Neural Network Training: From Vanilla NNs to Physics-Informed Methods

A Lecture on the Spectrum of Physics-Informedness

Paolo Franceschi
paolo.franceschi@supsi.ch

IDSIA-USI-SUPSI

May 27, 2026

Repository: github.com/paolofrance/piml_2026

Outline

The Problem with Pure Data-Driven ML in Science

Standard deep learning thrives on:

- Abundant, cheap data
- No need for interpretability
- In-distribution prediction

The Problem with Pure Data-Driven ML in Science

Standard deep learning thrives on:

- Abundant, cheap data
- No need for interpretability
- In-distribution prediction

In the physical sciences, none of these hold:

- Experiments and simulations are **expensive**
- Physical laws **must be respected**
- **Extrapolation** matters

The Problem with Pure Data-Driven ML in Science

Standard deep learning thrives on:

- Abundant, cheap data
- No need for interpretability
- In-distribution prediction

In the physical sciences, none of these hold:

- Experiments and simulations are **expensive**
- Physical laws **must be respected**
- **Extrapolation** matters

The core idea

Combine data with prior physical knowledge to compensate for both data scarcity *and* model incompleteness.

Three Pathways to Embed Physics (Karniadakis et al., 2021)

No predictive model can exist without assumptions. The question is *which* to encode and *how*.

Observational Bias	Inductive Bias	Learning Bias
Physics enters through **data**. Train on physics-respecting data; the network inherits structure statistically.	Physics enters through **architecture**. Constraints satisfied by construction; mathematically guaranteed.	Physics enters through **loss function**. Soft penalties for violating physical laws; very flexible.

Three Pathways to Embed Physics (Karniadakis et al., 2021)

No predictive model can exist without assumptions. The question is *which* to encode and *how*.

Observational Bias	Inductive Bias	Learning Bias
Physics enters through **data**. Train on physics-respecting data; the network inherits structure statistically.	Physics enters through **architecture**. Constraints satisfied by construction; mathematically guaranteed.	Physics enters through **loss function**. Soft penalties for violating physical laws; very flexible.

Most successful methods combine all three.

Three Representative Methods Covered Today

- **PINN** — Physics in the loss (learning bias)
  - Raissi, Perdikaris, Karniadakis (2019)
  - PDE residual penalized at collocation points
- **LNN** — Lagrangian structure in architecture (generic inductive bias)
  - Cranmer et al. (2020)
  - Learn scalar $L_\theta(q, \dot{q})$; derive dynamics
- **DeLaN** — Mechanical Lagrangian structure (specific inductive bias)
  - Lutter, Ritter, Peters (2019)
  - Parameterize $\mathcal{M}(q)$ and $V(q)$ separately

What is a Neural Network?

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 nonlinearities.

- Weights and biases start **random**
- Training = find θ that minimizes loss $\mathcal{L}(\theta)$
- Optimization via **gradient descent**
- Gradients computed efficiently via **backpropagation**

Four steps repeated millions of times: forward $\to$ loss $\to$ backward $\to$ update

Notation

For a network with L layers, at layer l:

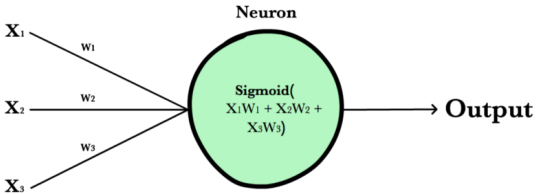
- $W^{(l)}$: weight matrix
- $b^{(l)}$: bias vector
- $z^{(l)}$: pre-activation
- $a^{(l)}$: activation (output) of layer
- σ: activation function
- x: input ($= a^{(0)}$)
- y: true target
- $\hat{y}$: prediction ($= a^{(L)}$)
- $\mathcal{L}$: scalar loss
- η: learning rate

Step 1: Forward Pass — One Neuron, One Layer

A single neuron computes a weighted sum, adds a bias, and applies a nonlinearity:

$$z = \sum_i w_i x_i + b$$

$$a = \sigma(z)$$



The basic neuron: weighted inputs, bias, activation

Stacking n_l neurons into layer l:

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

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

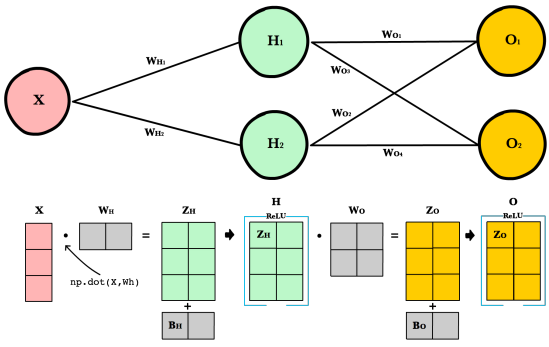
Step 1: Forward Pass — Full Network

Apply the layer recursion for $l = 1, \ldots, L$ with $a^{(0)} = x$ and $\hat{y} = a^{(L)}$.

Unrolling the recursion gives the full nested expression for N layers:

$$\hat{y} = \sigma\Big(W^{(N)}\sigma\big(W^{(N-1)}\sigma\big(\cdots \sigma\big(W^{(2)}\sigma\big(W^{(1)}x + b^{(1)}\big) + b^{(2)}\big)\cdots + b^{(N-1)}\big) + b^{(N)}\Big)$$

Nested linear maps interleaved with nonlinear "squishes."



Information flows layer by layer until the final prediction

Step 2: Loss Calculation

The loss quantifies how wrong $\hat{y}$ is.

Mean Squared Error (regression):

$$\mathcal{L} = \tfrac{1}{2}\|\hat{y} - y\|^2$$

Cross-Entropy (classification):

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

Batch average over N examples:

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

Step 3: Backpropagation — The Idea

Goal: compute $\partial \mathcal{L}/\partial W^{(l)}$ and $\partial \mathcal{L}/\partial b^{(l)}$ for every layer.

The trick: work backwards from the output, reusing computations via the chain rule.

Define the **error signal** at layer l:

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

Think of $\delta^{(l)}$ as "how much does the pre-activation at layer l affect the loss?"

Step 3: Backpropagation — The Recursion

At the output ($l = L$):

$$\delta^{(L)} = (\hat{y} - y) \odot \sigma'(z^{(L)})$$

Recurse backward for $l = L-1, \ldots, 1$:

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

Step 3: Backpropagation — The Recursion

At the output ($l = L$):

$$\delta^{(L)} = (\hat{y} - y) \odot \sigma'(z^{(L)})$$

Recurse backward for $l = L-1, \ldots, 1$:

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

Gradients fall out:

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

Intuition

A weight's gradient = (error at its output side) $\times$ (activation at its input side).

Step 4: Weight Update

Vanilla gradient descent:

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

SGD with momentum ($\beta \approx 0.9$):

$$v^{(l)} \leftarrow \beta v^{(l)} + \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \qquad W^{(l)} \leftarrow W^{(l)} - \eta v^{(l)}$$

Adam — adaptive learning rate per parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

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

The Full Training Loop

For each batch:

1. **Forward**: $a^{(l)} = \sigma(W^{(l)} a^{(l-1)} + b^{(l)})$
2. **Loss**: $\mathcal{L} = \text{loss\_fn}(a^{(L)}, y)$
3. **Backward**: propagate $\delta^{(l)}$ from output to input
4. **Gradients**: compute $\partial \mathcal{L} / \partial W^{(l)}$, $\partial \mathcal{L} / \partial b^{(l)}$
5. **Update**: $W^{(l)} \leftarrow W^{(l)} - \eta \cdot \partial \mathcal{L} / \partial W^{(l)}$

Repeat for every batch, every epoch, until convergence.

The Full Training Loop

For each batch:

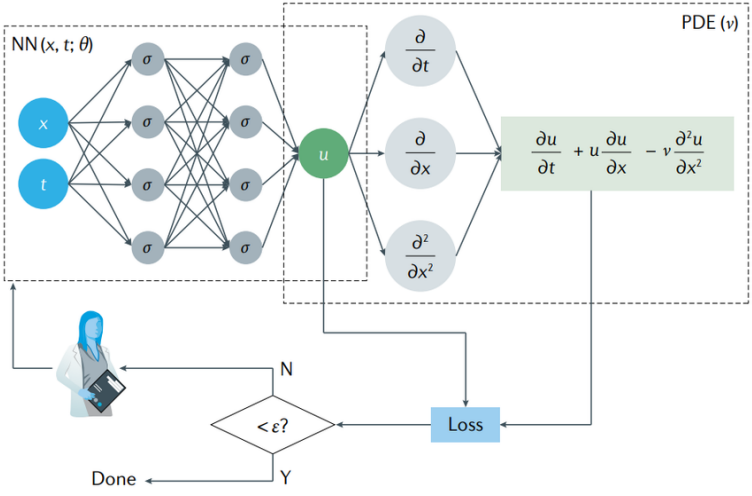
1. **Forward**: $a^{(l)} = \sigma(W^{(l)} a^{(l-1)} + b^{(l)})$
2. **Loss**: $\mathcal{L} = \text{loss\_fn}(a^{(L)}, y)$
3. **Backward**: propagate $\delta^{(l)}$ from output to input
4. **Gradients**: compute $\partial \mathcal{L} / \partial W^{(l)}$, $\partial \mathcal{L} / \partial b^{(l)}$
5. **Update**: $W^{(l)} \leftarrow W^{(l)} - \eta \cdot \partial \mathcal{L} / \partial W^{(l)}$

Repeat for every batch, every epoch, until convergence.

Key takeaway

Every physics-informed method in this lecture uses **exactly this loop**. What changes is the *loss function* and what the *forward pass* computes.

PINN: The Core Idea



PINN structure — Karniadakis et al. (2021)

Standard loop, but the loss enforces a **PDE residual**.

The Key Tool: Differentiation Through the Input

Vanilla NN: differentiate loss w.r.t. parameters θ.

PINN: also differentiate the network's *output* w.r.t. its *input*.

Given a network $u_\theta(x)$ with input coordinates $x \in \Omega \subset \mathbb{R}^d$ (which may include time), automatic differentiation provides any derivative

$$\frac{\partial^{|\alpha|} u_\theta}{\partial x^\alpha}$$

as a *differentiable expression* in θ.

Why this matters

We can plug u_θ into a differential equation, compute the residual, and backprop through the residual — "autodiff inside autodiff."

Generic Problem Statement

A PINN targets a generic differential problem (Partial Differential Equation):

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

closed by a boundary/initial condition operator

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

$\mathcal{N}$ is a (possibly nonlinear) differential operator; $\mathcal{B}$ collects Dirichlet, Neumann, periodic, or initial conditions.

Covers ODEs, PDEs, and systems of either. Examples ahead instantiate $\mathcal{N}$ as a 1st-order and a 2nd-order ODE.

The network $u_\theta(x)$ approximates u; training penalises the violation of these equations at sampled points.

PINN Loss Components

Physics residual at collocation points $\{x_i^r\}$:

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

Boundary / initial conditions at $\{x_i^{bc}\}$:

$$\mathcal{L}_{\text{bc}} = \frac{1}{N_{bc}} \sum_i \|\mathcal{B}[u_\theta](x_i^{bc})\|^2$$

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

$$\mathcal{L}_{\text{data}} = \frac{1}{N_d} \sum_i \|u_\theta(x_i^d) - u_i^d\|^2$$

Total PINN Loss

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

The λ's are weights, and balancing them is the central headache of PINN training.

Total PINN Loss

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

The λ's are weights, and balancing them is the central headache of PINN training.

Loss balancing

Multiple loss terms can have wildly different magnitudes and gradient scales. Naive weighting causes one term to dominate; the others are ignored.

A common fix: if $\mathcal{N}$ has a natural scale (stiffness, rate constant, etc.), divide the residual by it so all loss terms start out $\mathcal{O}(1)$.

How PINN Differs from Vanilla NN

- Network input = **coordinates** $x \in \Omega$, not features
- Network output = **field value** $u_\theta(x)$, not a class or label
- Forward pass requires **autodiff through inputs**
- Loss includes the **residual of** $\mathcal{N}[u] = 0$ at collocation points
- Trained with little or **no labeled data**
- Multiple competing loss terms must be **balanced**

Example 1: Newton's Law of Cooling

First-order ODE, scalar, with unknown rate:

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

Analytical solution: $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})e^{-Rt}$.

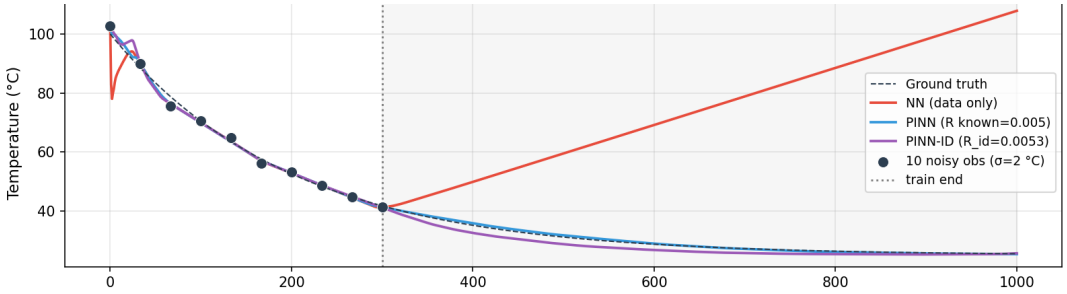
Adapted from T. Wolf, medium.com (2022)

Three models compared, all trained on 10 noisy observations:

- **NN**: data loss only
- **PINN**: data + physics residual (R known)
- **PINN-ID**: R becomes a learnable parameter

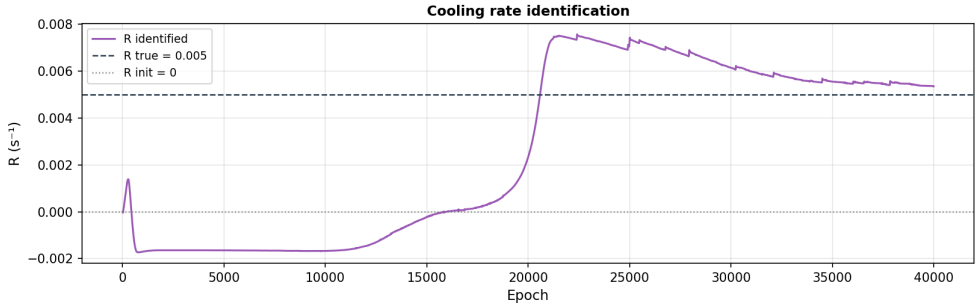
The ODE residual acts as a self-supervised identification signal.

Example 1: Results



- NN overfits the 10 observations → diverges beyond the training window
- PINN converges to the correct exponential decay
- PINN-ID also recovers the rate R from noisy data alone

Example 1: Parameter Identification



Learned R converges to the true value, with no extra labels.

Example 2: Mass-Spring-Damper (MCK)

A second-order ODE with a key implementation detail:

$$m\ddot{x} + c\dot{x} + kx = 0, \quad m = 1,\ c = 4,\ k = 400$$

Inspired by B. Moseley, benmoseley.blog (2021)

Residual via chained autodiff:

$$\mathcal{R}[x_\theta](t) = m\ddot{x}_\theta + c\dot{x}_\theta + kx_\theta$$

Requires **higher-order autodiff** (two derivatives w.r.t. t).

Example 2: Mass-Spring-Damper (MCK)

A second-order ODE with a key implementation detail:

$$m\ddot{x} + c\dot{x} + kx = 0, \quad m = 1,\ c = 4,\ k = 400$$

Inspired by B. Moseley, benmoseley.blog (2021)

Residual via chained autodiff:

$$\mathcal{R}[x_\theta](t) = m\ddot{x}_\theta + c\dot{x}_\theta + kx_\theta$$

Requires **higher-order autodiff** (two derivatives w.r.t. t).

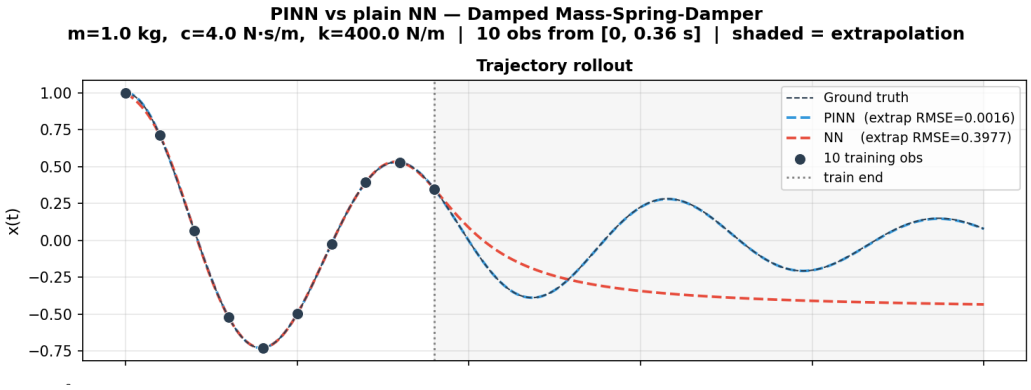
Residual normalization

$k = 400$ makes kx dominate. Dividing by k brings every term to $\mathcal{O}(1)$ — both loss terms are balanced from epoch 1, no warm-up needed:

$$\tilde{\mathcal{R}} = \mathcal{R}/k$$

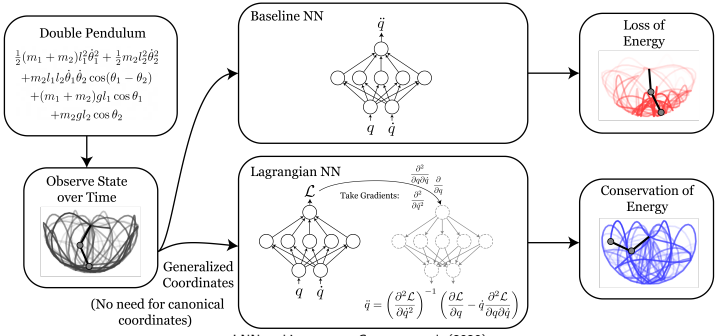
Collocation points must span the full evaluation domain $[0, 1\,s]$*, not just the training window.*

Example 2: Results



- Vanilla NN interpolates the training window but diverges afterwards
- PINN extrapolates the decaying oscillation correctly far past the data

LNN: The Core Idea



LNN architecture — Cranmer et al. (2020)

Key insight

Don't learn the dynamics directly. Learn a scalar **Lagrangian** $L_\theta(q, \dot{q})$, then derive dynamics from it.

For a system with positions $q \in \mathbb{R}^n$ and velocities $\dot{q} \in \mathbb{R}^n$, the network is a scalar map:

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

Euler–Lagrange in Gradient Form

Start from the **Euler–Lagrange equation**:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = 0$$

Identify the partial-derivative vectors with gradients:

$$\frac{\partial L}{\partial \dot{q}} \equiv \nabla_{\dot{q}} L, \qquad \frac{\partial L}{\partial q} \equiv \nabla_q L.$$

Rewrite Euler–Lagrange in compact form:

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

Chain Rule: Total Time Derivative

L depends on t only through $q(t)$ and $\dot{q}(t)$. Chain rule on the vector $\nabla_{\dot{q}} L$:

$$\frac{d}{dt}(\nabla_{\dot{q}} L) = \underbrace{\frac{\partial(\nabla_{\dot{q}} L)}{\partial q}}_{n \times n} \dot{q} + \underbrace{\frac{\partial(\nabla_{\dot{q}} L)}{\partial \dot{q}}}_{n \times n} \ddot{q}$$

Each Jacobian is a **Hessian block** of L:

$$\frac{\partial(\nabla_{\dot{q}} L)}{\partial \dot{q}} = \nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L \qquad \frac{\partial(\nabla_{\dot{q}} L)}{\partial q} = \nabla_{q} \nabla_{\dot{q}}^{\top} L$$

Chain Rule: Total Time Derivative

L depends on t only through $q(t)$ and $\dot{q}(t)$. Chain rule on the vector $\nabla_{\dot{q}} L$:

$$\frac{d}{dt}(\nabla_{\dot{q}} L) = \underbrace{\frac{\partial(\nabla_{\dot{q}} L)}{\partial q}}_{n\times n}\,\dot{q} + \underbrace{\frac{\partial(\nabla_{\dot{q}} L)}{\partial \dot{q}}}_{n\times n}\,\ddot{q}$$

Each Jacobian is a **Hessian block** of L:

$$\frac{\partial(\nabla_{\dot{q}} L)}{\partial \dot{q}} = \nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L \qquad \frac{\partial(\nabla_{\dot{q}} L)}{\partial q} = \nabla_{q} \nabla_{\dot{q}}^{\top} L$$

Substitute into Euler–Lagrange:

$$\boxed{\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L \cdot \ddot{q} + \nabla_{q} \nabla_{\dot{q}}^{\top} L \cdot \dot{q} - \nabla_{q} L = 0}$$

Solve for the Acceleration

From the previous slide:

$$\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L \cdot \ddot{q} \; + \; \nabla_{q} \nabla_{\dot{q}}^{\top} L \cdot \dot{q} \; - \; \nabla_{q} L \; = \; 0$$

Isolate $\ddot{q}$ (assuming the velocity Hessian is invertible):

$$\boxed{\ddot{q} = \left(\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L\right)^{-1} \left[\nabla_{q} L - \left(\nabla_{q} \nabla_{\dot{q}}^{\top} L\right) \dot{q}\right]}$$

Newtonian shape: generalised mass $\times$ acceleration $=$ generalised force.

What if There Are External Torques?

Anything *not* derivable from the potential $V(q)$ in L goes on the RHS as a generalised force τ:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} \;=\; \tau$$

The chain-rule expansion is identical; only the RHS picks up τ:

$$\boxed{\ddot{q} = \left(\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L\right)^{-1} \left[\, \tau + \nabla_q L - \left(\nabla_q \nabla_{\dot{q}}^{\top} L\right) \dot{q} \,\right]}$$

What if There Are External Torques?

Anything *not* derivable from the potential $V(q)$ in L goes on the RHS as a generalised force τ:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} \;=\; \tau$$

The chain-rule expansion is identical; only the RHS picks up τ:

$$\boxed{\ddot{q} = \left(\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L\right)^{-1} \left[\, \tau + \nabla_q L - \left(\nabla_q \nabla_{\dot{q}}^{\top} L\right) \dot{q} \,\right]}$$

- **Motor torques, control inputs:** τ commanded or measured
- **Friction / damping:** $\tau_{\text{diss}} = -\nabla_{\dot{q}} \mathcal{R}(q, \dot{q})$, with $\mathcal{R} \geq 0$ a Rayleigh dissipation function
- Training data becomes $(q, \dot{q}, \ddot{q}, \tau)$ — enables the DeLaN forward loss
- Energy is no longer conserved: τ pumps energy in/out

Parametrise the Lagrangian: the LNN

Replace the unknown scalar L with a neural network:

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

Plug L_θ into the equation of motion — all derivatives are computed by autodiff:

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

$$M_{\text{LNN}}(q, \dot{q}) := \nabla_{\dot{q}} \nabla_{\dot{q}}^\top L_\theta \qquad \text{(velocity Hessian)}$$

Parametrise the Lagrangian: the LNN

Replace the unknown scalar L with a neural network:

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

Plug L_θ into the equation of motion — all derivatives are computed by autodiff:

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

$$M_{\text{LNN}}(q, \dot{q}) := \nabla_{\dot{q}} \nabla_{\dot{q}}^\top L_\theta \qquad \text{(velocity Hessian)}$$

- Needs **first and second derivatives** of L_θ — autodiff inside autodiff
- Needs to **invert** the $n \times n$ velocity Hessian
- M_{LNN} is implicit: not guaranteed SPD or $\dot{q}$-independent

LNN: Loss Function

Training data $\{(q_i, \dot{q}_i, \ddot{q}_i)\}_{i=1}^{N}$ — no torque needed:

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

where $\hat{\ddot{q}}_i$ is the LNN forward solve at $(q_i, \dot{q}_i)$.

- Single, clean loss term — no λ-balancing
- Network outputs **energy**, not acceleration directly
- Conservation laws (energy, momentum) hold **by construction**
- Trade-off: forward pass needs one M_{LNN}^{-1} per sample

LNN vs Vanilla: Inductive Bias in Action

Implicit mass matrix (the velocity Hessian):

$$M_{\text{LNN}}(q, \dot{q}) = \nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} L_{\theta}$$

Nothing guarantees:

- symmetry
- positive-definiteness
- velocity independence

The LNN *hopes* the network learns these from data. Often works, but fragile.

Example: Simple Pendulum

System chosen near the separatrix (highly nonlinear):

$$\ddot{\theta} = -\frac{g}{\ell}\sin\theta, \quad \theta_0 = 2.5\,\text{rad}, \quad \ell = 1\,\text{m}$$

Inspired by M. Ross, magnusross.github.io & pytorch-lagrangian-nn (2021)

Setup:

- Only **20** training samples from $t \in [0, 3]$ s
- Evaluate over $t \in [0, 12]$ s ($4\times$ extrapolation)
- Same architecture (3×64) for both LNN and Vanilla NN
- *Identical loss*: MSE on acceleration

Example: Simple Pendulum

System chosen near the separatrix (highly nonlinear):

$$\ddot{\theta} = -\frac{g}{\ell}\sin\theta, \quad \theta_0 = 2.5\,\text{rad}, \quad \ell = 1\,\text{m}$$

Inspired by M. Ross, magnusross.github.io & pytorch-lagrangian-nn (2021)

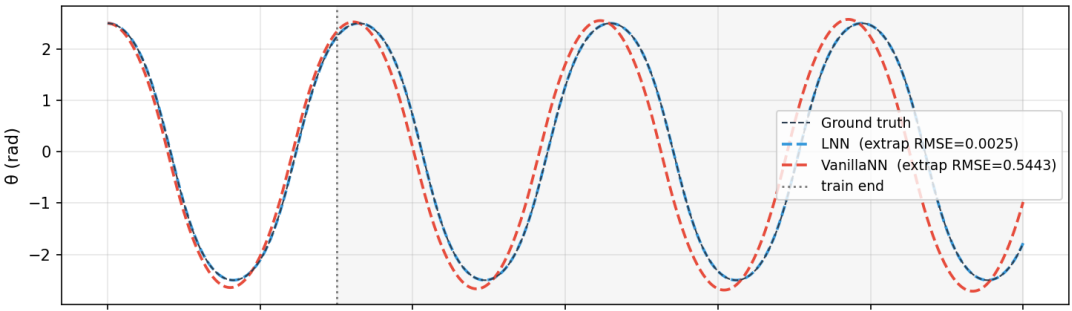
Setup:

- Only **20** training samples from $t \in [0, 3]$ s
- Evaluate over $t \in [0, 12]$ s ($4\times$ extrapolation)
- Same architecture (3×64) for both LNN and Vanilla NN
- *Identical loss*: MSE on acceleration

The only difference

How $\hat{\ddot{q}}$ is computed: VanillaNN directly outputs it; LNN derives it from a learned L_θ via Euler–Lagrange.

Example: Results



- LNN stays on the correct energy shell throughout extrapolation
- VanillaNN drifts immediately and diverges
- Same data, same loss, same architecture — only the *inductive bias* differs

Kinetic Energy in Generalised Coordinates

For a system of N point masses with Cartesian positions $\vec{r}_i$:

$$T = \tfrac{1}{2} \sum_{i=1}^{N} m_i \|\dot{\vec{r}}_i\|^2$$

Constraints (linkages, joints, etc.) reduce the DOFs to generalised coordinates $q \in \mathbb{R}^n$:

$$\vec{r}_i = \vec{r}_i(q) \quad \Longrightarrow \quad \dot{\vec{r}}_i = \frac{\partial \vec{r}_i}{\partial q}\, \dot{q}$$

Substituting yields a quadratic form in $\dot{q}$:

$$\boxed{T = \tfrac{1}{2}\, \dot{q}^\top \mathcal{M}(q)\, \dot{q}, \qquad \mathcal{M}(q) := \sum_i m_i \left(\frac{\partial \vec{r}_i}{\partial q}\right)^\top \frac{\partial \vec{r}_i}{\partial q}}$$

Kinetic Energy in Generalised Coordinates

For a system of N point masses with Cartesian positions $\vec{r}_i$:

$$T = \tfrac{1}{2} \sum_{i=1}^{N} m_i \|\dot{\vec{r}}_i\|^2$$

Constraints (linkages, joints, etc.) reduce the DOFs to generalised coordinates $q \in \mathbb{R}^n$:

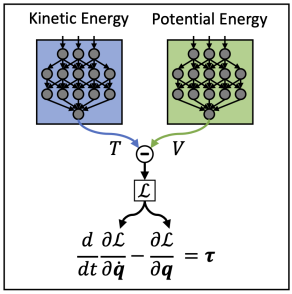
$$\vec{r}_i = \vec{r}_i(q) \quad \Longrightarrow \quad \dot{\vec{r}}_i = \frac{\partial \vec{r}_i}{\partial q}\, \dot{q}$$

Substituting yields a quadratic form in $\dot{q}$:

$$\boxed{T = \tfrac{1}{2}\, \dot{q}^\top \mathcal{M}(q)\, \dot{q}, \qquad \mathcal{M}(q) := \sum_i m_i \left(\frac{\partial \vec{r}_i}{\partial q}\right)^\top \frac{\partial \vec{r}_i}{\partial q}}$$

- $\mathcal{M}(q)$ is **symmetric and positive definite** by construction ($J^\top J$ sum)
- Depends only on q, not on $\dot{q}$ — this is a *theorem*, not a modelling choice
- Adding $V(q)$ gives the structured Lagrangian $L = T - V = \frac{1}{2}\dot{q}^\top \mathcal{M}(q)\dot{q} - V(q)$

DeLaN: A Specialized LNN for Rigid Bodies



DeLaN architecture — two networks

Key insight

Mechanical Lagrangians have **known structure**: $L = \frac{1}{2}\dot{q}^\top \mathcal{M}(q)\dot{q} - V(q)$. Exploit it.

From the Generic LNN to a Mechanical Lagrangian

Substitute $L = \frac{1}{2}\dot{q}^\top \mathcal{M}(q)\dot{q} - V(q)$ into the §LNN formulas:

Generic LNN term	**After substitution**
$M_{\text{LNN}} = \nabla_{\dot{q}} \nabla_{\dot{q}}^\top L$	$\mathcal{M}(q)$ — true inertia, q-only
$\nabla_{\dot{q}} L$	$\mathcal{M}(q)\,\dot{q}$ — generalised momentum
$\nabla_q L$	$\frac{1}{2}\nabla_q(\dot{q}^\top \mathcal{M}\dot{q}) - \nabla_q V$
$(\nabla_q \nabla_{\dot{q}}^\top L)\,\dot{q}$	$\dot{\mathcal{M}}(q)\,\dot{q}$

Inserted into the LNN implicit equation, terms group into the standard form:

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

with $c = \dot{\mathcal{M}}\dot{q} - \frac{1}{2}\nabla_q(\dot{q}^\top \mathcal{M}\dot{q})$ and $g = \nabla_q V$.

The Manipulator Equation

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

- $\mathcal{M}(q)$: inertia matrix (**must be symmetric & positive definite**)
- $V(q)$: potential energy
- $c(q, \dot{q})$: Coriolis and centrifugal forces
- $g(q) = \partial V / \partial q$: gravity
- τ: applied torque (motor torques for a robot)

Not a new dynamics law — this is what the generic LNN equation *becomes* when L is restricted to the mechanical class. DeLaN's job is to enforce that restriction architecturally.

The Cholesky Trick

Network 1 outputs a lower-triangular $L_d(q)$:

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

Inertia matrix is constructed as:

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

The Cholesky Trick

Network 1 outputs a lower-triangular $L_d(q)$:

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

Inertia matrix is constructed as:

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

By construction, $\mathcal{M}(q)$ is:

- Symmetric (form $AA^\top$)
- Positive definite (diagonal $l_{ii} > 0$ via softplus, plus ϵI)
- Depends only on q, not $\dot{q}$

Network 2 outputs a scalar potential $V_\varphi(q)$.

DeLaN Forward Pass

Given $(q, \dot{q}, \ddot{q})$, compute predicted torque $\hat{\tau}$:

1. $L_d(q) = f_\theta(q)$, then $\mathcal{M}(q) = L_d L_d^\top + \epsilon I$
2. $V_\varphi(q)$
3. Gravity: $g(q) = \partial V_\varphi / \partial q$ (autodiff)
4. $\dot{\mathcal{M}}(q) = \sum_k (\partial \mathcal{M} / \partial q_k)\, \dot{q}_k$
5. Kinetic gradient: $\partial / \partial q \, (\frac{1}{2} \dot{q}^\top \mathcal{M} \dot{q})$
6. Coriolis: $c(q, \dot{q}) = \dot{\mathcal{M}} \dot{q} - \frac{1}{2} \partial / \partial q \, (\dot{q}^\top \mathcal{M} \dot{q})$
7. Torque: $\hat{\tau} = \mathcal{M}(q)\ddot{q} + c(q, \dot{q}) + g(q)$

All steps are differentiable — backprop flows through to both networks.

DeLaN Loss

Training data $\{(q_i, \dot{q}_i, \ddot{q}_i, \tau_i)\}_{i=1}^{N}$ — torques are observed.

Forward loss (cheap — no matrix inverse):

$$\mathcal{L}_{\text{forward}} = \frac{1}{N} \sum_i \|\hat{\tau}_i - \tau_i\|^2, \qquad \hat{\tau} = \mathcal{M}\ddot{q} + c + g$$

Inverse loss (one $\mathcal{M}^{-1}$ per sample, but $\mathcal{M}$ is SPD by construction):

$$\mathcal{L}_{\text{inverse}} = \frac{1}{N} \sum_i \|\hat{\ddot{q}}_i - \ddot{q}_i\|^2, \qquad \hat{\ddot{q}} = \mathcal{M}^{-1}(\tau - c - g)$$

Combined:

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

Why different from LNN? LNN has no τ in its formulation — only $\mathcal{L}_{\text{inverse}}$-style is available, and its M_{LNN}^{-1} is not guaranteed well-conditioned.

Why the Cholesky Trick Matters: DeLaN vs Vanilla LNN

Aspect	Vanilla LNN	DeLaN
Output	Scalar $L_\theta(q, \dot{q})$	$L_d(q) \to \mathcal{M}(q)$, plus $V_\varphi(q)$
Mass matrix	Implicit Hessian	Explicit $L_d L_d^\top + \epsilon I$
Symmetric / PD?	Not guaranteed	Guaranteed
Velocity dependence	May depend on $\dot{q}$	Only on q
Derivatives needed	First + second order	First order only
Inductive bias	Generic Lagrangian	Mechanical Lagrangian

The principle

Don't penalize wrong outputs in the loss — make wrong outputs *unrepresentable*.

Example: Spring Pendulum

A 2-DOF system in polar coordinates (r, θ) with nonlinear coupling:

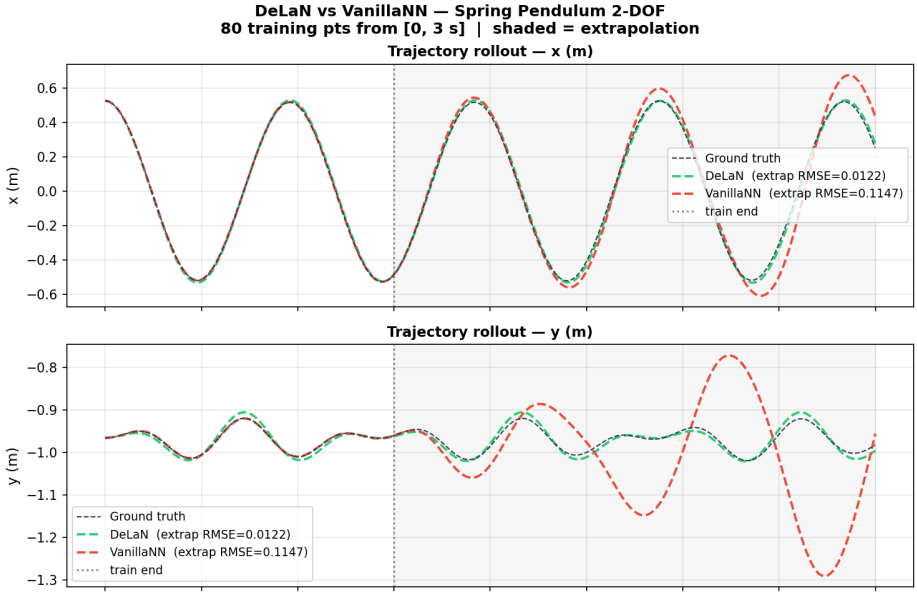
$$\ddot{r} = r\dot{\theta}^2 - g(1 - \cos\theta) - 2k(r - r_0)$$
$$\ddot{\theta} = \frac{-g\sin\theta - 2\dot{r}\dot{\theta}}{r}$$

Setup:

- 80 training samples over $t \in [0, 3]$ s
- Evaluate over $t \in [0, 8]$ s
- Compare DeLaN vs. VanillaNN

The nonlinear couplings make the mass matrix non-trivial: VanillaNN must learn that structure implicitly.

Example: Results



Putting It Together

- **Vanilla NN** — no physics; pure observational bias
- **PINN** — PDE residual in loss (soft, learning bias)
- **LNN** — variational structure in architecture (hard, generic inductive bias)
- **DeLaN** — mechanical structure in architecture (hard, specific inductive bias)

The trade-off

More baked-in physics $\Rightarrow$ less data needed, better extrapolation, narrower applicability.

All four share the same training loop. Only the loss and forward pass differ.

Beyond the Three Methods: A Quick Tour

The field is much bigger. A few directions worth knowing:

- **PINN variants**: XPINN, cPINN, VPINN, B-PINN, Causal PINN
- **Architectural priors**: HNN, SympNets, equivariant networks (NequIP, MACE)
- **Operator learning**: DeepONet, Fourier Neural Operators (FNO), PINO
- **Equation discovery**: SINDy, weak SINDy, PDE-FIND
- **Hybrid methods**: Neural ODEs, Universal Differential Equations

DeepONet: Learning Solution Operators

Instead of one solution to one PDE: learn the **solution operator**

$$\mathcal{G} : \mathcal{U} \to \mathcal{V}, \quad u \mapsto \mathcal{G}(u)$$

mapping input *functions* to output *functions*.

DeepONet: Learning Solution Operators

Instead of one solution to one PDE: learn the **solution operator**

$$\mathcal{G} : \mathcal{U} \to \mathcal{V}, \quad u \mapsto \mathcal{G}(u)$$

mapping input *functions* to output *functions*.

Architecture: two sub-networks

- **Branch** b_θ: encodes input function values at sensor points
- **Trunk** t_φ: encodes the query location y

Output combined via dot product:

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

SINDy: Discovering the Equations Themselves

Different goal: find **interpretable governing equations** from data.

Assume: $\dot{x}(t) = f(x(t))$, and f is **sparse** in some function library.

Build a candidate library:

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

Solve the regression:

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

with sparsity:

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

Output: a closed-form equation, not a black-box network.

Choosing a Method

When you need...	Reach for...
PDE solution, sparse data, known equations	PINN
Many solutions across a parameter family	DeepONet, FNO
Long-term stable rollouts of conservative systems	LNN, HNN, DeLaN
Atomistic / molecular systems with symmetries	Equivariant networks
Interpretable closed-form equations	SINDy
Hybrid known + unknown physics	UDEs

The lesson

No method dominates. Match the inductive bias to what you actually know.

Exercises Overview

All scripts and Colab-ready notebooks at:

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

Each exercise has:

- A Python script in `excercises/<method>/`
- A matching `.ipynb` with an *Open in Colab* badge at the top
- A README explaining the task and what to observe

How to run

Either: clone locally and `python excercises/.../<script>.py`, or click the Colab badge in the notebook on GitHub and *Runtime* → *Run all*.

PINN Exercises (Lecture 1)

Four exercises building on `pinn_vs_nn_mck.py`:

1. `pinn_identification.py` — **learn the damping** c from data. Compare *raw* vs *softplus* parameterisation; the ODE residual is the only signal that drives c.
2. `pinn_2dof_spring_damper.py` — **2-DOF coupled MCK**. Network outputs (x_1, x_2); one residual per DOF, normalised by $\lambda_{\max}(K)$.
3. `pinn_multi_traj.py` — **generalise across initial conditions**. Augment input to $(t, x_0, \dot{x}_0)$; add an autograd-enforced IC loss.
4. `pinn_spring_pendulum.py` — **PINN on the 2-DOF spring pendulum**. Position-only data; compare energy drift against LNN/DeLaN on the same system.

Key skills: autograd residuals, residual normalisation, parameter identification.

LNN Exercise (Lecture 2)

One exercise building on `lnn_vs_vanilla.py`:

`lnn_friction.py` — **Extend LNN to dissipative systems.**

System: pendulum with viscous friction $\ddot{\theta} = -(g/\ell)\sin\theta - b\,\dot{\theta}$.

Add an unconstrained friction network $\hat{F}(\theta, \dot{\theta})$ alongside the conservative LNN:

$$\hat{\ddot{\theta}} = \ddot{\theta}_{\text{LNN}} + \hat{F}/M_{\text{LNN}}$$

Joint loss with detached LNN outputs in the friction term:

$$\ell_{\text{dyn}} = \text{MSE}(\hat{\ddot{\theta}}, \ddot{\theta}_{\text{true}}), \qquad \ell_{\text{fric}} = \text{MSE}\big(\hat{F},\ M_{\text{LNN}} \cdot (\ddot{\theta}_{\text{true}} - \ddot{\theta}_{\text{LNN}})\big)$$

Key skill: jointly learn conservative dynamics and a separate residual force network.

DeLaN Exercises (Lecture 2)

Two exercises building on `delan_vs_vanilla.py`:

1. `delan_friction_pendulum.py` — **DeLaN-F on the 1-DOF pendulum**. Add a positive scalar $B(q) = b(q)^2 + \varepsilon$ to the EoM:

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

   Structured friction $\Rightarrow$ guaranteed $dE/dt \leq 0$.

2. `delan_friction.py` — **DeLaN-F on the 2-DOF spring pendulum**. Scale B to a 2×2 PSD matrix via Cholesky:

$$B(q) = L_B(q)\, L_B(q)^\top + \varepsilon_B I$$

   Same construction as the inertia matrix $\mathcal{M}$.

Key skill: extend a structurally constrained model with a structurally constrained dissipation term.

Summary

1. Neural networks are universal function approximators trained via the four-step loop: forward, loss, backward, update.
2. Pure data-driven networks struggle in science: too little data, too much physics, too important to extrapolate.
3. Three pathways to embed physics: **observational**, **inductive**, **learning** bias.
4. **PINN**: physics in the loss — flexible, but loss balancing is hard.
5. **LNN**: structure in the architecture — conservation guaranteed.
6. **DeLaN**: mechanical structure in the architecture — maximum physical correctness for rigid bodies.
7. The broader field offers operator learning (DeepONet), equation discovery (SINDy), and hybrid methods (UDEs).

Closing Thought

No model can be built without assumptions.

The question is which to encode, and how.

— Karniadakis et al., 2021

References

Key references:

- Karniadakis et al., *Physics-Informed Machine Learning*, Nature Reviews Physics (2021)
- Raissi, Perdikaris, Karniadakis, *Physics-Informed Neural Networks*, JCP (2019)
- Cranmer et al., *Lagrangian Neural Networks*, arXiv:2003.04630 (2020)
- Greydanus et al., *Hamiltonian Neural Networks*, NeurIPS (2019)
- Lutter, Ritter, Peters, *Deep Lagrangian Networks*, ICLR (2019)
- Lu et al., *Learning Nonlinear Operators via DeepONet*, Nat. Mach. Intell. (2021)
- Brunton, Proctor, Kutz, *Discovering Governing Equations via SINDy*, PNAS (2016)
- Moseley, *So, what is a physics-informed neural network?*, blog (2021): benmoseley.blog — inspired the MCK example
- Ross, *Lagrangian Neural Networks*, blog (2021) & pytorch-lagrangian-nn — inspired the LNN pendulum examples
- Wolf, *Physics-Informed Neural Networks: a simple tutorial with PyTorch*, Medium (2022) — adapted for the cooling-law example

Questions?