Neural Network Training

From Vanilla NNs to Physics-Informed Methods

Paolo Franceschi

IDSIA-USI-SUPSI, Lugano (Switzerland)

Email: paolo.franceschi@supsi.ch

Repository: https://github.com/paolofrance/piml_2026

Affiliation: IDSIA-USI-SUPSI

Contents

1 Introduction

1.1 From Vanilla Neural Networks to Physics-Aware Learning

A neural network is a parameterized function $f_\theta : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ built by composing affine maps with elementwise nonlinearities. Given a dataset $\{(x_i, y_i)\}_{i=1}^N$, training amounts to finding parameters θ that minimize a loss $\mathcal{L}(\theta)$ measuring the discrepancy between predictions $f_\theta(x_i)$ and targets y_i. The optimization is performed via gradient-based methods, with gradients computed efficiently by backpropagation, the reverse-mode application of the chain rule through the network's computation graph.

This paradigm has been spectacularly successful in domains where data is abundant and inexpensive, such as image recognition, language modelling, recommendation systems. In the physical sciences, however, the situation is fundamentally different. Experimental data are often scarce, noisy, and expensive to obtain; high-fidelity simulations are computationally prohibitive; and the underlying systems are governed by well-established physical laws expressed as partial differential equations (PDEs), conservation principles, or variational structures. A purely data-driven neural network trained in this regime tends to overfit, generalize poorly outside the training distribution, and produce predictions that violate basic physical constraints such as conservation of mass, momentum, or energy [1].

The recognition that physical prior knowledge can compensate for data scarcity, and conversely that data can compensate for incomplete physical models, has given rise to the field of physics-informed machine learning. The central insight is that no predictive model can be constructed without assumptions, so the question becomes which assumptions to encode and how to encode them. Karniadakis et al. [1] formalize this as the problem of introducing appropriate biases into a learning algorithm, and identify three complementary pathways: observational biases, inductive biases, and learning biases.

1.2 Three Pathways for Embedding Physics

Observational Biases

The simplest and most direct way to embed physics into a learning algorithm is through the data itself. Observational data drawn from a physical system implicitly encode the laws governing that system. If sufficient data are available to densely cover the input domain, a neural network can learn the underlying input–output mapping by interpolation, and the learned function will inherit, in a statistical sense, the symmetries, conservation laws, and structural regularities of the physics that produced the data [1].

This approach is appealing for its conceptual simplicity: standard supervised learning, applied to physics-respecting data, yields physics-respecting predictions. Its principal limitation is the volume of data required. Over-parameterized deep networks need large datasets to reliably internalize physical principles purely through observations, and in many scientific contexts such datasets are prohibitively expensive: they may require costly experiments, high-fidelity simulations, or rare in-situ measurements. Furthermore, predictions outside the training distribution remain unconstrained, so extrapolation is unreliable.

Inductive Biases

A more principled pathway is to design the network architecture itself so that physical constraints are satisfied by construction, regardless of training data. Predictions are then guaran-

teed to obey the encoded physics, even at points the network has never seen during training. Classical examples include convolutional neural networks [2], which enforce translation equivariance for image data; graph neural networks [3], which respect permutation symmetries of relational data; and equivariant networks [4], which encode rotation, reflection, or more general gauge symmetries.

For physical systems specifically, this approach has produced architectures that enforce symplecticity for Hamiltonian dynamics [5], anti-symmetry under particle exchange for fermionic wavefunctions [6], and Galilean invariance for turbulent flows [7]. The strength of inductive biases lies in their exactness: the imposed constraints are mathematical guarantees of the architecture, not approximations learned from data. The weakness is their narrowness: each architecture targets a specific class of symmetries, and extending the approach to systems with poorly understood or complex invariances remains difficult [1].

Learning Biases

The third pathway leaves the architecture generic but modifies the loss function to penalize predictions that violate physical constraints. Instead of designing a network that cannot violate the physics, one trains a network that is discouraged from doing so through soft penalty terms in the training objective. A typical loss takes the form

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{data}}(\theta) + \lambda \, \mathcal{L}_{\text{physics}}(\theta), \tag{1}$$

where $\mathcal{L}_{\text{data}}$ measures fit to observations and $\mathcal{L}_{\text{physics}}$ measures the residual of a governing equation, the violation of a conservation law, or any other physical desideratum.

This approach is enormously flexible: physical constraints expressed as differential equations, integral equations, variational principles, or even fractional operators can all be incorporated as long as they admit a differentiable residual [1]. The trade-off is that constraints are only approximately satisfied, and the relative weighting of competing terms (the λ in Equation 1) becomes a nontrivial hyperparameter, often the central pain point of practical training.

Hybrid Strategies

These three pathways are not mutually exclusive. The most successful real-world physics-informed models typically combine all three: they exploit observational data, encode known symmetries in the architecture, and impose remaining physical constraints through the loss [1]. The art lies in choosing the right balance for a given problem: baking in what is reliably known, softly enforcing what is approximately known, and learning from data what is unknown.

1.3 Three Representative Methods

This document reviews three influential physics-informed neural network architectures, each illustrating a different point along the bias spectrum.

Physics-Informed Neural Networks (PINNs). Introduced by Raissi, Perdikaris, and Karniadakis [8], PINNs sit squarely in the learning bias category. A generic feedforward network $u_\theta(x, t)$ is trained to satisfy a PDE by adding the PDE residual, evaluated at scattered collocation points via automatic differentiation, as a penalty term in the loss. Initial and boundary conditions enter as additional penalty terms. The architecture remains unconstrained; the physics is enforced only softly. This flexibility makes PINNs applicable to virtually any system whose governing equations are known, but at the cost of the loss-balancing problem and well-documented training pathologies for stiff or multiscale systems [9, 10].

Lagrangian Neural Networks (LNNs). Cranmer et al. [11] take a step further toward inductive bias. Rather than learning the dynamics directly, an LNN learns a scalar Lagrangian $L_\theta(q, \dot{q})$ and derives the equations of motion from the Euler–Lagrange equations. Because the dynamics are mathematically guaranteed to follow from a Lagrangian, conservation laws associated with continuous symmetries (via Noether's theorem) hold by construction. The price is computational: each forward pass requires the second derivatives and Hessian inversion implied by the Euler–Lagrange formulation. A closely related approach using Hamiltonians instead of Lagrangians, namely Hamiltonian Neural Networks (HNNs), was proposed concurrently by Greydanus et al. [12], with the same philosophy but cheaper first-order derivatives.

Deep Lagrangian Networks (DeLaN). Lutter, Ritter, and Peters [13] push the inductive bias still further by exploiting the specific structure of mechanical Lagrangians. For rigid-body systems, the Lagrangian decomposes as $L = \frac{1}{2}\dot{q}^\top \mathcal{M}(q)\dot{q} - V(q)$, where the inertia matrix $\mathcal{M}(q)$ must be symmetric and positive-definite for the system to be physically valid. DeLaN parameterizes $\mathcal{M}(q)$ directly via a Cholesky-like construction that guarantees these properties by design, and $V(q)$ via a separate scalar network. The result is a model that not only respects Lagrangian structure but additionally enforces the specific form of mechanical kinetic energy, yielding strong sample efficiency and direct compatibility with classical robotics control algorithms, at the cost of being restricted to systems where the rigid-body assumption holds.

1.4 The Spectrum of Physics-Informedness

Together, vanilla neural networks, PINNs, LNNs, and DeLaN form a natural progression of increasing physical commitment:

- Vanilla NN: no physics assumed; pure observational bias.
- PINN: governing equations enforced softly through the loss (learning bias).
- LNN / HNN: variational framework enforced architecturally (inductive bias, generic).
- DeLaN: mechanical Lagrangian structure enforced architecturally (inductive bias, specific).

Each step trades flexibility for built-in correctness: the more physics is baked in, the less data is needed and the better extrapolation tends to be, but the narrower the class of applicable systems becomes. There is no universally best choice, as the appropriate method depends on the available data, the completeness of physical knowledge, and the structural assumptions one is willing to make.

The remainder of this document develops the mathematics of each method in turn, starting from the training loop of a vanilla neural network and extending it progressively to incorporate physics through each of the three pathways.

2 Vanilla Neural Network Training

NN concepts

A neural network is a collection of interconnected "neurons" organized in layers. Each connection has a weight (a number), and each neuron has a bias. These weights and biases start as random numbers: the network knows nothing yet. Training is the process of finding the right values for all of them so the network produces useful outputs.

2.1 Notation

Consider a network with L layers. For layer l:

- $W^{(l)}$: weight matrix, shape $(n_l \times n_{l-1})$ where n_l is the number of neurons in layer l
- $b^{(l)}$: bias vector, shape $(n_l \times 1)$
- $a^{(l)}$: activations (output) of layer l
- $z^{(l)}$: pre-activation (the value before applying the activation function)
- σ: activation function (e.g., ReLU, sigmoid, tanh)
- x: input vector, equivalent to $a^{(0)}$
- y: true label (target)
- $\hat{y}$: prediction, equivalent to $a^{(L)}$
- $\mathcal{L}$: loss (scalar)
- η: learning rate

2.2 Step 1: Forward Pass

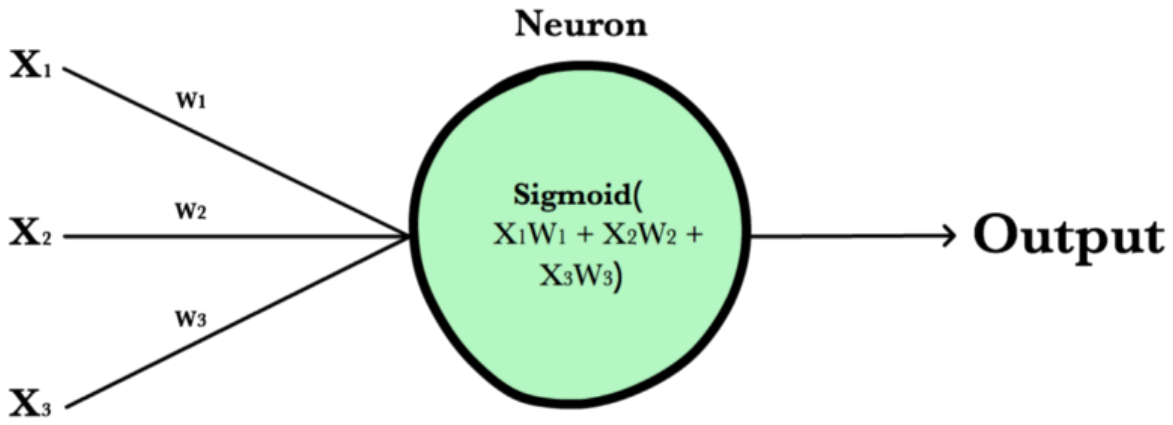

Figure 1: The basic neuron inputs and output

Forward Propagation

You feed an input into the network. Each neuron multiplies its inputs by its weights, adds a bias, and applies an activation function. The signal flows layer by layer until the final layer produces a prediction.

For each layer l from 1 to L, compute the pre-activation as a linear combination of the previous layer's outputs:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \tag{2}$$

Then apply the activation function elementwise:

$$a^{(l)} = \sigma(z^{(l)}) \tag{3}$$

The final layer's activation is the prediction:

$$\hat{y} = a^{(L)} \tag{4}$$

For a concrete example with a 3-layer network, the full forward pass is:

$$\hat{y} = \sigma\Big(W^{(3)}\,\sigma\Big(W^{(2)}\,\sigma\Big(W^{(1)}x + b^{(1)}\Big) + b^{(2)}\Big) + b^{(3)}\Big) \tag{5}$$

That nested structure is the entire network, simply repeated linear transformations interleaved with nonlinear "squishes."

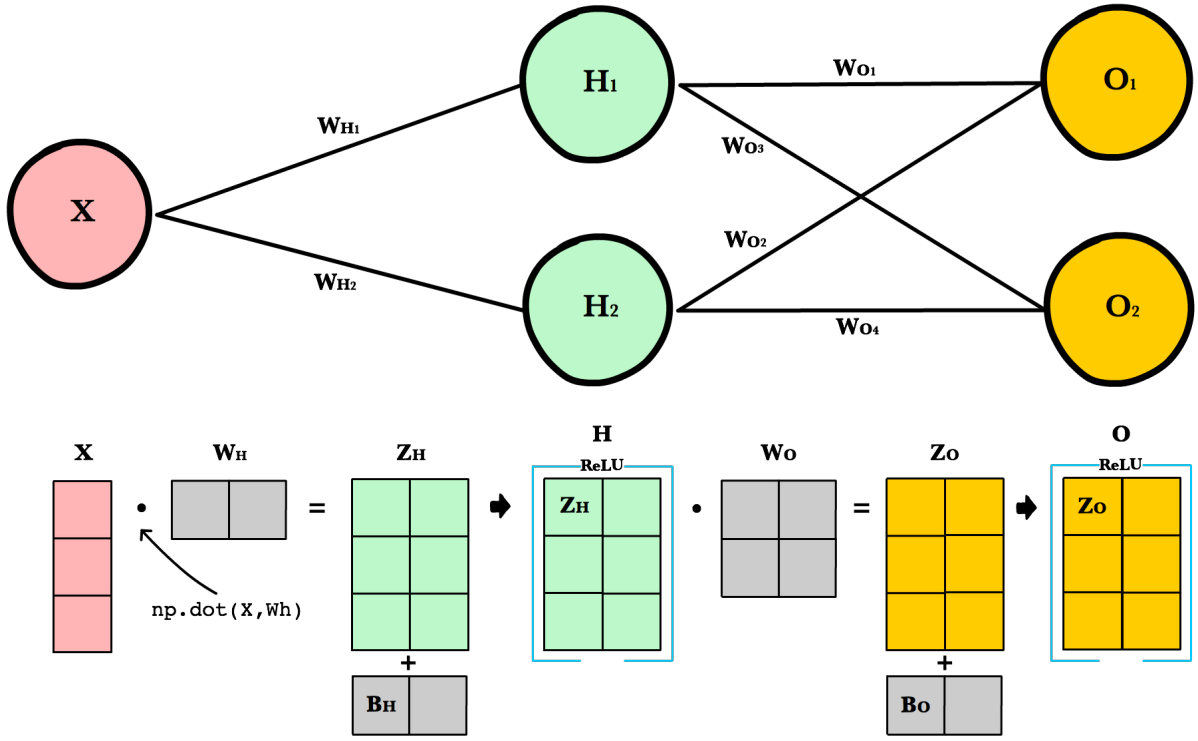

Figure 2: The forward pass for a MLP

2.3 Step 2: Loss Calculation

The loss measures how wrong $\hat{y}$ is compared to y. The choice depends on the task.

Mean Squared Error (regression):

$$\mathcal{L} = \frac{1}{2}\|\hat{y} - y\|^2 = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2 \tag{6}$$

Cross-Entropy (classification):

$$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \tag{7}$$

For a batch of N examples, you average:

$$\mathcal{L}_{\text{batch}} = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}^{(n)} \tag{8}$$

2.4 Step 3: Backpropagation

Back Propagation

Back prop

The goals of backpropagation are straightforward: adjust each weight in the network in proportion to how much it contributes to overall error.

This is the heart of the algorithm. The goal is to compute $\partial\mathcal{L}/\partial W^{(l)}$ and $\partial\mathcal{L}/\partial b^{(l)}$ for every layer. The trick is to compute these working backwards from the output, reusing intermediate quantities via the chain rule.

Define the error signal at layer l as the gradient of the loss with respect to the pre-activation:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} \tag{9}$$

At the output layer ($l = L$), this comes directly from the loss. For MSE with linear output:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot \sigma'(z^{(L)}) = (\hat{y} - y) \odot \sigma'(z^{(L)}) \tag{10}$$

where $\odot$ is elementwise multiplication and σ' is the derivative of the activation function.

For hidden layers ($l = L-1, L-2, \ldots, 1$), propagate the error backwards:

$$\delta^{(l)} = \left((W^{(l+1)})^\top \delta^{(l+1)}\right) \odot \sigma'(z^{(l)}) \tag{11}$$

This is the chain rule made concrete: the error at layer l depends on the error at layer $l+1$, multiplied through the weights that connect them, then modulated by how sensitive the activation function was at this layer.

Once you have $\delta^{(l)}$, the gradients fall out:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \, (a^{(l-1)})^\top \tag{12}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{13}$$

Intuition: a weight's gradient is the error signal at its output side, multiplied by the activation that came in on its input side. A weight matters more if it amplifies an erroneous output, and if its incoming signal was strong.

Why It Works (the Chain Rule)

Backprop is just the chain rule applied repeatedly. To compute $\partial \mathcal{L}/\partial W^{(1)}$ for the first layer, you'd in principle need:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdots \frac{\partial a^{(1)}}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial W^{(1)}} \tag{14}$$

That's a brutal product. Backprop's brilliance is that each $\delta^{(l)}$ caches the running product of the early factors, so you compute each piece exactly once.

2.5 Step 4: Weight Update

With gradients in hand, you update every parameter.

Vanilla gradient descent:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}} \tag{15}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}} \tag{16}$$

The minus sign is critical: gradients point uphill (toward higher loss), so you step opposite to descend.

SGD with momentum ($\beta \approx 0.9$) adds a velocity term that accumulates past gradients, helping push through flat regions and damp oscillations:

$$v^{(l)} \leftarrow \beta \, v^{(l)} + \frac{\partial \mathcal{L}}{\partial W^{(l)}} \tag{17}$$

$$W^{(l)} \leftarrow W^{(l)} - \eta \, v^{(l)} \tag{18}$$

Adam, the most popular optimizer in practice, tracks both a running average of gradients (first moment) and squared gradients (second moment), and adapts the effective learning rate per parameter:

$$m_t = \beta_1 \, m_{t-1} + (1 - \beta_1) \, g_t \tag{19}$$

$$v_t = \beta_2 \, v_{t-1} + (1 - \beta_2) \, g_t^2 \tag{20}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{21}$$

$$W \leftarrow W - \eta \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{22}$$

where g_t is the current gradient, $\beta_1 \approx 0.9$, $\beta_2 \approx 0.999$, and $\epsilon \approx 10^{-8}$ prevents division by zero. The bias-correction terms $(\hat{m}_t, \hat{v}_t)$ compensate for the moments being initialized at zero.

2.6 Putting It All Together

For each batch of training data, the full update cycle is:

1. Forward: $a^{(l)} = \sigma(W^{(l)} a^{(l-1)} + b^{(l)})$ for $l = 1, \ldots, L$
2. Loss: $\mathcal{L} = \text{loss\_fn}(a^{(L)}, y)$
3. Backward: $\delta^{(L)}$ at the output, then $\delta^{(l)} = (W^{(l+1)})^\top \delta^{(l+1)} \odot \sigma'(z^{(l)})$ propagating backwards
4. Gradients: $\partial \mathcal{L} / \partial W^{(l)} = \delta^{(l)} (a^{(l-1)})^\top$ and $\partial \mathcal{L} / \partial b^{(l)} = \delta^{(l)}$
5. Update: $W^{(l)} \leftarrow W^{(l)} - \eta \cdot \partial \mathcal{L} / \partial W^{(l)}$ (or use Adam, etc.)

Repeat for every batch, every epoch, until convergence.

3 Physics-Informed Neural Networks (PINNs)

pinn matlab

The core loop is the same (forward pass, loss, backprop, update), but the loss function is fundamentally different, and that changes almost everything downstream.

3.1 A Key Tool: Differentiating Through the Input

In vanilla training, you only ever differentiate the loss with respect to the parameters (W, b). In PINNs, you also need to differentiate the network's output with respect to its input. This is what automatic differentiation lets you do for free.

If the network is a function $u_\theta(x)$ with input coordinates $x \in \Omega \subset \mathbb{R}^d$ (which may include time as one component) and parameters θ, autodiff lets you compute any derivative $\partial^{|\alpha|} u_\theta / \partial x^\alpha$ as a differentiable expression. Crucially, those derivatives are themselves functions of θ, so you can backprop through them.

3.2 Generic Problem Statement

A PINN targets a generic differential problem of the form

$$\mathcal{N}[u](x) = 0, \quad x \in \Omega, \tag{23}$$

where $\mathcal{N}$ is a (possibly nonlinear) differential operator and $u : \Omega \to \mathbb{R}^{d_u}$ is the unknown field. The problem is closed by a boundary/initial condition operator

$$\mathcal{B}[u](x) = 0, \quad x \in \partial\Omega \cup \Omega_0, \tag{24}$$

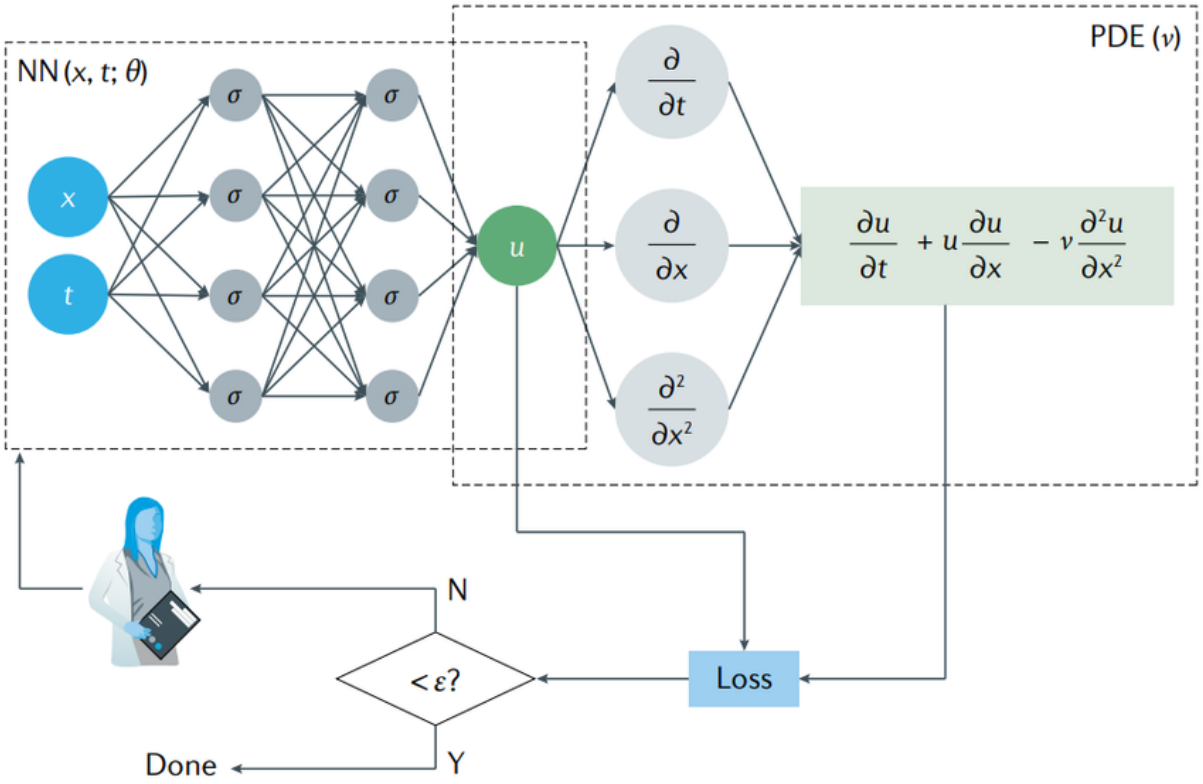

Figure 3: The structure of a PINN, from [1]

which collects whichever Dirichlet, Neumann, periodic, or initial conditions the problem requires. Equations (23)-(24) cover ODEs, PDEs, and systems of either; the worked examples in Sections 3.6 and 3.7 instantiate $\mathcal{N}$ as a first-order and a second-order ODE, respectively.

The network $u_\theta(x)$ approximates u. Training amounts to penalising the violation of (23)-(24) at sampled points, plus optionally fitting observational data when available.

3.3 Loss Components

Physics residual loss. Pick a set of collocation points $\{x_i^r\}_{i=1}^{N_r}$ scattered through Ω and penalise the squared residual of (23):

$$\mathcal{L}_{\text{phys}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \|\mathcal{N}[u_\theta](x_i^r)\|^2 . \tag{25}$$

Boundary/initial condition loss. At points $\{x_i^{bc}\}$ on $\partial\Omega \cup \Omega_0$:

$$\mathcal{L}_{\text{bc}} = \frac{1}{N_{bc}} \sum_{i=1}^{N_{bc}} \left\| \mathcal{B}[u_\theta](x_i^{bc}) \right\|^2 . \tag{26}$$

Data loss, if observations $\{(x_i^d, u_i^d)\}$ are available:

$$\mathcal{L}_{\text{data}} = \frac{1}{N_d} \sum_{i=1}^{N_d} \left\| u_\theta(x_i^d) - u_i^d \right\|^2 . \tag{27}$$

Total PINN loss:

$$\boxed{\mathcal{L}_{\text{PINN}} = \lambda_r \mathcal{L}_{\text{phys}} + \lambda_{bc} \mathcal{L}_{\text{bc}} + \lambda_d \mathcal{L}_{\text{data}}} \tag{28}$$

The λ's are weighting hyperparameters, and balancing them is the central pain point of PINN training. Some methods learn them adaptively. A common simplification when $\mathcal{N}$ has a natural scale (e.g. a stiffness or a rate constant) is to normalise the residual by that scale so that all loss terms are $\mathcal{O}(1)$ from the start; this removes one degree of freedom from the λ-tuning problem.

3.4 Backprop for a PINN

The gradient of the data and boundary terms with respect to θ is standard, following the same steps as vanilla NN backprop. The non-trivial part is the physics residual.

Denote $r_i = \mathcal{N}[u_\theta](x_i^r)$. The contribution to the gradient is

$$\frac{\partial \mathcal{L}_{\text{phys}}}{\partial \theta} = \frac{2}{N_r} \sum_i r_i^\top \frac{\partial r_i}{\partial \theta}. \tag{29}$$

The last term, $\partial r_i / \partial \theta$, requires differentiating derivatives of the network output with respect to θ. Mechanically this is "autodiff inside autodiff": a first autodiff pass over the input coordinates assembles $\mathcal{N}[u_\theta]$ as a function of θ, and a second pass backpropagates through that construction to obtain the weight gradient. Every operator $\mathcal{N}$ that can be written in terms of differentiable elementary operations is handled by this mechanism without any problem-specific implementation effort beyond writing down $\mathcal{N}$ itself.

3.5 How PINN Training Differs from Vanilla

- Multiple loss terms must be balanced via λ hyperparameters.
- Network input = coordinates (x, t), output = field value (not features $\rightarrow$ labels).
- Forward pass requires autodiff through the network's input derivatives.
- Trained on collocation points; little or no labeled data needed.

3.6 Example 1: Newton's Law of Cooling

The exposition of this example is adapted from the Medium tutorial by Wolf [14], recast in the notation of the present document and extended with a PINN-ID variant.

Newton's law of cooling is the simplest possible test bed for a PINN: a first-order, scalar ODE with a single unknown parameter. The governing equation is

$$\frac{dT}{dt} = R\,(T_{\text{env}} - T), \tag{30}$$

whose analytical solution is the exponential $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})\, e^{-Rt}$. We set $T_{\text{env}} = 25\,^\circ\text{C}$, $T_0 = 100\,^\circ\text{C}$, $R = 0.005\,\text{s}^{-1}$, and observe only 10 noisy temperature readings (noise $\sigma = 2\,^\circ\text{C}$) spread over the first 300 s, then evaluate over the full 1000 s interval.

Three models. We compare three trajectory networks, all sharing the same architecture $(t \rightarrow T_\theta(t))$:

- NN: data loss only; no knowledge of the ODE.
- PINN: data loss plus the physics residual of Eq. (30), with R treated as a known constant.
- PINN-ID: same residual, but R is an additional learnable scalar parameter initialised at zero.

Physics residual. The residual operator for Eq. (30) is

$$\mathcal{R}[T_\theta](t) = \frac{dT_\theta}{dt}(t) - R\left(T_{\text{env}} - T_\theta(t)\right). \tag{31}$$

Because the ODE is first-order, a single autodiff pass through the network's input t suffices to evaluate dT_θ/dt. The total loss is

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \lambda\,\mathcal{L}_{\text{phys}}, \qquad \mathcal{L}_{\text{phys}} = \frac{1}{N_c}\sum_{i=1}^{N_c}\left|\mathcal{R}[T_\theta](t_i^c)\right|^2, \tag{32}$$

where $\{t_i^c\}$ are collocation points sampled uniformly over $[0,\ 1000\,\text{s}]$. No residual normalisation is needed here because $R \approx 5 \times 10^{-3}$ keeps all terms naturally $\mathcal{O}(1)$.

Parameter identification with PINN-ID. Making R a learnable `nn.Parameter` turns the PINN into a system identification tool. The physics residual now provides a gradient signal that pulls R towards the value for which the ODE is exactly satisfied; no extra labelled data is required beyond the 10 noisy observations. To ensure positivity, R is reparametrised as $R = e^\rho$ with ρ unconstrained, and ρ is initialised at a value far from the truth. The convergence of R towards its true value can be tracked alongside the training loss.

Key result. The plain NN overfits the 10 noisy observations and diverges as t grows large, predicting temperatures well below T_{env}. The PINN recovers the correct exponential decay and converges to equilibrium, because the collocation loss penalises any prediction that violates Eq. (30) across the full evaluation domain. PINN-ID simultaneously achieves good trajectory fit and accurate parameter recovery, demonstrating that the ODE residual encodes enough structure to act as a self-supervised identification signal even under measurement noise.

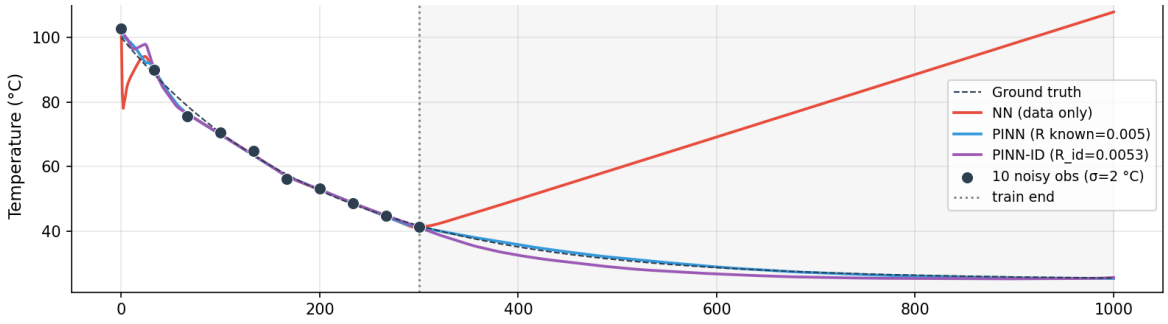

Figure 4: Newton's law of cooling. Temperature predictions of NN, PINN, and PINN-ID against the noisy observations and ground truth. Right: convergence of the learned cooling rate R during PINN-ID training; the dashed line marks the true value.

3.7 Example 2: Damped Mass-Spring-Damper (MCK)

The second example steps up to a second-order ODE and introduces a key implementation detail: residual normalisation. The setup follows the pedagogical exposition of Moseley [15], adapted here to the MCK form (m, c, k) with explicit residual normalisation by k.

The system is an underdamped mass-spring-damper

$$m\ddot{x} + c\,\dot{x} + k\,x = 0, \qquad m = 1\,kg, \quad c = 4\,Ns/m, \quad k = 400\,N/m, \tag{33}$$

with initial conditions $x(0) = 1\,\text{m}$, $\dot{x}(0) = 0$. We observe 10 noiseless measurements confined to $t \in [0,\ 0.36\,\text{s}]$, roughly one oscillation period, and ask both models to reconstruct the trajectory over the much longer window $[0,\ 1\,\text{s}]$.

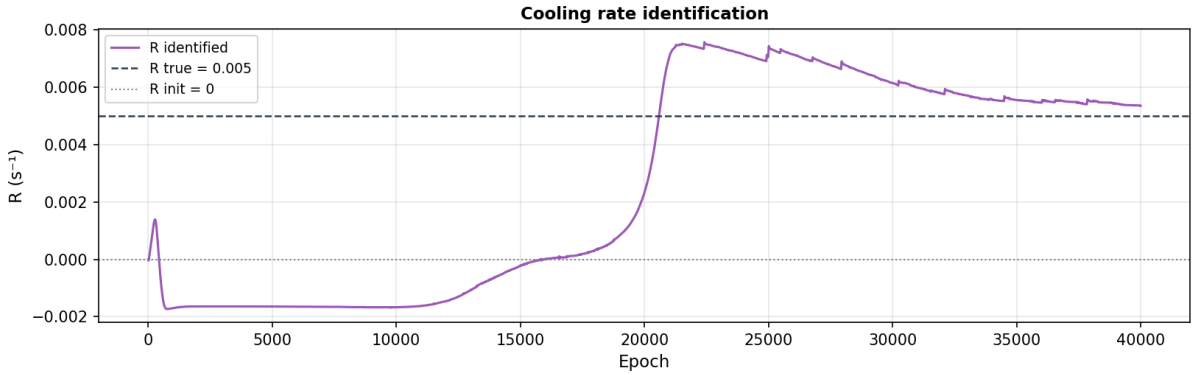

Figure 5: Convergence of the learned cooling rate R during PINN-ID training.

Second-order residual via chained autodiff. Because Eq. (33) involves $\ddot{x}$, the residual requires differentiating the network output twice with respect to t. Autodiff handles this by applying the same gradient computation a second time to the already-differentiated expression, a process sometimes called higher-order or nested autodiff. The residual operator is

$$\mathcal{R}[x_\theta](t) = m\,\ddot{x}_\theta(t) + c\,\dot{x}_\theta(t) + k\,x_\theta(t). \tag{34}$$

Residual normalisation. With $k = 400\,\mathrm{N/m}$, the spring term $k\,x \approx 400\,x$ is two orders of magnitude larger than typical displacement values. Without normalisation, the physics loss completely dominates $\mathcal{L}_{\text{data}}$ from the very first epoch, making the data term effectively invisible during optimisation. The fix is to divide the residual by k before squaring it:

$$\tilde{\mathcal{R}}[x_\theta](t) = \frac{m\,\ddot{x}_\theta + c\,\dot{x}_\theta + k\,x_\theta}{k}, \tag{35}$$

which brings every term to $\mathcal{O}(1)$ regardless of the system's stiffness. This normalisation step is crucial whenever the ODE coefficients span multiple orders of magnitude.

Collocation domain. A critical subtlety is where collocation points are placed. If they are drawn only from the training window $[0,\, 0.36\,\mathrm{s}]$, the physics residual is never evaluated in the region we actually care about, and the network is free to predict anything beyond that window. Collocation points must be sampled uniformly over the full evaluation domain $[0,\, 1\,\mathrm{s}]$ so that the ODE is enforced everywhere, not just where data exist.

Key result. The plain NN interpolates perfectly within the training window but has no reason to continue the decaying oscillation beyond $t = 0.36\,\mathrm{s}$: it simply collapses to a constant or diverges. The PINN, constrained by the ODE everywhere in $[0,\, 1\,\mathrm{s}]$, correctly extrapolates the amplitude decay and phase of the oscillation far past the last observation.

4 Lagrangian Neural Networks (LNNs)

LNNs take a more elegant, structural approach than PINNs. Instead of penalizing violations of an equation, they make it architecturally impossible for the network to violate certain physical laws.

4.1 The Core Idea

Classical mechanics offers a remarkable result: the dynamics of a very large class of physical systems can be derived from a single scalar function, the Lagrangian. Let the configuration of

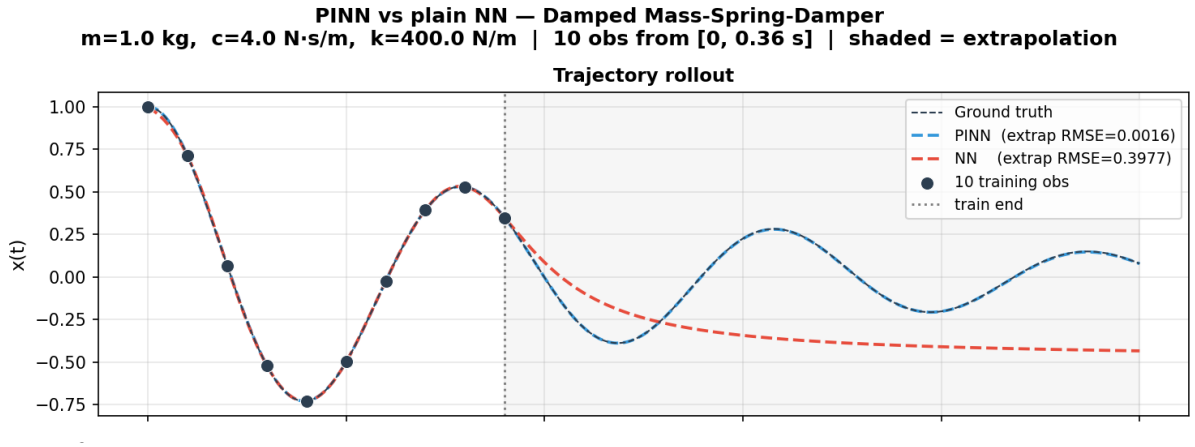

Figure 6: Comparison between PINN and vanilla NN predictions

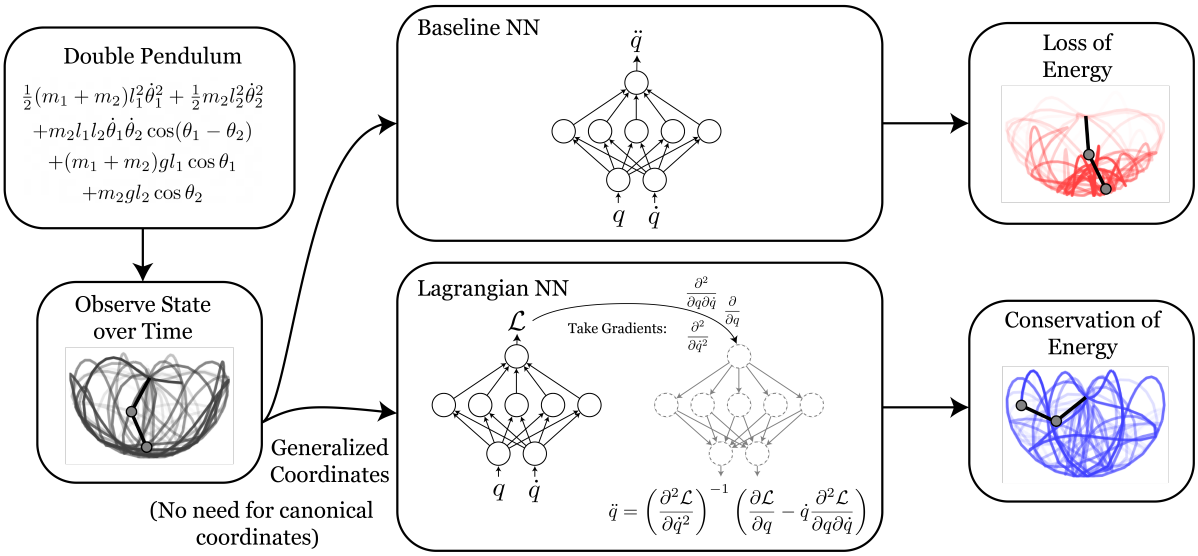

Figure 7: The structure of a from [11]

a system be described by n generalised coordinates $q = (q_1, \ldots, q_n) \in \mathcal{Q}$, with corresponding generalised velocities $\dot{q} \in T_q\mathcal{Q}$. The Lagrangian

$$L : T\mathcal{Q} \to \mathbb{R}, \qquad L(q, \dot{q}) = T(q, \dot{q}) - V(q), \tag{36}$$

is the difference between kinetic energy T and potential energy V.

Hamilton's principle states that the trajectory followed by the system between two fixed configurations $q(t_0)$ and $q(t_1)$ is a stationary point of the *action*

$$S[q] = \int_{t_0}^{t_1} L(q(t), \dot{q}(t))\, dt. \tag{37}$$

Imposing $\delta S = 0$ for all admissible variations δq with $\delta q(t_0) = \delta q(t_1) = 0$ and integrating by parts yields the Euler–Lagrange equations

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0, \qquad i = 1, \ldots, n, \tag{38}$$

or in vector form

$$\frac{d}{dt}(\nabla_{\dot{q}} L) - \nabla_q L = 0. \tag{39}$$

This single variational principle encodes Newton's second law, conservation laws (via Noether's theorem, one for each continuous symmetry of L), and the constraint structure of the system through the choice of q.

The LNN insight. Rather than learning the map $(q, \dot{q}) \mapsto \ddot{q}$ directly, parameterise the Lagrangian itself:

$$L_\theta(q, \dot{q}) : \mathbb{R}^{2n} \to \mathbb{R}, \tag{40}$$

and recover the dynamics by substituting L_θ into (38). Any trajectory generated this way satisfies a variational principle by construction; if L_θ has no explicit time dependence, energy is conserved automatically.

4.2 Equation of Motion

To use (38) as a dynamics predictor, we must isolate $\ddot{q}$. Expanding the total time derivative via the chain rule (for a Lagrangian with no explicit time dependence):

$$\frac{d}{dt}\nabla_{\dot{q}} L = \nabla_{\dot{q}}\nabla_{\dot{q}}^\top L \cdot \ddot{q} + \nabla_q \nabla_{\dot{q}}^\top L \cdot \dot{q}, \tag{41}$$

where $\nabla_{\dot{q}}\nabla_{\dot{q}}^\top L \in \mathbb{R}^{n\times n}$ and $\nabla_q \nabla_{\dot{q}}^\top L \in \mathbb{R}^{n\times n}$ are mixed second partials. Substituting into (38) gives the implicit equation of motion

$$\nabla_{\dot{q}}\nabla_{\dot{q}}^\top L \cdot \ddot{q} + \nabla_q \nabla_{\dot{q}}^\top L \cdot \dot{q} - \nabla_q L = 0. \tag{42}$$

Define the velocity Hessian

$$M_{\text{LNN}}(q, \dot{q}) := \nabla_{\dot{q}}\nabla_{\dot{q}}^\top L_\theta(q, \dot{q}) \in \mathbb{R}^{n\times n}. \tag{43}$$

When M_{LNN} is invertible, (42) solves to the explicit form

$$\boxed{\hat{\ddot{q}} = M_{\text{LNN}}^{-1}(q, \dot{q}) \left[\nabla_q L_\theta - \left(\nabla_q \nabla_{\dot{q}}^\top L_\theta\right)\dot{q}\right]} \tag{44}$$

which has the familiar Newtonian shape: a generalised mass term multiplying $\ddot{q}$ on the left, generalised forces (kinetic coupling and conservative force) on the right.

For a mechanical system with kinetic energy $T = \frac{1}{2}\dot{q}^\top \mathcal{M}(q)\dot{q}$ and potential $V(q)$, M_{LNN} recovers the true inertia matrix $\mathcal{M}(q)$ and the right-hand side recovers the Coriolis/centrifugal terms and the conservative force $-\nabla_q V$. The LNN is therefore expressive enough in principle to learn any mechanical system.

Caveat: the mass matrix is implicit. Nothing in the LNN architecture guarantees that M_{LNN} is symmetric positive definite, or that it depends only on q (not on $\dot{q}$, as physical inertia matrices must). The vanilla LNN relies on the data to push the network towards a Lagrangian whose Hessian is well-behaved. When the data is sparse or noisy, M_{LNN} can drift towards being singular or indefinite during training, in which case (44) becomes unstable. This is precisely the failure mode that motivates DeLaN's structured parameterisation in Section 5.

4.3 Loss Function

Given training data of states and their accelerations $\{(q_i, \dot{q}_i, \ddot{q}_i)\}_{i=1}^N$:

$$\boxed{\mathcal{L}_{\text{LNN}} = \frac{1}{N}\sum_{i=1}^{N} \left\| \hat{\ddot{q}}_i - \ddot{q}_i \right\|^2} \tag{45}$$

where $\hat{\ddot{q}}_i$ is computed via the formula above evaluated at $(q_i, \dot{q}_i)$.

4.4 How LNN Training Differs from Vanilla

- The network outputs a scalar (energy), not a prediction directly.
- Forward pass requires first and second derivatives of the network output.
- Forward pass requires inverting an $n \times n$ matrix (the velocity Hessian).
- Single, clean loss term, with no λ-balancing problem like PINNs.
- Conservation laws are guaranteed by construction, not penalized.

4.5 Example A: Simple Pendulum

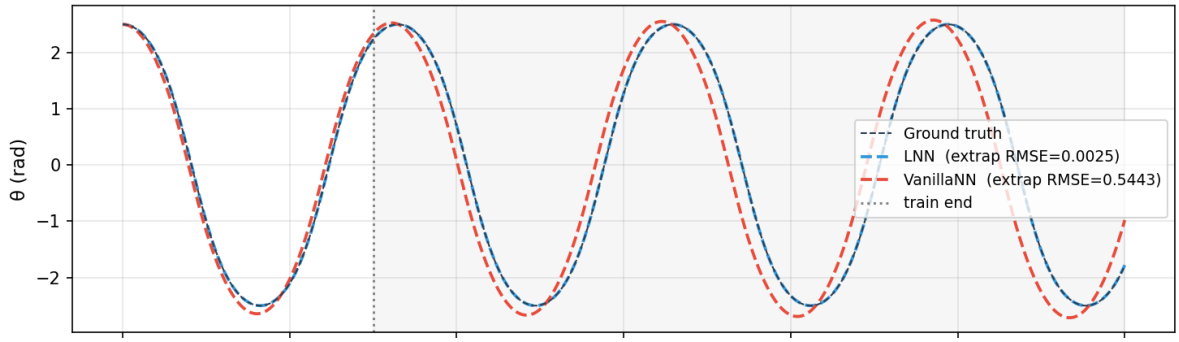

Figure 8: LNN vs. VanillaNN on the simple pendulum. Left: angle trajectory over 12 s (training window 0–3 s, shaded region is extrapolation). Right: absolute error on log scale.

The first example demonstrates the core advantage of the LNN on a classic 1-DOF nonlinear system: the simple pendulum,

$$\ddot{\theta} = -\frac{g}{\ell}\sin\theta, \qquad g = 9.81\,\mathrm{m/s^2}, \quad \ell = 1\,\mathrm{m}. \tag{46}$$

The pendulum and spring-pendulum setups in this section and the companion script `lnn_vs_delan.py` were inspired by Magnus Ross' blog post [16] and his PyTorch implementation [17], adapted to the comparison structure used throughout this document. The initial condition is set to $\theta_0 = 2.5\,\mathrm{rad}$, which is deliberately chosen near the separatrix, the boundary between oscillatory and rotational motion, where the trajectory is strongly nonlinear and sensitive to errors.

Data. Both models are trained on only $N = 20$ samples of $(q_i, \dot{q}_i, \ddot{q}_i)$ collected from a single trajectory over $t \in [0, 3]\,\mathrm{s}$. This is an unusually small dataset: the goal is to test whether the inductive bias of the Lagrangian structure compensates for the lack of data.

Training. Both models (LNN and VanillaNN, each with 3 hidden layers of 64 units) are trained to minimize the mean-squared error between predicted and true accelerations:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \|\hat{\ddot{q}}_i - \ddot{q}_i\|^2. \tag{47}$$

The loss is identical for both. The structural difference is in how $\hat{\ddot{q}}_i$ is computed: VanillaNN predicts it directly from a standard feedforward network; LNN derives it from the Euler-Lagrange equation applied to a learned scalar Lagrangian $L_\theta(q, \dot{q})$.

Evaluation. After training, a 4th-order Runge-Kutta integrator rolls both models forward for $t \in [0, 12]\,\mathrm{s}$, four times the training horizon.

Key result. The LNN stays on the correct energy shell throughout the extrapolation window. Since the Lagrangian structure guarantees energy conservation by construction, any errors in the predicted acceleration cannot accumulate into energy drift: the trajectory remains physically plausible. The VanillaNN, having no such constraint, begins to drift immediately after the training window and eventually diverges. This is visible both in the angle trajectory and in the absolute error plot (Figure 8), which shows the LNN error remaining bounded while the VanillaNN error grows exponentially.

5 Deep Lagrangian Networks (DeLaN)

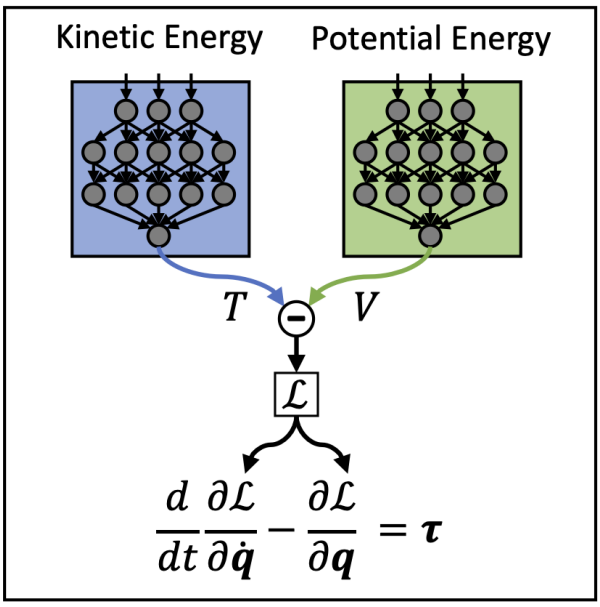

Figure 9: The structure of a DeLaN

DeLaN, introduced by Lutter, Ritter, and Peters in 2019, is a specialized LNN designed for rigid-body mechanics, particularly robotics. The key insight: instead of learning a generic scalar Lagrangian, DeLaN exploits the known structure of mechanical Lagrangians to enforce physical correctness by construction.

5.1 From the Generic LNN to a Mechanical Lagrangian

The LNN equation of motion derived in Section 4.2 is generic: it holds for any scalar Lagrangian $L_\theta(q, \dot{q})$, mechanical or not. DeLaN restricts the hypothesis class by assuming L_θ has the specific form of a rigid-body Lagrangian, and then specialises every term of the generic formula accordingly.

Assume L takes the mechanical form

$$L(q, \dot{q}) = \tfrac{1}{2}\, \dot{q}^\top \mathcal{M}(q)\, \dot{q} - V(q), \tag{48}$$

with $\mathcal{M}(q) = \mathcal{M}(q)^\top \succ 0$ depending only on q. Substituting (48) into each ingredient of the §4.2 formulas gives:

- Velocity Hessian (the LNN "mass matrix") collapses to the true inertia:

$$M_{\text{LNN}} = \nabla_{\dot{q}} \nabla_{\dot{q}}^\top L \;=\; \mathcal{M}(q). \tag{49}$$

The potential $V(q)$ contributes nothing because it does not depend on $\dot{q}$.

- Velocity gradient gives the generalised momentum:

$$\nabla_{\dot{q}} L \;=\; \mathcal{M}(q)\,\dot{q}. \tag{50}$$

- Configuration gradient splits into a kinetic-coupling term and a conservative force:

$$\nabla_q L \;=\; \tfrac{1}{2}\,\nabla_q\Big(\dot{q}^\top \mathcal{M}(q)\,\dot{q}\Big) \;-\; \nabla_q V(q). \tag{51}$$

- Mixed second derivative, contracted with $\dot{q}$, becomes

$$\big(\nabla_q \nabla_{\dot{q}}^\top L\big)\,\dot{q} \;=\; \dot{\mathcal{M}}(q)\,\dot{q}, \qquad \dot{\mathcal{M}}(q) := \sum_k \frac{\partial \mathcal{M}}{\partial q_k}\,\dot{q}_k. \tag{52}$$

Inserting these into the implicit LNN equation (42) yields

$$\mathcal{M}(q)\,\ddot{q} \;+\; \dot{\mathcal{M}}(q)\,\dot{q} \;-\; \tfrac{1}{2}\,\nabla_q\Big(\dot{q}^\top \mathcal{M}(q)\,\dot{q}\Big) \;+\; \nabla_q V(q) \;=\; 0, \tag{53}$$

which, defining the Coriolis/centrifugal vector and the gravitational force as

$$c(q,\dot{q}) \;:=\; \dot{\mathcal{M}}(q)\,\dot{q} \;-\; \tfrac{1}{2}\,\nabla_q\Big(\dot{q}^\top \mathcal{M}(q)\,\dot{q}\Big), \qquad g(q) \;:=\; \nabla_q V(q), \tag{54}$$

collapses to the standard manipulator equation

$$\boxed{\mathcal{M}(q)\,\ddot{q} \;+\; c(q,\dot{q}) \;+\; g(q) \;=\; \tau,} \tag{55}$$

where τ is any external generalised force not derivable from V (motor torques, in robotics). Setting $\tau = 0$ recovers the autonomous case.

Equation (55) is therefore §4.2 formula when L is restricted to the mechanical class. DeLaN's contribution is to enforce that restriction architecturally and to parameterise $\mathcal{M}(q)$ and $V(q)$ separately, as described next.

5.2 Structured Lagrangian

For a rigid-body system with generalized coordinates $q \in \mathbb{R}^n$ and velocities $\dot{q} \in \mathbb{R}^n$, the Lagrangian has the specific form derived in (48):

$$L(q,\dot{q}) = T(q,\dot{q}) - V(q) = \tfrac{1}{2}\,\dot{q}^\top \mathcal{M}(q)\,\dot{q} - V(q), \tag{56}$$

with $\mathcal{M}(q) \in \mathbb{R}^{n\times n}$ the inertia (mass) matrix and $V(q) \in \mathbb{R}$ the potential energy. For the system to be physically valid, $\mathcal{M}(q)$ must be symmetric and positive definite.

The dynamics follow the manipulator equation (55), with the Coriolis/centrifugal term given by

$$c(q,\dot{q}) = \dot{\mathcal{M}}(q)\,\dot{q} - \tfrac{1}{2}\,\nabla_q\Big(\dot{q}^\top \mathcal{M}(q)\,\dot{q}\Big), \tag{57}$$

and the gravitational force by $g(q) = \nabla_q V(q)$.

5.3 Architecture: Two Networks

Network 1: the inertia matrix network. A network $f_\theta(q)$ outputs the entries of a lower-triangular matrix $L_d(q) \in \mathbb{R}^{n\times n}$. The inertia matrix is then constructed as:

$$\mathcal{M}(q) = L_d(q) L_d(q)^\top + \epsilon I \tag{58}$$

This is a Cholesky-like construction. By multiplying a triangular matrix by its transpose, you get a symmetric positive semi-definite matrix automatically. Adding ϵI (with small $\epsilon > 0$) ensures strict positive definiteness.

To be concrete, if $n = 2$:

$$L_d(q) = \begin{pmatrix} l_{11}(q) & 0 \\ l_{21}(q) & l_{22}(q) \end{pmatrix} \tag{59}$$

The diagonal entries $l_{ii}(q)$ are typically passed through a softplus or exponential to keep them strictly positive.

Network 2: the potential energy network. A scalar-valued network $V_\varphi(q) : \mathbb{R}^n \to \mathbb{R}$ that outputs the potential energy.

The total parameters are $\{\theta, \varphi\}$, the weights of both networks.

5.4 Forward Pass

Given current state $(q, \dot{q}, \ddot{q})$, DeLaN computes the predicted torque $\hat{\tau}$ as follows:

1. Compute $L_d(q) = f_\theta(q)$ and form $\mathcal{M}(q) = L_d L_d^\top + \epsilon I$.
2. Compute $V_\varphi(q)$.
3. Compute the gravitational force using autodiff:

$$g(q) = \frac{\partial V_\varphi}{\partial q}$$

4. Compute the time derivative of H:

$$\dot{\mathcal{M}}(q) = \sum_{k=1}^{n} \frac{\partial \mathcal{M}}{\partial q_k} \dot{q}_k$$

5. Compute the kinetic energy gradient term:

$$\frac{\partial}{\partial q} \left(\frac{1}{2} \dot{q}^\top \mathcal{M}(q) \dot{q} \right)$$

6. Assemble the Coriolis force:

$$c(q, \dot{q}) = \dot{\mathcal{M}}(q)\dot{q} - \frac{1}{2} \frac{\partial}{\partial q} \left(\dot{q}^\top \mathcal{M}(q) \dot{q} \right)$$

7. Assemble the predicted torque:

$$\hat{\tau} = \mathcal{M}(q)\ddot{q} + c(q, \dot{q}) + g(q)$$

5.5 Loss Function

DeLaN is typically trained on trajectories of $(q, \dot{q}, \ddot{q}, \tau)$. The forward dynamics loss is:

$$\mathcal{L}_{\text{forward}} = \frac{1}{N} \sum_{i=1}^{N} \|\hat{\tau}_i - \tau_i\|^2 \tag{60}$$

Some DeLaN variants add an inverse dynamics loss by also computing predicted accelerations:

$$\hat{\ddot{q}} = \mathcal{M}(q)^{-1} \left(\tau - c(q, \dot{q}) - g(q) \right) \tag{61}$$

$$\mathcal{L}_{\text{inverse}} = \frac{1}{N}\sum_{i=1}^{N}\left\|\hat{\ddot{q}}_i - \ddot{q}_i\right\|^2 \tag{62}$$

The combined loss is:

$$\boxed{\mathcal{L}_{\text{DeLaN}} = \lambda_f \mathcal{L}_{\text{forward}} + \lambda_i \mathcal{L}_{\text{inverse}}} \tag{63}$$

Both losses are differentiable through all the autodiff machinery: backprop flows back through the matrix inverse, the Coriolis assembly, and the Cholesky construction, all the way to θ and φ.

5.6 Why the Cholesky Trick Matters

Compare DeLaN to a vanilla LNN. A vanilla LNN learns $L_\theta(q, \dot{q})$ as a generic scalar function. To extract the mass matrix, you compute the Hessian:

$$M_{\text{LNN}}(q, \dot{q}) = \frac{\partial^2 L_\theta}{\partial \dot{q}^2}$$

Nothing prevents this Hessian from being:

- Velocity-dependent (real mechanical mass matrices depend only on q, not $\dot{q}$)
- Asymmetric (numerical autodiff can give numerically asymmetric Hessians)
- Indefinite or singular (catastrophic, as the dynamics become unphysical or non-invertible)

DeLaN sidesteps all three problems by directly parameterizing $\mathcal{M}(q)$ as a function of position only, constructed to be symmetric positive-definite by design. This is the textbook example of structural inductive bias: instead of penalizing wrong outputs in the loss, you make wrong outputs unrepresentable.

5.7 DeLaN vs Vanilla LNN

Aspect	Vanilla LNN	DeLaN
Network output	Single scalar $L_\theta(q, \dot{q})$	$L_d(q) \to \mathcal{M}(q)$, and $V_\varphi(q)$
Mass matrix	Implicit Hessian $\nabla_{\dot{q}} \nabla_{\dot{q}}^\top L_\theta$	Explicit: $\mathcal{M} = L_d L_d^\top + \epsilon I$
Symmetric / PD?	Not guaranteed	Guaranteed by construction
Velocity dependence	May depend on $\dot{q}$ (unphysical)	Depends only on q (correct)
Required derivatives	First + second order autodiff	First order autodiff only
Inductive bias	Generic Lagrangian	Mechanical Lagrangian (rigid body)

5.8 Example: Spring Pendulum, DeLaN vs. VanillaNN

The example applies DeLaN and a VanillaNN to the spring pendulum, a 2-DOF conservative system described in polar coordinates $q = (r, \theta)$:

$$\ddot{r} = r\dot{\theta}^2 - g(1 - \cos\theta) - 2k(r - r_0), \tag{64}$$

$$\ddot{\theta} = \frac{-g\sin\theta - 2\dot{r}\dot{\theta}}{r}, \tag{65}$$

with parameters $g = 10\,\text{m/s}^2$, $k = 10\,\text{N/m}$, $r_0 = 1\,\text{m}$. The initial condition is $q_0 = (1.1\,\text{m},\ 0.5\,\text{rad})$, $\dot{q}_0 = (0, 0)$.

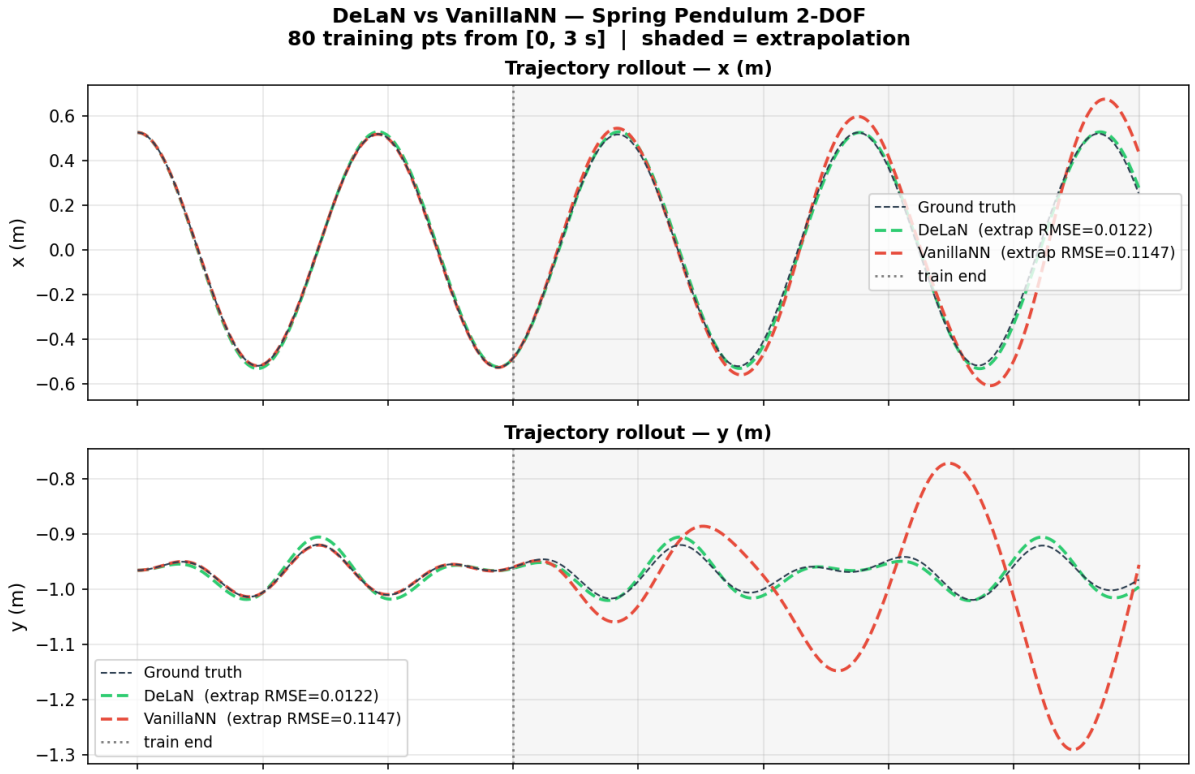

Figure 10: DeLaN vs. VanillaNN on the 2-DOF spring pendulum. Top two rows: Cartesian $x(t)$ and $y(t)$ over 8 s (training window 0–3 s, shaded region is extrapolation). Bottom row: Euclidean position error on log scale.

Why the spring pendulum. The spring pendulum is chosen because it couples two qualitatively different motions, radial oscillation of the spring and angular swing of the pendulum, through nonlinear inertial terms ($r\dot{\theta}^2$, $\dot{r}\dot{\theta}$). The coupling makes the mass matrix non-trivial: the VanillaNN must implicitly represent a position-dependent, 2-DOF inertia structure from data alone, with no constraint preventing it from learning something physically inconsistent.

Data and training. Both models use 3 hidden layers of 64 units and are trained for 2500 epochs on $N = 80$ samples of $(q_i, \dot{q}_i, \ddot{q}_i)$ from the first 3 s of the trajectory. The loss minimized is the mean-squared error on predicted acceleration $\hat{\ddot{q}}$. For DeLaN, the Softplus activation is used throughout to facilitate the non-negative potential energy constraint; the mass matrix is built via the Cholesky construction described in Section 5.

Evaluation. A 4th-order Runge-Kutta integrator rolls both models forward for $t \in [0, 8]$ s, nearly three times the training window. Performance is reported in Cartesian coordinates $(x, y) = (r\sin\theta,\ -r\cos\theta)$ to give a spatially meaningful error metric independent of coordinate singularities. Interpolation RMSE (within the training window) and extrapolation RMSE (outside it) are computed separately to distinguish in-distribution accuracy from generalization.

Key result. Within the training window both models achieve comparable accuracy. In the extrapolation region, DeLaN maintains the trajectory on the correct orbit while the VanillaNN diverges rapidly (Figure 10). The root cause is energy: the VanillaNN's predicted accelerations are not consistent with any conserved Hamiltonian, so small errors accumulate into an energy drift of approximately 54% over the extrapolation window. DeLaN's structured Lagrangian,

with its SPD mass matrix and non-negative potential, conserves the total mechanical energy $E = T + V$ by construction, keeping the energy drift near zero throughout the rollout.

This result illustrates the central advantage of structural inductive bias over soft penalties: DeLaN does not encourage energy conservation via a loss term; it makes energy-violating trajectories architecturally impossible.

6 The Spectrum of Physics-Informedness

It's useful to see these methods as points along a spectrum of how much physics you bake in:

- Vanilla NN: no physics assumed; minimize $\mathcal{L} = \sum(\hat{y} - y)^2$.
- PINN: PDE residual penalized in the loss (soft constraint).
- LNN: Lagrangian framework enforced architecturally (generic).
- DeLaN: Mechanical Lagrangian structure: $\mathcal{M}(q)$ positive-definite by construction.

Trade-off: the more physics you bake in, the less data you need and the better your extrapolation tends to be, but the narrower the class of systems you can handle. Vanilla NNs work on anything. PINNs work on systems whose equations you know. LNN/DeLaN work on Lagrangian (especially mechanical) systems.

All four methods share the same training loop: forward pass → loss → backprop → update. The differences lie entirely in what the forward pass computes and what the loss measures.

7 The Broader Landscape

The three methods covered in this document, namely PINNs, LNNs, and DeLaN, are representative examples, but the field of physics-informed machine learning is considerably broader. A brief, non-exhaustive survey helps situate what follows within the larger landscape.

7.1 Other physics-in-the-loss methods.

Beyond standard PINNs, many variants address specific failure modes. XPINNs [18] and cPINNs [19] use domain decomposition to handle multiscale problems and conservation laws. Variational PINNs (VPINNs) [20] replace the strong-form PDE residual with a weak (variational) form, reducing the regularity required of the network. Bayesian PINNs (B-PINNs) [21] provide uncertainty quantification by placing distributions over network weights. Causal PINNs [22] enforce temporal causality during training to fix a common pathology where networks find spurious late-time solutions while ignoring early dynamics.

7.2 Other physics-in-the-architecture methods.

Alongside LNN and DeLaN, several architectures hard-code different physical structures. Hamiltonian Neural Networks (HNNs) [12] learn a Hamiltonian instead of a Lagrangian, giving cheaper first-order derivatives at the cost of generality. Symplectic Networks (SympNets) [5] preserve the symplectic geometry of phase space exactly, enabling stable long-horizon rollouts. Equivariant neural networks (e.g., NequIP, MACE, Allegro [23]) encode rotation, translation, and permutation symmetries directly, now dominant in molecular dynamics.

7.3 Operator learning.

A complementary paradigm shifts the goal entirely. Instead of learning a single solution to a single PDE, one learns the solution operator of an entire PDE family, a map from input functions (initial conditions, source terms, boundary data, material coefficients) to output functions (the corresponding solutions). Once trained, the operator can be evaluated instantly on new inputs without retraining.

Formally, an operator $\mathcal{G} : \mathcal{U} \to \mathcal{V}$ maps an input function $u \in \mathcal{U}$ to an output function $\mathcal{G}(u) \in \mathcal{V}$, where $\mathcal{U}$ and $\mathcal{V}$ are infinite-dimensional function spaces. For example, in the heat equation $\partial_t s = \alpha \partial_{xx} s$ with initial condition $s(x, 0) = u(x)$, the solution operator $\mathcal{G}$ maps the initial condition u to the solution $s(\cdot, t)$ at any later time. The challenge is that neural networks naturally map finite-dimensional vectors to finite-dimensional vectors, not functions to functions, so a careful construction is required.

DeepONet [24] is the canonical method, built on a universal approximation theorem for nonlinear operators [25]. The architecture decomposes the operator into two sub-networks. A branch network b_θ takes a discretized representation of the input function, specifically its values at a fixed set of sensor points $\{x_1, x_2, \ldots, x_m\}$, and encodes it as a vector of latent features:

$$b_\theta(u(x_1), u(x_2), \ldots, u(x_m)) \in \mathbb{R}^p. \tag{66}$$

A trunk network t_φ takes a query location y in the output domain (which may be a spatial point, a spatiotemporal coordinate, or any element of the domain on which the output function is defined) and also produces a p-dimensional vector:

$$t_\varphi(y) \in \mathbb{R}^p. \tag{67}$$

The operator's output at y, given input function u, is then the inner product of the two:

$$\mathcal{G}_{\theta,\varphi}(u)(y) \;=\; \sum_{k=1}^{p} b_\theta^{(k)}(u(x_1), \ldots, u(x_m)) \cdot t_\varphi^{(k)}(y) \;+\; b_0, \tag{68}$$

where b_0 is a learnable bias. The branch encodes what the input function looks like, the trunk encodes where we want to evaluate the output, and their inner product reconstructs the value of $\mathcal{G}(u)$ at that location. Training proceeds by minimizing a standard supervised loss over triples $(u^{(i)}, y^{(i)}, \mathcal{G}(u^{(i)})(y^{(i)}))$:

$$\mathcal{L}_{\text{DeepONet}} = \frac{1}{N} \sum_{i=1}^{N} \left| \mathcal{G}_{\theta,\varphi}(u^{(i)})(y^{(i)}) - \mathcal{G}(u^{(i)})(y^{(i)}) \right|^2. \tag{69}$$

A key strength of DeepONet is its modularity: the trunk network is independent of how the input function is discretized, so a single trained model can be queried at arbitrary points in the output domain, effectively producing a continuous, mesh-free representation of the solution.

A related and now widely-used alternative is the Fourier Neural Operator (FNO) [26], which parameterizes the operator using learnable convolutions performed in Fourier space. FNOs are particularly effective for problems with translation invariance and have demonstrated remarkable scalability for fluid dynamics and weather forecasting. Physics-Informed Neural Operators (PINOs) [27] combine FNO-style architectures with PINN-style residual losses, blending operator learning with soft physics enforcement.

7.4 Discovery and hybrid methods.

A fundamentally different goal motivates a separate family of methods: instead of learning a black-box surrogate for the dynamics, can we discover the governing equations themselves from

data? An interpretable equation is more transferable, more diagnosable, and more trustworthy than any neural network, if it can be found.

SINDy (Sparse Identification of Nonlinear Dynamics) [28] is the most influential method in this direction. SINDy is not a neural network; it is a sparse regression framework that exploits a key empirical observation: most physical systems are governed by equations with only a handful of active terms. The Lorenz equations have three; the Navier–Stokes equations are dense in symbolic form but sparse relative to all conceivable polynomial combinations of state variables.

Given time-series measurements of a state $x(t) \in \mathbb{R}^n$ sampled at N time points, SINDy assumes the dynamics take the form

$$\dot{x}(t) = f(x(t)), \tag{70}$$

and seeks to identify f from data. The state observations are arranged into a matrix

$$X = \begin{pmatrix} x(t_1)^\top \\ x(t_2)^\top \\ \vdots \\ x(t_N)^\top \end{pmatrix} \in \mathbb{R}^{N \times n}, \tag{71}$$

and the corresponding time derivatives (estimated by finite differences or smoothed differentiation) into a matrix $\dot{X} \in \mathbb{R}^{N \times n}$.

The crucial step is the construction of a candidate function library $\Theta(X) \in \mathbb{R}^{N \times K}$ whose columns are nonlinear functions of the state evaluated at each time point:

$$\Theta(X) = \begin{pmatrix} \mathbf{1} & X & X^{P_2} & X^{P_3} & \sin(X) & \cos(X) & \cdots \end{pmatrix}, \tag{72}$$

where X^{P_k} denotes the matrix of all k-th order polynomial combinations of the state variables, and the library can be extended with any functions the user suspects might appear (trigonometric, exponential, rational, etc.). The library is the user's hypothesis space, comprising the set of all terms the dynamics might contain.

SINDy then poses dynamics identification as a linear regression in this library:

$$\dot{X} \;=\; \Theta(X)\,\Xi, \tag{73}$$

where $\Xi \in \mathbb{R}^{K \times n}$ is the coefficient matrix: each column ξ_j tells us which library terms appear in the equation for $\dot{x}_j$ and with what coefficients. Without further constraints, this would be an ordinary least-squares problem with a dense solution containing every library term, which would be useless for interpretability. The defining move of SINDy is to enforce sparsity: most entries of Ξ should be zero. This is accomplished by solving the regularized problem

$$\boxed{\xi_j \;=\; \arg\min_{\xi} \; \left\| \dot{X}_j - \Theta(X)\xi \right\|_2^2 \;+\; \lambda \|\xi\|_1} \tag{74}$$

for each state dimension j, where the ℓ_1 penalty (LASSO-style) drives most coefficients to exactly zero. In practice, sequential thresholded least squares (STLSQ) is often preferred: solve ordinary least squares, zero out coefficients below a threshold, refit on the remaining terms, and iterate until convergence.

The output is a parsimonious equation like

$$\dot{x}_1 = -0.1\,x_1 + 2.0\,x_1 x_2, \qquad \dot{x}_2 = 1.5\,x_2 - 0.05\,x_1^2, \tag{75}$$

written in closed form using only the library terms with nonzero coefficients, which is directly readable, dimensionally analyzable, and analytically tractable.

SINDy has been extended in numerous directions: SINDy with control inputs [29] for systems with external forcing; SINDy-PI [30] for implicit dynamics; weak SINDy [31] for noisy data via integration against test functions; and PDE-FIND [32] for spatiotemporal systems governed by PDEs (where the library contains spatial derivatives and the regression identifies which derivative terms appear).

7.5 Hybrid approaches.

Between black-box surrogates and pure equation discovery sit hybrid methods that combine known physics with learnable components. Neural ODEs [33] parameterize the dynamics with a neural network integrated by an ODE solver:

$$\frac{dx}{dt} = f_\theta(x, t), \tag{76}$$

treating the network as a continuous-time vector field rather than a discrete-layer feedforward map. Backpropagation through the ODE solver is performed via the adjoint method, giving memory-efficient gradients.

Universal Differential Equations (UDEs) [34] push this further by embedding neural networks inside otherwise-classical differential equations:

$$\frac{dx}{dt} = f_{\text{known}}(x, t) \; + \; \mathcal{N}_\theta(x, t), \tag{77}$$

where f_{known} encodes the physics you trust and $\mathcal{N}_\theta$ is a neural network learning the residual, namely the missing physics, the unmodeled closure term, the unknown forcing. This is arguably the cleanest expression of the philosophy "inject everything you know, let the network learn only what you don't," and it generalizes SINDy (whose library is replaced by a neural network) and Neural ODEs (whose f_θ is replaced by "known plus network").

7.6 Choosing a method.

No single approach dominates. PINNs excel when governing equations are known but data is scarce. Operator methods like DeepONet and FNO shine when one needs many solutions across a parameter family. LNN/HNN/DeLaN are ideal for conservative mechanical systems requiring long-term stability. Equivariant networks are the right tool when symmetries are the dominant structure. SINDy is unmatched when interpretable equations are the goal. The methods developed in this document, namely PINN, LNN, and DeLaN, are foundational examples that illustrate the core principles; the broader literature builds on these ideas in many directions.

8 Exercises

The following exercises extend each worked example by introducing one new concept or structural modification at a time. The goal is to implement the extension yourself; the provided scripts contain only the setup code and the comparison baseline; the core modification is left to you.

Repository. All scripts, notebooks, and supporting code are at

`https://github.com/paolofrance/piml_2026`

Each exercise has a Python script and a Colab-ready notebook (badge at the top of each `.ipynb`). Running locally requires `torch`, `numpy`, `matplotlib`, `scipy`; running on Colab needs only a browser.

PINN exercises

Exercise 1: Parameter identification on the MCK system (`pinn_identification.py`)

In the example `pinn_vs_nn_mck.py`, the physical parameters m, c, k are fully known. In many practical scenarios only some parameters are known and the rest must be inferred from measurements.

Task. Suppose the damping coefficient c is unknown. Modify the PINN so that c is treated as a learnable `nn.Parameter` alongside the network weights, initialised at a deliberately wrong value $c_0 = 1\,\mathrm{N{\cdot}s/m}$ (true value: 4). Train the model on the same 10 observations and verify that the ODE residual drives c towards its true value without any direct supervision on c.

Questions to answer.

- Which loss term provides the gradient signal for c? What happens to c in a plain NN (data loss only, no physics term)?
- How does the identified c compare to the true value? How does extrapolation accuracy compare to the version with known c?
- Does the raw parameterisation ever drive c below zero during training? If so, what does the trajectory look like at that point?

Hint. The script compares two parameterisations side by side:

- Raw: `self.c = nn.Parameter(`c_0`)`. Unconstrained; gradient descent can drive c negative (unphysical).
- Softplus: `self.c = softplus(raw_c)` $= \log(1+e^{\texttt{raw\_c}})$. Strictly non-negative by construction; initialise `raw_c` by inverting softplus: `raw_c`$_0 = \log(e^{c_0} - 1)$.

Substitute the resulting c into the residual $(\,m\ddot{x} + c\dot{x} + kx\,)/k$ at each forward pass and compare how the two variants converge.

Exercise 2: Two-DOF coupled mass-spring-damper (`pinn_2dof_spring_damper.py`)

The MCK example is 1-DOF: the network has a scalar output. Extend it to a 2-DOF coupled system:

$$M\ddot{x} + C\dot{x} + Kx = 0, \qquad M = I, \quad K = \begin{pmatrix} k_1{+}k_2 & -k_2 \\ -k_2 & k_2{+}k_3 \end{pmatrix}, \quad C = \begin{pmatrix} c_1{+}c_2 & -c_2 \\ -c_2 & c_2 \end{pmatrix} \tag{78}$$

with $k_1 = k_3 = 6$, $k_2 = 4$, $c_1 = 0.4$, $c_2 = 0.2$. The system has two coupled oscillation modes that produce a beat-like response.

Task. Build a PINN with a 2-dimensional output $t \mapsto (x_1(t), x_2(t))$ and write two ODE residuals, one per DOF. Normalise each residual by the largest eigenvalue of K (which is 14) to keep both loss terms $\mathcal{O}(1)$.

Questions to answer.

- How do you compute $\dot{x}_i$ and $\ddot{x}_i$ independently for each output dimension via `autograd.grad`?
- Does the PINN capture both oscillation modes in the extrapolation window? Does the plain NN?
- What happens if you normalise by the smallest eigenvalue instead of the largest?

Exercise 3: Generalisation across initial conditions (`pinn_multi_traj.py`)

All PINN examples so far learn the solution for a single initial condition. This exercise trains a single network to represent the solution family $x(t;\, x_0, \dot{x}_0)$ across a distribution of initial conditions.

Task. Augment the network input from t to $(t,\, x_0,\, \dot{x}_0)$ and train on `--n_train` trajectories with initial conditions drawn uniformly from $x_0 \in [-0.5,\, 0.5]$, $\dot{x}_0 \in [-0.1,\, 0.1]$. Add an initial condition loss that enforces $x(0;\, x_0, \dot{x}_0) = x_0$ and $\dot{x}(0;\, x_0, \dot{x}_0) = \dot{x}_0$ via `autograd`.

Evaluate on 8 held-out initial conditions (different random seed).

Questions to answer.

- How do you enforce the IC constraints using `autograd` rather than a separate analytical formula?
- How does generalisation to unseen ICs depend on `--n_train`? Find the minimum number of training trajectories for which the PINN reliably outperforms the plain NN on held-out ICs.
- Where should the collocation points be sampled from when training on multiple trajectories?

Hint. At collocation time, sample $(t, x_0, \dot{x}_0)$ triples where t spans $[0, T_{\text{eval}}]$ and the ICs are drawn from the training set; do not sample ICs outside the training distribution.

Exercise 4: PINN on the 2-DOF spring pendulum (`pinn_spring_pendulum.py`)

The LNN and DeLaN examples use the spring pendulum as their benchmark. This exercise applies a PINN to the same system so you can directly compare the two approaches on identical data.

Task. Build a PINN with a 2-dimensional output $t \mapsto (r(t),\, \theta(t))$ and write the two coupled ODE residuals:

$$\mathcal{R}_1 = \frac{\ddot{r} - r\dot{\theta}^2 + g(1 - \cos\theta) + 2k(r - r_0)}{2k},$$
$$\mathcal{R}_2 = \frac{\ddot{\theta} + (g\sin\theta + 2\dot{r}\dot{\theta})/r}{g/r_0}.$$

Train on position-only observations (t, r, θ), unlike LNN/DeLaN, no velocity or acceleration measurements are needed.

Questions to answer.

- The PINN needs $\dot{r}$, $\ddot{r}$, $\dot{\theta}$, $\ddot{\theta}$ to evaluate the residuals. How does it obtain these from a network that maps $t \mapsto (r, \theta)$?
- Plot the total mechanical energy $E(t) = T + V$ along the PINN rollout. Does it conserve energy? Compare to the LNN and DeLaN results on the same system.
- What is the normalisation strategy for $\mathcal{R}_2$? Why is dividing by g/r_0 a reasonable choice?

Key contrast. The PINN requires only position data, which is easier to collect experimentally than $(q, \dot{q}, \ddot{q})$. However, it loses the energy conservation guarantee that the Lagrangian methods provide by construction. Both trade-offs are visible in the energy and trajectory panels of the comparison plot.

LNN exercise

Exercise 5: Extending LNN to dissipative systems (`lnn_friction.py`)

The LNN example (‘lnn_vs_vanilla.py‘) handles a conservative pendulum. Real mechanical systems are dissipative. This exercise adds friction learning following the implicit approach of Ragusano et al. (Politecnico di Milano, 2024).

Task. The system is the simple pendulum with viscous friction:

$$\ddot{\theta} = -\frac{g}{\ell}\sin\theta - b\,\dot{\theta}, \qquad b = 0.3, \quad \ell = 1\,\text{m}. \tag{79}$$

Add a separate unconstrained DNN $\hat{F}(\theta, \dot{\theta})$ alongside the conservative LNN. The combined prediction is:

$$\hat{\ddot{\theta}} = \ddot{\theta}_{\text{LNN}} + \frac{\hat{F}(\theta, \dot{\theta})}{M_{\text{LNN}}(\theta, \dot{\theta})}, \tag{80}$$

where $M_{\text{LNN}} = \partial^2 L_\theta / \partial \dot{\theta}^2$ is the scalar velocity Hessian from the learned Lagrangian. Train with a joint loss:

$$\ell_{\text{dyn}} = \text{MSE}\Big(\hat{\ddot{\theta}},\ \ddot{\theta}_{\text{true}}\Big), \tag{81}$$

$$\ell_{\text{fric}} = \text{MSE}\Big(\hat{F},\ M_{\text{LNN}} \cdot (\ddot{\theta}_{\text{true}} - \ddot{\theta}_{\text{LNN}})\Big), \tag{82}$$

where the LNN outputs in ℓ_{fric} are `.detach()`ed so gradients flow only to $\hat{F}$.

Questions to answer.

- Why does the conservative LNN fail on this system even though it trains to a low loss? Plot its energy over the rollout to confirm.
- Why must the LNN outputs be detached when computing ℓ_{fric}? What would happen if they were not?
- Does the learned $\hat{F}$ resemble the true friction $-b\dot{\theta}$? Plot $\hat{F}$ as a function of $\dot{\theta}$ at a fixed θ.
- Compare LNN-F, conservative LNN, and VanillaNN in terms of extrapolation accuracy and energy profile over $[0, 12\,\text{s}]$.

Hint. Joint training creates a moving-target problem for $\hat{F}$: the LNN keeps changing while $\hat{F}$ tries to track its residual. Detaching the LNN outputs when computing ℓ_{fric} decouples the two gradient flows and is sufficient to stabilise convergence in practice.

DeLaN exercises

Exercise 6: DeLaN-F on the simple pendulum (`delan_friction_pendulum.py`)

Repeat Exercise 5 using DeLaN instead of LNN as the conservative baseline, and replace the unconstrained friction DNN with a structured Rayleigh dissipation function.

Task. Add a learnable scalar $B(q) > 0$ (for 1-DOF, this is just a positive-valued network output) to DeLaN. The modified equations of motion are:

$$\mathcal{M}(q)\,\ddot{q} = \nabla_q L_\theta - [\nabla_q \nabla_{\dot{q}} L_\theta]\,\dot{q} - B(q)\,\dot{q}. \tag{83}$$

Enforce $B(q) > 0$ using the same Cholesky trick as the mass matrix: let $B(q) = b(q)^2 + \varepsilon$ where $b(q)$ is the unconstrained network output (for 1-DOF this reduces to squaring a scalar).

Questions to answer.

- What physical guarantee does $B(q) > 0$ provide? Write the expression for $\mathrm{d}E/\mathrm{d}t$ and show it is non-positive.
- Compare the energy profiles of DeLaN-F, conservative DeLaN, and VanillaNN. Which model best captures the monotone energy decay?
- The LNN-F in Exercise 5 uses an unconstrained $\hat{F}$. What could go wrong if a DeLaN used an unconstrained friction term instead of the Rayleigh form?

Hint. Start with the conservative DeLaN from `delan_vs_vanilla.py`, add $B(q)$ as a third network, and modify only the force assembly step. The training loop and RK4 rollout are otherwise identical.

Exercise 7: DeLaN-F on the 2-DOF spring pendulum (`delan_friction.py`)

Scale Exercise 6 to the full 2-DOF spring pendulum with friction on both degrees of freedom:

$$\ddot{r} = r\dot{\theta}^2 - g(1 - \cos\theta) - 2k(r - r_0) - b_r\,\dot{r}, \tag{84}$$

$$\ddot{\theta} = \frac{-g\sin\theta - 2\dot{r}\dot{\theta}}{r} - b_\theta\,\dot{\theta}, \tag{85}$$

with $b_r = 0.1$, $b_\theta = 0.3$.

Task. Extend the scalar $B(q)$ from Exercise 6 to a full 2×2 Rayleigh dissipation matrix $B(q) \in \mathbb{R}^{2\times 2}$, positive semi-definite by the same Cholesky construction used for the mass matrix:

$$B(q) = L_B(q)\,L_B(q)^\top + \varepsilon_B I, \tag{86}$$

where $L_B(q)$ is a learned lower-triangular matrix.

Questions to answer.

- How many independent entries does a 2×2 lower-triangular matrix have? Which entries must be strictly positive, and how do you enforce this?
- Does a full 2×2 matrix $B(q)$ allow cross-DOF dissipation coupling? Is there physical motivation for such coupling in this system?
- Compare DeLaN-F, conservative DeLaN, and VanillaNN on the trajectory and energy plots. Is the improvement of DeLaN-F over conservative DeLaN larger or smaller than in the 1-DOF case? Why might the answer differ?

Hint. The Cholesky construction for $B(q)$ is identical in structure to the one used for $\mathcal{M}(q)$ in the base DeLaN model: inspect `models/delan.py` to see how the mass matrix is assembled and replicate the pattern for B.

References

[1] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[2] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, 3361(10):1995, 1995.

[3] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[4] Taco Cohen, Maurice Weiler, Berkay Kicanaoglu, and Max Welling. Gauge equivariant convolutional networks and the icosahedral CNN. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2019.

[5] Pengzhan Jin, Zhen Zhang, Aiqing Zhu, Yifa Tang, and George Em Karniadakis. SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems. *Neural Networks*, 132:166–179, 2020.

[6] David Pfau, James S Spencer, Alexander G Matthews, and W Matthew C Foulkes. Ab initio solution of the many-electron Schrödinger equation with deep neural networks. *Physical Review Research*, 2(3):033429, 2020.

[7] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[8] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[9] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why PINNs fail to train: A neural tangent kernel perspective. *Journal of Computational Physics*, 449:110768, 2022.

[10] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[11] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020.

[12] Sam Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[13] Michael Lutter, Christian Ritter, and Jan Peters. Deep Lagrangian networks: Using physics as model prior for deep learning. In *International Conference on Learning Representations (ICLR)*, 2019.

[14] Theo Wolf. Physics-Informed Neural Networks: a simple tutorial with PyTorch. Medium article, 2022. Accessed 2026.

[15] Ben Moseley. So, what is a physics-informed neural network? Blog post, 2021. Accessed 2026.

[16] Magnus Ross. Lagrangian Neural Networks (blog post). Blog post, 2021. Accessed 2026.

[17] Magnus Ross. pytorch-lagrangian-nn. GitHub repository, 2021. Accessed 2026.

[18] Ameya D Jagtap and George Em Karniadakis. Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. *Communications in Computational Physics*, 28(5):2002–2041, 2020.

[19] Ameya D Jagtap, Ehsan Kharazmi, and George Em Karniadakis. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering*, 365:113028, 2020.

[20] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 374:113547, 2021.

[21] Liu Yang, Xuhui Meng, and George Em Karniadakis. B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data. *Journal of Computational Physics*, 415:109913, 2021.

[22] Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality for training physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421:116813, 2024.

[23] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E Smidt, and Boris Kozinsky. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications*, 13(1):2453, 2022.

[24] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[25] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[26] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations (ICLR)*, 2021.

[27] Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations. *ACM/JMS Journal of Data Science*, 1(3):1–27, 2024.

[28] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[29] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Sparse identification of nonlinear dynamics with control (SINDYc). *IFAC-PapersOnLine*, 49(18):710–715, 2016.

[30] Kadierdan Kaheman, J Nathan Kutz, and Steven L Brunton. SINDy-PI: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics. *Proceedings of the Royal Society A*, 476(2242):20200279, 2020.

[31] Daniel A Messenger and David M Bortz. Weak SINDy for partial differential equations. *Journal of Computational Physics*, 443:110525, 2021.

[32] Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4):e1602614, 2017.

[33] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 31, 2018.

[34] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.