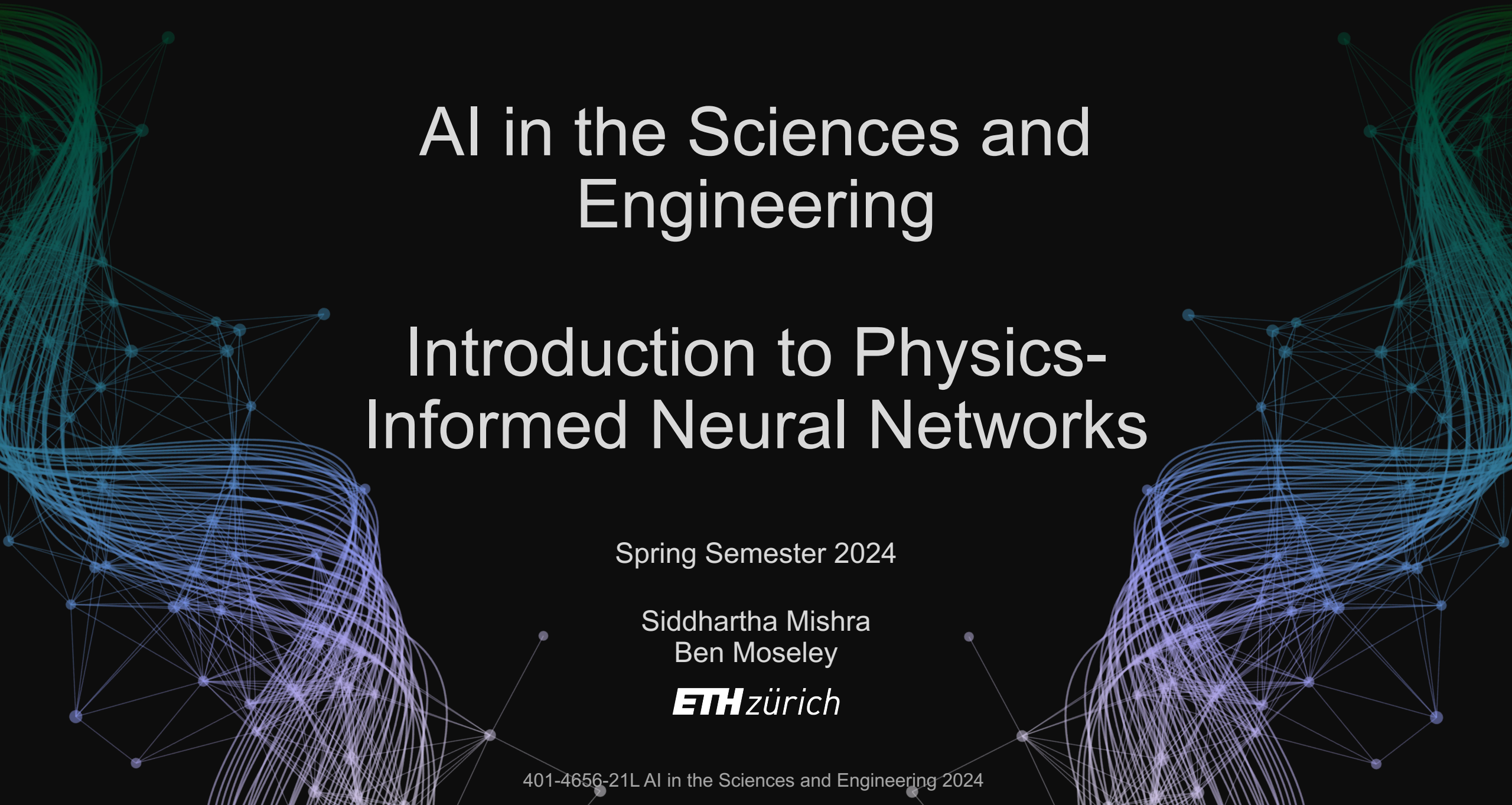


AI in the Sciences and Engineering

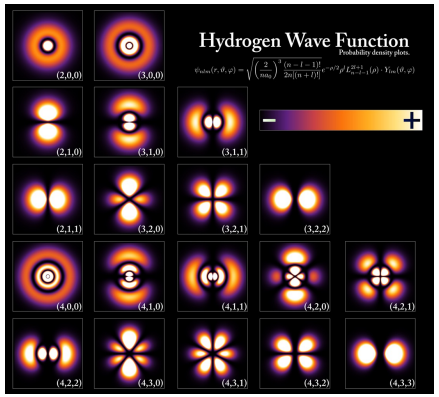
Introduction to Physics-Informed Neural Networks

Spring Semester 2024

Siddhartha Mishra
Ben Moseley

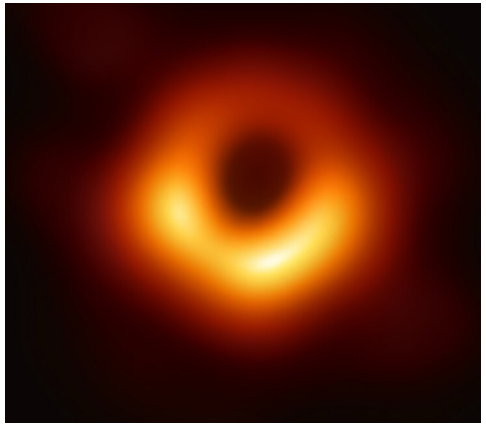
ETH zürich

Recap – importance of PDEs



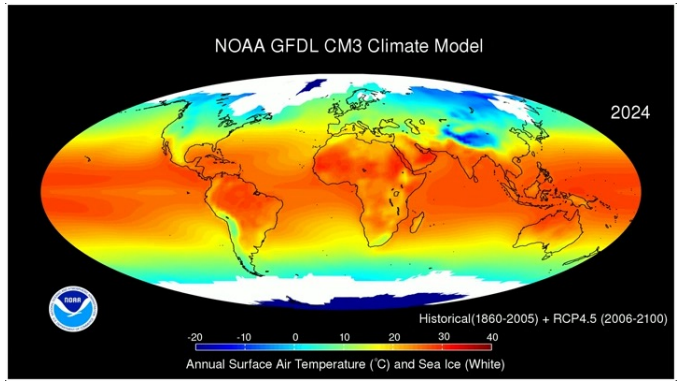
Source: Wikipedia

Schrödinger equation



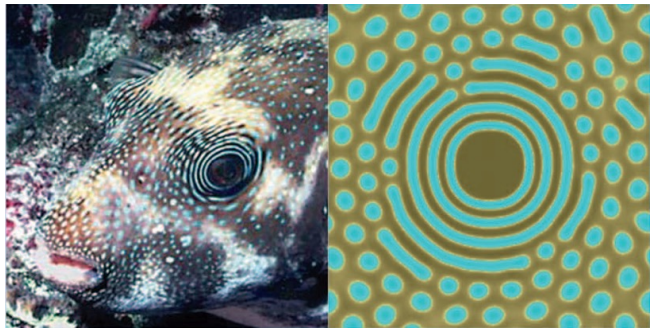
Source: The Event Horizon Telescope (2019)

Einstein field equations



Source: NOAA

Navier-Stokes equations



Source: Kondo and Miura, Science (2010)

Reaction-diffusion equation

Course timeline

Tutorials		Lectures			
Mon 12:15-14:00 HG E 5		*Wed 08:15-10:00 ML H 44*		*Fri 12:15-13:00 ML H 44*	
19.02.		21.02.	~~Course introduction~~	23.02.	~~Introduction to deep learning I~~
26.02.	Introduction to PyTorch	28.02.	~~Introduction to deep learning II~~	01.03.	~~Importance of PDEs in science~~
04.03.	CNNs and surrogate modelling	06.03.	**Introduction to physics-informed neural networks**	08.03.	PINNs – limitations and extensions
11.03.	Implementing PINNs I	13.03.	PINNs – extensions and theory	15.03.	PINNs – theory
18.03.	Implementing PINNs II	20.03.	Introduction to operator learning	22.03.	DeepONets and spectral neural operators
25.03.	Operator learning I	27.03.	Fourier- and convolutional- neural operators	29.03.	
01.04.		03.04.		05.04.	
08.04.	Operator learning II	10.04.	Operator learning – limitations and extensions	12.04.	Introduction to transformers
15.04.		17.04.	Foundational models for operator learning	19.04.	Graph neural networks for PDEs
22.04.	GNNs	24.04.	GNNs for PDEs / introduction to diffusion models	26.04.	Introduction to diffusion models
29.04.	Transformers	01.05.		03.05.	Diffusion models - applications
06.05.	Diffusion models	08.05.	Introduction to differentiable physics	10.05.	Hybrid workflows
13.05.	Coding autodiff from scratch	15.05.	Neural differential equations	17.05.	Introduction to JAX
20.05.		22.05.	Symbolic regression and equation discovery	24.05.	Course summary and future trends
27.05.	Introduction to JAX / NDEs	29.05.	Guest lecture: ML in chemistry and biology	31.05.	Guest lecture: ML in chemistry and biology

Lecture overview

- What are physics-informed neural networks (PINNs)?
- How to train PINNs
  - Live-coding a PINN in PyTorch
- Applications of PINNs
  - Simulation
  - Inversion
  - Equation discovery

Lecture overview

- What are physics-informed neural networks (PINNs)?
- How to train PINNs
  - Live-coding a PINN in PyTorch
- Applications of PINNs
  - Simulation
  - Inversion
  - Equation discovery

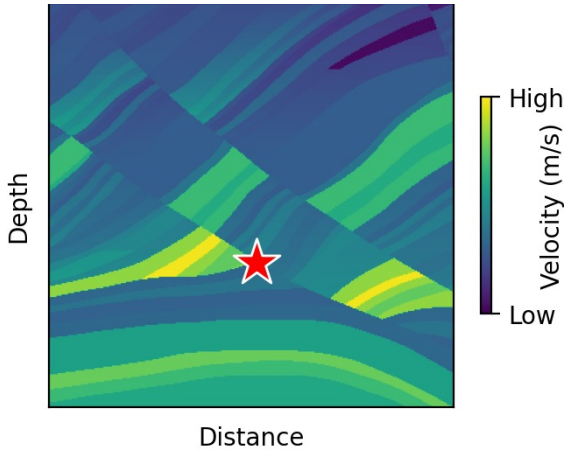
Learning objectives

- Explain what a PINN is, and how they are trained
- Understand how to apply PINNs to different scientific tasks

What is a physics-informed neural network (PINN)?

Using neural networks for simulation

How could we solve this PDE using neural networks?



Wave equation:

$$\nabla^2 u - \frac{1}{c(x,y)^2}\frac{\partial^2 u}{\partial t^2} = s(x,y,t)$$

Initial conditions:

$$u(x,y,t=0) = 0$$
$$u_t(x,y,t=0) = 0$$

What is a PINN?

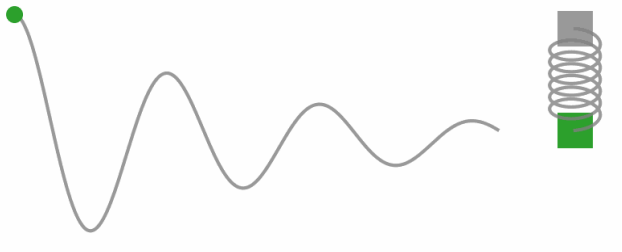
Damped harmonic oscillator:

$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

Initial conditions:

$$u(t=0) = 1$$
$$u_t(t=0) = 0$$

$u =$ displacement
$m =$ mass of oscillator
$\mu =$ coefficient of friction
$k =$ spring constant



Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

What is a PINN?

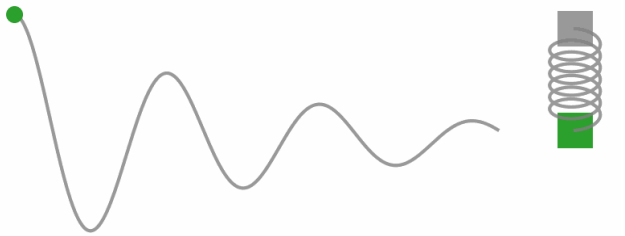
Damped harmonic oscillator:

$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

Initial conditions:

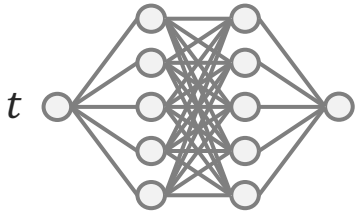
$$u(t=0) = 1$$
$$u_t(t=0) = 0$$

$u =$ displacement
$m =$ mass of oscillator
$\mu =$ coefficient of friction
$k =$ spring constant



Key idea: use a neural network to directly approximate the solution

$NN(t;\theta) \approx u(t)$



$NN(t;\theta) \approx u(t)$

Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

What is a PINN?

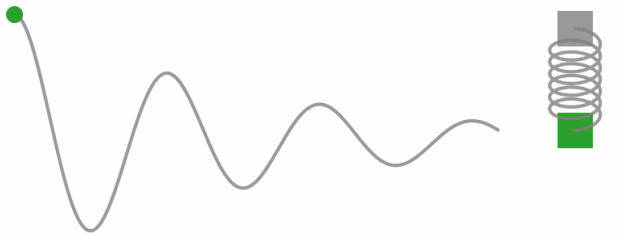
Damped harmonic oscillator:

$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

Initial conditions:

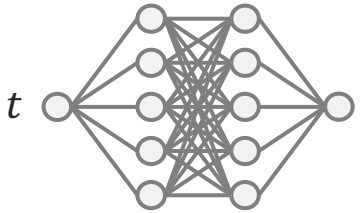
$$u(t=0) = 1$$
$$u_t(t=0) = 0$$

u = displacement
m = mass of oscillator
μ = coefficient of friction
k = spring constant



Key idea: use a neural network to directly approximate the solution

$$NN(t;\theta) \approx u(t)$$



$$NN(t;\theta) \approx u(t)$$

Train the network using the loss function:

Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1(NN(t=0;\theta) - 1)^2 + \lambda_2\left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$ (aka PDE residual)

$$+ \frac{1}{N_p}\sum_i^{N_p}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_i;\theta)\right)^2$$

Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

What is a PINN?

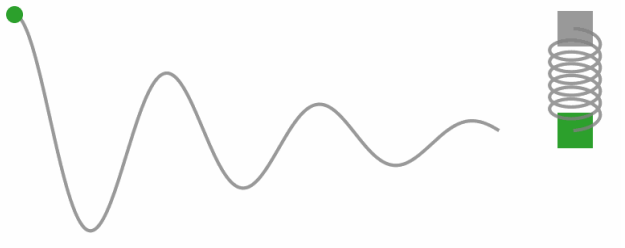
Damped harmonic oscillator:

$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

Initial conditions:

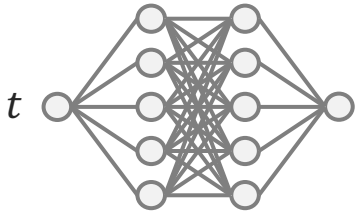
$$u(t=0) = 1$$
$$u_t(t=0) = 0$$

u = displacement
m = mass of oscillator
μ = coefficient of friction
k = spring constant



Key idea: use a neural network to directly approximate the solution

$$NN(t;\theta) \approx u(t)$$



$$NN(t;\theta) \approx u(t)$$

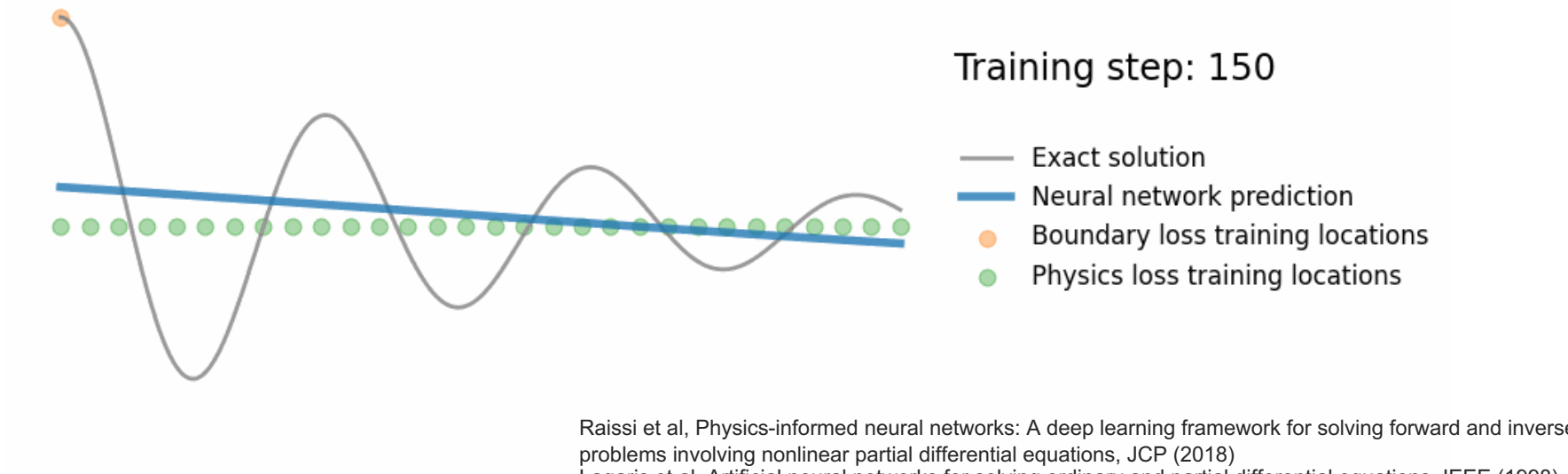
Train the network using the loss function:

Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1(NN(t=0;\theta) - 1)^2 + \lambda_2\left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$ (aka PDE residual)

$$+ \frac{1}{N_p}\sum_i^{N_p}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_i;\theta)\right)^2$$



Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

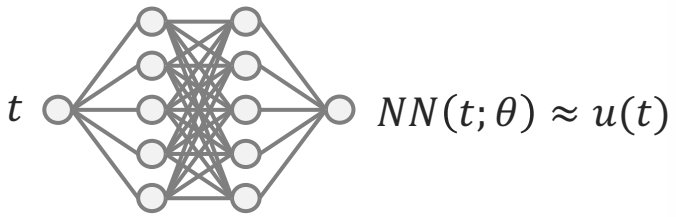
What is a PINN?

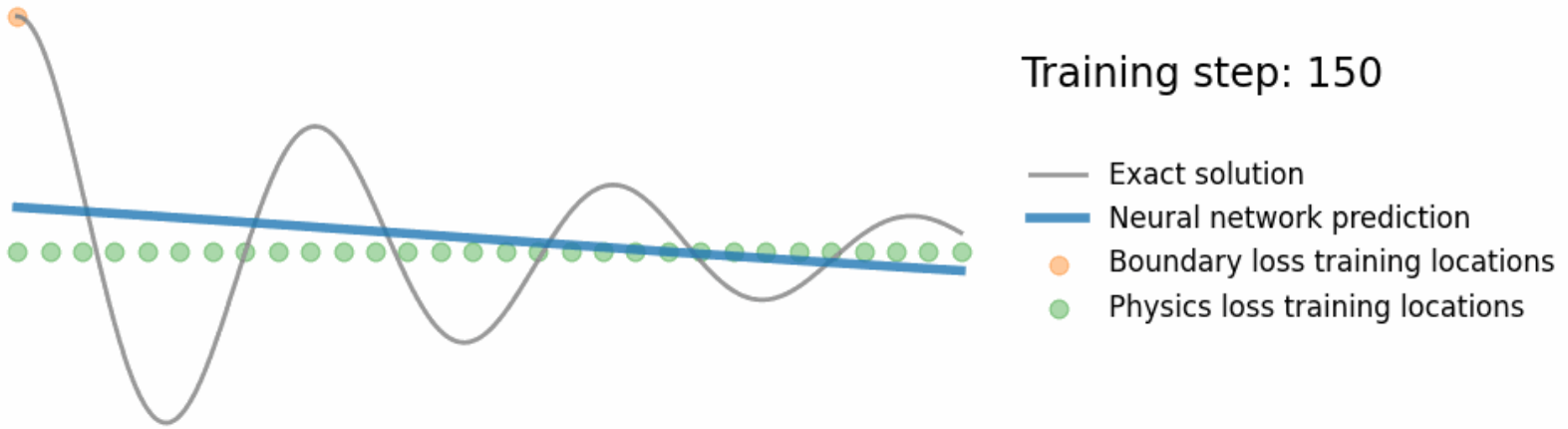
- $\{t_i\}_{i=1}^{N_p}$ are known as **collocation points**, which are sampled throughout the domain
- λ_1, λ_2 are scalar hyperparameters which **balance** the contribution of each loss term

Train the network using the loss function:

$$\text{Boundary loss } L_b(\theta) \begin{cases} L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 \\ \quad + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0 \right)^2 \end{cases}$$

$$\text{Physics loss } L_p(\theta) \text{ (aka PDE residual)} \begin{cases} \quad + \frac{1}{N_p} \sum_i^{N_p} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2 \end{cases}$$





Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

PINNs are a general framework for solving PDEs

Given a PDE and its boundary/initial conditions

$$\mathcal{D}[u(x)] = f(x), \qquad x \in \Omega \subset \mathbb{R}^d$$
$$\mathcal{B}_k[u(x)] = g_k(x), \qquad x \in \Gamma_k \subset \partial\Omega$$

Where $\mathcal{D}$ is some differential operator, $\mathcal{B}_k$ are a set of boundary operators, and $u(x)$ is the solution to the PDE

PINNs train a neural network to **approximate** the solution to the PDE $NN(x;\theta) \approx u(x)$ using the following loss function:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k\left[NN(x_{kj};\theta)\right] - g_k(x_{kj}) \right\|^2 \quad \text{Boundary loss}$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left\| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \right\|^2 \quad \text{Physics loss}$$

PINNs are a general framework for solving PDEs

Given a PDE and its boundary/initial conditions

$$\mathcal{D}[u(x)] = f(x), \qquad x \in \Omega \subset \mathbb{R}^d$$
$$\mathcal{B}_k[u(x)] = g_k(x), \qquad x \in \Gamma_k \subset \partial\Omega$$

Where $\mathcal{D}$ is some differential operator, $\mathcal{B}_k$ are a set of boundary operators, and $u(x)$ is the solution to the PDE

PINNs train a neural network to **approximate** the solution to the PDE $NN(x;\theta) \approx u(x)$ using the following loss function:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k\left[NN(x_{kj};\theta)\right] - g_k(x_{kj}) \right\|^2 \quad \text{Boundary loss}$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left\| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \right\|^2 \quad \text{Physics loss}$$

For example, the 1+1D viscous Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$$

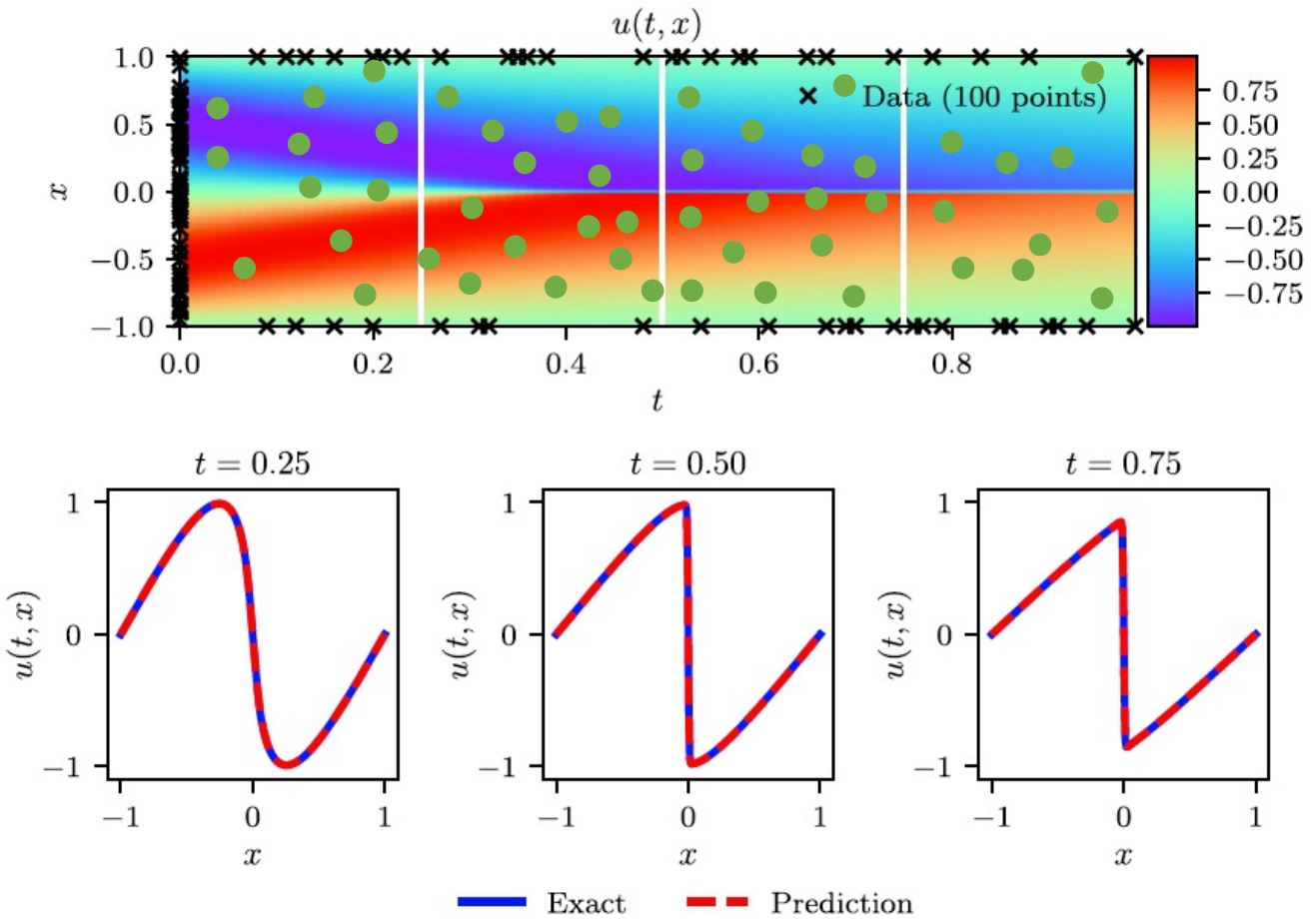
$$u(x,0) = -\sin(\pi x)$$
$$u(-1,t) = u(+1,t) = 0$$

$$u(x,t) \approx NN(x,t;\theta)$$

$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_j^{N_{b1}} \left(NN\left(x_j, 0;\theta\right) + \sin(\pi x_j)\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_k^{N_{b2}} \left(NN(-1,t_k;\theta) - 0\right)^2 + \frac{\lambda_3}{N_{b3}} \sum_l^{N_{b3}} \left(NN(+1,t_l;\theta) - 0\right)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN\frac{\partial NN}{\partial x} - \nu\frac{\partial^2 NN}{\partial x^2} \right)(x_i, t_i;\theta) \right)^2$$

PINNs for solving viscous Burgers' equation

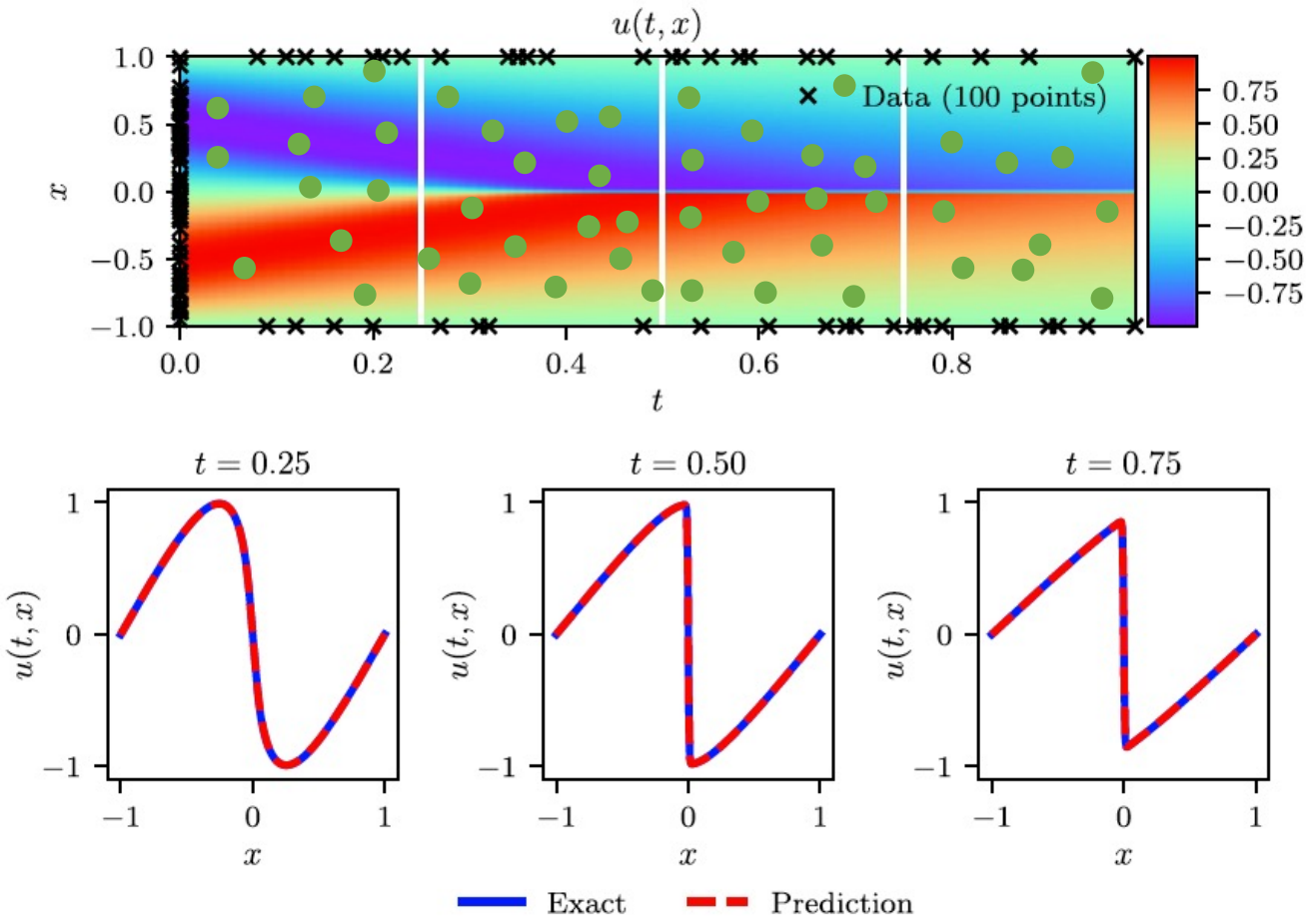


$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_j^{N_{b1}} \left(NN(x_j, 0; \theta) + \sin(\pi x_j)\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_k^{N_{b2}} (NN(-1, t_k; \theta) - 0)^2 + \frac{\lambda_3}{N_{b3}} \sum_l^{N_{b3}} (NN(+1, t_l; \theta) - 0)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN \frac{\partial NN}{\partial x} - \nu \frac{\partial^2 NN}{\partial x^2} \right) (x_i, t_i; \theta) \right)^2$$

Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)

PINNs for solving viscous Burgers' equation



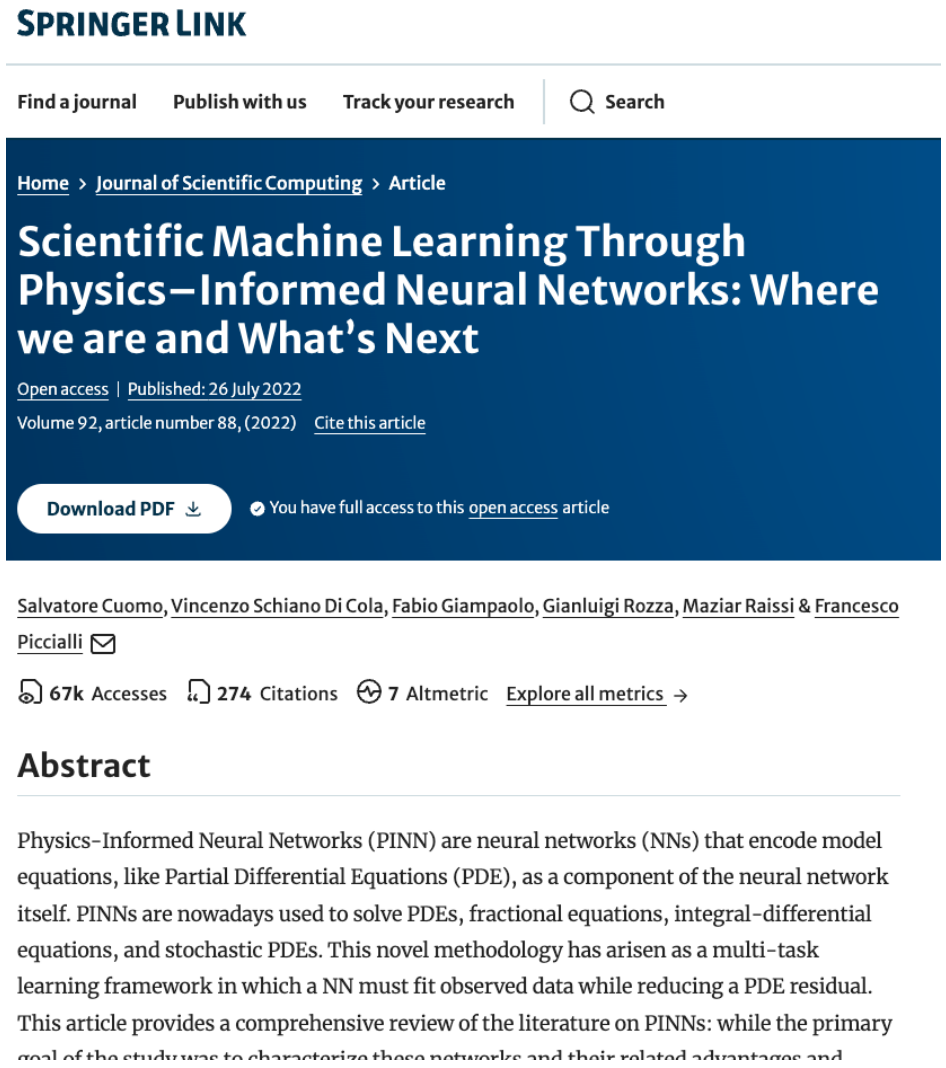
Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)

$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_j^{N_{b1}} \left(NN(x_j, 0; \theta) + \sin(\pi x_j) \right)^2 + \frac{\lambda_2}{N_{b2}} \sum_k^{N_{b2}} (NN(-1, t_k; \theta) - 0)^2 + \frac{\lambda_3}{N_{b3}} \sum_l^{N_{b3}} (NN(+1, t_l; \theta) - 0)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN \frac{\partial NN}{\partial x} - \nu \frac{\partial^2 NN}{\partial x^2} \right) (x_i, t_i; \theta) \right)^2$$

$\nu = 0.01/\pi$
$N_p = 10{,}000$ (Latin hypercube sampling)
$N_{b1} + N_{b2} + N_{b3} = 100$
Fully connected network with 9 layers, 20 hidden units (3021 free parameters)
Tanh activation function
L-BFGS optimiser

PINNs – an entire research field



SPRINGER LINK

Find a journal | Publish with us | Track your research | Search

Home > Journal of Scientific Computing > Article

Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next

Open access | Published: 26 July 2022

Volume 92, article number 88, (2022) Cite this article

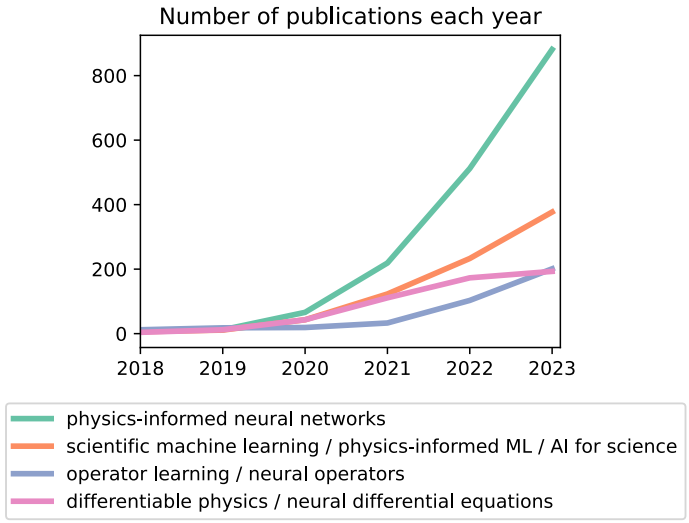
Download PDF | You have full access to this open access article

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi & Francesco Piccialli

67k Accesses | 274 Citations | 7 Altmetric | Explore all metrics →

Abstract

Physics-Informed Neural Networks (PINN) are neural networks (NNs) that encode model equations, like Partial Differential Equations (PDE), as a component of the neural network itself. PINNs are nowadays used to solve PDEs, fractional equations, integral-differential equations, and stochastic PDEs. This novel methodology has arisen as a multi-task learning framework in which a NN must fit observed data while reducing a PDE residual. This article provides a comprehensive review of the literature on PINNs: while the primary goal of the study was to characterize these networks and their related advantages and



Source: Scopus keyword search (Feb 2024)

- The basic concepts behind PINNs were introduced in the 1990s (Lagaris et al, IEEE (1998) and others)
- Raissi et al, JCP (2018) is the seminal paper which reimplemented and extended PINNs using a modern deep learning framework

Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

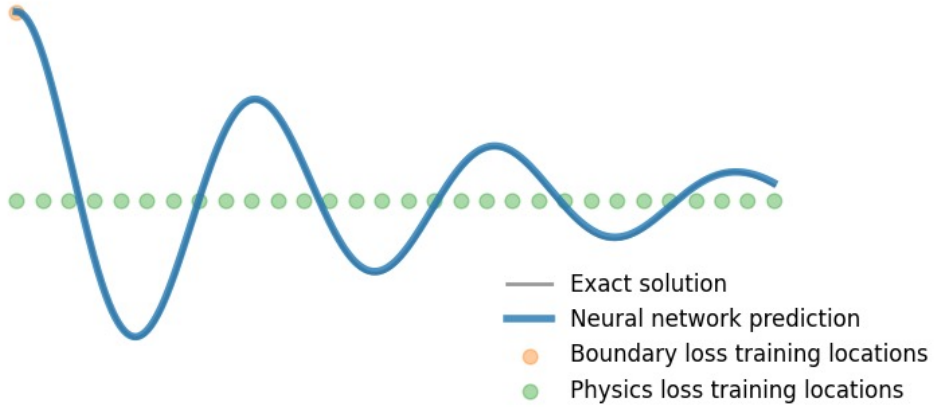
Lecture overview

- What are physics-informed neural networks (PINNs)?
- How to train PINNs
  - Live-coding a PINN in PyTorch
- Applications of PINNs
  - Simulation
  - Inversion
  - Equation discovery

Learning objectives

- Explain what a PINN is, and how they are trained
- Understand how to apply PINNs to different scientific tasks

PINN training loop

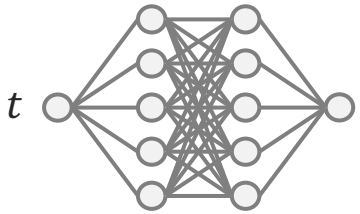


Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

$$+ \frac{1}{N_p} \sum_i^{N_p} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2$$



$$NN(t;\theta) \approx u(t)$$

Training loop:

1. Sample boundary/ physics training points
2. Compute network outputs
3. Compute 1st and 2nd order gradient of network output **with respect to network input**
4. Compute loss
5. Compute gradient of loss function **with respect to network parameters**
6. Take gradient descent step

How can we compute the gradients (e.g. $\frac{dNN}{dt}$ and $\frac{dL}{d\theta}$) required in steps 3 and 5?

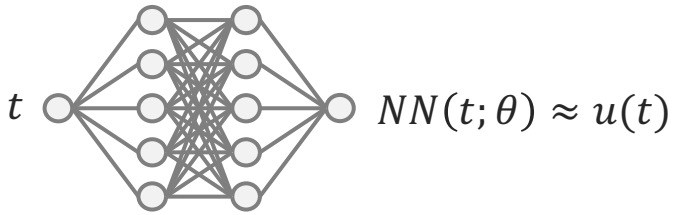
Gradient computation for PINNs

Assume network is a MLP, e.g.:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3 \approx u(t)$$

Analogous to backpropagation, we can use the multivariate **chain rule** to compute gradients with respect to network **inputs**:

$$NN(t;\theta) = f \circ \boldsymbol{g} \circ \boldsymbol{h}(t;\theta)$$
$$f = W_3\sigma(\boldsymbol{g}) + b_3$$
$$\boldsymbol{g} = W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2$$
$$\boldsymbol{h} = W_1 t + \boldsymbol{b}_1$$



$$\frac{\partial NN}{\partial t} = \frac{\partial f}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial t}$$

$$\frac{\partial NN}{\partial t} = W_3\,\mathrm{diag}(\sigma'(\boldsymbol{g}))W_2\,\mathrm{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$

(see Lecture 2: Introduction to Deep Learning Part I)

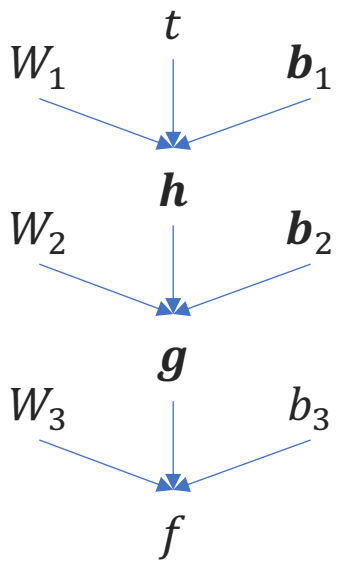
Extending computational graph

Computing gradients can simply be thought of as **extending** the network's computational graph:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3$$

$$\begin{aligned} f &= W_3\sigma(\boldsymbol{g}) + b_3 \\ \boldsymbol{g} &= W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2 \\ \boldsymbol{h} &= W_1 t + \boldsymbol{b}_1 \end{aligned}$$

$$\frac{\partial NN}{\partial t} = W_3\,\mathrm{diag}(\sigma'(\boldsymbol{g}))W_2\,\mathrm{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$



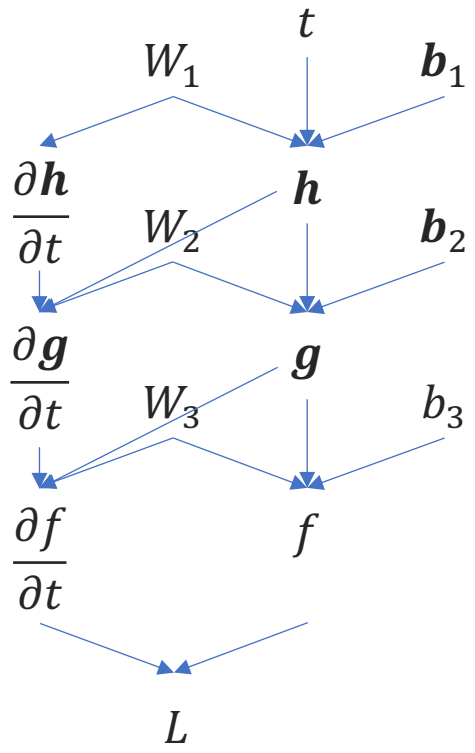
Extending computational graph

Computing gradients can simply be thought of as **extending** the network's computational graph:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3$$

$$\begin{aligned} f &= W_3\sigma(\boldsymbol{g}) + b_3 \\ \boldsymbol{g} &= W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2 \\ \boldsymbol{h} &= W_1 t + \boldsymbol{b}_1 \end{aligned}$$

$$\frac{\partial NN}{\partial t} = W_3\,\mathrm{diag}(\sigma'(\boldsymbol{g}))W_2\,\mathrm{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$



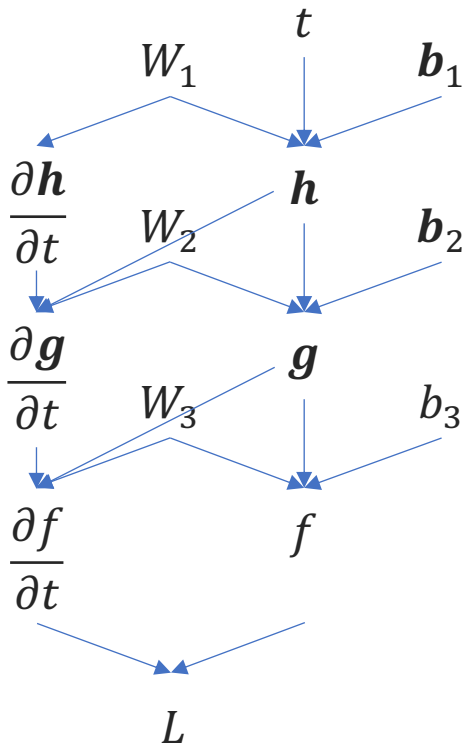
Extending computational graph

Computing gradients can simply be thought of as **extending** the network's computational graph:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3$$

$$f = W_3\sigma(\boldsymbol{g}) + b_3$$
$$\boldsymbol{g} = W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2$$
$$\boldsymbol{h} = W_1 t + \boldsymbol{b}_1$$

$$\frac{\partial NN}{\partial t} = W_3\,\mathrm{diag}(\sigma'(\boldsymbol{g}))W_2\,\mathrm{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$



We can recursively apply **autodifferentiation** to compute gradients and extend the graph

PINN training loop

Training loop:

1. Sample boundary/ physics training points
2. Compute network outputs
3. Compute 1st and 2nd order gradient of network output **with respect to network input <= (recursively) apply autodiff, extending graph**
4. Compute loss
5. Compute gradient of loss function **with respect to network parameters <= apply autodiff on extended graph**
6. Take gradient descent step

We can recursively apply **autodifferentiation** to compute gradients and extend the graph

```
# PINN training psuedocode

#2.
t.requires_grad_(True)# tells PyTorch to start tracking graph
theta.requires_grad_(True)
u = NN(t, theta)

#3.
dudt = torch.autograd.grad(u, t, torch.ones_like(u),
                           create_graph=True)[0]
d2udt2 = torch.autograd.grad(dudt, t, torch.ones_like(u),
                             create_graph=True)[0]

#4.
physics_loss = torch.mean((m*d2udt2 + mu*dudt + k*u)**2)
loss = physics_loss + lambda_*boundary_loss

#5.
dtheta = torch.autograd.grad(loss, theta)[0]
```

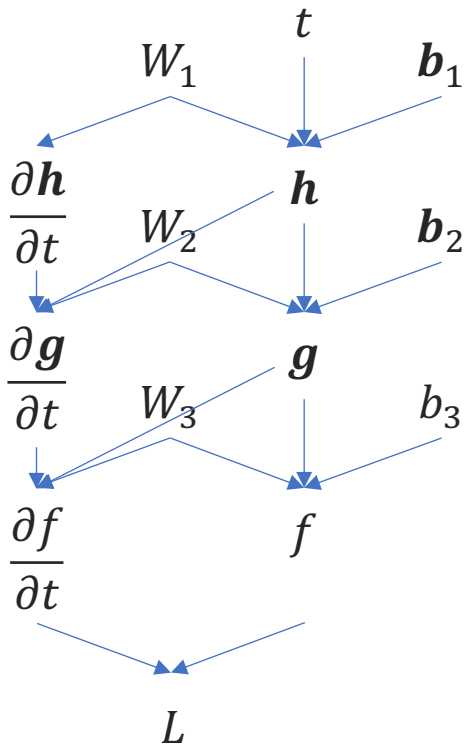
Live-coding a PINN in PyTorch

Computational cost of higher order derivatives

⚠ Note, gradient computation roughly **doubles** the size of the computational graph*

⇒ The cost of evaluating $\frac{\partial^n NN}{\partial t^n}$ grows **exponentially** with n (!)

⇒ Most time is spent computing gradients, not the forward pass, when training PINNs



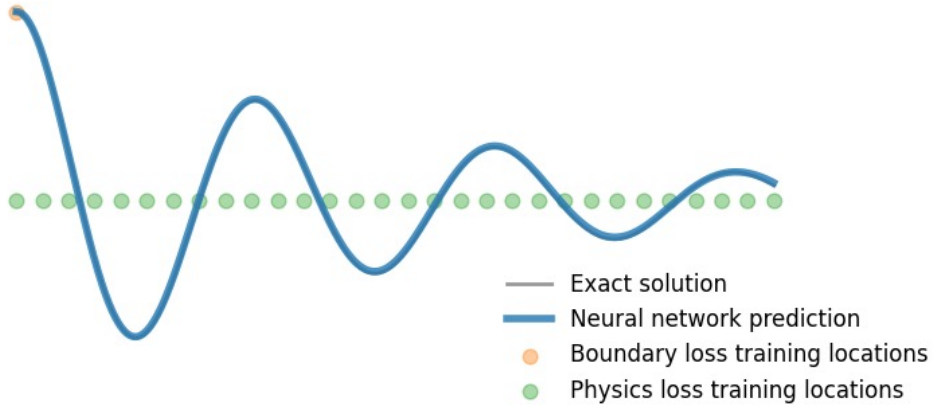
*More precisely, given some $f: \mathbb{R}^n \to \mathbb{R}^m$, its Jacobian $J \in \mathbb{R}^{m \times n}$ and some vector $v \in \mathbb{R}^n$

$$TIME\{f, Jv\} \leq \omega \, TIME\{f\}$$

With a constant $\omega \in [2, 2.5]$ using autodifferentiation

For detailed derivation, see eg: Griewank and Walther, Evaluating Derivatives, Ch 3.1, SIAM (2008))

Other important training considerations

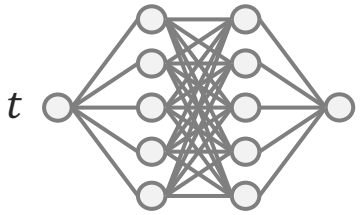


Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

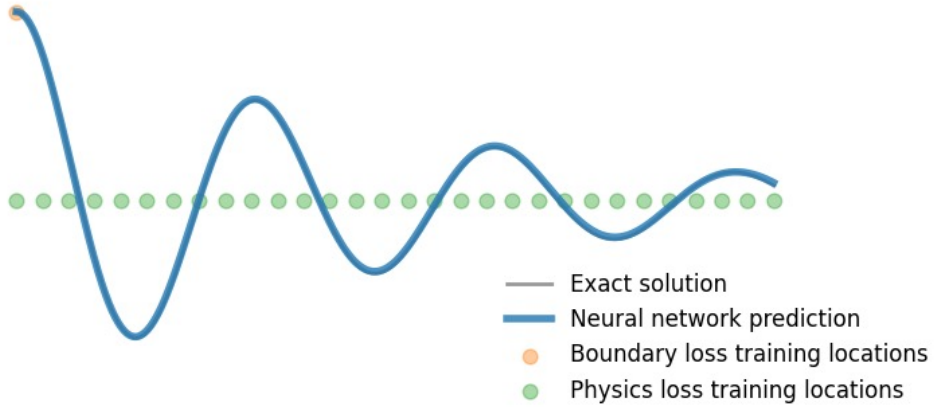
$$+ \frac{1}{N_p} \sum_i^{N_p} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2$$



$NN(t;\theta) \approx u(t)$

- λ values often significantly affect convergence
- "Enough" collocation points $\{t_i\}_{i=1}^{N_p}$ must be chosen so that the learned solution is accurate across the full domain
- Similarly, "enough" boundary points must be chosen such that the learned solution is unique
- Training points are usually sampled uniformly, randomly or quasirandomly
- PINNs still suffer from approximation, estimation and optimisation error (just like normal neural networks, see Lecture 3: Introduction to Deep Learning Part II)!

PINNs from a ML perspective

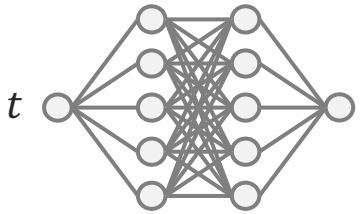


Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

$$+ \frac{1}{N_p} \sum_i^{N_p} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2$$



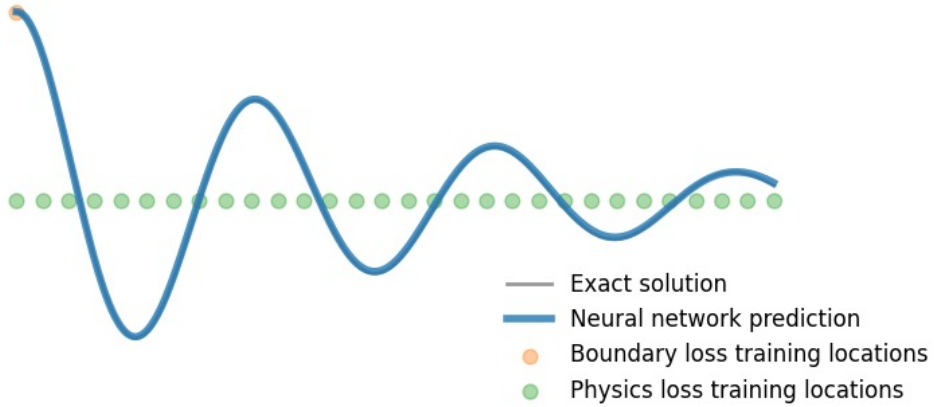
$NN(t;\theta) \approx u(t)$

From a mathematical perspective:

- PINNs provide a way to solve PDEs:
  - Neural network is a mesh-free, **functional** approximation of PDE solution
  - Physics loss is used to assert solution is **consistent** with PDE
  - Boundary loss is used to assert boundary/initial conditions, to ensure solution is **unique**

What can we say about PINNs from a ML perspective?

PINNs from a ML perspective

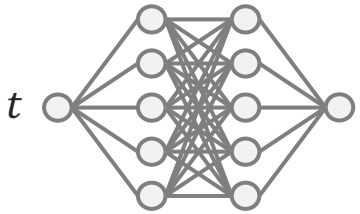


Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

$$+ \frac{1}{N_p} \sum_i^{N_p} \left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2$$



$NN(t;\theta) \approx u(t)$

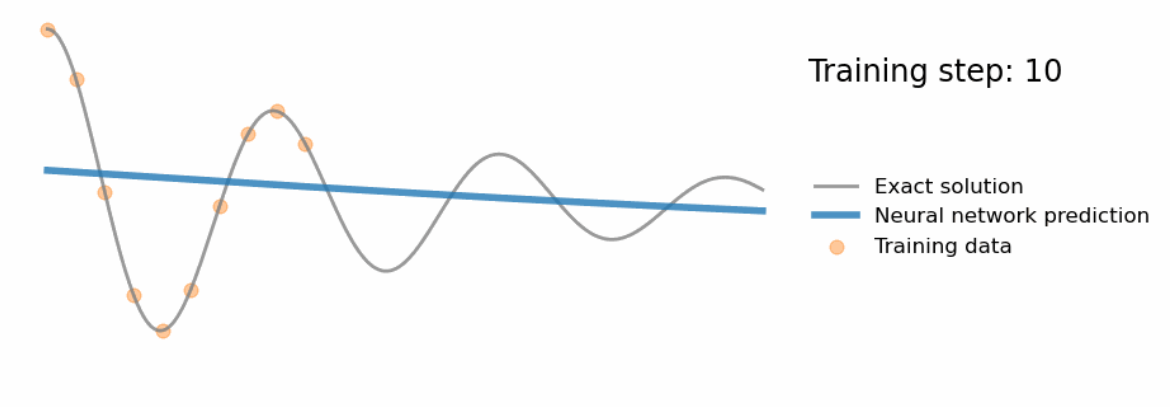
From a mathematical perspective:

- PINNs provide a way to solve PDEs:
  - Neural network is a mesh-free, **functional** approximation of PDE solution
  - Physics loss is used to assert solution is **consistent** with PDE
  - Boundary loss is used to assert boundary/initial conditions, to ensure solution is **unique**

From a ML perspective:

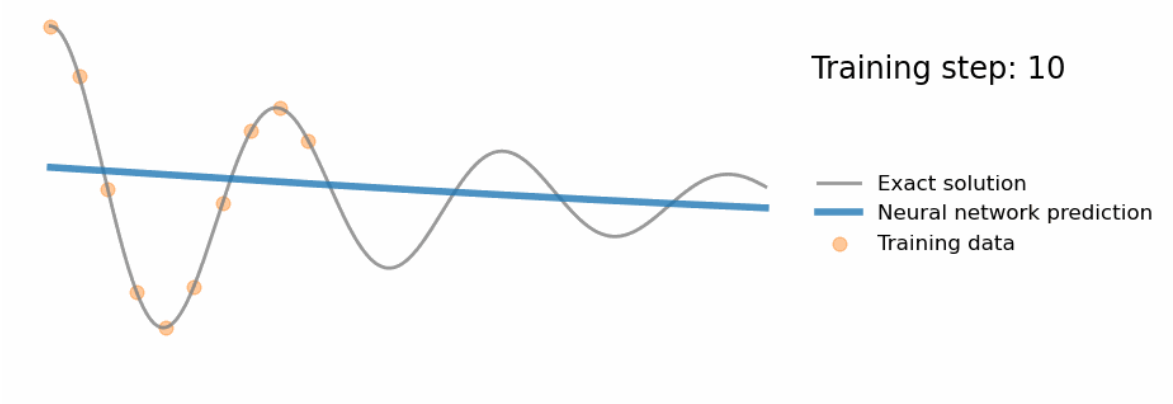
- Physics loss is an **unsupervised** regulariser, which adds prior knowledge

PINNs from a ML perspective

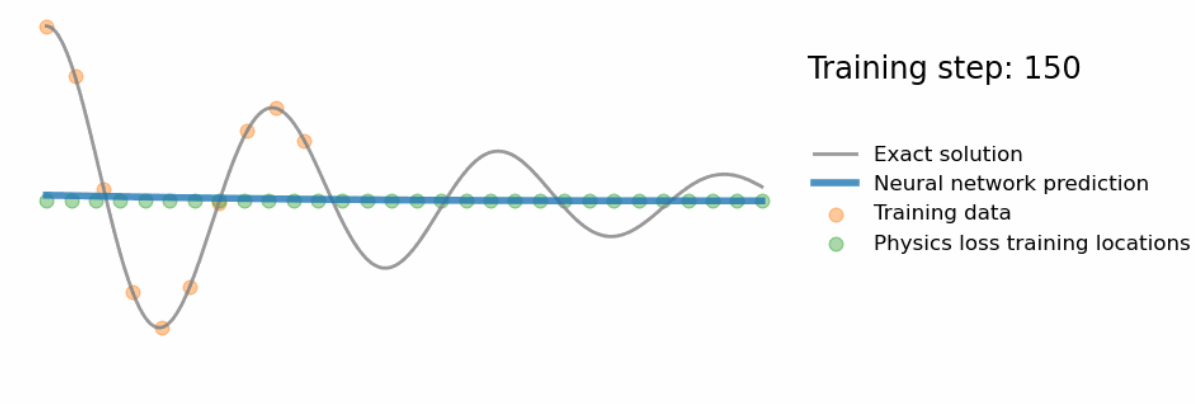


$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u_i)^2$$ Supervised loss

PINNs from a ML perspective



$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u_i)^2$$ Supervised loss



$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u_i)^2$$ Supervised loss

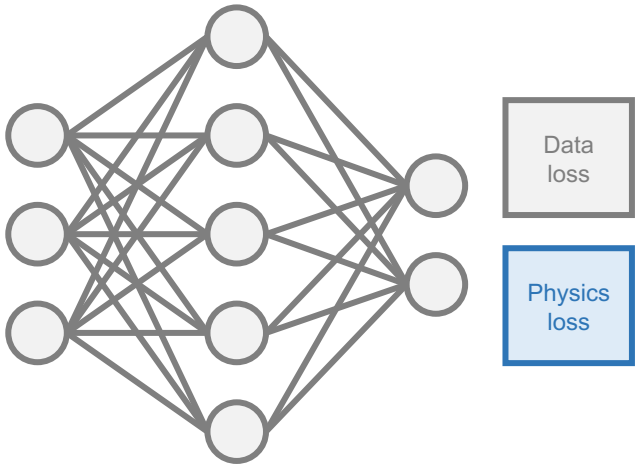
$$+ \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2$$ Physics loss

From a ML perspective:

- Physics loss is an **unsupervised** regulariser, which adds prior knowledge

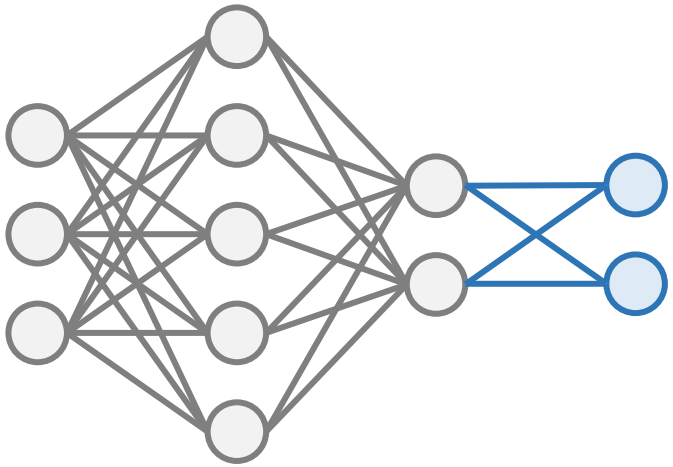
Ways to incorporate scientific principles into machine learning

Loss function



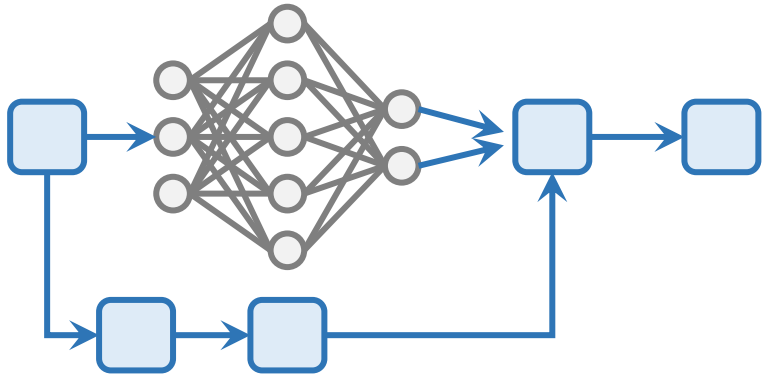
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



Example:
Encoding symmetries / conservation laws
(e.g. energy conservation, rotational invariance)

Hybrid approaches



Example:
Neural differential equations
(incorporating neural networks into PDE models)

Lecture overview

- What are physics-informed neural networks (PINNs)?
- How to train PINNs
  - Live-coding a PINN in PyTorch
- Applications of PINNs
  - Simulation
  - Inversion
  - Equation discovery

Learning objectives

- Explain what a PINN is, and how they are trained
- Understand how to apply PINNs to different scientific tasks

5 min break

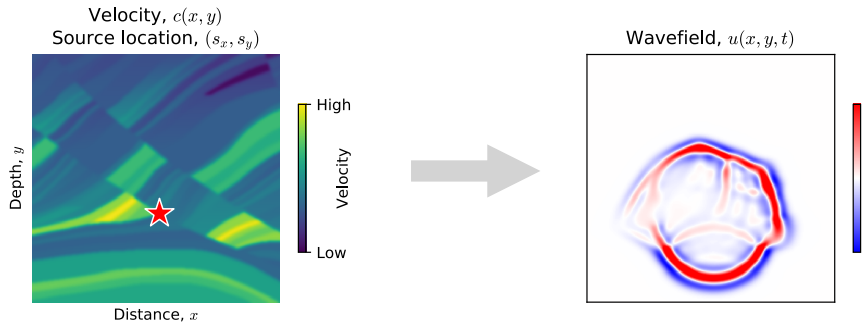
Lecture overview

- What are physics-informed neural networks (PINNs)?
- How to train PINNs
  - Live-coding a PINN in PyTorch
- Applications of PINNs
  - Simulation
  - Inversion
  - Equation discovery

Learning objectives

- Explain what a PINN is, and how they are trained
- Understand how to apply PINNs to different scientific tasks

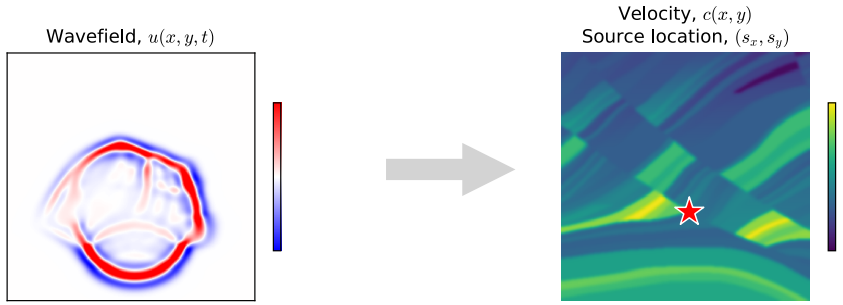
Key scientific tasks



Forward simulation

Estimate wavefield $u(x, y, t)$
Given velocity $c(x, y)$
and source location (s_x, s_y)

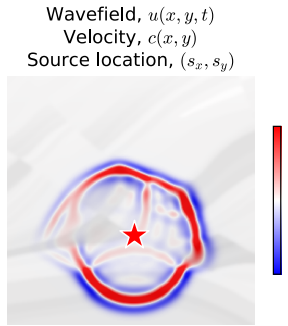
$$b = F(a)$$



Inversion

Estimate velocity $c(x, y)$
and source location (s_x, s_y)
Given wavefield $u(x, y, t)$

$$b = F(a)$$



Wave equation

$$\nabla^2 u - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = f$$

Equation discovery

Estimate governing equation
Given wavefield $u(x, y, t)$,
velocity $c(x, y)$,
and source location (s_x, s_y)

$$b = F(a)$$

PINNs can be used to solve **forward**, **inverse** and **equation discovery** problems related to PDEs

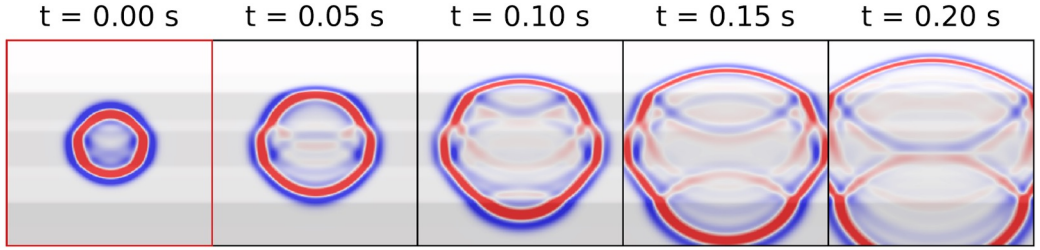
a = set of input conditions

F = physical model of the system (usually a PDE)

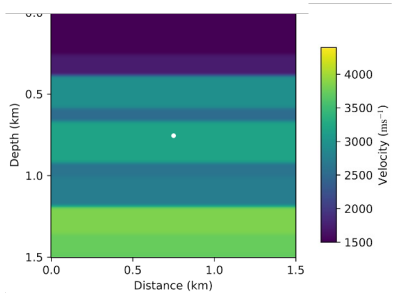
b = resulting properties given F and a

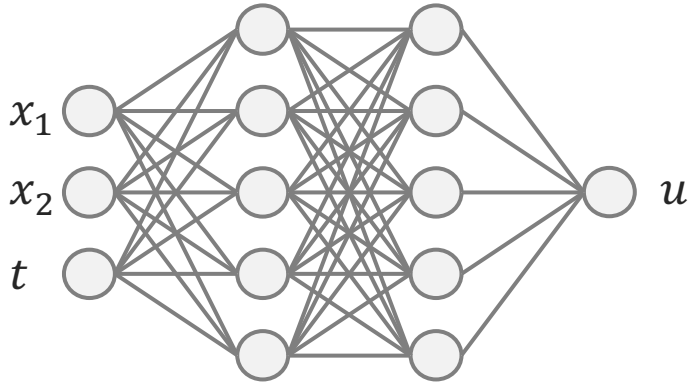
PINNs for solving wave equation

Ground truth FD simulation



Velocity model, $c(x)$





Moseley et al, Solving the wave equation with physics-informed deep learning, ArXiv (2020)

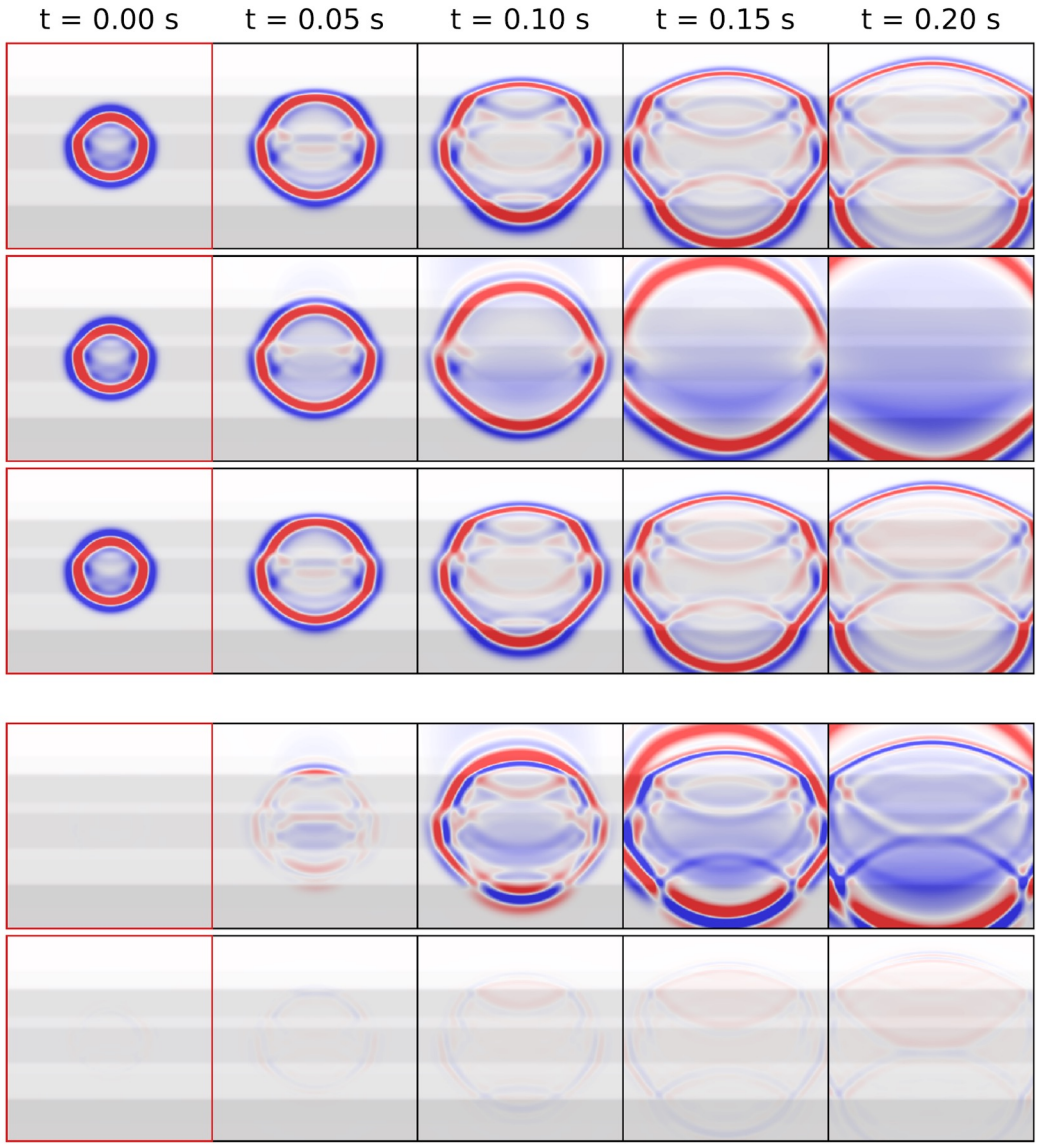
$$L_b(\theta) = \frac{\lambda}{N_b}\sum_{j}^{N_b}\left(NN(x_j, t_j; \theta) - u_{FD}(x_j, t_j)\right)^2$$

Boundary data from FD simulation (first 0.02 seconds)

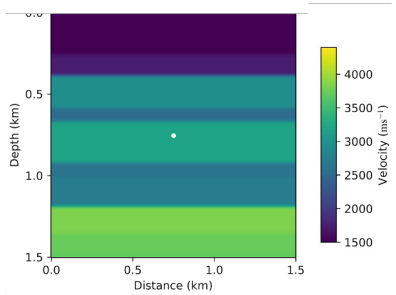
$$L_p(\theta) = \frac{1}{N_p}\sum_{i}^{N_p}\left(\left[\nabla^2 - \frac{1}{c(x_i)^2}\frac{\partial^2}{\partial t^2}\right] NN(x_i, t_i; \theta)\right)^2$$

Collocation points randomly sampled over entire domain (up to 0.2 seconds)

PINNs for solving wave equation



Velocity model, $c(x)$



Moseley et al, Solving the wave equation with physics-informed deep learning, ArXiv (2020)

Mini-batch size $N_b = N_p = 500$ (random sampling)
Fully connected network with 10 layers, 1024 hidden units
Softplus activation
Adam optimiser

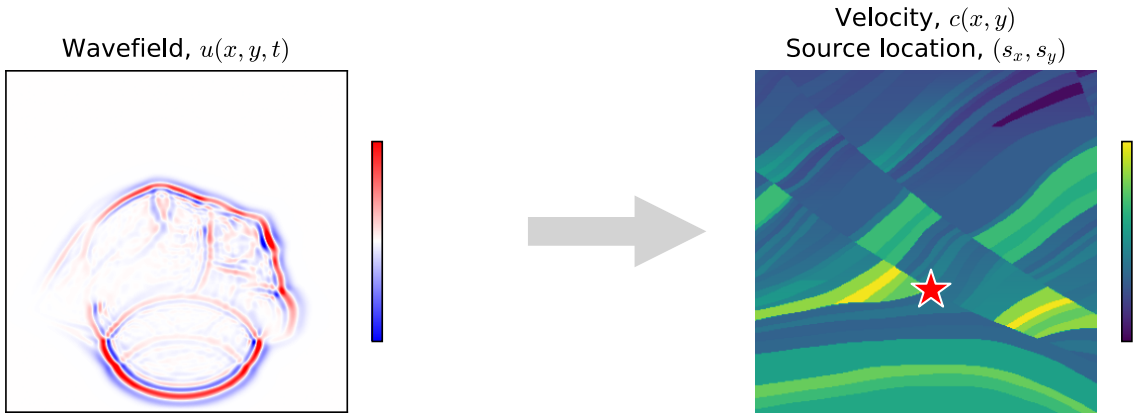
Training time: **~1 hour**

PINNs for inverse problems

What is an inverse problem?

Wave equation:

$$\nabla^2 u - \frac{1}{c(x)^2}\frac{\partial^2 u}{\partial t^2} = 0$$



$b =$ observed wavefield $u(x, t)$

$a =$ (velocity model $c(x)$, source location)

- Fundamentally, inverse problems are **search** problem
- It is often useful to frame them as an optimisation problem, for example:

$$\min_{\hat{a}} \|b - F(\hat{a})\|^2$$

$$b = F(a)$$

$a =$ set of input conditions

$F =$ physical model of the system

$b =$ resulting properties given F and a

PINNs for inversion

PINNs for solving **forward** simulation:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k\left[NN(x_{kj};\theta)\right] - g_k(x_{kj}) \right\|^2 \text{ Boundary loss}$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \|^2 \text{ Physics loss}$$

For example:

$$D = \left[\nabla^2 - \frac{1}{c(x)^2} \frac{\partial^2}{\partial t^2} \right]$$
$$f = 0$$

PINNs for inversion

PINNs for solving **forward** simulation:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k\left[NN(x_{kj};\theta)\right] - g_k(x_{kj}) \right\|^2 \text{ Boundary loss}$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \|\mathcal{D}[NN(x_i;\theta)] - f(x_i)\|^2 \text{ Physics loss}$$

For example:

$$D = \left[\nabla^2 - \frac{1}{c(x)^2}\frac{\partial^2}{\partial t^2}\right]$$
$$f = 0$$

PINNs for solving **inverse** problems:

$$L(\theta, \phi) = L_p(\theta, \phi) + L_d(\theta)$$

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_i^{N_p} \|\mathcal{D}[NN(x_i;\theta);\phi] - f(x_i)\|^2 \text{ Physics loss}$$

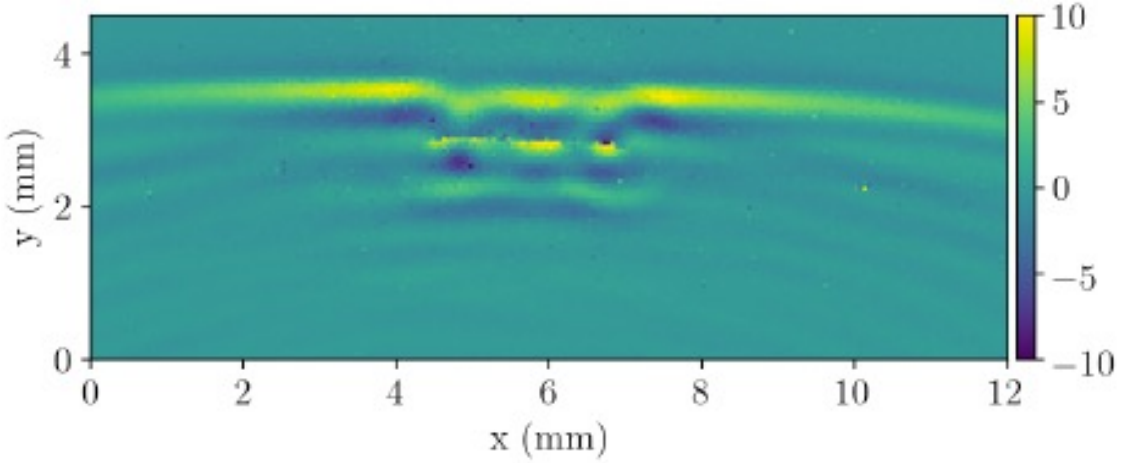
$$L_d(\theta) = \frac{\lambda}{N_d} \sum_l^{N_d} \|NN(x_l;\theta) - u_l\|^2 \quad \textbf{Data loss}$$

Where ϕ are unknown, underlying PDE parameters we wish to invert for, and $\{x_l, u_l\}$ are a set of (potentially noisy) observational data

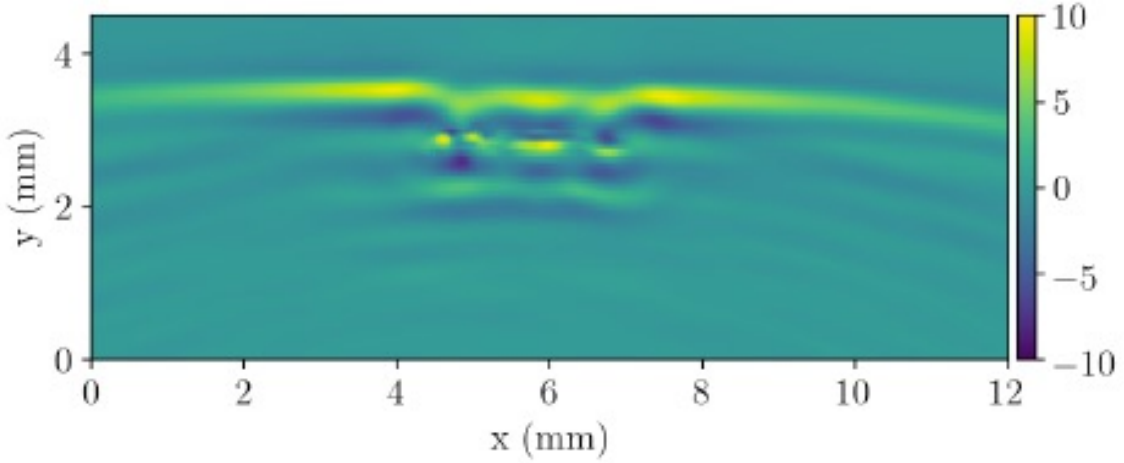
We **simultaneously** learn θ and ϕ when training the PINN

PINNs for wave equation

Shukla et al, Physics-Informed Neural Network for Ultrasound Nondestructive Quantification of Surface Breaking Cracks, Journal of Nondestructive Evaluation (2020)



(a) Actual data at $t = 12.38\ \mu s$.

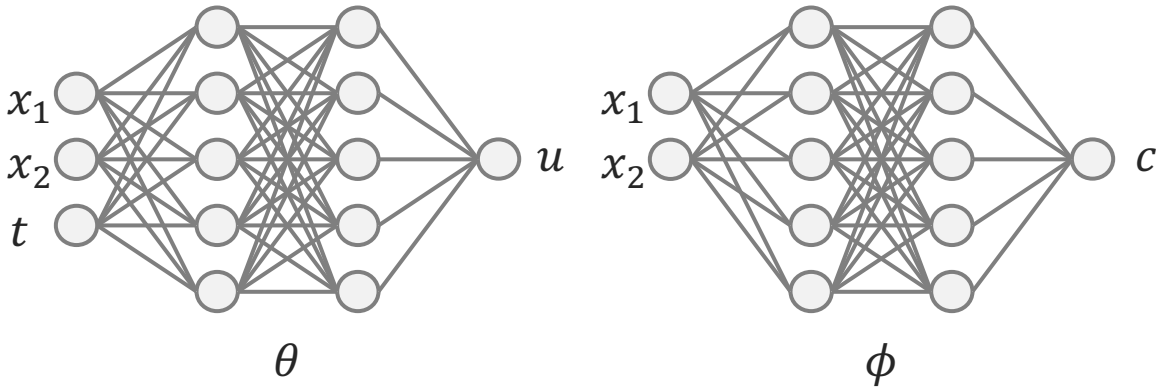


(b) Data recovered from PINN simulation at $t = 12.38\ \mu s$.

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_i^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i; \phi)^2} \frac{\partial^2}{\partial t^2} \right] NN(x_i, t_i; \theta) \right)^2$$

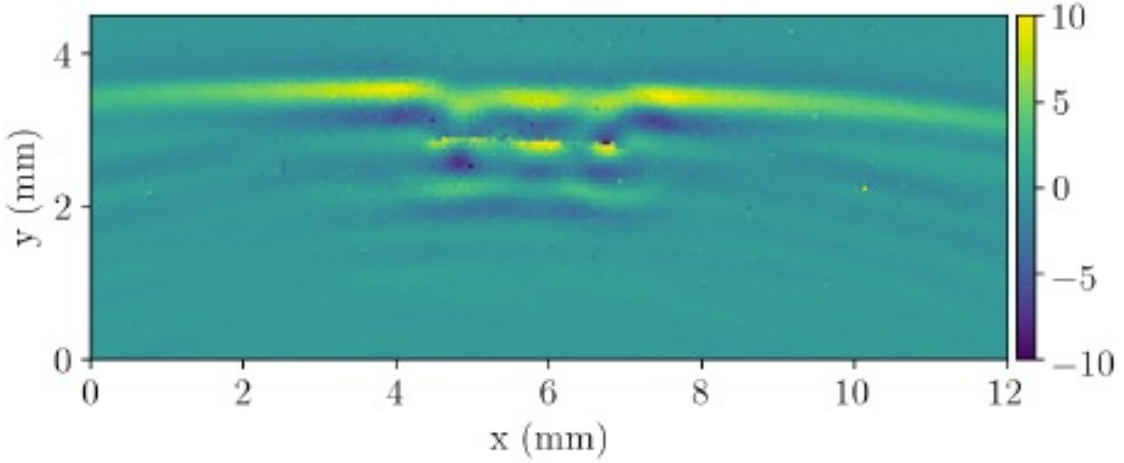
$$L_d(\theta) = \frac{\lambda}{N_d} \sum_l^{N_d} (NN(x_l, t_l; \theta) - u_{obs\,l})^2$$

Treat velocity model as **another** neural network, and simultaneously learn it

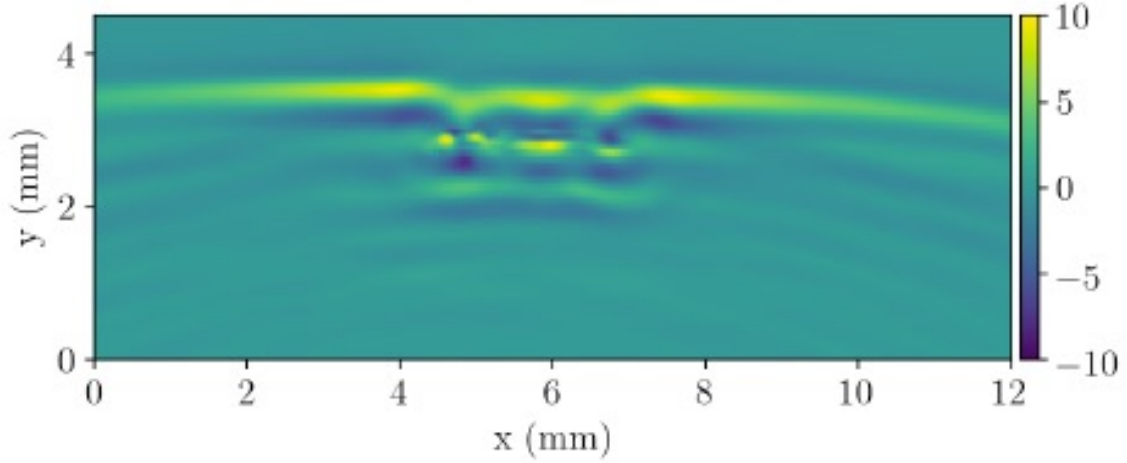


PINNs for wave equation

Shukla et al, Physics-Informed Neural Network for Ultrasound Nondestructive Quantification of Surface Breaking Cracks, Journal of Nondestructive Evaluation (2020)



(a) Actual data at $t = 12.38\ \mu s$.

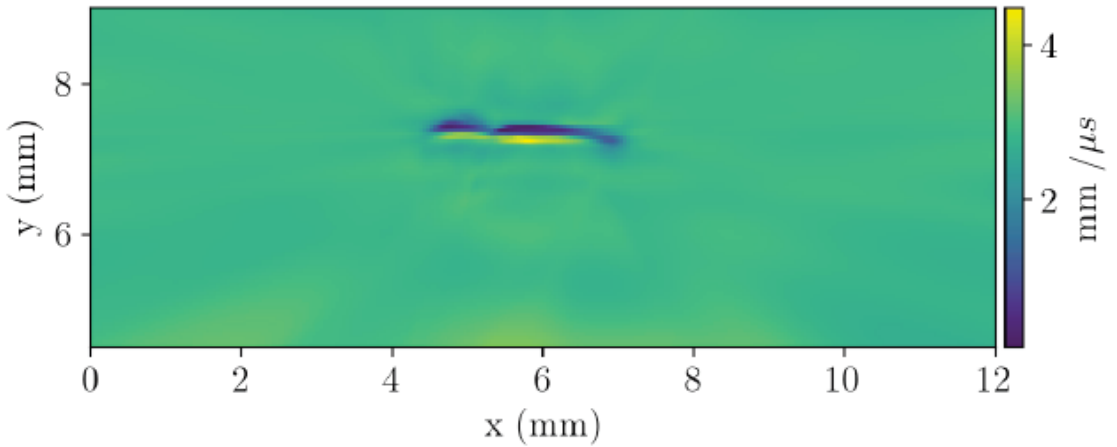


(b) Data recovered from PINN simulation at $t = 12.38\ \mu s$.

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i; \phi)^2} \frac{\partial^2}{\partial t^2} \right] NN(x_i, t_i; \theta) \right)^2$$

$$L_d(\theta) = \frac{\lambda}{N_d} \sum_{l}^{N_d} (NN(x_l, t_l; \theta) - u_{obs\,l})^2$$

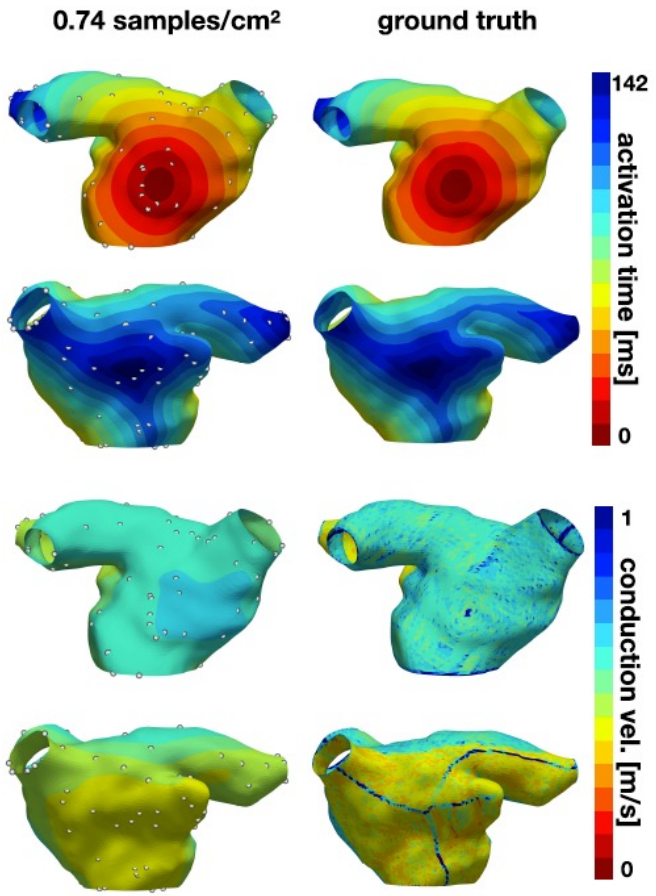
Treat velocity model as **another** neural network, and simultaneously learn it

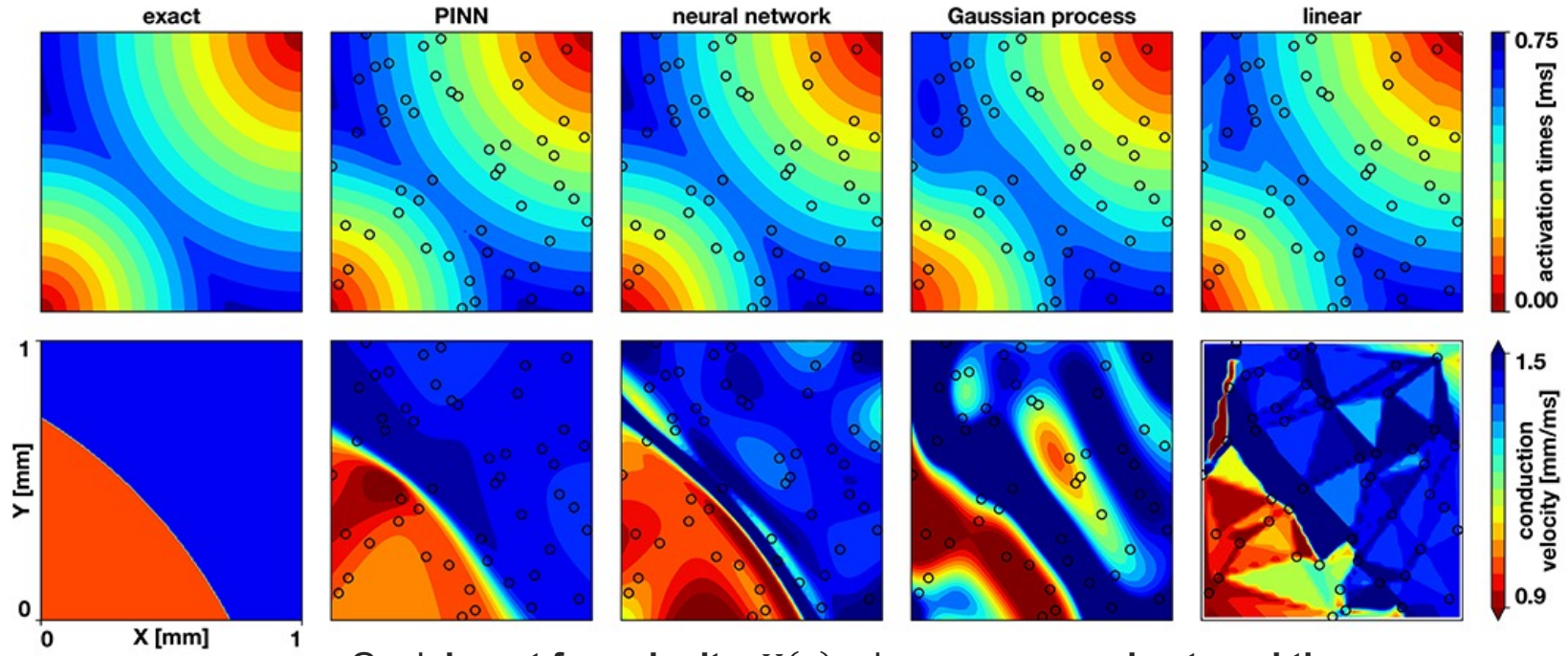


(d) Speed $v(x, y)$ recovered from PINN simulation.

Live-coding a PINN in PyTorch

PINNs for cardiac activation mapping





Goal: **invert for velocity**, $V(x)$, given **sparse, noisy travel time** measurements T_{obs}

Eikonal equation: $\|\nabla T(x)\| = 1/V(x)$

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_{i}^{N_p} (\|\nabla NN(x_i; \theta)\| V(x_i; \phi) - 1)^2$$

$$L_d(\theta) = \frac{\lambda}{N_d} \sum_{l}^{N_d} (NN(x_l; \theta) - T_{obs\,l})^2$$

Sahli Costabal et al, Physics-Informed Neural Networks for Cardiac Activation Mapping, Frontiers in Physics (2020)

PINNs for equation discovery

PINNs for equation discovery

PINNs for solving **inverse** problems:

$$L(\theta, \phi) = L_p(\theta, \phi) + L_d(\theta)$$

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_{i}^{N_p} \|\mathcal{D}[NN(x_i; \theta); \phi] - f(x_i)\|^2 \quad \text{Physics loss}$$

$$L_d(\theta) = \frac{\lambda}{N_d} \sum_{l}^{N_d} \|NN(x_l; \theta) - u_l\|^2 \quad \text{Data loss}$$

Where $\{x_l, u_l\}$ are a set of (potentially noisy) observational data

But how do we learn the **entire** differential operator $\mathcal{D}$, rather than its parameters ϕ?

PINNs for equation discovery

How do we learn an **entire** differential operator $\mathcal{D}$?

Build a **library** of n operators, such as:

$$\phi = (1, \partial_x, \partial_t, \partial_{xx}, \partial_{tt}, \partial_{xt})^T$$

Then assume the differential operator can be represented as

$$\mathcal{D} = \Lambda\phi$$

Where Λ is a (sparse) matrix of shape (d_u, n)

E.g. for 1D damped harmonic oscillator:

$$\mathcal{D} = \begin{pmatrix} k & \mu & m & 0 \end{pmatrix} \begin{pmatrix} 1 \\ d_t \\ d_{tt} \\ d_{ttt} \end{pmatrix}$$

$$= m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k$$

PINNs for equation discovery

How do we learn an **entire** differential operator $\mathcal{D}$?

Build a **library** of n operators, such as:

$$\phi = (1, \partial_x, \partial_t, \partial_{xx}, \partial_{tt}, \partial_{xt})^T$$

Then assume the differential operator can be represented as

$$\mathcal{D} = \Lambda\phi$$

Where Λ is a (sparse) matrix of shape (d_u, n)

E.g. for 1D damped harmonic oscillator:

$$\mathcal{D} = (k \quad \mu \quad m \quad 0)\begin{pmatrix} 1 \\ d_t \\ d_{tt} \\ d_{ttt} \end{pmatrix}$$

$$= m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k$$

PINNs for **equation discovery**:

$$L(\theta, \Lambda) = L_p(\theta, \Lambda) + L_d(\theta)$$

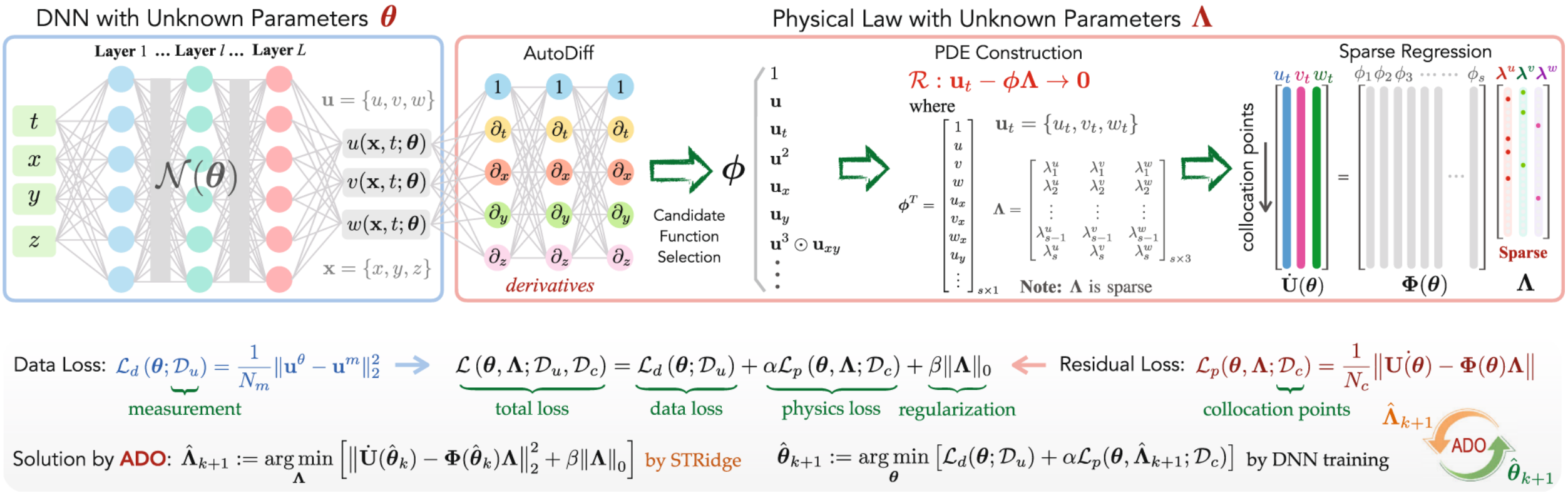
$$L_p(\theta, \Lambda) = \frac{1}{N_p}\sum_i^{N_p} \|\Lambda\phi[NN(x_i; \theta)]\|^2 + \|\Lambda\|^2 \quad \text{Physics loss}$$

$$L_d(\theta) = \frac{\lambda}{N_d}\sum_l^{N_d} \|NN(x_l; \theta) - u_l\|^2 \quad \text{Data loss}$$

Where Λ are treated as **learnable** parameters and $\{x_l, u_l\}$ are a set of (potentially noisy) observational data

Typically, some regularization / prior on Λ (e.g. sparsity) is needed, as this optimisation problem can be very **ill-posed**

PINNs for equation discovery



Data Loss: $\mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) = \frac{1}{N_m}\|\mathbf{u}^\theta - \mathbf{u}^m\|_2^2$ (measurement)

$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_u, \mathcal{D}_c) = \mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) + \alpha\mathcal{L}_p(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_c) + \beta\|\boldsymbol{\Lambda}\|_0$ (total loss = data loss + physics loss + regularization)

Residual Loss: $\mathcal{L}_p(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_c) = \frac{1}{N_c}\|\dot{\mathbf{U}}(\boldsymbol{\theta}) - \boldsymbol{\Phi}(\boldsymbol{\theta})\boldsymbol{\Lambda}\|$ (collocation points)

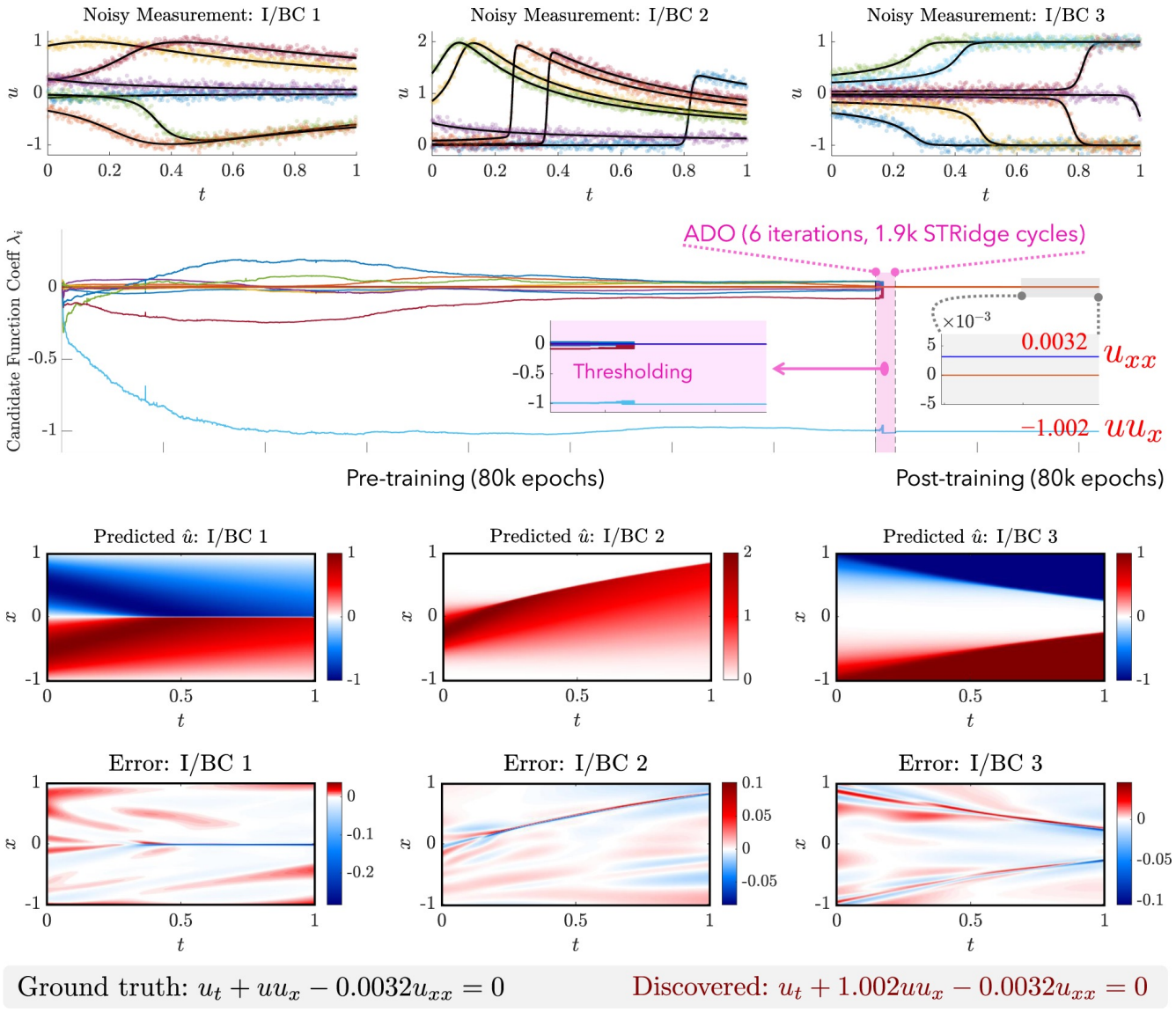
Solution by **ADO**: $\hat{\boldsymbol{\Lambda}}_{k+1} := \arg\min_{\boldsymbol{\Lambda}} \left[\|\dot{\mathbf{U}}(\hat{\boldsymbol{\theta}}_k) - \boldsymbol{\Phi}(\hat{\boldsymbol{\theta}}_k)\boldsymbol{\Lambda}\|_2^2 + \beta\|\boldsymbol{\Lambda}\|_0\right]$ by STRidge

$\hat{\boldsymbol{\theta}}_{k+1} := \arg\min_{\boldsymbol{\theta}} \left[\mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) + \alpha\mathcal{L}_p(\boldsymbol{\theta}, \hat{\boldsymbol{\Lambda}}_{k+1}; \mathcal{D}_c)\right]$ by DNN training

- Trains by alternating between updating Λ and θ

Chen et al, Physics-informed learning of governing equations from scarce data, Nature communications (2021)

PINNs for equation discovery



- PINN “discovering” Burgers’ equation
- By combining datasets sampled under three different I/BCs with 10% noise

Chen et al, Physics-informed learning of governing equations from scarce data, Nature communications (2021)

Lecture summary

- PINNs are an alternative approach for solving **forward, inverse** and **equation discovery** problems related to differential equations
- They are **mesh-free**
- They use a neural network to **directly approximate** the solution to the PDE