{w}_{\mathcal{R}}$.

Our NCM formulation is a design template: depending on the specific problem, available prior knowledge, and desired level of interpretability, some components of the NCM in (6) may be removed. In the following, we describe each component in detail, highlighting the design choices and the corresponding `nnodely` coding blocks that implement them.

3.2.1. Regime-Dependent Activation Functions and Local Models

A first class of building blocks for NCMs is devoted to regime-dependent modeling, in which a nonlinear effect is learned as a weighted combination of local models associated with different operating conditions. Inside NCMs, regime-dependent local models are formulated using Eq. (6), with the exclusion of the residual term $\mathcal{R}$:

$$y_{LM} = \sum_{s=1}^{S} g\Big(\phi_s(\bar{\boldsymbol{z}}_k) \odot f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{f,s}); \boldsymbol{w}_{g,s}\Big), \tag{7}$$

where y_{LM} denotes the scalar local-model output, $\phi_s(\bar{\boldsymbol{z}}_k)$ is the activation function associated with the s-th regime, $\bar{\boldsymbol{z}}_k$ are the scheduling variables defined in Eq. (2), while $f(\cdot)$ and $g(\cdot)$ are the input and output functions of the local model. A schematic representation is shown in Fig. 3.

We design the shape of the activation functions based on prior knowledge about the system and the scheduling variables. For continuous scheduling variables, we adopt smooth activation functions, such as triangular or Gaussian functions. For discrete or categorical variables, rectangular activation functions can instead be used to generate differentiable analogues of one-hot encodings. In our framework, the local-model formulation is based on activation functions satisfying the partition-of-unity property, and is closely related to Takagi–Sugeno models [55] and adaptive neuro-fuzzy systems (ANFIS) [56], where smooth membership functions are used to interpolate among multiple local models. Differently from these general-purpose local modeling approaches, we employ the same principle as a structured weighting mechanism inside our NCMs, but specialize it

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

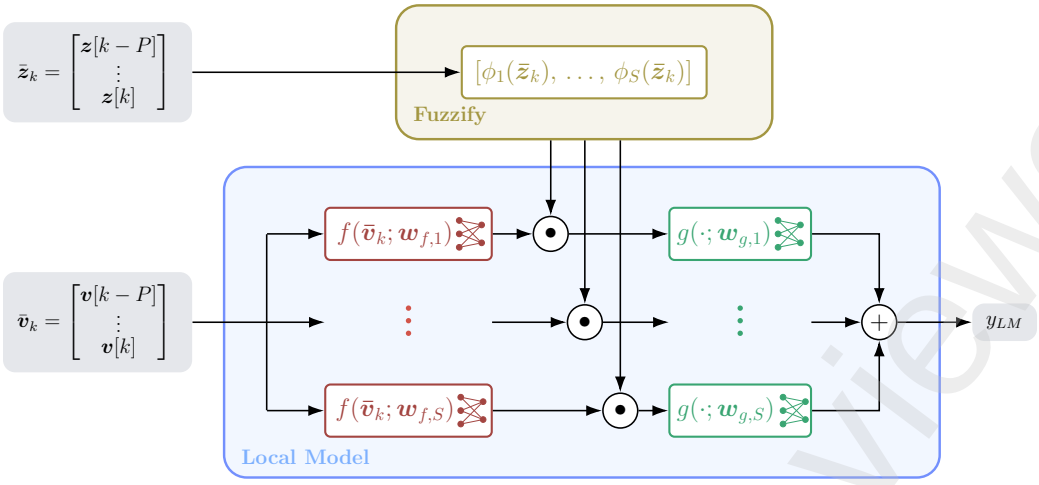


Figure 3: Structure of the `LocalModel` block. The scheduling variable $\bar{z}_k$ is processed by the `Fuzzify` block, producing the activation coefficients $[\phi_1(\bar{z}_k), \ldots, \phi_S(\bar{z}_k)]$. The input $\bar{v}_k$ is processed by local input functions $f(\bar{v}_k; \boldsymbol{w}_{f,s})$, weighted by the corresponding activation coefficients, and mapped to the output through local functions $g(\cdot; \boldsymbol{w}_{g,s})$. The resulting local contributions are finally summed to produce the output y_k.

to dynamics learning of physical systems by embedding the local models within an architecture whose input and output functions can be shaped by prior knowledge, enabling physically meaningful nonlinear relations and temporal dependencies to be represented while preserving continuity and interpretability.

The regime-dependent activation functions $\phi_s(\bar{z}_k)$ introduced above are scalar functions applied pointwise to the scheduling variables $\boldsymbol{z}$ over the temporal window collected in $\bar{z}_k$, where the index s identifies the specific activation function associated with a local regime. To clarify their construction, consider a single scalar scheduling variable z at a generic time instant. The variable z can be interpreted as a shifted representation of an underlying, unshifted variable ζ with respect to the center ζ_s associated with the s-th local regime. More precisely, we define $z = \zeta - \zeta_s$.

When the activation functions share the same shape and differ only in the location of their centers, they can be generated from a common prototype function $\phi(\cdot)$ according to

$$\phi_s(z) = \phi\left(\zeta - \zeta_s\right). \tag{8}$$

In this case, $\phi(\cdot)$ is independent of the specific local regime, while the different activations are obtained by changing the center ζ_s, as illustrated by the triangular and rectangular functions in Fig. 4(a)–(b). In the more general case, however, each activation may also depend on the complete set of centers. For example, when a Gaussian kernel $\phi(\cdot)$ is used, the activation functions must be normalized to satisfy the partition-of-unity property, as illustrated in Fig. 4(c), yielding

$$\phi_s(z) = \phi_s(\zeta - \zeta_s) = \frac{\phi\left(\zeta - \zeta_s\right)}{\sum_{r=1}^{S} \phi\left(\zeta - \zeta_r\right)}. \tag{9}$$

The normalization in Eq. (9) provides a practical way to enforce the partition-of-unity property when the underlying activation kernels do not satisfy it directly. More generally, regardless of the specific functional form adopted, the activation functions $\phi_s(\bar{z}_k)$ are selected or normalized such that

$$\sum_{s=1}^{S} \phi_s(\bar{z}_k) = 1, \quad \phi_s(\bar{z}_k) \geq 0, \quad s = 1, \ldots, S, \tag{10}$$

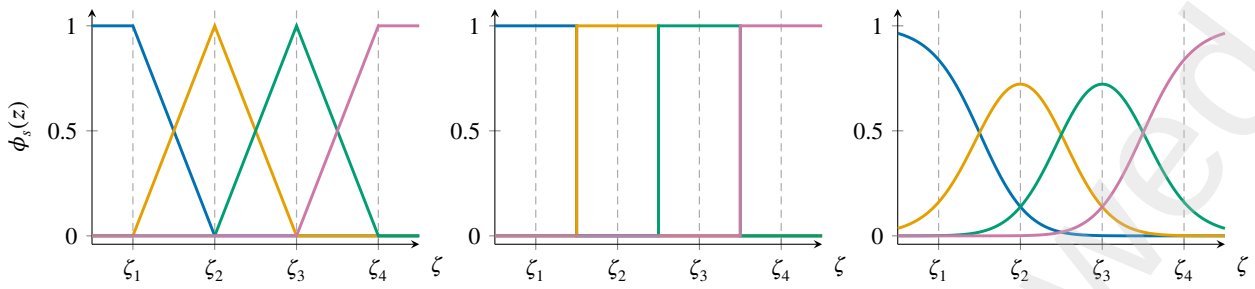
where S is the number of local models or operating regimes. This property ensures that the local models are combined through a convex interpolation across the operating regimes, with their relative contributions determined by the corresponding activation values.

When the scheduling variables are multidimensional, i.e., $\bar{z}_k = [\bar{z}_{k,1}, \ldots, \bar{z}_{k,n_z}] \in \mathbb{R}^{(P+1)\times n_z}$, the activation functions of the local models must account for the combinations of operating regimes along the different scheduling dimensions. Let S_ℓ denote the number of local regimes associated with the ℓ-th scheduling variable, and let $\phi_{s_\ell,\ell}(\bar{z}_{k,\ell})$, with $s_\ell = 1, \ldots, S_\ell$, be the corresponding one-dimensional activation functions. The total number of multidimensional local regimes is then $S = \prod_{\ell=1}^{n_z} S_\ell$.

We enumerate the multidimensional regimes with a scalar index $s = 1, \ldots, S$. Each s identifies one combination of values $(s_1, \ldots, s_\ell, \ldots, s_{n_z})$, where $s_\ell \in \{1, \ldots, S_\ell\}$ is the selected one-dimensional regime for the ℓ-th scheduling variable. The corresponding multidimensional activation is then defined as the product of the selected one-dimensional activations:

$$\phi_s(\bar{z}_k) = \prod_{\ell=1}^{n_z} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}). \tag{11}$$

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813



(a) Triangular (b) Rectangular (c) Normalized Gaussian

Figure 4: Examples of regime-dependent activation functions for a scalar scheduling variable. The horizontal axis represents the unshifted scheduling variable ζ, while ζ_s, $s \in \{1, \dots, 4\}$, denote the activation centers; equivalently, each activation can be expressed in terms of the shifted variable $z = \zeta - \zeta_s$. (a) Triangular and (b) rectangular activation functions obtained by translating a common prototype function $\phi(\cdot)$ across the different centers. (c) Normalized Gaussian activation functions, for which each activation also depends on the other centers through the normalization term. In all cases, the activation functions satisfy the partition-of-unity property $\sum_{s=1}^{S} \phi_s(z) = 1$. The colors identify the activation functions associated with the four centers.

Each one-dimensional activation family satisfies the partition-of-unity property independently along its scheduling variable:

$$\sum_{s_\ell=1}^{S_\ell} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) = 1, \quad \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \geq 0, \qquad s_\ell = 1, \dots, S_\ell, \quad \ell = 1, \dots, n_z. \tag{12}$$

Since the scalar index s enumerates all combinations of the one-dimensional regime indices $(s_1, \dots, s_{n_z})$, summing the multidimensional activations over all $s = 1, \dots, S$ is equivalent to summing over the full Cartesian product of the one-dimensional regimes. Therefore,

$$\begin{aligned} \sum_{s=1}^{S} \phi_s(\bar{\mathbf{z}}_k) &= \sum_{s=1}^{S} \prod_{\ell=1}^{n_z} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \\ &= \sum_{s_1=1}^{S_1} \cdots \sum_{s_{n_z}=1}^{S_{n_z}} \prod_{\ell=1}^{n_z} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \\ &= \prod_{\ell=1}^{n_z} \left(\sum_{s_\ell=1}^{S_\ell} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \right) = \prod_{\ell=1}^{n_z} 1 = 1. \end{aligned} \tag{13}$$

Thus, the separable multidimensional activations inherit the partition-of-unity property from the one-dimensional activation families. See Example 1.

In our `nnodely` library, regime-dependent local models and their activation functions are implemented with the `LocalModel` (Sec. 4.1.2) and `Fuzzify` (Sec. 4.1.1) coding blocks, respectively.

Example 1. *Consider $n_z = 2$ scheduling components with $S_1 = 2$ and $S_2 = 3$ one-dimensional activations, so that $S = S_1 S_2 = 6$. In this example, the scalar index s is obtained from the pair (s_1, s_2) through the mixed-radix enumeration $s = (s_1 - 1)S_2 + s_2$, with $s_1 \in \{1, 2\}$ and $s_2 \in \{1, 2, 3\}$. This assigns a unique scalar index to each combination of one-dimensional regimes and gives the correspondence*

s	**1**	**2**	**3**	**4**	**5**	**6**
s_1	1	1	1	2	2	2
s_2	1	2	3	1	2	3

Suppose that at a given operating point $\bar{\mathbf{z}}_k = [\bar{z}_{k,1}, \bar{z}_{k,2}]$, the activation functions satisfy the partition-of-unity property:

$$\begin{aligned} \sum_{s=1}^{6} \phi_s(\bar{\mathbf{z}}_k) &= \sum_{s=1}^{6} \phi_s(\bar{z}_{k,1}, \bar{z}_{k,2}) \\ &= \sum_{s=1}^{6} \prod_{\ell=1}^{2} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \\ &= \sum_{s_1=1}^{2} \sum_{s_2=1}^{3} \prod_{\ell=1}^{2} \phi_{s_\ell,\ell}(\bar{z}_{k,\ell}) \\ &= \sum_{s_1=1}^{2} \sum_{s_2=1}^{3} \left(\phi_{s_1,1}(\bar{z}_{k,1}) \cdot \phi_{s_2,2}(\bar{z}_{k,2}) \right) \\ &= \left(\sum_{s_1=1}^{2} \phi_{s_1,1}(\bar{z}_{k,1}) \right) \left(\sum_{s_2=1}^{3} \phi_{s_2,2}(\bar{z}_{k,2}) \right) \\ &= 1 \cdot 1 = 1. \end{aligned} \tag{14}$$

If the one-dimensional activations take the values

$$\begin{aligned} \phi_{1,1} &= 0.7, & \phi_{1,2} &= 0.2, \\ \phi_{2,1} &= 0.3, & \phi_{2,2} &= 0.5, \\ & & \phi_{3,2} &= 0.3, \end{aligned}$$

each family satisfying (12): $\phi_{1,1} + \phi_{2,1} = 1$ and $\phi_{1,2} + \phi_{2,2} + \phi_{3,2} = 1$. The six multidimensional activations from (11) are then

$$\begin{aligned} \phi_1 &= \phi_{1,1}\phi_{1,2} = 0.7 \times 0.2 = 0.14, \\ \phi_2 &= \phi_{1,1}\phi_{2,2} = 0.7 \times 0.5 = 0.35, \\ \phi_3 &= \phi_{1,1}\phi_{3,2} = 0.7 \times 0.3 = 0.21, \\ \phi_4 &= \phi_{2,1}\phi_{1,2} = 0.3 \times 0.2 = 0.06, \end{aligned}$$

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

$$\phi_5 = \phi_{2,1}\phi_{2,2} = 0.3 \times 0.5 = 0.15,$$
$$\phi_6 = \phi_{2,1}\phi_{3,2} = 0.3 \times 0.3 = 0.09,$$

and indeed $\sum_{s=1}^{6} \phi_s = 0.14 + 0.35 + 0.21 + 0.06 + 0.15 + 0.09 = 1$*, confirming* (14).

3.2.2. Input Functions

The input functions $f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f)$ in the NCM (6) transform the inputs $\bar{\boldsymbol{v}}_k$ into learned features, capturing static nonlinearities and interactions among the input variables of the physical system. Specifically, $f(\cdot)$ is one of the parts of the NCM where prior physics-based knowledge about the system can be directly embedded. Depending on the available prior knowledge, $f(\cdot)$ can be implemented as a learnable parametric physics-based function, an equation-learning operator, a Taylor-like expansion, or a generic neural mapping.

When prior domain knowledge is available, $f(\cdot)$ can be designed as a *parametric physics-based function*, namely a component whose functional form is derived from prior physical knowledge and whose parameters are learned from data. In abstract form, such a block can be represented as:

$$f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f) = f^{\text{phys}}(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f), \tag{15}$$

In our `nnodely` library, this is implemented through the `ParametricFunction` block (Sec. 4.1.3), which allows the user to define a custom function $f^{\text{phys}}(\cdot)$ and its learnable parameters $\boldsymbol{w}_f$.

When limited physics priors are available but a meaningful set of candidate terms can be defined, the same `ParametricFunction` block can implement Taylor-like or basis-expansion representations. In this case, the input function is expressed as a learnable combination of prescribed basis functions:

$$f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f) = \sum_{r=1}^{N} w_r \, \rho_r(\bar{\boldsymbol{v}}_k), \tag{16}$$

where $\rho_r(\cdot)$ are fixed analytical terms, such as polynomial, Taylor-like, or physically motivated basis functions, w_r are learnable coefficients, and N denotes the number of basis terms. A relevant special case is the first-order Taylor approximation around an operating point $\boldsymbol{v}_s \in \mathbb{R}^{n_v}$. Assuming that the underlying function $\tilde{f}(\boldsymbol{v})$ is continuously differentiable in a neighborhood of $\boldsymbol{v}_s$, its local expansion can be written as

$$\begin{aligned} f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f) &= \overbrace{\tilde{f}(\boldsymbol{v}_s)}^{\tilde{w}_0} + \sum_{j=1}^{n_v} \overbrace{\left.\frac{\partial \tilde{f}(\boldsymbol{v})}{\partial v_j}\right|_{\boldsymbol{v}_s}}^{w_j} (\bar{v}_{k,j} - v_{j,s}) \\ &= \overbrace{\tilde{w}_0 - \sum_{j=1}^{n_v} w_j v_{j,s}}^{w_0} + \sum_{j=1}^{n_v} w_j \bar{v}_{k,j} \\ &= w_0 + \sum_{j=1}^{n_v} w_j \bar{v}_{k,j}. \end{aligned} \tag{17}$$

where $\boldsymbol{w}_f = [w_0, w_1, \ldots, w_{n_v}]$, and $N = n_v + 1$ and $\bar{v}_{k,j}$ denotes the j-th scalar component of the temporal vector $\bar{v}_k$. This formulation provides a locally affine approximation around $\boldsymbol{v}_s$ without requiring explicit knowledge of either the underlying function or its analytical derivatives. The linearization point will be identified by $\bar{\boldsymbol{z}}_k$ as shown in Section 3.2.5.

This provides an intermediate representation between a fully-prescribed physical law and a black-box neural approximation: the admissible functional forms are constrained by the chosen basis, while the relative contribution of each term is learned from data.

Alternatively, when the qualitative structure of the input relation is partially known, or when the relevant nonlinear terms are expected to belong to a predefined functional library, $f(\cdot)$ can be implemented as an *equation-learning operator*. For example, robot dynamics often involve trigonometric functions of joint angles, while fluid dynamics may involve polynomial terms in velocity or pressure.

The operator first builds intermediate affine features from the input variables, then applies a library of nonlinear functions, and finally combines the resulting candidate terms through learnable coefficients. A scalar time-wise representation of the equation-learning operator is

$$\begin{aligned} f(\boldsymbol{v}[j]; \boldsymbol{w}_f) = \boldsymbol{w}_{\text{out}} \, h\left(\boldsymbol{w}_{\text{in}} \, \boldsymbol{v}[j] + \boldsymbol{b}_{\text{in}}\right) + \boldsymbol{b}_{\text{out}}, \\ j = k - P, \ldots, k. \end{aligned} \tag{18}$$

where $\boldsymbol{v}[j] \in \mathbb{R}^{n_v}$ denotes the input vector at a single time instant j, $\boldsymbol{w}_{\text{in}} \in \mathbb{R}^{R \times n_v}$ and $\boldsymbol{b}_{\text{in}} \in \mathbb{R}^{R}$ are the input affine parameters, and R is the total number of intermediate variables provided to the nonlinear vector function $\boldsymbol{h}(\cdot) = [h_1(\cdot), \ldots, h_S(\cdot)]^\top : \mathbb{R}^R \to \mathbb{R}^S$ that maps these intermediate variables into S candidate nonlinear terms. The functions h_* are selected from a structured library of predefined functions. The output parameters $\boldsymbol{w}_{\text{out}} \in \mathbb{R}^S$ and $b_{\text{out}} \in \mathbb{R}$ combine the candidate terms into the scalar output.

In the NCM formulation, the input function is applied to the temporal input $\bar{\boldsymbol{v}}_k \in \mathbb{R}^{(P+1) \times n_v}$. For notational simplicity, Eq. (18) is written for a single time instant j; in practice, the same equation-learning operator is applied to each sample $j = k - P, \ldots, k$ of the temporal window. Therefore, $f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_f)$ is obtained by repeating the operation along the temporal dimension $P + 1$.

Fig. 5 shows a single-stage `EquationLearner`, where the model identifies relevant functional components and their coefficients while preserving an explicit and interpretable structure. More complex relations can be represented by stacking multiple stages. In this case, the output affine map of one stage produces the intermediate variables required by the nonlinear functions $\boldsymbol{h}(\cdot)$ of the following stage, so that the corresponding weight matrix maps the candidate terms generated at the current stage to the input dimension of the next one.

This approach is related to the Sparse Identification of Nonlinear Dynamics (SINDy) in [26] and the equation learner networks in [11], where governing equations are discovered by selecting combinations of candidate functions

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

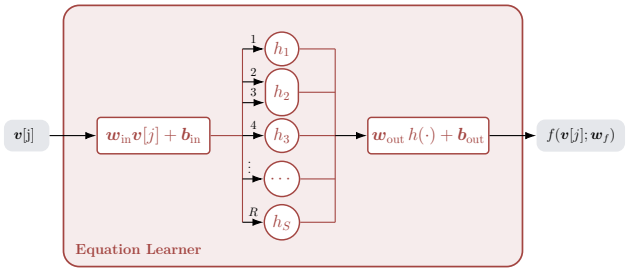


Figure 5: Structure of the `EquationLearner` block. At each time instant j of the temporal window, the input $\boldsymbol{v}[j]$ is first transformed through the affine map $\boldsymbol{w}_{\text{in}}\boldsymbol{v}[j]+\boldsymbol{b}_{\text{in}}$, producing R intermediate features. These features are processed by a library of predefined nonlinear functions $h_i(\cdot)$, which may include algebraic, trigonometric, or custom operators. The resulting S candidate terms are then combined through the output affine map $\boldsymbol{w}_{\text{out}}h(\cdot)+\boldsymbol{b}_{\text{out}}$, yielding $f(\boldsymbol{v}[j];\boldsymbol{w}_f)$. The same operation is repeated for all samples $j = k-P,\dots,k$ of the temporal input window. The figure shows a single-stage `EquationLearner`; more complex relations can be obtained by stacking multiple stages, where the output affine map of one stage provides the intermediate variables required by the nonlinear functions of the following stage.

from predefined libraries. Here, the functional library is embedded directly into a differentiable computational graph, allowing structure selection and parameter estimation to be performed jointly through gradient-based learning. The specific neural architecture adopted in this work follows the formulation introduced in [57].

In Sec. 5, we will show how the input function $f(\cdot)$ can be used to embed prior knowledge and retain physical interpretability in the learned model, while still allowing for data-driven flexibility.

Hence, the learnable input function $f(\cdot)$ provides an interface between prior knowledge and data-driven flexibility inside the NCM. The corresponding coding components in `nnodely` are `ParametricFunction` (Sec. 4.1.3) and `Equation-Learner` (Sec. 4.1.4).

3.2.3. Output Functions

The output function $g(\cdot)$ in the NCM structure (6) maps the activated input features $\phi_s(\bar{\boldsymbol{z}}_k)\odot f(\bar{\boldsymbol{v}}_k)$ into the scalar output of the s-th local regime. While the input function $f(\cdot)$ captures static nonlinearities and interactions among the input variables, the output function $g(\cdot)$ learns the transient response of the system, including temporal dependencies, delays, and memory effects, and can also be used to mitigate input noise (see Section 5.1). Let

$$\bar{\xi}_{k,s} = \phi_s(\bar{\boldsymbol{z}}_k)\odot f(\bar{\boldsymbol{v}}_k;\boldsymbol{w}_f) = \begin{bmatrix} \xi_s[k-P] \\ \vdots \\ \xi_s[k-1] \\ \xi_s[k] \end{bmatrix} \in \mathbb{R}^{P+1} \quad (19)$$

denote the activated local features entering the output operator $g(\cdot)$. Depending on the modeled relation, $g(\cdot)$ can either be an identity operator, or a temporal operator acting on the finite history of activated features $\bar{\xi}_{k,s}$ (with temporal length $P+1$). When learning sampled-time dynamical models of physical systems, we design $g(\cdot)$ as a finite-memory operator, learning the temporal dependencies of the activated features over the finite window of past samples in $\bar{\xi}_{k,s}$:

$$y_s[k] = g(\bar{\xi}_{k,s}) = b_s + \sum_{p=0}^{P} w_{p,s}\,\xi_s[k-p]. \quad (20)$$

In (20), $w_{p,s}$ are learnable weights to capture the temporal dependencies, b_s is an optional learnable offset, P defines the temporal window used by the operator, and $\xi_s[k-p]$ denotes the activated feature at time $k-p$. This formulation provides a compact representation of causal temporal dependencies inside the NCM, and implements a learnable Finite Impulse Response (FIR) filter. In NN terminology, $g(\cdot)$ is a fully connected layer with $P+1$ weights and one bias.

When used in combination with S local models and regime-dependent activations $\phi_s(\bar{\boldsymbol{z}}_k)$, the output function $g(\cdot)$ allows the NCM to learn the dynamic response of the physical system in different operating regimes, with different sets of weights $w_{p,s}$ and biases b_s for each regime, $s \in \{1,\dots,S\}$.

Our formulation of the output function $g(\cdot)$ as a learnable FIR operator provides a clear physical interpretation of the learned weights $w_{p,s}$, which represent the contribution of past activated features to the current output. In Sec. 5, we will show how the interpretability of $g(\cdot)$ can be exploited in practical applications, allowing us to analyze the temporal dependencies of the system and gain insights into its underlying dynamics.

In our `nnodely` library, the output function $g(\cdot)$ is implemented as a `Fir` coding block (Sec. 4.1.5), while the construction of temporal windows on signals is described in Sec. 4.1.

3.2.4. Residual Function

The residual block implements the term $\mathcal{R}(\bar{\boldsymbol{v}}_k;\boldsymbol{w}_{\mathcal{R}})$ of our NCM in Eq. (6). Its role is to compensate for unmodeled effects, higher-order contributions, neglected dynamics, or approximation errors that remain after the structured part of the NCM is specified. In this way, the main NCM contribution (input and output functions with local activations) can preserve its physical or local interpretability, while the residual provides additional flexibility when the available prior knowledge is incomplete. We implement the residual block as a Multi-Layer Perceptron (MLP) acting on the input $\bar{\boldsymbol{v}}_k$:

$$\begin{aligned} \boldsymbol{v}^{(0)} &= \bar{\boldsymbol{v}}_k, \\ \boldsymbol{v}^{(a)} &= h_a\left(\boldsymbol{w}_a\boldsymbol{v}^{(a-1)} + \boldsymbol{b}_a\right), \quad a = 1,\dots,L, \\ \mathcal{R}(\bar{\boldsymbol{v}}_k;\boldsymbol{w}_{\mathcal{R}}) &= \boldsymbol{w}_{\text{out}}\boldsymbol{v}^{(L)} + \boldsymbol{b}_{\text{out}}. \end{aligned} \quad (21)$$

where $\boldsymbol{v}^{(0)}$ is the input, $\boldsymbol{v}^{(a)}$ is the hidden representation at layer a of the MLP, $h_a(\cdot)$ is the corresponding activation function, and L is the number of hidden layers. The vector $\boldsymbol{w}_{\mathcal{R}}$ collects all weights and biases of the residual network,

Preprint not peer reviewed
This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

namely $\boldsymbol{w}_{\mathcal{R}} = \{\boldsymbol{w}_a, \boldsymbol{b}_a\}_{a=1}^{L} \cup \{\boldsymbol{w}_{\text{out}}, \boldsymbol{b}_{\text{out}}\}$. If the residual is intended to act as a bounded correction, the final affine map can be followed by a bounded output activation, so that the residual contribution remains within a controlled range.

The corresponding implementation in the `nnodely` library, based on the `Linear` block and nonlinear activation functions, is described in Sec. 4.1.6.

3.2.5. Classes of Systems Represented by NCMs

This section illustrates how the NCM structure in Eq. (6) can represent different classes of (physical) dynamical systems, depending on the choice of local models, input functions, and output operators.

Finite-Memory Linear Dynamical Systems: In the simplest case, a discrete-time, single-input, linear time-invariant system with finite memory is exactly represented by a single local model ($S = 1$) with no input function. In this case, the NCM in (6) reduces to a learnable output function $g(\cdot)$ with an optional bias residual b:

$$\Psi(\bar{v}_k; \boldsymbol{w}) = g(\bar{v}_k; \boldsymbol{w}_g) = b + \sum_{p=0}^{P} w_p \, v[k-p], \qquad (22)$$

where $\bar{v}_k = [v[k-P], \dots, v[k]]^\top$ is the past history of the single input variable v. The NCM (22) is a linear Finite Impulse Response (FIR) system, and the learnable weights $\boldsymbol{w}_g = [w_0, \dots, w_P]^\top$ are the FIR coefficients. The FIR system is a finite-memory approximation of a general linear time-invariant system, where the output at time k is a linear combination of the current and past input samples. The optional bias term b accounts for a constant offset in the output. Thus, the output function $g(\cdot)$ is a linear fully-connected layer with $P+1$ weights and one bias, acting on the input $\bar{v}_k$.

Finite-Memory Locally-Linear Dynamical Systems: A locally-linear extension of (22) is obtained by introducing multiple local models of type (22), each activated by a regime-dependent function $\phi_s(\bar{\boldsymbol{z}}_k)$, $s = \{1, \dots, S\}$:

$$\begin{aligned}\Psi(\bar{v}_k, \bar{\boldsymbol{z}}_k; \boldsymbol{w}) &= \sum_{s=1}^{S} g\Big(\overbrace{\phi_s(\bar{\boldsymbol{z}}_k) \odot \bar{v}_k}^{\bar{\xi}_{k,s}}; \boldsymbol{w}_{g,s}\Big) \\ &= \sum_{s=1}^{S} g\Big(\bar{\xi}_{k,s}; \boldsymbol{w}_{g,s}\Big) \\ &= \sum_{s=1}^{S} \overbrace{\left(b_s + \sum_{p=0}^{P} w_{p,s}\, \xi_s[k-p]\right)}^{g(\bar{\xi}_{k,s}; \boldsymbol{w}_{g,s})}.\end{aligned} \qquad (23)$$

where b_s and $w_{p,s}$ are the learnable bias and FIR weights of the s-th local model, respectively. Equivalently, the same NCM (23) can be interpreted as a finite-memory operator with coefficients that depend on the operating regime, as described by the scheduling variables $\bar{\boldsymbol{z}}_k$:

$$\begin{aligned}&\Psi(\bar{v}_k, \bar{\boldsymbol{z}}_k; \boldsymbol{w}) = b(\bar{\boldsymbol{z}}_k) + \sum_{p=0}^{P} w_p(\bar{\boldsymbol{z}}_k)\, v[k-p], \quad \text{with} \\ &b(\bar{\boldsymbol{z}}_k) = \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{z}}_k)\, b_s, \\ &w_p(\bar{\boldsymbol{z}}_k) = \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{z}}_k)\, w_{p,s}.\end{aligned} \qquad (24)$$

Locally-Linearized Functions: A locally-linearized NCM can be used when the nonlinear unknown relation $\tilde{f}(\boldsymbol{v})$ is approximated through a set of first-order expansions around local operating points $\boldsymbol{v}_s$, $s \in \{1, \dots, S\}$ as in Eq. (17). Each local approximation is weighted by the corresponding regime-dependent activation function $\phi_s(\bar{\boldsymbol{z}}_k)$. In this case, the scheduling variables are defined as the displacement of the input variables from the local operating point, i.e., $\bar{\boldsymbol{z}}_k = \bar{\boldsymbol{v}}_k - \boldsymbol{v}_s$, as in Eq. (8). By applying a first-order Taylor expansion around $\boldsymbol{v}_s$, the NCM can be written as

$$\begin{aligned}\Psi(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k; \boldsymbol{w}) &= \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{z}}_k) \odot f(\bar{\boldsymbol{v}}_k; \boldsymbol{w}_{f,s}) \\ &= \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{z}}_k) \odot \left(\overbrace{\tilde{f}(\boldsymbol{v}_s)}^{\tilde{w}_{0,s}} + \sum_{j=1}^{n_v} \overbrace{\left.\frac{\partial \tilde{f}(\boldsymbol{v})}{\partial v_j}\right|_{\boldsymbol{v}_s}}^{w_{j,s}} (\bar{v}_{k,j} - v_{j,s})\right) \\ &= \sum_{s=1}^{S} \phi_s(\overbrace{\bar{\boldsymbol{v}}_k - \boldsymbol{v}_s}^{\bar{\boldsymbol{z}}_k}) \odot \left(\overbrace{\tilde{w}_{0,s} - \sum_{j=1}^{n_v} w_{j,s} v_{j,s} + \sum_{j=1}^{n_v} w_{j,s} \bar{v}_{k,j}}^{w_{0,s}}\right) \\ &= \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{v}}_k - \boldsymbol{v}_s) \odot \left(w_{0,s} + \sum_{j=1}^{n_v} w_{j,s} \bar{v}_{k,j}\right).\end{aligned} \qquad (25)$$

Here, $\bar{v}_{k,j}$ is the temporal history associated with the j-th column of the temporal input matrix $\bar{\boldsymbol{v}}_k$, and $v_{j,s}$ is the corresponding component of $\boldsymbol{v}_s$. Equivalently, the same NCM in Eq. (25) can be interpreted as an affine input function whose coefficients vary with the operating regime according to the scheduling variables $\bar{\boldsymbol{z}}_k = \bar{\boldsymbol{v}}_k - \boldsymbol{v}_s$:

$$\begin{aligned}&\Psi(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k; \boldsymbol{w}) = w_0(\bar{\boldsymbol{z}}_k) + \sum_{j=1}^{n_v} w_j(\bar{\boldsymbol{z}}_k) \odot \bar{v}_{k,j}, \quad \text{with} \\ &w_0(\bar{\boldsymbol{z}}_k) = \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{v}}_k - \boldsymbol{v}_s) w_{0,s}, \\ &w_j(\bar{\boldsymbol{z}}_k) = \sum_{s=1}^{S} \phi_s(\bar{\boldsymbol{v}}_k - \boldsymbol{v}_s) w_{j,s}.\end{aligned} \qquad (26)$$

Finite-Memory Locally-Linear Dynamical Systems with Input Nonlinearities: When we combine the learnable input functions $f(\cdot)$ to the model (23), the resulting NCM takes

Preprint not peer reviewed

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

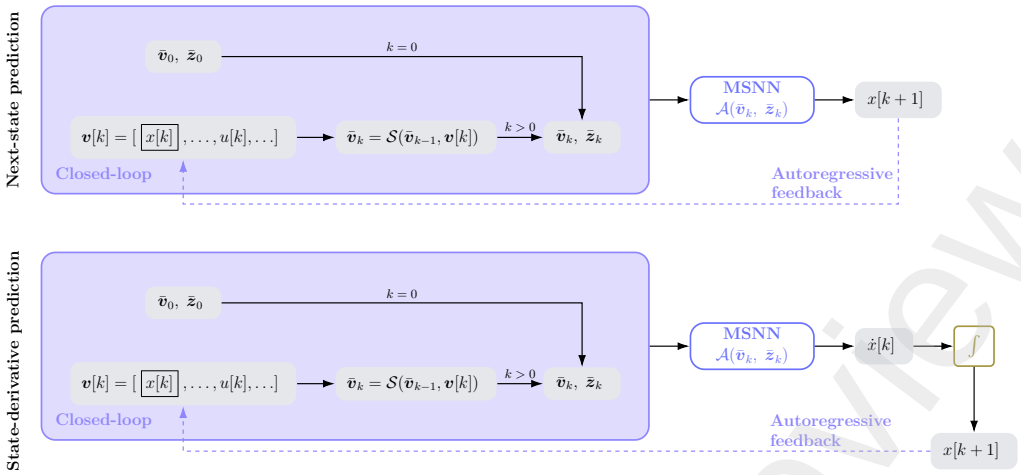


Figure 6: System dynamic evolution using NCM-based MSNNs. The figure shows two typical autoregressive realizations for temporal prediction. In the next-step prediction case, the MSNN directly predicts the next state $x[k+1]$, which is then fed back to the model. In the state derivative prediction case, the MSNN represents the vector field $y(t) \approx \dot{x}(t)$, and an integration block computes the sampled-time transition through a numerical integration operator over the sampling interval. In both cases, the closed-loop block initializes the inputs and scheduling variables $(\bar{\boldsymbol{v}}_0, \bar{\boldsymbol{z}}_0)$ at $k=0$; for $k>0$, the predicted state is inserted into the instantaneous vector $\boldsymbol{v}[k]$, and the shift operator $\mathcal{S}$ updates the finite-memory inputs $(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k)$. This feedback mechanism enables multi-step prediction and time-marching simulation, and corresponds to an autoregressive neural network realization.

the form in Eq. (7), which corresponds to a locally-linear dynamical system with input nonlinearities. This class of systems is particularly relevant in physical modeling, where the system may exhibit nonlinear behavior in its inputs, while still maintaining a locally linear dynamic response. As the number of local models S increases, the NCM can approximate a wide range of nonlinear behaviors, while still preserving interpretability through the physics-encoded design of the input functions $f(\cdot)$, activation functions $\phi_s(\cdot)$, and the FIR-type output functions $g(\cdot)$.

Finite-Memory Nonlinear Dynamical Systems: The most general class of systems represented by the NCM in Eq. (6) is obtained by including the residual term $\mathcal{R}(\cdot)$, in the form of a learnable MLP. In this case, the NCM can represent general nonlinear dynamical systems, with a structure that combines interpretable, physics-encoded local modules with flexible, data-driven residual terms. Moreover, the combination of multiple specialized NCMs into MSNN architectures (Sec. 3.1.2) allows the representation of complex, multi-output systems with several interacting components, while still preserving interpretability and physical insights through the design of the individual NCMs.

3.3. Multi-Step Dynamic Prediction using Model-Structured Neural Networks

This section presents two approaches for multi-step simulation of the system dynamics with MSNNs, enabling long-horizon time-marching prediction. The first method is based on next-state prediction (Fig. 6, top), where the MSNN predicts the next state, which is recursively fed back as input. The second is based on state-derivative prediction (Fig. 6, bottom), where the MSNN predicts the state derivative (vector field), and the next state is obtained by integrating it over a sampling interval. In both approaches, long-horizon prediction is achieved by recursively feeding back the predicted state, while updating the input through a shift operator that preserves a finite memory of past states and controls. In this section, we can interpret $\boldsymbol{\mathcal{A}}$ as a collection of operators $\mathcal{A}$, each producing a single output.

Next-Step Prediction: In this case, the MSNN predicts the next state $\boldsymbol{x}[k+1]$ of the system (Fig. 6, top):

$$\boldsymbol{x}[k+1] = \boldsymbol{\mathcal{A}}\left(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k\right). \tag{27}$$

For *one-step* prediction, the input $\bar{\boldsymbol{v}}_k$ is constructed from measured past states, controls, or other quantities (Sec. 3.1.1). For *multi-step* prediction, instead, the predicted state $\boldsymbol{x}[k+1]$ is fed back into the input $\boldsymbol{v}[k+1]$ used at the next step, yielding a recurrent update:

$$\bar{\boldsymbol{v}}_{k+1} = \mathcal{S}\left(\bar{\boldsymbol{v}}_k, \boldsymbol{v}[k+1]\right), \tag{28}$$

where $\mathcal{S}(\cdot)$ denotes the shift operator that removes the oldest samples from the memory window and appends the newly available variables. The sampled-time transition in Eq. (27)-(28) has an autoregressive character: the predicted states $\boldsymbol{x}[k+1]$ are fed back into the next input $\boldsymbol{v}[k+1]$, allowing the same MSNN architecture to be reused recursively over multiple time steps.

If the current state is sufficient to describe the past, the model reduces to a Markovian state-space update. If the input $\bar{\boldsymbol{v}}_k$ contains finite histories of measured states and controls, the same mechanism realizes finite-memory or

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

partially observed dynamics (Sec. 3.1.1). Thus, the autoregressive construction of $\bar{\boldsymbol{v}}_{k+1}$ determines how the system's memory is propagated during episodic rollouts.

State Derivative Prediction: In this case, the MSNN predicts the time derivative of the state $\dot{\boldsymbol{x}}[k]$ (Fig. 6, bottom):

$$\dot{\boldsymbol{x}}[k] = \boldsymbol{\mathcal{A}}\left(\bar{\boldsymbol{v}}_k, \bar{\boldsymbol{z}}_k\right). \tag{29}$$

Here, the MSNN output $\dot{\boldsymbol{x}}[k]$ represents the vector field of the system dynamics, which we then integrate numerically over the sampling interval $[t_k, t_{k+1}]$ to obtain the next state:

$$\boldsymbol{x}[k+1] = \boldsymbol{x}[k] + \mathcal{I}_{t_k}^{t_{k+1}}\left[\dot{\boldsymbol{x}}_k\right], \qquad t_{k+1} = t_k + \Delta t, \tag{30}$$

where $\mathcal{I}_{t_k}^{t_{k+1}}[\cdot]$ denotes the selected numerical integration scheme and Δt is the sampling time.

In this setting, the MSNN is employed as an executable neural operator: the integration scheme $\mathcal{I}_{t_k}^{t_{k+1}}[\cdot]$ may need to evaluate the vector field not only at the time t_k, but also at intermediate steps within the interval $[t_k, t_{k+1}]$ (*e.g.*, in Runge-Kutta methods). After integration, the updated state $\boldsymbol{x}[k+1]$ is inserted into the input $\boldsymbol{v}[k+1]$ through the same shift mechanism in Eq. (28), enabling time-marching simulation.

In our `nnodely` library, autoregressive connections are handled by the composition mechanisms described in Sec. 4.6, while numerical integration is implemented through the `Integrate` command presented in Sec. 4.1.7, which allows the user to select the desired integration scheme and its parameters.

4. nnodely Framework

The `nnodely` framework provides the computational implementation for the compositional modeling approach introduced in Section 3. In particular, it enables the construction of NCMs and of the MSNN architectures obtained by composing them. In this setting, MSNNs are represented as explicit and structured computational models, rather than as opaque neural network architectures. The framework is released as open-source software and is publicly available through the repository `https://github.com/tonegas/nnodely`, with an archived release available on Zenodo.

At the same time, the framework is not limited to NCM-based MSNNs. More generally, `nnodely` supports different ways of embedding prior physical knowledge into neural models, including physics-guided input transformations, physics-encoded architectural structures, and physics-informed loss terms. Thus, physical knowledge can be introduced at different levels of the learning pipeline: in the definition of the input variables and features, in the computational graph of the neural architecture, and in the objective functions used during training.

The framework formalizes model structure, learnable parameters, temporal dependencies, and composition rules through a unified abstraction. This makes architectural choices explicit and reusable, allowing the user to build neural models from structured components such as parametric functions, local models, finite-memory operators, integration blocks, residual networks, and loss constraints. Therefore, `nnodely` can be used both to implement the specific MSNN methodology proposed in this work and to construct more general physics-aware neural models.

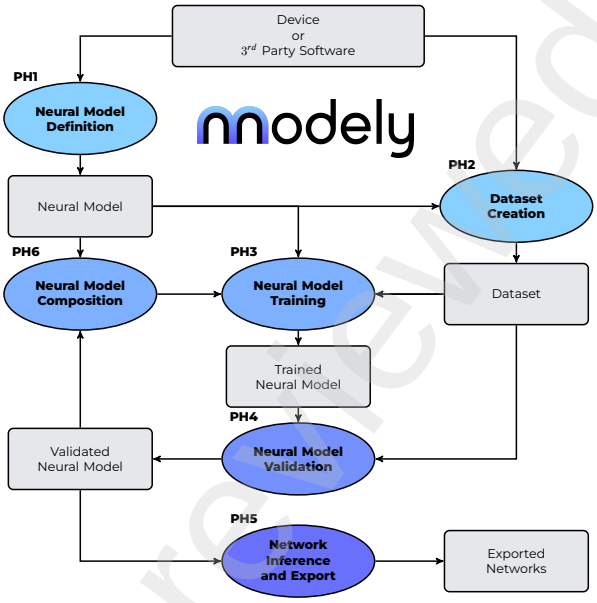


Figure 7: Overview of the `nnodely` development pipeline, covering model design (PH1), dataset construction consistent with the network architecture (PH2), training (PH3), domain-specific validation (PH4), model export (PH5), and composition of complex models (PH6). Ellipses denote the main pipeline phases, while rectangles represent the artifacts produced at each phase.

From this perspective, `nnodely` adopts a model-centric approach, where neural architectures are treated as computational representations of mathematical models. This viewpoint aligns the construction of neural models with established practices in modeling and simulation, supporting modular formulations that can be interpreted, validated, exported, and integrated into simulation-based workflows.

The main phases of `nnodely` are summarized in Fig. 7. The following subsections describe the corresponding computational steps, focusing on how structured and physics-aware neural models are realized in practice. Illustrative code snippets are provided to highlight the main functionalities supporting the compositional modeling approach, without aiming at a complete description of all implementation details or configuration options. For this reason, the examples are intentionally kept compact and as release-agnostic as possible, so that the methodological aspects remain valid across future framework extensions and backend updates. The code snippets reported below follow the syntax of `nnodely` version `v1.5.4`. Detailed usage instructions are available in the project repository.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

4.1. Neural Model Definition (PH1)

In `nnodely`, a neural model is specified through a small set of explicit and modular components: *Inputs*, *Parameters* or *Constants*, *Relations*, and *Outputs*. Together, these components define the structure, data flow, and learnable quantities of an MSNN, corresponding to the model definition (PH1) phase in Fig. 7. This decomposition makes the architectural choices explicit and provides a direct computational counterpart to the compositional modeling methodology introduced in Section 3.

Inputs define the interface between the model and the data. Each input is uniquely identified, dimensioned, and associated with a temporal context window, allowing access to past and/or future samples (or seconds). Inputs may originate from external datasets or from internal connections to other model components, enabling flexible stream routing within composite architectures (Section 4.6).

```
x = Input('x')                # 'x' ∈ ℝ
u = Input('u', dimensions=2) # 'u' ∈ ℝ^2
z = Input('z')                # 'z' ∈ ℝ
xs = x.sw(6)                  # last 6 samples of 'x'
ul = u.last()                 # last sample of 'u'
zt = z.tw([-2, 1])            # 'z' 2 past and 1 future samples
```

Temporal windows are supported through the `tw` and `sw` operators. The former defines windows in seconds, while the latter defines windows in samples. These operators allow inputs and intermediate variables to be transformed into temporally extended representations.

Parameters represent the learnable quantities of the model and explicitly identify which elements of the architecture are estimated from data. Each parameter is uniquely named and dimensioned, ensuring unambiguous reference even in highly structured models. Parameters can also be defined over temporal windows, which is useful for filters or time-dependent effects. Their initialization is treated as a modeling choice: besides standard random initialization, `nnodely` supports structured initialization schemes, such as smooth interpolation or decaying profiles, to encode prior assumptions directly into the learnable representation. *Constants* are defined analogously to parameters, but are not trainable.

```
w = Parameter('w', sw=3, dim=5)   # 5 features, 3 samples
c = Constant('c', values=[1,2,3]) # constant [1, 2, 3]
```

Relations define the computational structure of the model and constitute the core of the MSNN architecture. They implement elementary or higher-level neural operators, including filters, linear mappings, local models, parametric functions, equation learners, and temporal operators. All data tensors follow a shared shape convention, $[B, P, D]$, where B denotes the batch size, P the temporal window length, and D the stream dimension. This convention allows relations to be composed consistently while preserving the temporal and structural dependencies required by the methodology described in Section 3. The remainder of this section describes the main relation blocks provided by the framework.

Outputs expose selected relations as model predictions and define the interface between the internal architecture and the quantities of interest. Each output is associated with a specific computational stream, corresponding to a structured flow of data through the network, and is identified by a semantic tag. Outputs affect training only when explicitly included in the loss function; otherwise, they serve as access points for inference, analysis, or model composition.

```
y_out = Output('y', stream)  # output definition
```

Outputs can be combined to construct one or more neural models. Once a model is specified, `nnodely` performs a *neuralization* step that prepares the architecture for learning. During this step, temporal sampling windows are constructed according to the selected sampling time, providing the context required for dynamical modeling (Fig. 8a).

```
nn = nnodely()                          # nnodely object
nn.addModel('model_a', [y_out, ...]) # model definition
nn.neuralizeModel(0.5)                  # sample time 0.5 s
```

The complete model specification, including architecture, parameters, temporal windows, and output definitions, is internally stored in a framework-agnostic representation. This decouples model definition from any specific backend and supports reproducibility, interoperability, and deployment across heterogeneous software and hardware platforms. Since multiple models can be defined within the same `nnodely` object and interconnected to form a larger MSNN, these individual models are referred to in the following as *sub-models*. Their composition is discussed in Section 4.6.

4.1.1. `Fuzzify`

The `Fuzzify` relation implements the activation mechanism used in the local and regime-dependent structures introduced in Section 3.2.1. It maps a scalar scheduling variable to a set of activation coefficients that determine the contribution of different operating regimes.

Given an input tensor of dimensions $[B, P, 1]$, the block produces an output of size $[B, P, S]$, where S is the number of activation functions. Each output channel corresponds to a membership or activation function associated with a specific operating region.

```
fuzzy_xs = Fuzzify(centers=[-1,0,1],
    functions="Triangular")(xs)  # 3 activations on 'xs'
```

For discrete or categorical variables, rectangular activation functions can be used to obtain a differentiable analogue of one-hot encoding. For continuous variables, triangular or smooth membership functions can instead be employed to generate overlapping activation regions and smooth transitions between local behaviors.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

The block also supports custom activation functions, allowing the same mechanism to be used for more general distributed encodings or application-specific partitioning strategies, such as channel coding. In most MSNN architectures, `Fuzzify` is used together with `LocalModel` to weight and interpolate multiple local regime-dependent models.

4.1.2. `LocalModel`

The `LocalModel` relation implements the regime-dependent modeling structure introduced in Section 3.2.1. Its role is to combine multiple local regime dependent models according to activation coefficients, typically generated by a `Fuzzify` block (Section 4.1.1).

As shown in Fig. 3, each local branch is composed of an input transformation $f(\cdot; \boldsymbol{w}_{f,s})$, and an output transformation $g(\cdot; \boldsymbol{w}_{g,s})$. These transformations can be implemented using other relations, such as `Linear`, `Fir`, `ParametricFunction`, or `EquationLearner`. For the implementation of an NCM, we use `ParametricFunction` for f and `Fir` for g. This modular organization allows local models to be purely linear, dynamic, physics-based, or hybrid.

```
local_y = LocalModel(
    input_function = Sin,     # f(·) = Sin
    output_function = Fir     # g(·; w_{g,s}) = Fir
)(ul, fuzzy_xs)               # one local model per activation
```

Given an input tensor of dimensions [B, P, 1] and a fuzzy variable of dimensions [B, P, S], the output dimension in the code example is [B, 1, 1] due to the `Fir` output function.

Operationally, the block is particularly useful when the system behavior changes across operating conditions. Instead of relying on a single global model, the user defines a set of local behaviors together with a scheduling mechanism that blends them smoothly through the activation coefficients (commonly generated by `Fuzzify`). This enables nonlinear dynamics to be represented through compositions of simpler and more interpretable local structures.

The `LocalModel` relation naturally supports multiple scheduling variables, allowing for multi-axis partitioning of the operating space and the representation of more complex regime-dependent behaviors through a *tuple* of activation functions.

4.1.3. `ParametricFunction`

The `ParametricFunction` relation implements the parametric physics-based functions introduced in Section 3.2.2. It allows known analytical or empirical relations to be embedded directly into the neural architecture while exposing selected coefficients as learnable parameters. This block is particularly useful when the functional form of a phenomenon is known *a priori*, but the corresponding system-specific parameters remain uncertain and must be identified from data. Typical applications include friction laws, rigid-body dynamics parameterizations, or other analytical models involving unknown masses, inertias, damping coefficients, stiffnesses, or friction parameters.

```
def par_fun(x, W):                # parametric function
    ...                           # backend operations
    return y                      # function output

parfun_y = ParamFun(par_fun)(ul) # apply par_fun to ul
```

The `ParametricFunction` relation allows the user to define custom differentiable functions directly using the backend language, providing maximum flexibility while preserving end-to-end gradient-based optimization within the computational graph. The tensors passed to the function follow the shared tensor convention [B, P, D], requiring tensor operations and dimension handling to be explicitly managed inside the user-defined implementation. The distinction between inputs and learnable parameters is automatically inferred at call time from the number and type of streams provided to the relation.

For NCM implementations, the output tensor must comply with the shape [B, P, 1]. Additionally, the block supports integration with external libraries and third-party software, as demonstrated in Section 5.3, enabling complex physical models to be embedded directly into MSNN architectures. This facilitates the combination of classical modeling tools with data-driven parameter identification within a unified structured framework.

4.1.4. `EquationLearner`

The `EquationLearner` relation implements the equation-learning structures introduced in Section 3.2.2. It is designed for scenarios in which the relevant nonlinear terms are partially known or can be selected from a candidate library, while the final analytical relation remains unknown. The internal computational structure of the `EquationLearner` is illustrated in Fig. 5.

```
eq_learner_y = EquationLearner(
    functions = [Tan, Mul, Identity, ...], # op library
    linear_out = Linear(1)                 # scalar output
)(ul)
```

The block first projects the input variables through a learnable affine transformation, then applies a library of predefined nonlinear functions, and finally recombines the resulting candidate terms through an output affine mapping. The function library may include algebraic, trigonometric, polynomial, or user-defined operators, enabling the model to identify both the relevant functional terms and their coefficients while preserving an explicit and interpretable structure.

When several `EquationLearner` blocks are connected in a cascade, the output affine mapping ($\boldsymbol{w}_{out}, \boldsymbol{b}_{out}$) is omitted in the intermediate stages. In this case, the intermediate blocks only generate candidate nonlinear features, while the final affine recombination is performed by the output block.

Within the proposed framework, `EquationLearner` provides a partially structured alternative to standard black-box

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

neural components, combining the flexibility of neural approximators with a higher degree of symbolic interpretability.

4.1.5. `Fir`

The `Fir` relation implements the finite-memory operator introduced in Section 3.2.3. It is used to model temporal dependencies by applying a learnable discrete-time convolution along the temporal dimension of the input tensor. Given an input stream with dimensions $[B,\ P,\ 1]$, the block produces an output of size $[B,\ 1,\ 1]$, or $[B,\ 1,\ N]$ when a family of N filters is used.

```
fir_y = Fir(output_dimension=2)(zt)  # 2 filters on zt
```

The coefficients of the `Fir` block directly represent the impulse response of the modeled component. For this reason, their initialization is not only a numerical detail, but also a modeling choice. `nnodely` supports standard random initialization as well as structured profiles, such as decaying or smooth initializations, which are useful when modeling dissipative or low-pass mechanical effects. The block can be applied to raw inputs, state variables, or intermediate relations, enabling input filtering, sensor smoothing, and finite-memory dynamics.

4.1.6. `Linear`

The `Linear` relation implements instantaneous linear mappings across the feature dimension of a tensor. In the NCM, this operation is used to implement one layer of the residual component presented in Section 3.2.4. While `Fir` operates along the temporal dimension P, `Linear` operates along the feature dimension D of the input tensor $\boldsymbol{v}$, with shape $[B,\ P,\ D]$, independently at each time step.

```
linear_y = Linear(output_dimension=3)(ul)  # 3 output features
```

Given an input tensor of dimensions $[B,\ P,\ D]$, a family of N linear mappings produces an output of size $[B,\ P,\ N]$. An optional bias can be included.

More generally, this block can be used to represent instantaneous couplings and transformations between variables, including coordinate changes, scaling between physical quantities, direct input–output gains and the layers of standard MLPs used for residual terms or black-box neural networks. The parameters of the relation can be interpreted as the elements of a transformation matrix, preserving a direct engineering interpretation of the mapping. As for the `Fir` relation, parameter initialization can also be used to encode prior assumptions on the modeled transformations, improving convergence and preserving physically meaningful behaviors.

4.1.7. `Integrate` *and* `Differentiate`

The `Integrate` and `Differentiate` relations implement the numerical operators introduced in Section 3.3. These blocks provide the computational realization of discrete-time integration and differentiation within structured neural dynamical models. Both can be used inside larger MSNN architectures and are compatible with recurrent rollout. The `Integrate` relation approximates sampled-time integration using the selected sampling time and numerical integration scheme. Standard methods such as Euler and trapezoidal integration are supported, allowing continuous-time structured models to be transformed into executable discrete-time dynamics.

```
# Time integral of yl, method: 'euler' or 'trapezoidal'
int_yl = Integrate(yl, method=integration_method)
```

Complementarily, the `Differentiate` relation implements discrete-time differentiation operators that can be applied either along the temporal dimension or with respect to selected input stream inside differentiable computational graphs.

```
new_yl = Differentiate(int_yl)          # d(int_yl)/dt
dfir_zt_dzt = Differentiate(fir_zt, zt) # d(fir_zt)/d(zt)
```

The initialization of integration and differentiation states is explicitly supported through user-defined initial values provided in the dataset, ensuring consistent behavior during training, recurrent prediction, and deployment.

4.1.8. Elementary Functions and Tensor Operations

In addition to the structured relations described above, `nnodely` provides a set of elementary differentiable operators that can be composed to construct more complex MSNN architectures. These include activation functions, trigonometric and arithmetic operators, as well as tensor manipulation utilities for selecting, reshaping, and concatenating streams across temporal and feature dimensions.

Standard activation functions such as `Relu`, `Tanh`, `Sigmoid`, and custom differentiable activations are supported and can be applied to any relation output. Trigonometric and arithmetic operators allow analytical expressions to be constructed directly within the computational graph, facilitating the implementation of physically motivated relations and custom nonlinear structures.

The framework also provides tensor manipulation operators for selecting and combining signals along temporal or spatial dimensions. These operations are useful when constructing structured architectures involving delayed signals, multi-channel relations, or composite local models.

```
# first sample of [xs, xs], then ReLU
y = (SampleSelect(Concatenate(xs,xs), 0) + Relu(fir_y))
```

These elementary operations constitute the low-level computational primitives used by higher-level blocks such as `LocalModel`, `EquationLearner`.

4.2. Dataset Creation (PH2)

Dataset creation in `nnodely` is designed to ensure consistency between data representation and the temporal structure of

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

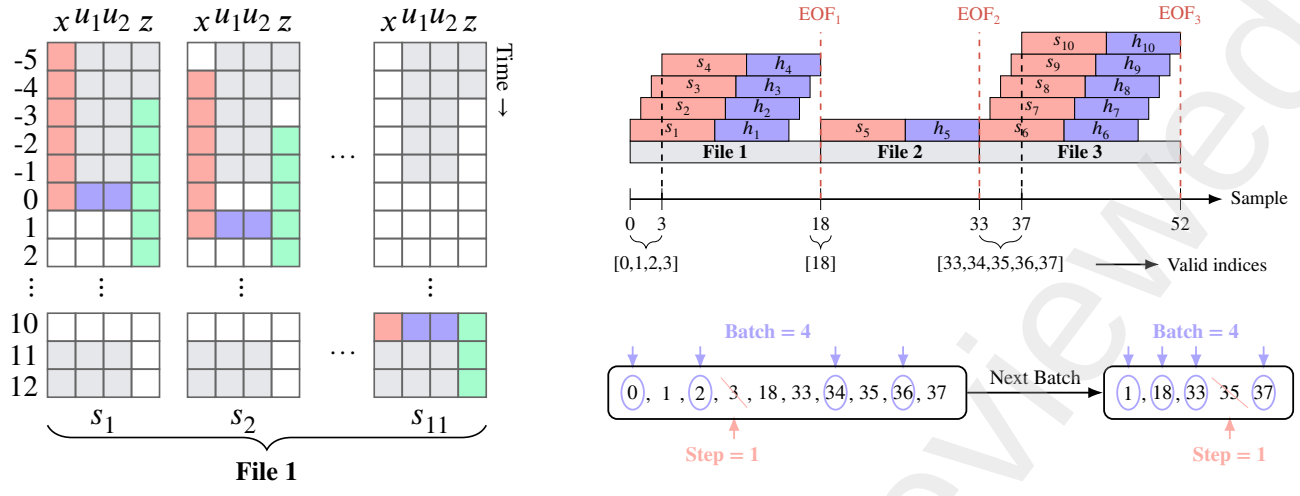


Figure 8: Training data handling in `nnodely`. (a) Temporal window extraction by the data loader: columns denote input variables (x, u (two elements, u_1 and u_2), z as presented in the code Section 4.1) and rows time samples; independent window configurations are applied to each input, with x using six past samples, u the current sample, and z with 4 past and 2 future samples (obtained from the sampling time defined during neuralization). From 18 available samples, only 11 valid windows ($s_1, s_2, \ldots, s_{11}$) can be formed, while gray boxes indicate excluded data. (b) Multi-file data handling during *autoregressive training*: samples s_i and their prediction windows h_i are extracted independently from each file without overlap across files, preserving temporal coherence. For a total temporal window of 8 samples and `prediction_samples` of 7 samples, the three example files containing, respectively 18, 15 and 19 samples, yield 10 valid training indices. Then, temporally coherent Batch (`train_batch_size` in the code) indices are randomly selected at each iteration, and a fixed number of samples Step (`step` in the code) are discarded before the next selection; when shuffling is disabled, Batch and the Step indices are taken sequentially.

the neural model (PH2 in Fig. 7). The framework supports multiple data sources, including external files, pandas data frames, and Python dictionaries, and provides utilities for resampling and filtering.

```
nn.loadData(name = 'data',            # dataset name
            source = 'folder',        # CSV folder
            format = ['x', 'u', 'z']) # column order
```

When using external data, `nnodely` supports CSV files, including multi-file datasets in which each file represents an independent simulation with consistent temporal ordering. When loading multiple files from a directory, temporal windows of length P are constructed while preventing windows from spanning across different files (see Fig. 8a). The resulting dataset can be used for both feedforward and autoregressive training. In the autoregressive setting, temporal consistency is preserved not only within the window dimension P, but also across prediction samples, when present, as illustrated in Fig. 8b.

To accommodate different experimental settings, the framework includes utilities for resampling (e.g., via interpolation) and filtering for extracting specific temporal intervals, enabling controlled evaluations on selected windows or scenarios.

4.3. Neural Model Training (PH3)

Training in `nnodely` (PH3 in Fig. 7) identifies model parameters by minimizing the discrepancy between predicted and target outputs under user-defined loss functions and metrics. The predicted and target variables must have compatible dimensions, `dim`, and temporal windows `sw` or `tw`.

```
nn.addMinimize('z_error', y_out, zt)  # loss: y_out vs zt
```

The framework supports standard gradient-based optimization and multi-objective losses through optional weighting, together with regularization mechanisms such as early stopping.

```
params = {'num_of_epochs': 100,
          'train_batch_size': 4,
          'lr': 0.001}            # hyperparameters
nn.trainModel(splits = [100, 0, 0],   # train/val/test [%]
              early_stopping = ...,   # early stopping
              minimize_gain = ...,    # loss weights
              prediction_samples = 7, # horizon
              step = 1,               # step
              training_params = params)
```

The code snippet corresponds to Figure 8b. In the example, all the data are used for training.

Two complementary training modalities are supported. *Feed-forward training* follows the conventional mini-batch paradigm, in which samples are shuffled and propagated through an acyclic computation graph to update the model parameters based on the instantaneous loss. This modality is

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

suited to architectures without internal feedback, where predictions depend only on the provided inputs. *Autoregressive training* targets architectures with closed-loop dependencies, such as feedback connections or coupled multi-model structures. In this case, training is performed over a finite prediction horizon, and parameter updates are applied only after the full recurrent rollout has been completed. Since the loss is evaluated over complete prediction horizons, batching is performed at the level of temporal sequences: the input and target windows required for each rollout are kept intact within the batch, rather than shuffling individual time instants independently. See Section 4.6 for the implementation of closed-loop dependencies.

4.4. Neural Model Validation (PH4)

Model validation in `nnodely` (PH4 in Fig. 7) follows established practices in mechanical and control engineering, combining predictive accuracy assessment with explicit consideration of model complexity. To this end, the framework supports metrics such as the Akaike Information Criterion (AIC) and the Fraction of Variance Unexplained (FVU). AIC balances goodness of fit against model complexity, while FVU quantifies the portion of output variance not explained by the model, providing an inverse measure of predictive performance. Validation is typically performed on a dedicated validation set and can be integrated into the training process for continuous monitoring, while final performance assessment is carried out on an independent test set.

```
nn.analyzeModel(minimize_gain = ...,    # loss weights
                prediction_samples = 7, # horizon
                step = 1)               # step
```

During training and validation, `nnodely` provides textual and graphical feedback through configurable visualization backends, enabling inspection of convergence behavior, loss evolution, and model fit quality. Validation results can be exported as a structured PDF report, including loss trends, learned fuzzy and parametric functions, and additional diagnostic metrics, supporting systematic post-analysis and documentation.

4.5. Network Inference and Export (PH5)

After training and validation, neural models defined in `nnodely` can be used for inference and exported for integration into external software pipelines (PH5 in Fig. 7). During inference, the model receives input tensors associated with the predefined *Inputs* and returns the selected *Outputs* specified in the network structure. Inference can be performed either through a single forward pass or autoregressively over a temporal horizon, which is particularly useful for multi-step prediction and simulation of dynamical systems. In the autoregressive setting, the required temporal windows are automatically generated and updated during the rollout procedure.

```
# one-step inference
nn({'x': [-5,-4,-3,-2,-1,0], 'u': [[2,2]],
'z': [-3,-2,-1,0,1,2]})
# recurrent inference (2 steps)
nn({'x': [-5,-4,-3,-2,-1,0,1], 'u': [[2,2],[4,4]],
    'z': [-3,-2,-1,0,1,2,3]}, prediction_samples = 2)
```

The complete model can be serialized into a framework-independent representation, which encodes the hierarchical organization of inputs, relations, outputs, parameters, and temporal dependencies. This representation supports reproducibility, portability, and long-term reusability, while decoupling the model specification from a specific execution backend.

For efficient execution, `nnodely` models can also be translated into optimized PyTorch classes through symbolic tracing. This produces a system-specific implementation that can be used for fine-tuning, standalone inference, or integration into existing deep learning pipelines. In addition, export to the `ONNX` format is supported as an engineering convenience for deployment across heterogeneous platforms and inference engines. Although ONNX export is natively supported by PyTorch, not all PyTorch operations and dynamic graph features are directly exportable. For this reason, `nnodely` explicitly constructs export-compatible inference models, using ONNX-compatible operations whenever possible and generating dedicated representations for autoregressive inference, temporal-window handling, and closed-loop connections.

All export modalities can be applied either to the complete model or to selected *sub-models* (defined in Section 4.6), including autoregressive architectures. In the case of autoregressive inference, the exported representation preserves temporal-window handling and closed-loop connections by explicitly accounting for the prediction-horizon dimension. This allows the exported model to reproduce the same inference behavior defined within `nnodely`, including multi-step rollouts and time-windowed signal dependencies.

4.6. Neural Model Composition (PH6)

Building on the definition introduced in Section 4.1, `nnodely` supports the composition of multiple *sub-models* within the same framework and consistently across all development phases (PH6 in Fig. 7). Each sub-model can represent a complete MSNN, an NCM-based component, a controller, or an auxiliary neural block, and several sub-models can be interconnected to form a larger structured architecture. Since sub-models are defined independently, they can also be selectively optimized: some components can be trained while others are kept frozen. This enables staged identification, controlled refinement of complex architectures, and controller training over fixed or partially fixed dynamics.

Connections can be defined at different levels: within a block by operating on streams, between previously defined models, or dynamically at training or inference time for temporary testing purposes. The three connection modalities are illustrated below.

Preprint not peer reviewed

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Multiple sub-models can be combined through the `Connect` operation, which allows outputs, intermediate relations, or state-dependent quantities to be routed across different components of the architecture. This enables the construction of cascaded, interacting, or hierarchical structured neural models.

```
y.connect(x)                  # stream connection: y -> x
nn.addConnect('y','x')        # model connection: y -> x
nn(..., connect = {'y':'x'}) # runtime connection: y -> x
```

Autoregressive dynamical behaviors are implemented through the `ClosedLoop` mechanism, which feeds selected outputs back into subsequent prediction steps. This enables multi-step inference, rollout-based training, and time-marching simulation while preserving the same structured model representation used for one-step prediction.

```
y.closedLoop(x)                   # stream feedback: y -> x
nn.addClosedLoop('y','x')         # model feedback: y -> x
nn(..., closed_loop = {'y':'x'}) # runtime feedback: y -> x
```

Within the framework, the `Integrate` and `Differentiate` relations introduced in Section 4.1.7 are internally constructed using combinations of `Connect` and `ClosedLoop` operations. In this way, temporal integration and differentiation are realized through explicit recurrent state propagation mechanisms consistent with the same compositional modeling framework.

The composition mechanisms are also preserved during export, allowing individual sub-models or complete autoregressive architectures to be deployed independently while maintaining the original temporal and structural dependencies defined during model construction.

4.7. Future Extensions

The current implementation of `nnodely` is based on a tensor representation of the form $[B,\ P,\ D]$, where B denotes the batch dimension, P the temporal dimension, and D the feature dimension. Future releases will extend this abstraction to support more general tensor layouts, including autoregressive and sequential structures of the form $[B, P, S_1, \dots, S_m]$, where $S_1, \dots, S_m$ denote additional sequence dimensions. This extension will enable the native integration of autoregressive and sequence-based neural architectures within the same structured modeling framework.

In addition, the feature dimension D will be generalized to support higher-dimensional tensor structures $D_1, \dots, D_n$, allowing the direct representation of matrix- and tensor-valued streams. In its most general form, the framework will support the unified layout $[B, D_1, \dots, D_n, P, S_1, \dots, S_m]$ for temporal, sequential, and high-dimensional structured data.

From the software perspective, future versions of `nnodely` will also introduce a new backbone abstraction based on Keras 3.0. This extension will provide native compatibility with multiple execution backends, including PyTorch, TensorFlow, and JAX, while preserving the same high-level compositional modeling interface. In this way, model specification, training, and deployment will remain backend-independent, further improving interoperability and long-term portability of structured neural models.

5. Case Studies

This section validates the `nnodely` framework through a set of representative case studies covering simulated and real-world physical systems. The examples include rigid-body dynamics learned from simulation, friction modeling for a six dof robotic manipulator, and lateral vehicle dynamics estimation from real on-track telemetry data.

Together, these studies span the modeling spectrum, from fully physics-based to hybrid and data-driven structured architectures, and assess the framework's ability to embed physical priors, handle heterogeneous data, and deliver accurate, interpretable models for control-oriented applications.

All experiments are implemented using the same modeling, training, and validation pipeline, with parameter specifications reported in Table 1. The full source code, configuration files, and scripts associated with the reported results, together with additional examples, are publicly available at `https://github.com/tonegas/nnodely-applications`. An archived version of the case-study repository, compatible with `nnodely v1.5.4`, is available on Zenodo.

5.1. Dynamics Learning of Inverted Pendulum on a Cart

This case study evaluates how different network designs capture the dynamics of an inverted pendulum on a cart (IPC) within the MSNN implemented in the `nnodely` framework.

Physics Priors. The IPC consists of a cart of mass M moving along a horizontal axis and a pendulum of mass m and length $2l$ mounted on the cart. The system state is $\boldsymbol{x} = [\mathbf{q}, \dot{\mathbf{q}}]$, with $\mathbf{q} = [p, \theta]$ representing the position of the cart and the angle of the rod, respectively. An external force F is applied to the cart. The system is modeled as follows:

$$\begin{cases} (m+M)\ddot{p} + ml\ddot{\theta}\cos(\theta) + b\dot{p} - ml\dot{\theta}^2\sin(\theta) = F, \\ ml\ddot{p}\cos(\theta) + (I+ml^2)\ddot{\theta} - mgl\sin(\theta) + d\dot{\theta} = 0. \end{cases} \tag{31}$$

Neural Models. The following models are evaluated. (i) The *Physics-Based Model* (**PBM**) exploits the known physical structure of the system while considering all physical parameters as trainable variables, thereby framing the problem as parameter identification. Two variants are considered: the *Physics-Based Model Powell* ($\mathbf{PBM}_P$), where parameters are identified using the derivative-free Powell optimization method [58] available in the SciPy library [59], and the *Physics-Based Model Gradient* ($\mathbf{PBM}_G$), where parameters are optimized through gradient-based backpropagation. (ii) The *Partially Neuralized Model* (**PNM**) merges identifiable parameter groupings wherever possible, while isolating nonlinear terms as trainable weights, thereby increasing model

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Example	Net	Param.	Ep.	Batch	Pred. (s)	lr
IPC	**PBM**$_G$ (ours)	7	50*	64	1	10^{-3}
	PNM (ours)	12	50*			
	PANN (ours)	187	129*			
	LFIR (ours)	56	66*			
	MLP (benchmark)	563	75*			
	PBM$_P$ (benchmark)	7	-	-		-
DIPC	**EQL** (ours)	571	274*	256	0.01	10^{-3}
	LM (ours)	375	738*			
	MLP (benchmark)	168	158*			
Z1	**MSNN** (ours)	256	143*	128	1	10^{-3}
	LSTM (benchmark)	92550	103*			
	PBM (benchmark)	24	-	-		-
F250LM	**MSNN** (ours)	110	60	256	25	10^{-3}
	MLP (benchmark)	299	500			
	PBM (benchmark)	3	-	-		-

Table 1
Model complexity and training parameters used in the reported case studies. **IPC** denotes the Inverted Pendulum on a Cart, **DIPC** the Double Inverted Pendulum on a Cart, **Z1** the Unitree Z1 manipulator, and **F250LM** the Ferrari 250 LM vehicle. The compared models include the Physics-Based Model (**PBM**), Partially Neuralized Model (**PNM**), Physics-Augmented Neural Network (**PANN**), Learned FIR Filter model (**LFIR**), Equation Learner model (**EQL**), Local model (**LM**), Model-Structured Neural Network (**MSNN**), Long Short-Term Memory network (**LSTM**), Multi-Layer Perceptron (**MLP**). **Param.** denotes the number of learnable parameters, **Ep.** the number of training epochs, **Batch** the training batch size, **Pred. (s)** the prediction horizon used during autoregressive training, and **lr** the learning rate. All neural models are trained using the Adam optimizer. The symbol (*) indicates the effective number of epochs resulting from early stopping. The Regression baseline is fitted through unconstrained optimization and therefore has no associated training epochs.

flexibility during training while preserving the known nonlinear structure of the system. The two preceding models are based on NCM design using only an input function (see Section 3.2.2). (iii) The *Physics-Augmented Neural Network* (**PANN**) combines the analytical physical formulation with a residual black-box component. Here, the input function is complemented by the residual block introduced in Section 3.2.4. The resulting prediction can be thought of as $\boldsymbol{x}[k+1] = f(\boldsymbol{x}[k], F[k]) + \mathcal{R}(\boldsymbol{x}[k], F[k])$ where $\mathcal{R}(\cdot)$ is the learned residual correction. (iv) The *Learned FIR Filter Model* (**LFIR**) exploits a time window of past inputs, rather than relying solely on the most recent measurement, to mitigate the effects of measurement noise, as described in Section 3.2.3. (v) The **MLP** model was chosen using a dedicated library called `optuna` [60], an automated tool for searching for optimal hyperparameters, such as the number of layers and nodes in the MLP. This MSNN comprises several NCMs. Some of them rely only on the output function to filter the input signals, while the others are used to represent the nonlinear model and the residual component, as shown in Fig. 9.

Dataset. Each dataset consists of 150 simulations of the IPC system conducted in MuJoCo, each lasting 3 s and driven by randomly generated input actions. The simulation procedure is independently repeated three times to generate the training, validation, and test datasets. The implementation and scripts for this case study are available in the corresponding `nnodely-applications` directory `inverted_pendulum_on_a_cart`.

Training. Models are first trained for one-step prediction and subsequently fine-tuned through recurrent training over a 1 s horizon (50 samples), minimizing the sum of the MSE losses between the predicted and true states. To evaluate robustness, two simulation conditions are considered: a noise-free setting and a noisy setting. In the noisy condition, zero-mean Gaussian noise is added during training to both state measurements and control inputs to model sensing and actuation uncertainties. Lower noise variance is applied to position and angle measurements, while higher variance is assigned to velocity-related states to account for the effects of numerical differentiation. Input noise is scaled as a small fraction of the admissible control range, representing actuator imperfections. The standard deviation of the injected noise is set to $\sigma = [0.01, 0.01, 0.02, 0.1]$ for the state variables $(q, \dot{q})$ and to $\sigma = 0.06$ for the actuator input.

Results. Table 2 reports the state estimation performance of the considered models under noise-free and noisy conditions. All values are obtained using the `analyzeModel` function in `nnodely`, as described in Section 4.4. In the absence of noise, all models achieve low and comparable estimation errors. In the presence of noise, the LFIR model is the best performing, as expected, due to its effective handling of noisy inputs. These results are qualitatively consistent with previous studies on IPC modeling and identification based on Volterra series [61] and artificial neural networks [62], with the proposed models achieving comparable accuracy while providing a unified framework that integrates physical insight with learning-based components.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

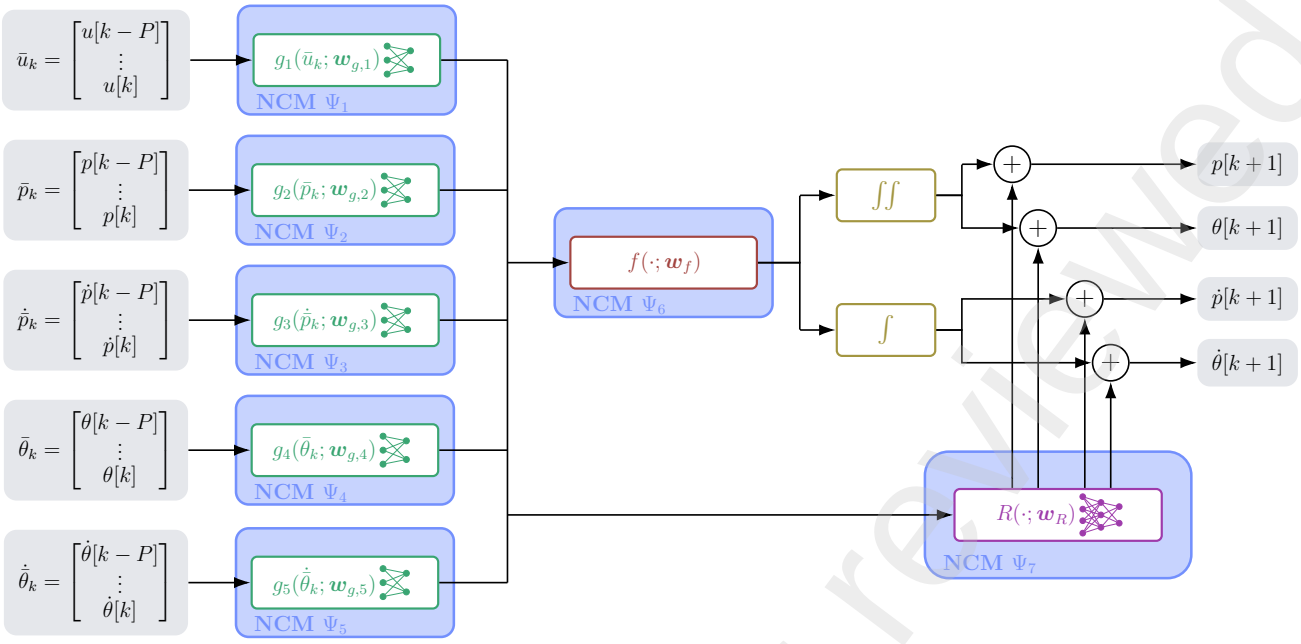


Figure 9: Architecture of the MSNN for the inverted pendulum on a cart (IPC). The network receives the finite-memory inputs $\bar{u}_k$, $\bar{p}_k$, $\dot{\bar{p}}_k$, $\bar{\theta}_k$, and $\dot{\bar{\theta}}_k$. NCMs Ψ_1–Ψ_5 implement output functions $g_i(\cdot; \boldsymbol{w}_{g,i})$ as FIR filters acting on the temporal history of each input. NCM Ψ_6 implements the input function $f(\cdot; \boldsymbol{w}_f)$ as a `ParametricFunction` encoding the known nonlinear IPC dynamics; its output is integrated, using a fourth-order Runge–Kutta scheme, to obtain the next state. NCM Ψ_7 implements a residual network $R(\cdot; \boldsymbol{w}_R)$ as a multilayer perceptron, whose state-wise corrections are added to the physics-based prediction. The resulting outputs are $p[k+1]$, $\theta[k+1]$, $\dot{p}[k+1]$, and $\dot{\theta}[k+1]$. The **PBM** uses only Ψ_6, with the physical parameters collected in $\boldsymbol{w}_f$ as learnable quantities. The **PNM** uses the same architecture but adopts a partially neuralized parameterization of $f(\cdot; \boldsymbol{w}_f)$, in which independent learnable coefficients represent identifiable parameter combinations and nonlinear terms. The **PANN** complements Ψ_6 with Ψ_7. The **LFIR** additionally introduces Ψ_1–Ψ_5.

5.2. Dynamics Learning of Double Inverted Pendulum on a Cart

This case study evaluates three strategies for learning the dynamics of a double inverted pendulum on a cart (DIPC).

Physics Priors. The DIPC consists of a cart of mass m that moves along a horizontal axis, with a first pole of mass m_1, length l_1, and inertia I_1 attached to the cart, and a second pole of mass m_2, length l_2, and inertia I_2 connected to the other end of the first pole. $\boldsymbol{q} = [p, \theta_1, \theta_2]^T$ collects the cart position and the two pole angles, while $\dot{\boldsymbol{q}} = [v, \omega_1, \omega_2]^T$ contains the corresponding velocities.

Neural Models. The following models are evaluated.

(i) The *Black-Box Model* (**MLP**) is adopted as a purely data-driven baseline. The network receives the current state and control input and predicts the state variation or, equivalently, the state at the following time instant. No prior knowledge of the mechanical structure is embedded in the model. Consequently, the MLP must learn the inertial, gravitational, and velocity-dependent effects directly from the available data.

(ii) The *Local Model* (**LM**) learns the same dynamics using the formulation introduced in Section 3.2.1. Rather than explicitly identifying the complete symbolic form of the governing equations, the LM approximates the nonlinear dynamics through a set of locally valid affine models combined by fuzzy membership functions. In the present case, the operating space is partitioned using the angular coordinates as scheduling variables, $\mathbf{z} = [\theta_1, \theta_2]^T \in \mathbb{R}^{n_z}$ with $n_z = 2$. The corresponding local activations are constructed as separable products of one-dimensional membership functions:

$$\phi_s(\theta_1, \theta_2) = \phi_{s_1,1}(\theta_1)\phi_{s_2,2}(\theta_2)$$

According to the *Locally-Linearized Functions* introduced in Section 3.2.5, each local contribution can be obtained from a Taylor approximation around an operating point. After collecting the constant, state-dependent, and input-dependent terms, each local model can be expressed in affine form. In this case, the regressor vector is $\varphi = [u, v, \omega_1, \omega_2]^T$. Thus, for each angular region (i, j), the learned local model for one acceleration component is written as

$$\tilde{f}_{ij}(\varphi) = b_{ij} + a_{ij,u}u + a_{ij,v}v + a_{ij,\omega_1}\omega_1 + a_{ij,\omega_2}\omega_2 \quad (32)$$

The fuzzy model for the r-th acceleration component is therefore

$$\hat{y}_r = \sum_{i,j} \phi_{ij}(\theta_1, \theta_2) \left(b_{ij,r} + a_{ij,u,r}u + a_{ij,v,r}v + a_{ij,\omega_1,r}\omega_1 + a_{ij,\omega_2,r}\omega_2\right). \quad (33)$$

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Model	q	RMSE on q		RMSE on $\dot{q}$		Total RMSE	
		best	std dev	best	std dev	best	std dev
1 s recurrent state estimation without added noise							
MLP (benchmark)	p (m)	1.00×10^{-2}	1.20×10^{-3}	2.65×10^{-2}	2.81×10^{-3}	8.24×10^{-2}	4.05×10^{-3}
	θ (rad)	4.79×10^{-2}	2.05×10^{-3}	1.55×10^{-1}	8.00×10^{-3}		
PBM$_P$ (benchmark)	p (m)	7.27×10^{-2}	3.63×10^{-2}	1.59×10^{-1}	7.78×10^{-2}	8.92×10^{-2}	3.78×10^{-2}
	θ (rad)	1.98×10^{-2}	4.93×10^{-3}	1.03×10^{-1}	3.36×10^{-2}		
PBM$_G$ (ours)	p (m)	$\mathbf{4.67 \times 10^{-4}}$	8.87×10^{-5}	2.45×10^{-3}	1.18×10^{-4}	$\mathbf{7.02 \times 10^{-3}}$	3.11×10^{-5}
	θ (rad)	5.57×10^{-3}	2.24×10^{-5}	1.96×10^{-2}	7.34×10^{-5}		
PNM (ours)	p (m)	4.85×10^{-4}	$\mathbf{1.26 \times 10^{-5}}$	2.46×10^{-3}	$\mathbf{1.17 \times 10^{-5}}$	7.06×10^{-3}	$\mathbf{1.75 \times 10^{-5}}$
	θ (rad)	5.59×10^{-3}	$\mathbf{1.03 \times 10^{-5}}$	1.97×10^{-2}	$\mathbf{3.07 \times 10^{-5}}$		
PANN (ours)	p (m)	9.28×10^{-4}	3.12×10^{-4}	2.78×10^{-3}	1.39×10^{-3}	7.18×10^{-3}	6.28×10^{-4}
	θ (rad)	$\mathbf{5.50 \times 10^{-3}}$	3.29×10^{-4}	1.95×10^{-2}	1.03×10^{-3}		
LFIR (ours)	p (m)	2.17×10^{-3}	1.87×10^{-3}	$\mathbf{2.13 \times 10^{-3}}$	5.15×10^{-3}	9.70×10^{-3}	2.80×10^{-3}
	θ (rad)	1.65×10^{-2}	2.86×10^{-4}	$\mathbf{1.80 \times 10^{-2}}$	5.79×10^{-3}		
1 s recurrent state estimation with added noise							
MLP (benchmark)	p (m)	1.03×10^{-1}	1.19×10^{-3}	2.71×10^{-1}	2.81×10^{-3}	2.72×10^{-1}	3.67×10^{-3}
	θ (rad)	1.57×10^{-1}	1.89×10^{-3}	5.57×10^{-1}	7.22×10^{-3}		
PBM$_P$ (benchmark)	p (m)	1.30×10^{-1}	1.30×10^{-2}	2.63×10^{-1}	3.23×10^{-2}	2.81×10^{-1}	5.37×10^{-3}
	θ (rad)	1.62×10^{-1}	2.78×10^{-5}	5.69×10^{-1}	3.88×10^{-4}		
PBM$_G$ (ours)	p (m)	$\mathbf{6.85 \times 10^{-2}}$	2.36×10^{-5}	$\mathbf{1.26 \times 10^{-1}}$	9.92×10^{-5}	2.24×10^{-1}	$\mathbf{1.54 \times 10^{-5}}$
	θ (rad)	1.58×10^{-1}	1.27×10^{-5}	5.44×10^{-1}	$\mathbf{2.30 \times 10^{-5}}$		
PNM (ours)	p (m)	6.86×10^{-2}	$\mathbf{5.90 \times 10^{-6}}$	1.27×10^{-1}	$\mathbf{1.57 \times 10^{-5}}$	2.24×10^{-1}	$\mathbf{1.54 \times 10^{-5}}$
	θ (rad)	1.58×10^{-1}	$\mathbf{3.16 \times 10^{-6}}$	5.44×10^{-1}	$\mathbf{2.30 \times 10^{-5}}$		
PANN (ours)	p (m)	$\mathbf{6.85 \times 10^{-2}}$	5.71×10^{-5}	1.27×10^{-1}	1.93×10^{-4}	2.24×10^{-1}	2.47×10^{-4}
	θ (rad)	1.57×10^{-1}	1.56×10^{-4}	5.44×10^{-1}	4.86×10^{-4}		
LFIR (ours)	p (m)	8.29×10^{-2}	2.35×10^{-3}	1.52×10^{-1}	3.71×10^{-3}	$\mathbf{1.74 \times 10^{-1}}$	1.87×10^{-3}
	θ (rad)	$\mathbf{9.96 \times 10^{-2}}$	2.37×10^{-3}	$\mathbf{3.61 \times 10^{-1}}$	5.42×10^{-3}		

Table 2
Prediction performance (**RMSE**) for the inverted pendulum on a cart (IPC). The table reports the RMSE on the state $\boldsymbol{q} = [p, \theta]^\mathsf{T}$ and on its derivative $\dot{\boldsymbol{q}}$, together with the Total RMSE, for 1 s recurrent prediction on the test set, under noise-free and noisy conditions. The compared models are the *Multi-Layer Perceptron* (**MLP**), the *Physics-Based Powell Model* (**PBM**$_P$), the *Physics-Based Gradient Model* (**PBM**$_G$), the *Partially Neuralized Model* (**PNM**), the *Physics-Augmented Neural Network* (**PANN**), and the *Learned FIR Filter Model* (**LFIR**). We report the best value and the standard deviation. All models achieve comparable accuracy in the noise-free case, while the LFIR model performs best under noisy conditions thanks to the FIR filters, which effectively handle noisy inputs. Best results are highlighted in boldface.

By collecting the coefficients associated with each regressor, the same model can be expressed as

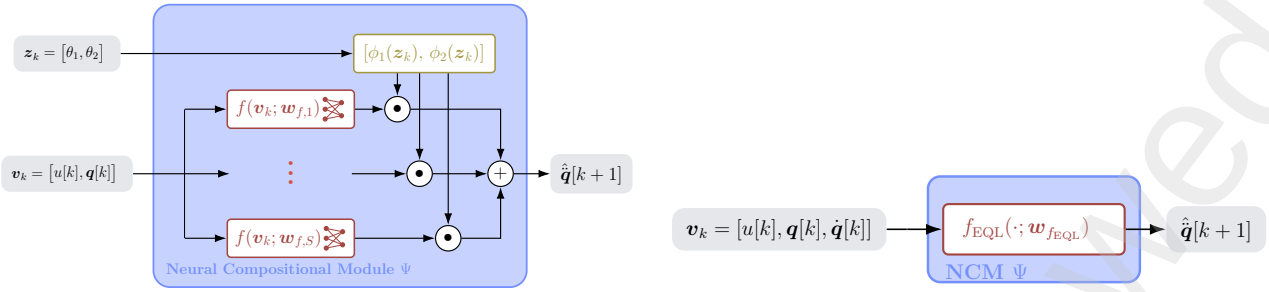
$$\begin{aligned}\hat{y}_r = w_{0,r}(\theta_1, \theta_2) + w_{u,r}(\theta_1, \theta_2)u + w_{v,r}(\theta_1, \theta_2)v \\ + w_{\omega_1,r}(\theta_1, \theta_2)\omega_1 + w_{\omega_2,r}(\theta_1, \theta_2)\omega_2.\end{aligned} \tag{34}$$

Since the local affine model includes only the linear regressors ω_1 and ω_2, quadratic and mixed velocity-dependent terms, such as $\omega_1^2(\cdot)$, $\omega_2^2(\cdot)$, and $\omega_1\omega_2(\cdot)$, are not explicitly represented in the approximation basis. In the considered case, the system operates in a low-velocity regime, where quadratic and mixed velocity contributions are small compared with the input-driven and configuration-dependent terms. These effects should therefore be regarded as neglected, or only weakly absorbed into the residual approximation error, rather than explicitly learned by the model. To explicitly represent these effects within the local decomposition, the velocities $[v, \omega_1, \omega_2]$ could also be included among the scheduling variables used by the fuzzy membership functions.

(iii) The *Equation Learner* (**EQL**) is based on the operator introduced in Section 3.2.2. This approach constructs a nonlinear mapping by composing affine learnable transformations with a predefined library of elementary symbolic operators. The input vector $\boldsymbol{v}_k = [u[k], \boldsymbol{q}[k], \dot{\boldsymbol{q}}[k]]$ where $\boldsymbol{q}[k] = [p[k], \theta_1[k], \theta_2[k]]^T$, and estimates the three accelerations $\hat{\dot{\boldsymbol{q}}}[k] = [\hat{\dot{p}}[k], \hat{\dot{\theta}}_1[k], \hat{\dot{\theta}}_2[k]]^T$. Each equation-learning block applies an affine transformation to its input, routes the resulting features through a structured library of nonlinear functions, and combines the generated candidate terms through a further learnable linear map, as described in Eq. (18).

Dataset. The dataset consists of 500 simulations of the DIPC system conducted in MuJoCo, each lasting 3 seconds and driven by randomly generated input actions. The simulation procedure is repeated independently three times to generate training, validation, and test datasets. The implementation and scripts associated with this case study are available in the corresponding `nnodely-applications` directory `double_inverted_pendulum_on_cart`.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813



(a) Local Model (b) Equation Learner

Figure 10: Architecture of the MSNNs for the double inverted pendulum on a cart (DIPC). (a) Local Model. The scheduling vector $z_k = [\theta_1[k], \theta_2[k]]^\mathsf{T}$ is processed by five triangular membership functions per angular coordinate, producing 25 activation coefficients $\phi_s(z_k)$. Each local affine input function $f(v_k; w_{f,s})$ in NCM Ψ receives the regressor $v_k = [u[k], v[k], \omega_1[k], \omega_2[k]]^\mathsf{T}$; the local predictions are weighted by the corresponding activations and summed to estimate one acceleration component. Three independent instances of this module estimate $\hat{\hat{p}}[k]$, $\hat{\hat{\theta}}_1[k]$, and $\hat{\hat{\theta}}_2[k]$. (b) Equation Learner. The input $v_k = [u[k], q[k], \dot{q}[k]]$ is processed by NCM Ψ, implemented as four cascaded `EquationLearner` stages with input function $f_{\text{EQL}}(\cdot; w_{f_{\text{EQL}}})$, to estimate $\hat{\hat{q}}[k+1]$.

Model	q	RMSE on $\ddot{q}$		Total RMSE	
		best	std dev	best	std dev
LM (ours)	p (m)	$\mathbf{1.65 \times 10^{-3}}$	$\mathbf{1.37 \times 10^{-3}}$	$\mathbf{7.50 \times 10^{-3}}$	$\mathbf{6.53 \times 10^{-3}}$
	θ_1 (rad)	$\mathbf{8.93 \times 10^{-3}}$	$\mathbf{7.62 \times 10^{-3}}$		
	θ_2 (rad)	$\mathbf{9.30 \times 10^{-3}}$	$\mathbf{8.25 \times 10^{-3}}$		
EQL (ours)	p (m)	1.51×10^{-2}	4.35×10^{-2}	3.13×10^{-2}	11.30×10^{-2}
	θ_1 (rad)	3.07×10^{-2}	14.54×10^{-2}		
	θ_2 (rad)	4.20×10^{-2}	12.95×10^{-2}		
MLP (benchmark)	p (m)	2.96×10^{-2}	3.56×10^{-3}	3.21×10^{-2}	2.80×10^{-3}
	θ_1 (rad)	3.51×10^{-2}	5.73×10^{-3}		
	θ_2 (rad)	3.13×10^{-2}	2.57×10^{-3}		

Table 3
Prediction performance (**RMSE**) for the double inverted pendulum on a cart (DIPC). The table reports the RMSE on the predicted accelerations $\ddot{q} = [\ddot{p}, \ddot{\theta}_1, \ddot{\theta}_2]^\mathsf{T}$, together with the Total RMSE, on the test set. The compared models are the *Local Model* (**LM**), the *Equation Learner* (**EQL**), and the *Multi-Layer Perceptron* (**MLP**). We report the best value and the standard deviation. The LM achieves the lowest RMSE among the three approaches. Best results are highlighted in boldface.

Training. All models are trained by minimizing the MSE of the three predicted and measured accelerations using Adam with a learning rate of 10^{-3} and a batch size of 256. Training is performed for up to 1000 epochs, with early stopping after 20 epochs.

Results. Table 3 reports the state estimation performance of the considered models. Overall, the results show that the MSNN based on the *Local Model* component achieves the best prediction performance among the three approaches. The comparison highlights the trade-off among predictive flexibility, interpretability, and specialization over the operational space. The **MLP** provides a general-purpose black-box approximation but does not expose the physical relationships learned from the data. The **EQL** provides a more interpretable representation by identifying combinations of nonlinear terms associated with the system dynamics. Conversely, the **LM** addresses the complexity of the global dynamics by decomposing the operating space into simpler local regions, making it particularly suitable when the available data cover multiple distinct dynamic regimes.

5.3. Simulation-Based Friction Learning of a Unitree Z1 Model

This case study demonstrates how external differentiable software can be integrated within an MSNN. The objective is learning the joint friction of a Unitree Z1 6-DOF manipulator. This work focuses on a simulation-based study of friction learning in a Unitree Z1 model. All simulations are conducted in a controlled MuJoCo environment, where friction parameters are modified while the rigid-body dynamics remain fixed.

Physics Priors. The manipulator dynamics are described by $\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{f}(\dot{\boldsymbol{q}}) = \boldsymbol{\tau}_a$, where $\boldsymbol{\tau}_a$ denotes the measured joint torques and $\boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ represents the unknown friction vector. The friction in this case is modeled with viscous, Coulomb and Stribeck effects.

$$\boldsymbol{f}_{cv}(\dot{\boldsymbol{q}}) = K_{v,s}\dot{\boldsymbol{q}} + K_{c,s}\,\mathrm{sign}(\dot{\boldsymbol{q}}), \tag{35}$$

$$\boldsymbol{f}_{st}(\dot{\boldsymbol{q}}) = \mathrm{sign}(\dot{\boldsymbol{q}})\left(K_{st,s} - K_{c,s}\right) e^{-\left(\dot{\boldsymbol{q}}/\dot{q}_{st,s}\right)^2}, \tag{36}$$

where the parameters $K_{v,s}$, $K_{c,s}$, $K_{st,s}$, $\dot{q}_{st,s}$ are the friction parameters.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

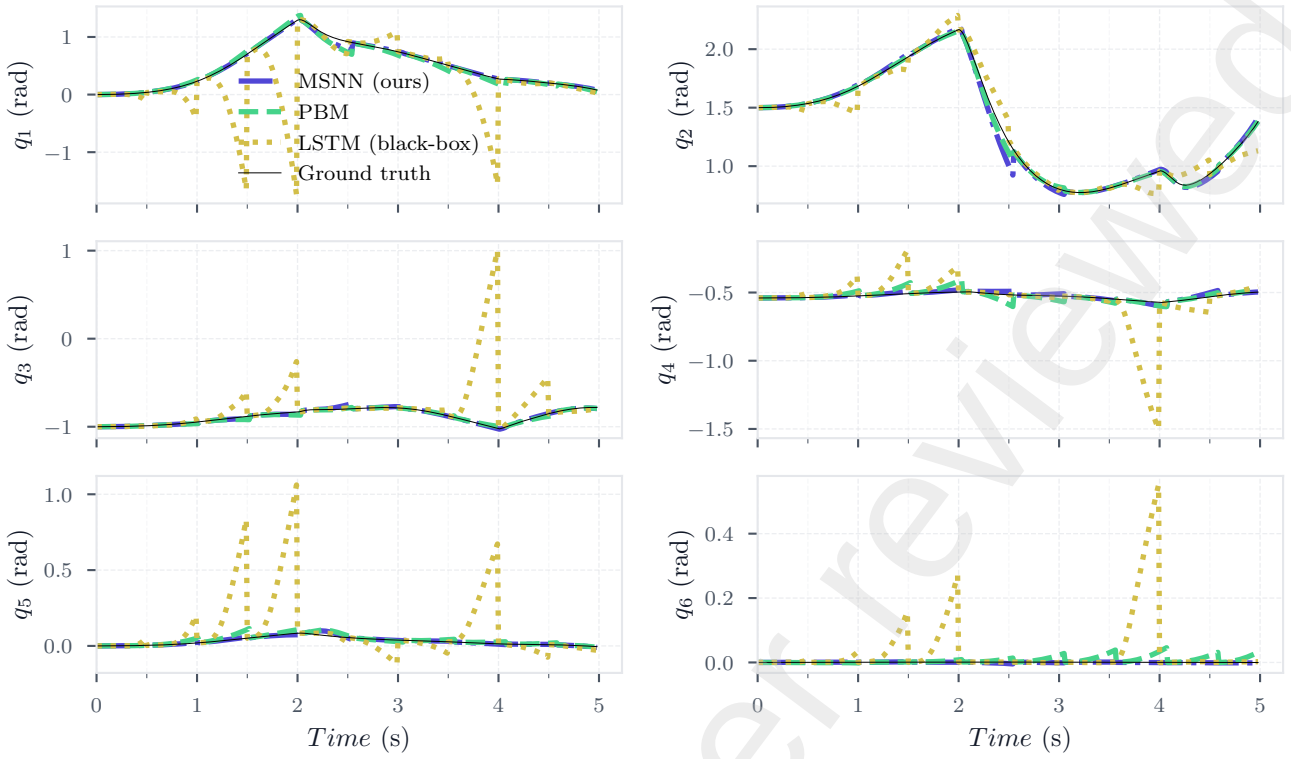


Figure 11: Comparison of joint position tracking accuracy for the Z1 manipulator over a $0.5s$ receding horizon. The **MSNN** achieves the lowest overall RMSE and outperforms the baselines on most joints for both the **LSTM** and the **PBM** baseline, particularly in the later joints (q_4, q_6), where it better captures the nonlinear dynamics and maintains stability over multiple prediction steps. The **LSTM** model exhibits significant drift, while the **PBM** model shows intermediate performance but fails to track rapid variations.

Neural Models. The following models are compared. (i) The *Model-Structured Neural Network* (**MSNN**), which uses local models for learning friction parameters. (ii) The *Black-Box* (**LSTM**), a Long Short-Term Memory network for learning the friction model. (iii) The *Physics-Based Model* (**PBM**), which performs classical system identification to retrieve the friction parameters.

The **MSNN** model uses two different Neural Compositional Modules (NCMs) for the task: one is responsible for learning the friction, and the other uses the library ADAM for solving the forward dynamics. ADAM[2] is a differentiable robotics library that computes all dynamic quantities directly from the robot URDF. Consequently, the inertia matrix $\boldsymbol{M}(\boldsymbol{q})$ and the bias force vector $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ are assumed to be known, whereas only the friction model is learned.

The first NCM consists of six independent `LocalModel` (Section 4.1.2), one for each joint. Since all variables refer to the discrete time instant k, the time index $[k]$ is omitted throughout this section for the sake of brevity. For each joint, the local friction model combines viscous, Coulomb, and Stribeck friction effects. The resulting input function (Section 3.2.2) of the NCM is the following

$$\boldsymbol{f}_{cv}(\dot{\boldsymbol{q}}; \boldsymbol{w}_{cv,s}) = K_{v,s}\dot{\boldsymbol{q}} + K_{c,s}\,\mathrm{sign}(\dot{\boldsymbol{q}}), \tag{37}$$

$$\boldsymbol{f}_{st}(\dot{\boldsymbol{q}}; \boldsymbol{w}_{st,s}) = \mathrm{sign}(\dot{\boldsymbol{q}})\left(K_{st,s} - K_{c,s}\right) e^{-\left(\dot{q}/\dot{q}_{st,s}\right)^2}, \tag{38}$$

where the parameters $K_{v,s}$, $K_{c,s}$, $K_{st,s}$, $\dot{q}_{st,s}$ are learned during training. The discontinuous sign function is approximated by $\tanh(\alpha\,\dot{\boldsymbol{q}})$, where α controls the approximation accuracy. The smoothing parameter α is fixed to $\alpha = 50$, which provides a close approximation of the discontinuous sign function while maintaining stable gradients during training. The scheduling variables z_n are the joint positions, and, following the principles of Section 3.2.1, the fuzzy activations are constructed as follows:

$$\phi_s(\boldsymbol{q}) = \prod_{\ell=1}^{n_z=6} \phi_{s_\ell,\ell}(q_\ell), \quad \text{with} \quad \sum_{j=1}^{S_\ell=2} \phi_{j,\ell}(q_\ell) = 1$$

where the centers of the membership functions S_ℓ are chosen at $\{0, \pi/2\}$ rad for every joint. This choice is motivated by the need to provide a uniform representation of the joint configuration space and, consequently, an effective coverage of the robot workspace. For each operating region s, the corresponding local friction model is

$$\tilde{\boldsymbol{f}}_s(\dot{\boldsymbol{q}}; \boldsymbol{w}_{s,s}) = \boldsymbol{f}_{cv}(\dot{\boldsymbol{q}}; \boldsymbol{w}_{cv,s}) + \boldsymbol{f}_{st}(\dot{\boldsymbol{q}}; \boldsymbol{w}_{st,s}), \tag{39}$$

[2] https://github.com/gbionics/adam

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

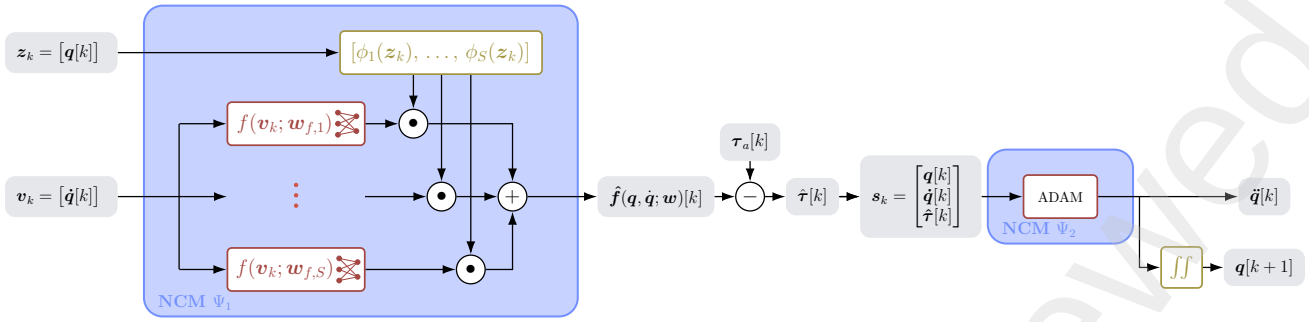


Figure 12: Architecture of the MSNN for the Unitree Z1 manipulator. NCM Ψ_1 receives the joint velocities $v_k = \dot{q}[k]$ and uses the joint positions $z_k = q[k]$ as scheduling variables; local input functions $f(v_k; w_{f,s})$ weighted by the activations $\phi_s(z_k)$ estimate the friction $\hat{f}(q, \dot{q}; w)[k]$. Subtracting this estimate from the applied torque $\tau_a[k]$ yields the net torque $\hat{\tau}[k]$. NCM Ψ_2 embeds the ADAM formulation and maps $s_k = [q[k], \dot{q}[k], \hat{\tau}[k]]^\mathsf{T}$ to the joint accelerations $\ddot{q}[k]$, which are integrated to obtain $q[k+1]$.

and the overall friction estimate is obtained through the weighted interpolation as described by Eq. (7)

$$\hat{f}(q, \dot{q}; w) = \sum_{s}^{S=2^6} \phi_s(q) \odot \tilde{f}_s(\dot{q}; w_{s,s}). \tag{40}$$

The estimated friction torque is then subtracted from the measured actuator torque,

$$\hat{\tau} = \tau_a - \hat{f}(q, \dot{q}; w), \tag{41}$$

The second NCM consists of an input function composed of a `ParametricFunction` that embeds the ADAM library for solving the forward dynamics problem

$$\hat{\ddot{q}} = M^{-1}(q)\,(\hat{\tau} - h(q, \dot{q}))\,. \tag{42}$$

The complete architecture is illustrated in Fig. 12.

The *black-box* **(LSTM)** model consists of a double-layer Long Short-Term Memory (LSTM) network with 128 hidden units in each layer, followed by a fully connected feedforward head composed of a linear layer, a ReLU activation function, and a final linear layer that maps the hidden representation to a 6-dimensional output vector corresponding to the predicted joint accelerations $\ddot{q} \in \mathbb{R}^6$. The model receives as input a sequence of joint positions $q \in \mathbb{R}^6$, joint velocities $\dot{q} \in \mathbb{R}^6$, and joint torques $\tau \in \mathbb{R}^6$ over a fixed window of length $T = 20$ time steps, corresponding to 0.2 s at a sampling rate of 100 Hz ($\Delta t = 0.01$ s), resulting in an input tensor of shape $(B, T, 18)$ where B denotes the batch size. The LSTM processes this input in a batch-first manner and outputs a hidden state sequence of shape $(B, T, 128)$, which is then passed through the feedforward head to produce predictions for the entire sequence. During training, the model minimizes the mean squared error (MSE) between the predicted and ground-truth accelerations using the Adam optimizer with a learning rate of 1×10^{-3}, while during evaluation the model is assessed on its ability to predict only the final timestep $\ddot{q}_T$ (i.e., the one-step-ahead prediction given the past 20 observations). The total number of trainable parameters in the model is 92,550, with 75,264 parameters in the LSTM layers and 17,286 parameters in the feedforward prediction head.

The *Physics-Based Model* **(PBM)** model performs classical system identification by estimating the friction parameters directly from the data using the known rigid-body dynamics. Specifically, the approach leverages the physical model of the manipulator and the measured joint torques to isolate the friction contribution, which is then identified through standard least-squares minimization over the collected trajectories. In our implementation, the PBM baseline is obtained by fitting the friction parameters in Equations (37) and (38) to the simulation data using unconstrained optimization, resulting in a compact physics-based model with only 24 learnable parameters. Unlike the data-driven baselines, this method does not require iterative training or large datasets, as it exploits the known structure of the dynamics to directly recover the friction coefficients. However, its performance is inherently limited by the fidelity of the assumed friction model and may not capture unmodeled nonlinear effects present in real-world scenarios.

Dataset. The dataset was generated in MuJoCo [63] using the original robot description files provided by the manufacturer[3]. Only the friction parameters of the simulator were modified, while all inertial quantities remained unchanged. The training set consists of sixteen trajectories of 500 samples each, generated through random excitation profiles with varying accelerations. The implementation and scripts associated with this case study are available in the corresponding `nnodely-applications` directory `friction_learning_z1`.

Training. For the neural network models (MSNN and LSTM), training is performed in an autoregressive setting using a prediction horizon of 100 samples (corresponding to 1 s, while in the test is used 0.5 s). At each time step, the predicted accelerations are integrated using an explicit Euler integration scheme with sampling time $T_s = 0.01$ s. The predicted state is recursively fed back to the model until the end of the prediction horizon, after which the state is reset to

[3] https://github.com/unitreerobotics/unitree_ros

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Model	q	RMSE on q		Total RMSE	
		best	std dev	best	std dev
PBM (benchmark)	q_1 (rad)	4.95×10^{-2}	2.10×10^{-3}	3.39×10^{-2}	1.50×10^{-3}
	q_2 (rad)	$\mathbf{2.23 \times 10^{-2}}$	$\mathbf{1.00 \times 10^{-3}}$		
	q_3 (rad)	1.22×10^{-2}	5.80×10^{-4}		
	q_4 (rad)	2.40×10^{-2}	1.10×10^{-3}		
	q_5 (rad)	1.44×10^{-2}	7.00×10^{-4}		
	q_6 (rad)	1.16×10^{-2}	5.50×10^{-4}		
LSTM (benchmark)	q_1 (rad)	5.72×10^{-1}	1.70×10^{-2}	2.96×10^{-1}	8.70×10^{-3}
	q_2 (rad)	8.11×10^{-2}	3.20×10^{-3}		
	q_3 (rad)	3.33×10^{-1}	1.00×10^{-2}		
	q_4 (rad)	1.67×10^{-1}	5.20×10^{-3}		
	q_5 (rad)	2.05×10^{-1}	6.40×10^{-3}		
	q_6 (rad)	9.86×10^{-2}	3.10×10^{-3}		
MSNN (ours)	q_1 (rad)	$\mathbf{2.85 \times 10^{-2}}$	$\mathbf{1.10 \times 10^{-3}}$	$\mathbf{3.11 \times 10^{-2}}$	$\mathbf{1.20 \times 10^{-3}}$
	q_2 (rad)	4.18×10^{-2}	1.60×10^{-3}		
	q_3 (rad)	$\mathbf{1.08 \times 10^{-2}}$	$\mathbf{4.20 \times 10^{-4}}$		
	q_4 (rad)	$\mathbf{1.09 \times 10^{-2}}$	$\mathbf{4.30 \times 10^{-4}}$		
	q_5 (rad)	$\mathbf{6.54 \times 10^{-3}}$	$\mathbf{2.60 \times 10^{-4}}$		
	q_6 (rad)	$\mathbf{9.91 \times 10^{-4}}$	$\mathbf{4.00 \times 10^{-5}}$		

Table 4
Prediction performance (**RMSE**) for the Unitree Z1 manipulator. The table reports the RMSE on the joint positions $\boldsymbol{q} = [q_1, \dots, q_6]^\mathsf{T}$, together with the Total RMSE, for 0.5 s recurrent prediction on the test set. The compared models are the *Physics-Based Model* (**PBM**), the *Long Short-Term Memory* network (**LSTM**), and the *Model-Structured Neural Network* (**MSNN**). We report the best value and the standard deviation over 5 random seeds. The MSNN achieves the lowest RMSE with fewer learnable parameters than the LSTM, while the PBM attains competitive performance, highlighting the effectiveness of the underlying physics-based structure. Best results are highlighted in boldface.

the measured one before continuing with the next segment. As a consequence, the model is evaluated in a multi-segment prediction setting rather than a fully autonomous long-horizon rollout. The model parameters are optimized by minimizing the mean squared error (MSE) between the predicted and measured joint positions over the complete prediction horizon. Specifically, the resulting predicted joint positions ($\hat{\boldsymbol{q}}_k$) are compared with the measured positions ($\boldsymbol{q}_k$) at each prediction step. The loss is evaluated over the full prediction horizon, and the gradient is backpropagated 2through the complete computational graph, including the differentiable dynamics provided by the ADAM library.

To ensure a fair comparison, all neural network-based baselines are trained under the same optimization budget. In particular, both the MSNN and the LSTM are optimized using the Adam optimizer with learning rate 10^{-3} for 1000 training epochs and identical batch size. All models are trained on the same training and validation splits without hyperparameter tuning beyond standard architectural choices. At evaluation time, performance is computed by comparing only the last timestep of each sequence to reduce correlation between overlapping sliding windows.

Results. Figure 11 and Table 4 report the joint position prediction over a prediction horizon of 0.5 s. The proposed MSNN achieves performance comparable to the classical PBM baseline, with a test RMSE of 3.10×10^{-2} rad (Table 4). The predicted trajectories follow the overall trend of the reference signals, although non-negligible deviations remain, indicating that accurate long-horizon prediction under the considered excitation regime remains challenging.

The LSTM shows significantly higher prediction error over the same horizon, with an RMSE of 2.96×10^{-1} rad, approximately one order of magnitude larger than the MSNN. This can be attributed to the fact that it performs purely data-driven sequence modeling without an explicit structural bias from rigid-body dynamics, which affects its robustness in recursive prediction settings. In contrast, the PBM baseline achieves an RMSE of 3.39×10^{-2} rad with only 24 learnable parameters, demonstrating that even a simple physics-based model can provide competitive performance when the underlying dynamics are well characterized. The PBM model exhibits intermediate performance, outperforming the black-box LSTM but slightly underperforming the MSNN. This result is noteworthy, as it shows that a PBM approach with minimal parameters can effectively capture the dominant dynamics of the system, even in the presence of complex friction effects. However, the PBM's reliance on a predefined friction structure limits its ability to generalize beyond the assumed model class, whereas the MSNN retains the flexibility to learn nonlinearities through its local model architecture.

5.4. Lateral Vehicle Dynamics Learning of a Ferrari 250LM

This case study evaluates the effectiveness of the proposed Model-Structured Neural Network (MSNN) framework for modeling the lateral dynamics of a vehicle. The model takes the steering angle, longitudinal speed, and longitudinal

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

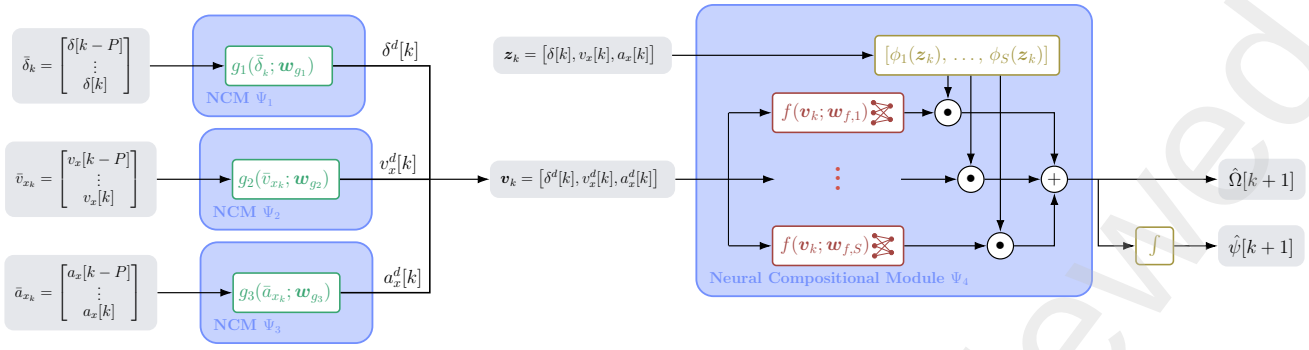


Figure 13: Architecture of the MSNN for the Ferrari 250 LM lateral dynamics. NCMs Ψ_1–Ψ_3 implement output functions $g_i(\cdot;\boldsymbol{w}_{g_i})$ as FIR filters acting on the temporal inputs $\bar{\delta}_k$, $\bar{v}_{x_k}$, and $\bar{a}_{x_k}$, producing the dynamic responses $\delta^d[k]$, $v_x^d[k]$, and $a_x^d[k]$, collected into $\boldsymbol{v}_k = [\delta^d[k], v_x^d[k], a_x^d[k]]$. NCM Ψ_4 is a Local Model: the scheduling vector $\boldsymbol{z}_k = [\delta[k], v_x[k], a_x[k]]^\top$ generates the activations $\phi_s(\boldsymbol{z}_k)$, and the local input functions $f(\boldsymbol{v}_k;\boldsymbol{w}_{f,s})$ implement the first-order linearization of Eq. (43) around the corresponding equilibrium points. The output of Ψ_4 is the yaw rate $\hat{\Omega}[k+1]$, which is integrated, using either the Euler or trapezoidal method, to obtain the yaw angle $\hat{\psi}[k+1]$.

acceleration as inputs and predicts the yaw rate, from which the yaw angle is obtained through numerical integration. The vehicle considered in this study is a Ferrari 250 LM driven on the Palm Beach circuit. Since the circuit can be represented as a two-dimensional track, three-dimensional effects such as elevation changes and road banking are neglected, as they are assumed to have a negligible influence on the vehicle dynamics considered in this study.

Physics Priors. The physical structure of the network is derived from the yaw rate gain formulation given in Eq. (43):

$$\Omega(v_x, a_x, \delta) = \frac{v_x}{K_{us}(v_x, a_x)v_x^2 + L} \cdot \delta \tag{43}$$

where v_x and a_x denote the forward speed and longitudinal acceleration, respectively, δ is the steering angle, and K_{us} is the understeering coefficient, which is generally a nonlinear function of v_x and a_x.

Neural Models. To assess the effectiveness of the proposed approach, the following models are compared. (i) The *Model-Structured Neural Network* (**MSNN**), which exploits a physically motivated structure composed of dynamic input models and local models to describe the vehicle lateral dynamics. (ii) The *Multi-Layer Perceptron* (**MLP**), a black-box neural network used to learn the vehicle dynamics without explicitly imposing the physical structure of the system. (iii) The *Single-Track* (**ST**) model, based on the classical linear vehicle dynamics formulation, used as a physically interpretable reference model.

The **MSNN** is illustrated in Figure 13. The three input signals are processed independently by three NCMs, each consisting of an FIR filter that captures the corresponding transient dynamics. The resulting dynamic responses are then provided to a fourth NCM, which implements the local model that describes the lateral dynamics of the vehicle. The output of this local model is the yaw rate Ω, which is subsequently integrated to obtain the yaw angle ψ.

Let:

$$\begin{aligned}\bar{\delta}_k &= [\delta[k-P], \ldots, \delta[k]]^T,\\ \bar{v}_{x_k} &= [v_x[k-P], \ldots, v_x[k]]^T,\\ \bar{a}_{x_k} &= [a_x[k-P], \ldots, a_x[k]]^T\end{aligned} \tag{44}$$

be the sampled histories of the three input signals provided to the model. The history length is set to $P = 9$ (i.e., 10 samples), corresponding to approximately 0.5s of past data at the considered sampling frequency. Each input history is processed by an NCM containing a Finite Impulse Response filter (FIR), which accounts for the transient response associated with the corresponding input. The resulting dynamic responses are denoted by:

$$\begin{aligned}\delta^d[k] &= g_1(\bar{\delta}_k;\boldsymbol{w}_{g_1}),\\ v_x^d[k] &= g_2(\bar{v}_{x_k};\boldsymbol{w}_{g_2}),\\ a_x^d[k] &= g_3(\bar{a}_{x_k};\boldsymbol{w}_{g_3})\end{aligned} \tag{45}$$

and are collected into the input vector of the output NCM:

$$\boldsymbol{v}_k = \left[\delta^d[k], v_x^d[k], a_x^d[k]\right]^T \tag{46}$$

which therefore consists of three scalar dynamic responses. The steady-state relation in Eq. (43) indicates that the yaw rate depends on the steering angle through a gain that varies with the longitudinal operating condition. To incorporate this dependency while preserving the physical structure of the model, the nonlinear discrete-time mapping is formulated as a convex combination of local models. The scheduling variables are defined as $\boldsymbol{z}_k = \boldsymbol{v}_k - \boldsymbol{v}_s$, while $\phi_s(\boldsymbol{z}_k)$ denotes the activation function associated with the s-th operating regime. These activation functions are constructed from triangular fuzzy membership functions. The resulting formulation of the yaw-rate prediction is:

$$\hat{\Omega}[k+1] = \sum_{s=1}^{S} \phi_s(\boldsymbol{z}_k) \odot f(\boldsymbol{v}_k;\boldsymbol{w}_{f,s}) \tag{47}$$

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

which corresponds to the Local Model formulation in (7), where the output function has unitary gain and $f(\cdot;\boldsymbol{w}_{f,s})$ is the input function associated with the s-th operating condition. Following a first-order Taylor expansion of Eq. (43), the local input function is expressed according to *Locally-Linearized Function* presented in Eq. (25):

$$\hat{\Omega}[k+1] = \sum_{s=1}^{S} \phi_s(\boldsymbol{v}_k - \boldsymbol{v}_s) \odot \left(w_{0,s} + \sum_{j=1}^{n_v} w_{j,s} v_{k,j} \right) \quad (48)$$

where $w_{j,s}$ are the learnable parameters of the local model and $n_v = 3$ is the number of elements contained in the vector $\boldsymbol{v}_k$. By grouping the contributions associated with each input variable, the local-model parameters can be interpreted as functions of the current operating point $\boldsymbol{z}_k$. The resulting expression can therefore be written as:

$$\begin{aligned} \hat{\Omega}[k+1] = {} & w_0(\boldsymbol{z}_k) + w_1(\boldsymbol{z}_k)\delta^d[k] \\ & + w_2(\boldsymbol{z}_k)v_x^d[k] + w_3(\boldsymbol{z}_k)a_x^d[k] \end{aligned} \quad (49)$$

The linearization points, which also define the centers of the `Fuzzy` membership functions, are selected as $\delta_s \in [-85, -42.5, 0, 42.5, 85]$ deg, $v_{x_s} \in [12.5, 58.0]$ m/s, and $a_{x_s} \in [-8.0, 5.0]$ m/s^2, resulting in a total of $S = 20$ local models. With $N_k = 4$ learnable parameters assigned to each local model, the resulting MSNN contains a total of $N_k \cdot S + 3(P+1) = 110$ learnable parameters.

The black-box neural network is implemented as a fully connected Multi-Layer Perceptron (**MLP**). The same three input histories used in the MSNN, as defined in (44), are concatenated into a single input vector, resulting in 30 input features. The network consists of two hidden layers with 8 and 5 neurons, respectively, both using ReLU activation functions, followed by a single linear output neuron predicting the yaw rate. This architecture results in a total of 299 learnable parameters.

The Single-Track model (**ST**) is considered as a conventional physics-based reference for the vehicle lateral dynamics. In its linear formulation, presented in [64], three parameters are identified from the available training data: the front and rear tire cornering stiffnesses and the vehicle yaw inertia, while the remaining vehicle parameters are assumed to be known a priori. This assumption may be restrictive when only limited information about the vehicle is available. The model is evaluated using Euler and fourth-order Runge–Kutta (RK4) integration, with the resulting yaw angle used to assess long-term prediction performance.

Dataset. The dataset used for training and validation is the open-source telemetry dataset released by the Stanford REVS Program [65], collected during the 2014 Targa Sixty-Six event. It consists of telemetry records from a Ferrari 250 LM driven on the Palm Beach circuit. The complete dataset comprises multiple runs, of which two were used for training, one for validation, and one for testing. The vehicle instrumentation operated at sampling rates ranging from 100 Hz to 500 Hz. However, the dominant vehicle dynamics lie well below these frequencies, with a natural frequency around 5 Hz and residual informative content extending to approximately 10 Hz. Some measured signals, particularly the accelerations, were also affected by high-frequency noise. The dataset was therefore preprocessed and downsampled to 20 Hz. Prior to downsampling, a fourth-order Butterworth low-pass filter with a 5 Hz cutoff frequency was applied as a trade-off between noise suppression and preservation of the relevant vehicle dynamics. Owing to its gradual roll-off, the filter provides approximately 53 dB of attenuation at the 20 Hz sampling frequency, effectively suppressing high-frequency noise while only partially attenuating, rather than eliminating, signal content up to 10 Hz. This choice is further supported by the inherent robustness of the proposed MSNN to residual noise and moderate signal attenuation. The implementation and scripts associated with this case study are available in the corresponding `nnodely-applications` directory `vehicle_lateral_dynamics_estimation`.

Training. Recurrent training is performed over a prediction horizon of 500 samples, corresponding to approximately 25 s. At each time step, the network predicts the yaw rate $\hat{\Omega}[k+1]$, while the yaw angle $\hat{\psi}[k+1]$ is obtained by numerically integrating the predicted yaw rate. Since the yaw angle is an autoregressive variable, its predicted value is propagated through the prediction horizon and used as the state for the subsequent integration step. Therefore, the yaw angle cannot be evaluated independently at each time step, but must be recursively predicted over the entire horizon. Accordingly, the training is performed in a recurrent fashion, where the prediction at each step is used to obtain the subsequent state. The loss function accounts for both the yaw-rate error and the yaw-angle error over the prediction horizon. The yaw angle is computed through the `Integrate` block, using either Euler or trapezoidal numerical integration. This recurrent formulation allows the training procedure to account for the accumulation of errors in the integrated yaw angle and therefore constrains the network to provide consistent yaw-rate predictions over long-term horizons. In particular, this is important in the presence of sensor offsets, since even a small bias in the predicted yaw rate can accumulate through the integration and lead to significant drift in the yaw-angle prediction.

Results. The results in Fig. 14 and Table 5 highlight the advantages of the proposed MSNN for long-term prediction. The results are reported using both Euler and trapezoidal integration schemes to assess the influence of the numerical integration method on the prediction. In both cases, the MSNN closely reproduces the yaw-rate oscillations over the entire 25 s horizon, with the trapezoidal integration achieving the lowest RMSE for both yaw rate (1.302 deg/s) and yaw angle (7.499 deg). The small standard deviations across the different training runs also indicate good robustness to weight initialization and data shuffling. In comparison, the MLP network, despite using 299 learnable parameters

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Model	Integrator	q	RMSE on q		RMSE on $\dot{q}$		Total RMSE	
			best	std dev	best	std dev	best	std dev
ST (benchmark)	Euler	ψ (deg)	14.29		3.18		4.61	
ST (benchmark)	RK4	ψ (deg)	14.61		3.01		4.47	
MLP (benchmark)	Trapezoidal	ψ (deg)	9.61	9.49×10^{-1}	2.08	1.21×10^{-1}	3.04	2.11×10^{-1}
MSNN (ours)	Euler	ψ (deg)	7.60	$\mathbf{2.86 \times 10^{-2}}$	1.31	$\mathbf{5.37 \times 10^{-3}}$	2.07	$\mathbf{5.05 \times 10^{-3}}$
MSNN (ours)	Trapezoidal	ψ (deg)	**7.50**	5.63×10^{-2}	**1.30**	8.40×10^{-3}	**2.05**	1.11×10^{-2}

Table 5
Prediction performance (**RMSE**) for the Ferrari 250 LM lateral dynamics. The table reports the RMSE on the state $q = \psi$ (yaw angle) and on its derivative $\dot{q} = \Omega$ (yaw rate), together with the Total RMSE, computed as $\mathrm{RMSE}_{\dot{q}} + 0.1 \cdot \mathrm{RMSE}_{q}$, for 25 s forward integration on an unseen dataset. The compared models are the *Single-Track* model (**ST**), the *Multi-Layer Perceptron* (**MLP**), and the *Model-Structured Neural Network* (**MSNN**). The ST benchmark is fitted by unconstrained optimization over the full training set, using either Euler or RK4 integration; the MLP and the MSNN are trained with 10 random seeds and data shuffles, and both Euler and trapezoidal integration are compared for the MSNN. We report the best value and the standard deviation. The MSNN with the trapezoidal scheme achieves the best accuracy and the lowest standard deviation on the Total RMSE, while having fewer learnable parameters than the MLP. Best results are highlighted in boldface.

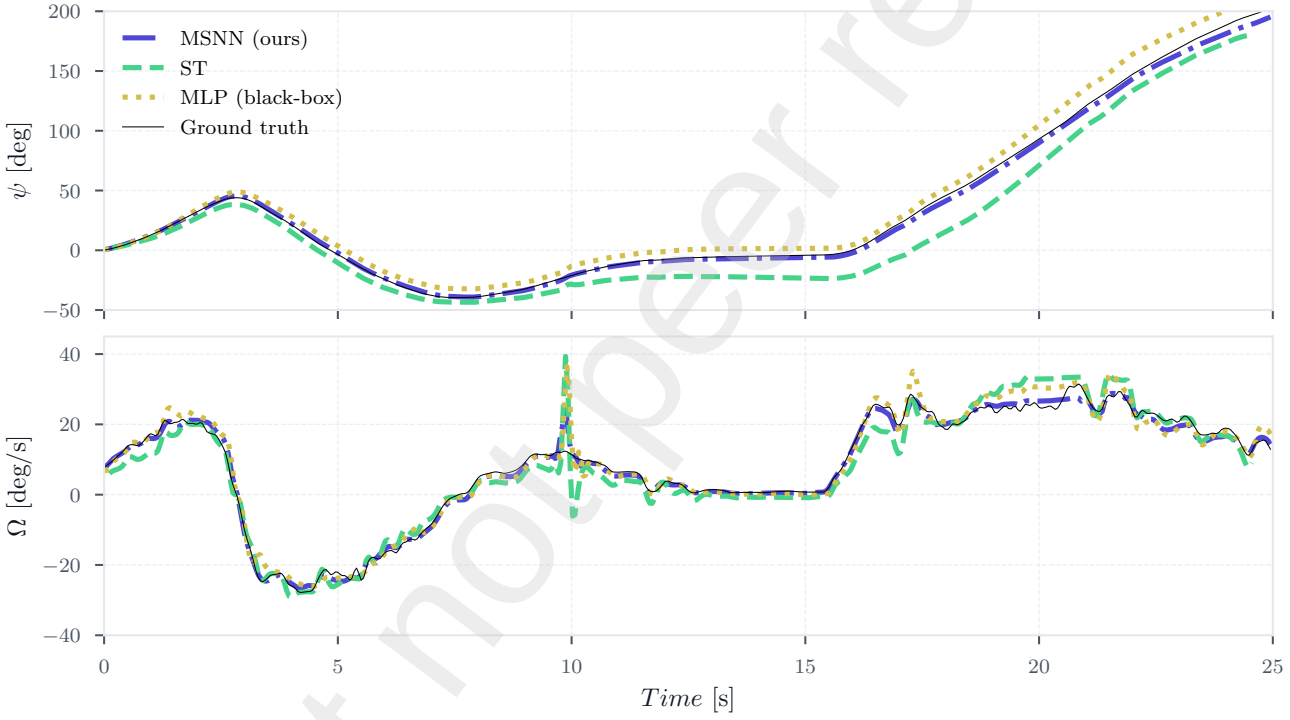


Figure 14: Comparison of the yaw angle ψ and yaw rate Ω predictions of the Ferrari 250 LM over a 25s receding horizon. The Single-Track (**ST**) model, with only three learnable parameters, exhibits a long-term accumulated drift, particularly in the yaw angle prediction. The **MSNN** provides the most consistent prediction of the oscillations in the yaw rate, resulting in an accurate long-term yaw angle prediction. In contrast, the general-purpose neural network (**MLP**), despite having approximately three times as many learnable parameters, exhibits poorer predictive performance than the proposed **MSNN**.

against only 110 for the MSNN, exhibits larger prediction errors and significantly higher variability. The MLP achieves yaw-rate and yaw-angle RMSE values of 2.082 deg/s and 9.608 deg, respectively. The Single-Track model provides a reasonable yaw-rate prediction; however, small errors accumulate through the numerical integration, resulting in a substantially larger yaw-angle error. In particular, the RK4 formulation results in a yaw-angle RMSE of 14.611 deg.

Overall, the results demonstrate that the MSNN provides a favorable trade-off between model complexity, prediction accuracy, and long-term prediction stability.

5.5. Sample Efficiency of Model-Structured Neural Networks

In this subsection, we evaluate the data efficiency of the proposed MSNNs by comparing their performance against general-purpose (black-box) neural networks. Specifically, we analyze how predictive accuracy, measured in terms of

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

R^2, evolves as a function of the available training data. This comparison aims to assess whether incorporating prior structural knowledge enables MSNNs to achieve accurate models with fewer data samples.

Evaluation metric and aggregation. Since several of the considered models predict multiple output variables, a single (R^2) value is reported in Fig. 15 by averaging the coefficient of determination computed independently for each output variable. In particular, for a model with n_y outputs, the R^2 score is first computed separately for each output j as

$$R_j^2 = 1 - \frac{\sum_{i=1}^{N}(y_{i,j} - \hat{y}_{i,j})^2}{\sum_{i=1}^{N}(y_{i,j} - \bar{y}_j)^2}, \tag{50}$$

where $y_{i,j}$ and $\hat{y}_{i,j}$ denote the true and predicted values of output j for sample i, respectively, and $\bar{y}_j$ is the mean value of the corresponding ground-truth output over the test set. The reported score is then obtained as the arithmetic mean across the output variables,

$$R^2_{\text{avg}} = \frac{1}{n_y}\sum_{j=1}^{n_y} R_j^2. \tag{51}$$

Thus, the reported R^2 does not result from concatenating the different outputs, nor does it represent the worst-performing output; instead, it provides an equally weighted measure of prediction accuracy across all outputs. This definition is applied consistently to both the MSNN and black-box models.

For each training-data fraction, the reported value corresponds to the mean R^2_{avg} over the independent training runs. The shaded bands represent ± 1 standard deviation across these runs, thereby quantifying the variability associated with different random initializations and, where applicable, different training-data selections. Consequently, a narrower band indicates greater robustness to the stochasticity of the training procedure, whereas a wider band indicates higher sensitivity to the specific training realization.

Dynamic Estimation of Inverted Pendulum on a Cart: For the Inverted Pendulum on a Cart case study, we compared the PBM_G model with an MLP model, both described in Section 5.1. Both models are trained using varying proportions of the available training dataset to assess their data efficiency and generalization capabilities under limited data regimes. The results show the superior performance of the MSNN, with consistently higher accuracy regardless of the percentage of the dataset used for training.
Dynamics Estimation of Double Pendulum on a Cart: For the Double Pendulum on a Cart case study, we compared a purely black-box based on MLP with a structured neural network based Local Model. Both models were trained using varying proportions of the available training dataset to evaluate their data efficiency and generalization capabilities under limited-data conditions. As shown in Fig. 15b, the MSNN reached near-perfect prediction accuracy even with a small fraction of the available data. In contrast, the MLP model required substantially more training data to achieve comparable performance, highlighting the improved sample efficiency and stronger inductive bias provided by the structured architecture.

Friction Estimation of a Unitree Z1: For the Friction Estimation of a Unitree Z1 case study, we compared the MSNN model described in Section 5.3 with an LSTM model. The training dataset consists of 16 episodes, each containing 500 samples recorded at 100 Hz, corresponding to a total duration of 80 seconds. For this study, the dataset was progressively reduced to smaller fractions (75%, 50%, 25%, 20%, 15%, and 10%). For each fraction, a model was trained and evaluated five times with different random seeds to estimate the variability of the results.

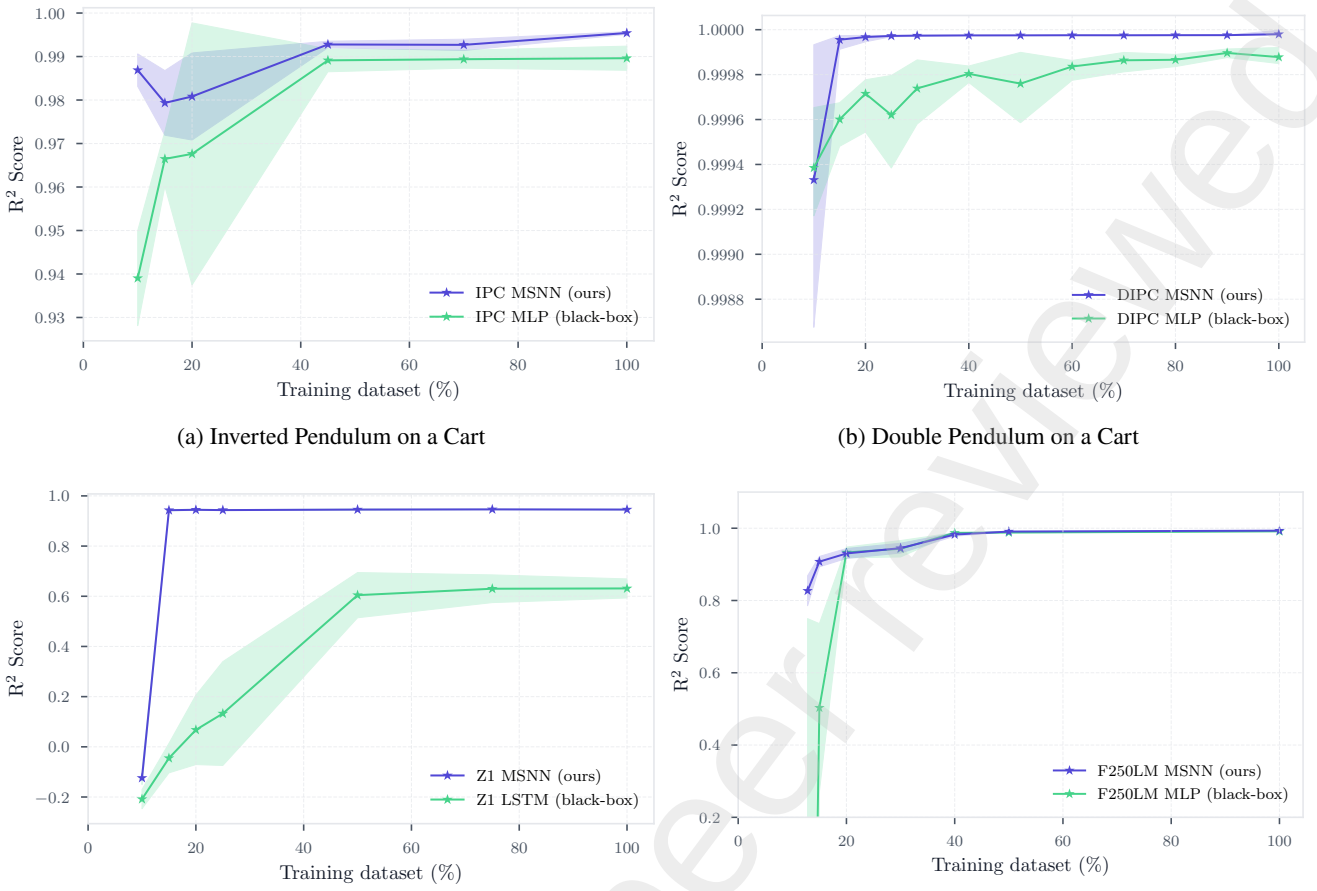
Figure 15c shows the results obtained from the different tests. As can be seen, the MSNN achieves an R^2 score above 0.9 much earlier than the LSTM, using only 15% of the dataset. Another observation is that the LSTM exhibits higher variance compared to the MSNN, supporting the claim that embedding physics into a neural network improves the learning of the true dynamical laws of the system.

Lateral Dynamics Estimation of a Ferrari 250LM: For the lateral dynamics prediction, we compare the proposed MSNN, described in Section 5.4, with a general-purpose neural network (MLP) trained using a loss function based solely on the yaw rate. The training dataset consists of 500 samples recorded at 20 Hz, corresponding to approximately 25 s of data. Figure 15d reports the R^2 score for different fractions of the dataset (100%, 50%, 40%, 30%, 20%, 15%, and 12.5%). For each fraction, both models are trained 20 times, with randomization of both the weight initialization and the starting index used to select the training window. Although both the MSNN and MLP achieve an R^2 close to 1 when sufficient training data are available, the MSNN generally exhibits a lower standard deviation and maintains higher accuracy as the available fraction of the dataset decreases. In particular, the MLP performance deteriorates sharply when reducing the training fraction from 15% to 12.5%, resulting in an out-of-scale R^2 value and a significantly higher variance. This highlights the greater robustness of the proposed MSNN to reductions in the amount of available data.

6. Conclusions

This work introduced a compositional methodology for constructing model-structured neural networks (MSNNs) for the model-based, data-driven modeling of dynamical systems. The methodology is built around Neural Compositional Modules (NCMs), configurable computational blocks designed to represent specific physical, temporal, or functional contributions to the system dynamics. By composing NCMs through physically meaningful connections, prior knowledge about system decomposition, operating regimes, temporal dependencies, and partially known dynamics can be embedded directly into the neural architecture. The resulting MSNN provides an explicit, trainable, and interpretable

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813



(a) Inverted Pendulum on a Cart

(b) Double Pendulum on a Cart

(c) Unitree Z1 Friction

(d) Ferrari 250LM Lateral Dynamics

Figure 15: Data efficiency comparison on an independent test set. The reported R^2 is the arithmetic mean of the R^2 scores computed independently for all predicted output variables. For each training-data fraction, the solid line represents the mean score across the independent training runs, while the shaded region corresponds to ± 1 standard deviation. The proposed MSNNs consistently achieve higher accuracy with fewer samples, demonstrating superior sample efficiency. In contrast, black-box models exhibit a marked decline in performance in the low-data regime, with variability increasing substantially for the smallest dataset sizes. This high sensitivity to the training set forces repeated retraining with different initializations to obtain reliable results, ultimately undermining their sample efficiency compared to the structurally informed approach. In (a), IPC MSNN refers to the PBM_G network, while in (b), DIPC MSNN refers to the LM network.

representation of the modeled system within a unified differentiable computational graph.

The proposed methodology is directly implemented in the open-source `nnodely` library, which translates its compositional principles into an operational modeling workflow. Unlike general-purpose machine-learning frameworks such as PyTorch, where implementing structured models requires expertise in both neural-network programming and model-based system representation, `nnodely` provides high-level abstractions tailored to dynamical-system modeling. It supports the complete development process, from model definition and data processing to training, validation, inference, and export. Analytical relations and trainable components can thus be integrated into a single computational graph, reducing implementation complexity while preserving model transparency and flexibility.

The case studies demonstrate the applicability of the proposed methodology to dynamical systems characterized by different levels and forms of prior knowledge. For the inverted pendulum on a cart, the same NCM framework was used to construct physics-based, partially neuralized, residual, and finite-memory models. Under noise-free conditions, the structured models achieved comparable accuracy, with the physics-based gradient model reaching a total RMSE of 7.02×10^{-3}. When measurement and actuation noise were introduced, the FIR-based formulation achieved the best performance, reducing the total RMSE to 1.74×10^{-1}, compared with approximately 2.24×10^{-1} for the other structured NCM formulations and 2.72×10^{-1} for the black-box MLP. This example highlights how the NCM structure can be adapted to different assumptions on the available physics and how explicitly modeling temporal dependencies can improve robustness to noisy measurements.

For the double inverted pendulum on a cart, the local-model MSNN decomposed the nonlinear dynamics into 25 locally affine regimes scheduled by the two pole angles,

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

achieving a total RMSE of 7.50×10^{-3}. The Equation Learner model also slightly outperformed the black-box MLP, with total RMSE values of 3.13×10^{-2} and 3.21×10^{-2}, respectively. Although less accurate than the local-model formulation, the Equation Learner represents an interesting intermediate solution when the exact governing dynamics are unavailable but qualitative knowledge about the relevant functional relations can still be encoded in the candidate-function library. This case therefore illustrates how different NCM realizations can exploit progressively different levels of prior knowledge.

For the Unitree Z1 manipulator, the MSNN combined a structured, regime-dependent friction model with differentiable rigid-body dynamics provided by the external ADAM library. Over a 0.5 s prediction horizon, the resulting model achieved a total joint-position RMSE of 3.11×10^{-2} rad, compared with 3.39×10^{-2} rad for the physics-based parameter-identification model and 2.96×10^{-1} rad for the black-box LSTM. The MSNN achieved this performance using only 24 learnable parameters, compared with 92,550 for the LSTM, showing how known rigid-body dynamics can be retained while learning only the uncertain friction contribution.

Finally, on real Ferrari 250 LM telemetry, the MSNN represented the lateral vehicle dynamics through 20 regime-dependent local models preceded by FIR-based input dynamics and was trained recurrently over a 25 s prediction horizon. Using trapezoidal integration, it achieved RMSE values of 1.30 deg/s for the yaw rate and 7.50 deg for the yaw angle, corresponding to a total RMSE of 2.05. This compares with 3.04 for the black-box MLP and approximately 4.47 for the fitted single-track models. Moreover, the MSNN used 110 learnable parameters compared with 299 for the MLP and exhibited substantially lower variability across training runs, demonstrating the benefit of encoding operating-regime and temporal structure for stable long-horizon prediction.

Overall, the four case studies show how the same compositional formulation can accommodate different degrees of prior knowledge, ranging from known analytical dynamics and partially unknown physical effects to qualitative knowledge about nonlinear relations and operating regimes. Rather than prescribing a single neural architecture, the NCM formulation provides a common mechanism for selecting and combining physics-based functions, equation-learning components, local models, finite-memory operators, residual terms, and external differentiable models according to the information available for a given dynamical system. The resulting MSNNs retain physically meaningful structure while providing the flexibility required to learn unmodeled effects, with benefits in prediction accuracy, parameter efficiency, robustness to noise, and recurrent long-horizon simulation.

Future work will generalize the NCM formulation to vector- and tensor-valued variables and parameters, allowing structured models to incorporate high-dimensional sensing modalities such as images and LiDAR data. This generalization will be reflected in `nnodely` through multidimensional tensor representations. Separately, multi-sequence data representations will be introduced to support autoregressive and sequence-based architectures. A future release based on a Keras 3 backbone will also provide compatibility with PyTorch, TensorFlow, and JAX while preserving the same compositional modeling interface.

A. Remarks on Sampled-Time Representations

This appendix collects representative choices of the predictive input variables $\bar{\boldsymbol{v}}_k \in \mathbb{R}^{(P+1)\times n_v}$ and memory length P, associated with the sampled-time formulation introduced in Section 3.1.1. These cases are not separate modeling frameworks: they are specializations of the same finite-memory representation used in our methodology.

Partially Observable Systems: When the state $\boldsymbol{x}$ is only partially available, the accessible measurements may contain a subset of observable state variables $\boldsymbol{x}_m$. In this case, one can choose $\boldsymbol{v}$ as a concatenation of the measured state $\boldsymbol{x}_m$ and the controls $\boldsymbol{u}$:

$$\boldsymbol{v}[k] \triangleq \left[\boldsymbol{x}_m[k], \boldsymbol{u}[k]\right]. \tag{52}$$

Then $\bar{\boldsymbol{v}}_k \triangleq \{\boldsymbol{x}_m[k-p], \boldsymbol{u}[k-p]\}_{p=0}^{P}$. This corresponds to a partially observable finite-memory representation, where delayed measurements and controls encode the effect of hidden or unobservable state components without explicitly reconstructing them. More generally, $\boldsymbol{x}_m[k]$ can be replaced by a measured output $\mathbf{y}_m[k]$ whenever the available measurements are outputs rather than directly observable state components.

Delayed Systems: When the physical system exhibits an explicit input delay d, delayed control inputs can be incorporated by choosing $P \geq d$ or by explicitly including the corresponding delayed samples in the module input. For example, an input-delay model may use

$$\bar{\boldsymbol{v}}_k \triangleq \left[\boldsymbol{x}[k], \left[\boldsymbol{u}[k-P]^\top, \boldsymbol{u}[k-P+1]^\top, \dots, \boldsymbol{u}[k]^\top\right]^\top\right]. \tag{53}$$

This representation remains finite-memory but selects samples according to the system's known delay structure.

Markovian Systems: A simpler specialization is obtained when the full state is available and contains all the information required to predict the future evolution. In this case, the model can be written in a Markovian form by choosing

$$\bar{\boldsymbol{v}}_k \triangleq [\boldsymbol{x}[k], \boldsymbol{u}[k]], \qquad P = 0. \tag{54}$$

The finite-memory input $\bar{\boldsymbol{v}}_k$ then reduces to the current state–control pair. This corresponds to a one-step state-space representation, where the next state is a function of the current state and control input, and the model is Markovian.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Input-Output Models: A further specialization is obtained when the modeled output is assumed to depend predominantly on the input history, for instance when the zero-input free response associated with unknown initial conditions is negligible over the chosen window. In this case, one can choose

$$\boldsymbol{v}[k] \triangleq \boldsymbol{u}[k], \qquad P > 0. \tag{55}$$

Then $\bar{\boldsymbol{v}}_k \triangleq \left[\boldsymbol{u}[k-P]^\top, \dots, \boldsymbol{u}[k]^\top\right]^\top$, which gives an input-output finite-memory representation, and is appropriate when the relevant dynamic effects can be approximated from a finite window of past inputs.

In all these cases, the scheduling variables can be selected independently from the prediction variables. For example, $\boldsymbol{z}[k]$ may include the current state, a measured output, a control, a categorical mode, or any other quantity that determines the active operating regime. The memory length P allows the scheduling mechanism itself to depend on a finite history, which is useful when regime changes exhibit delay, hysteresis, or transient effects. Thus, by selecting $\bar{\boldsymbol{v}}_k$ and $\bar{\boldsymbol{z}}_k$, the proposed architecture provides a unified sampled-time representation for Markovian state-space models, partially observed systems, delayed systems, finite-memory input-output models, and regime-dependent dynamics.

Data and code availability

The `nnodely` framework developed and used in this work is open-source and publicly available at `https://github.com/tonegas/nnodely`. An archived version of the framework, corresponding to `v1.5.4`, is available on Zenodo.

The source code, configuration files, and scripts associated with the case studies are publicly available at `https://github.com/tonegas/nnodely-applications`. Direct links to the implementations of the individual experiments are provided in the corresponding case-study sections. An archived version of the case-study repository, compatible with `nnodely v1.5.4`, is available on Zenodo.

The datasets used in the simulated case studies were generated using MuJoCo as described in the corresponding sections. The Ferrari 250 LM case study uses the publicly available telemetry dataset released by the Stanford REVS Program [65].

Declaration of generative AI

Generative AI tools were used solely to assist with language editing and rewriting parts of the manuscript. They were not used to develop the methodology, generate results, perform data analysis, or formulate the scientific contributions. The authors reviewed the resulting text and take full responsibility for the content of the article.

References

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational physics*, vol. 378, pp. 686–707, 2019.

[2] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, Jun 2021.

[3] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics–informed neural networks: Where we are and what's next," *Journal of Scientific Computing*, vol. 92, no. 3, p. 88, Jul 2022.

[4] S. A. Faroughi, N. M. Pawar, C. Fernandes, M. Raissi, S. Das, N. K. Kalantari, and S. Kourosh Mahjour, "Physics-guided, physics-informed, and physics-encoded neural networks and operators in scientific computing: Fluid and solid mechanics," *Journal of Computing and Information Science in Engineering*, vol. 24, no. 4, p. 040802, 01 2024.

[5] J. Watson, C. Song, O. Weeger, T. Gruner, A. T. Le, K. Hansel, A. Hendawy, O. Arenz, W. Trojak, M. Cranmer, C. D'Eramo, F. Buelow, T. Goyal, J. Peters, and M. W. Hoffmann, "Machine learning with physics knowledge for prediction: A survey," *Transactions on Machine Learning Research*, 2025.

[6] M. Da Lio, M. Piccinini, and F. Biral, "Robust and sample-efficient estimation of vehicle lateral velocity using neural networks with explainable structure informed by kinematic principles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 12, pp. 13 670–13 684, 2023.

[7] M. Piccinini, M. Zumerle, J. Betz, and G. P. Rosati Papini, "A road friction-aware anti-lock braking system based on model-structured neural networks," *IEEE Open Journal of Intelligent Transportation Systems*, vol. 6, pp. 522–536, 2025.

[8] A. Plebe, M. Da Lio, and D. Bortoluzzi, "On reliable neural network sensorimotor control in autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 2, pp. 711–722, 2019.

[9] M. Piccinini, A. Mungiello, G. Jank, G. P. Rosati Papini, F. Biral, and J. Betz, "Model-structured neural networks to control the steering dynamics of autonomous race cars," in *2025 IEEE 28th International Conference on Intelligent Transportation Systems (ITSC)*, 2025, pp. 4129–4136.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

[10] M. Lutter, C. Ritter, and J. Peters, "Deep lagrangian networks: Using physics as model prior for deep learning," *CoRR*, vol. abs/1907.04490, 2019. [Online]. Available: http://arxiv.org/abs/1907.04490

[11] H. Perez-Villeda, J. Piater, and M. Saveriano, "Learning and extrapolation of robotic skills using task-parameterized equation learner networks," *Robotics and Autonomous Systems*, vol. 160, p. 104309, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889022001981

[12] A. Saviolo, G. Li, and G. Loianno, "Physics-inspired temporal learning of quadrotor dynamics for accurate model predictive trajectory tracking," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 256–10 263, 2022.

[13] D. Bianchi, N. Epicoco, M. Di Ferdinando, S. Di Gennaro, and P. Pepe, "Physics-informed neural networks for unmanned aerial vehicle system estimation," *Drones*, vol. 8, no. 12, 2024.

[14] W. De Groote, E. Kikken, E. Hostens, S. Van Hoecke, and G. Crevecoeur, "Neural network augmented physics models for systems with partially unknown dynamics: Application to slider–crank mechanism," *IEEE/ASME Transactions on Mechatronics*, vol. 27, no. 1, pp. 103–114, 2021.

[15] S. H. Hong, J. Ou, and Y. Wang, "Physics-guided neural network and gpu-accelerated nonlinear model predictive control for quadcopter," *Neural Computing and Applications*, vol. 35, no. 1, pp. 393–413, 2023.

[16] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, "Lagrangian neural networks," in *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020, workshop paper.

[17] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[18] T. Duong and N. Atanasov, "Hamiltonian-based neural ODE networks on the SE(3) manifold for dynamics learning and control," in *Robotics: Science and Systems (RSS)*, 2021.

[19] S. Saemundsson, A. Terenin, K. Hofmann, and M. Deisenroth, "Variational integrator networks for physically structured embeddings," in *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, S. Chiappa and R. Calandra, Eds., vol. 108. PMLR, 26–28 Aug 2020, pp. 3078–3087.

[20] A. Havens and G. Chowdhary, "Forced variational integrator networks for prediction and control of mechanical systems," in *Proceedings of the 3rd Conference on Learning for Dynamics and Control*, ser. Proceedings of Machine Learning Research, vol. 144. PMLR, 07 – 08 June 2021, pp. 1142–1153. [Online]. Available: https://proceedings.mlr.press/v144/havens21a.html

[21] V. Duruisseaux, T. P. Duong, M. Leok, and N. Atanasov, "Lie group forced variational integrator networks for learning and control of robot systems," in *Learning for dynamics and control conference*. PMLR, 2023, pp. 731–744.

[22] F. Djeumou, T. J. Lew, N. DING, M. Thompson, M. Suminaka, M. Greiff, and J. Subosits, "One model to drift them all: Physics-informed conditional diffusion model for driving at the limits," in *Proceedings of The 8th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, P. Agrawal, O. Kroemer, and W. Burgard, Eds., vol. 270. PMLR, 06–09 Nov 2025, pp. 604–630.

[23] O. Dikici, E. Ghignone, C. Hu, N. Baumann, L. Xie, A. Carron, M. Magno, and M. Corno, "Learning-based on-track system identification for scaled autonomous racing in under a minute," *IEEE Robotics and Automation Letters*, 2025.

[24] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman, "Universal differential equations for scientific machine learning," *arXiv preprint arXiv:2001.04385*, 2020.

[25] F. Díaz Ledezma and S. Haddadin, "First-order-principles-based constructive network topologies: An application to robot inverse dynamics," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, 2017, pp. 438–445.

[26] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[27] D. Papageorgiou, G. P. Siguroardottir, E. Falotico, and S. Tolu, "Sliding-mode control of a soft robot based on data-driven sparse identification," *Control Engineering Practice*, vol. 144, p. 105836, 2024.

[28] M. Omar, K. Wang, D. Kun, R. Li, and A. Asker, "Robust data-driven dynamic model discovery of industrial robots with spatial manipulation capability using simple trajectory," *Nonlinear Dynamics*, vol. 112, no. 11, pp. 9155–9177, Jun 2024.

[29] M. Da Lio, D. Bortoluzzi, and G. P. Rosati Papini, "Modelling longitudinal vehicle dynamics with neural networks," *Vehicle System Dynamics*, vol. 58, no. 11, pp. 1675–1693, 2020.

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

[30] S. S. James, S. R. Anderson, and M. Da Lio, "Longitudinal vehicle dynamics: A comparison of physical and data-driven models under large-scale real-world driving conditions," *Ieee Access*, vol. 8, pp. 73 714–73 729, 2020.

[31] A. Mungiello, F. Jahncke, S. Santini, J. Betz, G. P. Rosati Papini, and M. Piccinini, "Vehicle dynamics learning from physics priors with model-structured neural networks," *IEEE Open Journal of Intelligent Transportation Systems*, vol. 7, pp. 1181–1196, 2026.

[32] M. Piccinini, S. Taddei, M. Larcher, M. Piazza, and F. Biral, "A physics-driven artificial agent for online time-optimal vehicle motion planning and control," *IEEE Access*, vol. 11, pp. 46 344–46 372, 2023.

[33] E. Pagot, M. Piccinini, E. Bertolazzi, and F. Biral, "Fast planning and tracking of complex autonomous parking maneuvers with optimal control and pseudo-neural networks," *IEEE Access*, vol. 11, pp. 124 163–124 180, 2023.

[34] A. Antonucci, G. P. Rosati Papini, P. Bevilacqua, L. Palopoli, and D. Fontanelli, "Efficient prediction of human motion for real-time robotics applications with physics-inspired neural networks," *IEEE Access*, vol. 10, pp. 144–157, 2021.

[35] T. Kamp, J. Ultsch, and J. Brembeck, "Closing the sim-to-real gap with physics-enhanced neural odes," in *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics - Volume 2: ICINCO*, INSTICC. SciTePress, 2023, pp. 77–84.

[36] X. Yang, Z. Zhou, L. Li, and X. Zhang, "Collaborative robot dynamics with physical humanrobot interaction and parameter identification with pinn," *Mechanism and Machine Theory*, vol. 189, p. 105439, 2023.

[37] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "Deepxde: A deep learning library for solving differential equations," *SIAM review*, vol. 63, no. 1, pp. 208–228, 2021.

[38] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, N. Vinchhi, K. Balakrishnan, D. Upadhyay, and C. Rackauckas, "Neuralpde: Automating physics-informed neural networks (pinns) with error approximations," *arXiv*, 2021. [Online]. Available: https://arxiv.org/abs/2107.09443

[39] O. Hennigh, S. Narasimhan, M. A. Nabian, A. Subramaniam, K. Tangsali, M. Rietmann, J. del Aguila Ferrandis, W. Byeon, Z. Fang, and S. Choudhry, "NVIDIA simnet^{TM}: an AI-accelerated multi-physics simulation framework," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2012.07938

[40] NVIDIA, "Physicsnemo: Open-source physics-ML framework (formerly NVIDIA modulus)," GitHub repository, 2026, accessed: 2026-04-13. [Online]. Available: https://github.com/NVIDIA/physicsnemo

[41] F. Chen, D. Sondak, P. Protopapas, M. Mattheakis, S. Liu, D. Agarwal, and M. Di Giovanni, "Neurodiffeq: A python package for solving differential equations with neural networks," *Journal of Open Source Software*, vol. 5, no. 46, p. 1931, 2020.

[42] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[43] M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park, "Torchdyn: A neural differential equations library," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2009.09346

[44] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, "Diffeqflux.jl – a julia library for neural differential equations," *arXiv*, 2019. [Online]. Available: https://arxiv.org/abs/1902.02376

[45] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, "Brax – a differentiable physics engine for large scale rigid body simulation," *arXiv*, 2021. [Online]. Available: https://arxiv.org/abs/2106.13281

[46] K. Werling, D. Omens, J. Lee, I. Exarchos, and C. K. Liu, "Fast and feature-complete differentiable physics for articulated rigid bodies with contact," *arXiv*, 2021. [Online]. Available: https://arxiv.org/abs/2103.16021

[47] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, "Neuralsim: Augmenting differentiable simulators with neural networks," *arXiv*, 2021, accepted at IEEE International Conference on Robotics and Automation (ICRA) 2021. [Online]. Available: https://arxiv.org/abs/2011.04217

[48] M. Macklin, N. Capens, Z. Zong, P. Ma, P. Y. Chen, and G. Klár, "Warp: Differentiable spatial computing for python," in *ACM SIGGRAPH 2024 Courses*, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3664475.3664543

[49] B. M. De Silva, K. Champion, M. Quade, J.-C. Loiseau, J. N. Kutz, and S. L. Brunton, "Pysindy: a python package for the sparse identification of nonlinear dynamics from data," *arXiv preprint arXiv:2004.08424*, 2020.

[50] G. Beintema, R. Tóth, and M. Schoukens, "Nonlinear state-space identification using deep encoder networks," in *Proceedings of the 3rd Conference on Learning for Dynamics and Control (L4DC)*, ser. Proceedings of Machine Learning

Preprint not peer reviewed

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813

Research, 2021, pp. 241–250. [Online]. Available: https://proceedings.mlr.press/v144/beintema21a.html

[51] W. R. Lacerda Junior, L. P. C. d. Andrade, S. C. P. Oliveira, and S. A. M. Martins, "Sysidentpy: A python package for system identification using narmax models," *Journal of Open Source Software*, vol. 5, no. 54, p. 2384, 2020. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.02384

[52] M. Hoffmann, M. Scherer, T. Hempel, A. Mardt, B. de Silva, B. E. Husic, S. Klus, H. Wu, N. Kutz, S. L. Brunton, and F. Noé, "Deeptime: a python library for machine learning dynamical models from time series data," *Machine Learning: Science and Technology*, vol. 3, no. 1, p. 015009, dec 2021. [Online]. Available: https://doi.org/10.1088/2632-2153/ac3de0

[53] J. Drgona, A. Tuor, J. Koch, M. Shapiro, B. Jacob, and D. Vrabie, "NeuroMANCER: Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations," 2023. [Online]. Available: https://github.com/pnnl/neuromancer

[54] G. P. Rosati Papini, "Neural network framework for modelling, control, and estimation of physical systems (nnodely)," Open-source software, 2026, gitHub repository; archived release available on Zenodo. [Online]. Available: https://github.com/tonegas/nnodely

[55] T. Takagi and M. Sugeno, "Fuzzy identification of systems and its applications to modeling and control," *IEEE transactions on systems, man, and cybernetics*, no. 1, pp. 116–132, 1985.

[56] J.-S. Jang, "Anfis: adaptive-network-based fuzzy inference system," *IEEE transactions on systems, man, and cybernetics*, vol. 23, no. 3, pp. 665–685, 1993.

[57] S. Sahoo, C. Lampert, and G. Martius, "Learning equations for extrapolation and control," in *International conference on machine learning*. Pmlr, 2018, pp. 4442–4450.

[58] M. J. Powell, "A method for nonlinear constraints in minimization problems," *Optimization*, pp. 283–298, 1969.

[59] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[60] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[61] G. Ronquillo-Lomeli, G. J. Ríos-Moreno, A. Gómez-Espinosa, L. A. Morales-Hernández, and M. Trejo-Perea, "Nonlinear identification of inverted pendulum system using volterra polynomials," *Mechanics Based Design of Structures and Machines*, vol. 44, no. 1-2, pp. 5–15, 2016. [Online]. Available: https://doi.org/10.1080/15397734.2015.1028551

[62] P. Gautam, "System identification of nonlinear inverted pendulum using artificial neural network," in *2016 International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, 2016, pp. 1–5.

[63] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[64] D. Schramm, M. Hiller, and R. Bardini, *Vehicle Dynamics: Modeling and Simulation*, 2nd ed. Berlin, Heidelberg: Springer, 2018.

[65] J. C. Kegelman, L. K. Harbott, and J. C. Gerdes, "2014 targa sixty-six," Stanford Digital Repository, Tech. Rep., 2016. [Online]. Available: http://purl.stanford.edu/hd122pw0365

Preprint not peer reviewed

This preprint research paper has not been peer reviewed. Electronic copy available at: https://ssrn.com/abstract=7634813