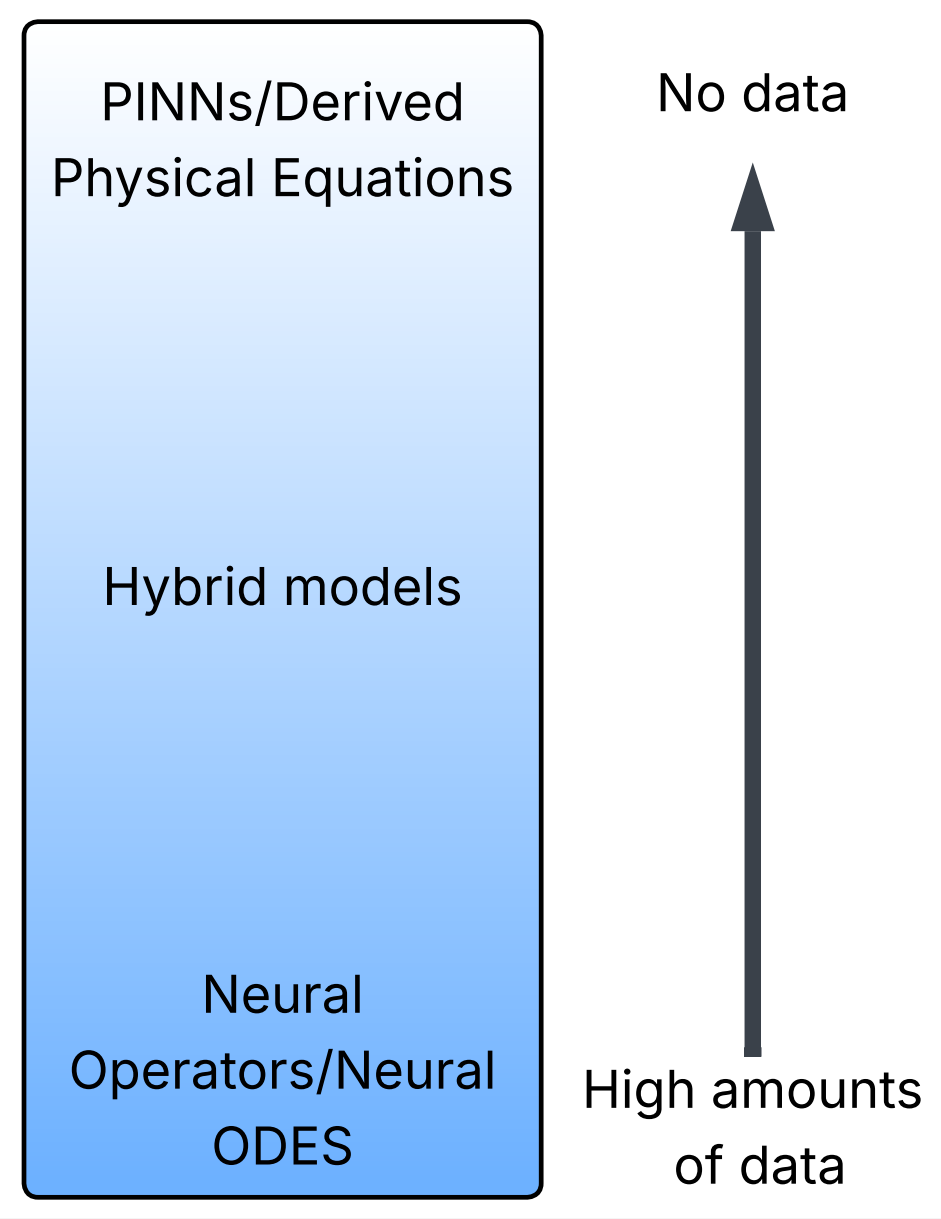
ECE 228 Spring 2026 Lecture 15: Physics Informed Neural Networks

Luke Bhan

May 19, 2026

Is data-driven the sole approach?

- What happens if I just know the equations of my process, but do not have any data?
- What if I have data **and** know the equations that govern the physical phenomena?
- What happens if I only have data?



What are PINNs

Consider our transport example:

$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} &= \frac{\partial u(x,t)}{\partial x}\,, \quad x \in (0,1), \quad t \in \mathbb{R}^+\,, \\ u(x,0) &= g(x)\,, \qquad \text{Initial Condition}\,, \\ u(1,t) &= 0\,, \qquad \text{Boundary Condition}\,, \end{aligned}$$

- We used our PDE solver (expensive) to compute $u(x,t)$ for a bunch of points $x_1, ..., x_n$ and times $t_1, ..., t_m$.

> But, imagine we don't have this PDE solver. What can we do?

So what can we do?

Let us define a neural network $u_\theta(x, t)$ and we would like $u_\theta(x, t)$ to solve this equation such that $u_\theta(x, t) = u(x, t)$.

How would we learn such a PDE?

- Perform supervised learning with the loss:

$$\mathcal{L}(u, u_\theta) = \sum_{i=1}^{n} \sum_{j=1}^{m} \| u(x_i, t_j) - u_\theta(x_i, t_j) \| .$$

- Then, we hope our neural network can interpolate at points around $x_1, ..., x_n$, $t_1, ..., t_m$ for our query.

Warm up: advantages and disadvantages

What advantages/disadvantages can you think of for learning u_θ?

- Can query u_θ at any x, t?
- Not guaranteed that u_θ will be accurate at all points, but possible to do out-of-sample prediction.
- Not obvious that this is better than just using some interpolation solver (Curse of dimensionality and computational cost).
- Completely fails to leverage the decades of physics and math used to develop the equations for u.

PINNs

Idea: Since we have both **data** $u(x_i, t_i)$ for $i = 1, ..., n_{\text{data}}$ and **the governing equation** $u_t = u_x$, $u(x, 0) = g(x)$, $u(1, t) = 0$, can we use both?

Construct the following loss function with $\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1$ as weighting terms

$$\mathcal{L}(u, u_\theta) = \lambda_1 \mathcal{L}_{\text{data}}(u, u_\theta) + \lambda_2 \mathcal{L}_{IC}(u_\theta) + \lambda_3 \mathcal{L}_{BC}(u_\theta) + \lambda_4 \mathcal{L}_{eq}(u_\theta),$$

where

- $\mathcal{L}_{\text{data}}(u, u_\theta) = \sum_{i=1}^{n_{\text{data}}} |u(x_i, t_i) - u_\theta(x_i, t_i)|$ where x_i, t_i are given by dataset availability.
- $\mathcal{L}_{IC}(u_\theta) = \sum_{i=1}^{k_1} |u_\theta(x_i, 0) - g(x_i)|, \quad x_i \in (0, 1)$ chosen by user
- $\mathcal{L}_{BC}(u_\theta) = \sum_{i=1}^{k_2} |u_\theta(0, t_i) - 0|, \quad t_i \in \mathbb{R}^+$ chosen by user
- $\mathcal{L}_{eq}(u_\theta) = \sum_{i=1}^{k_3} \left| \frac{\partial u_\theta}{\partial t}(x_i, t_i) + \frac{\partial u_\theta}{\partial x}(x_i, t_i) \right|, \quad x_i, t_i$ chosen by user. Derivatives computed via auto differentiation.

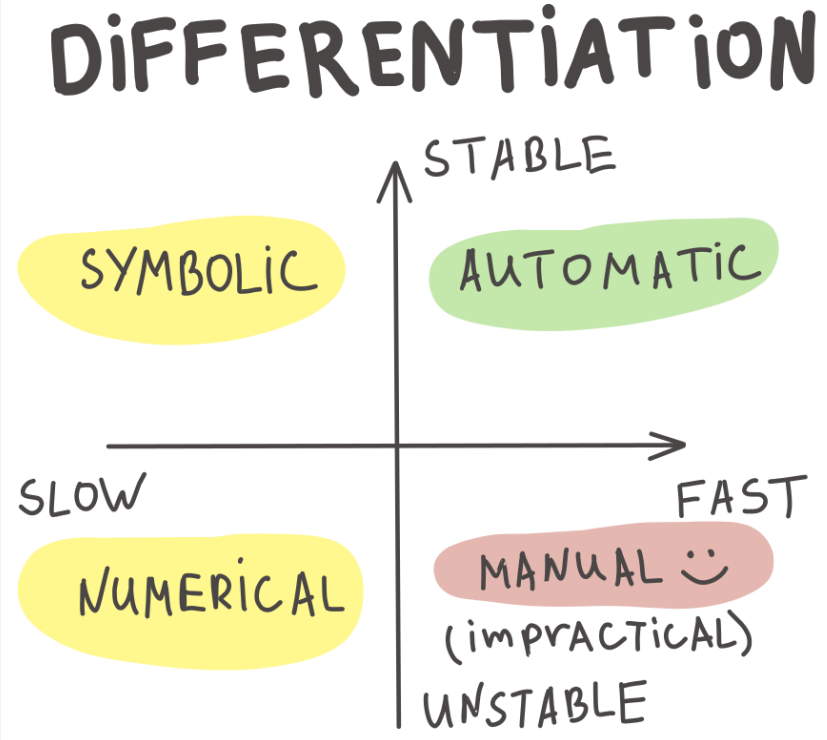
Key tool: Auto Differentiation

Autodiff = A tool that computes derivatives *exactly*, by tracking operations inside the neural network via the chain rule. The same reverse-mode machinery used for backprop.

How It Works:

- Breaks your function into simple steps (e.g., add, multiply, sin).
- Applies the *chain rule* automatically and efficiently.
- No need to write or approximate derivatives by hand!

Analogy: Like a cooking assistant that keeps track of how changing each ingredient affects the final dish.



Let's do a second example together (Schrödinger's Eqn.)

For the following PDE,

$$-iu_t(x,t) = 0.5u_{xx}(x,t) + |u(x,t)|^2 u(x,t) \quad x \in (-5,5), t \in (0,\pi/2]\,,$$

$$\begin{aligned} u(x,0) &= 2\text{sech}(x)\,, && \text{Initial Condition} \\ u(-5,t) &= u(5,t)\,, && \text{Periodic Dirichlet BC} \\ u_x(-5,t) &= u_x(5,t)\,, && \text{Periodic Neumann BC} \end{aligned}$$

Let's define a physics informed loss.

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{data}}(u, u_\theta) + \lambda_2 \mathcal{L}_{IC}(u_\theta) + \lambda_3 \mathcal{L}_{BC}(u_\theta) + \lambda_4 \mathcal{L}_{eq}(u_\theta)\,.$$

Data and Initial condition components

Data loss: Assume we have the solution u at the points (x_i, t_i), $i \in [1, .., n_{\text{data}}]$. Then, we can just use our classic supervised loss:

$$\mathcal{L}_{\text{data}}(u, u_\theta) = \frac{1}{n_{\text{data}}} \sum_{i=1}^{n_{\text{data}}} |u(x_i, t_i) - u_\theta(x_i, t_i)|^2 .$$

Initial condition loss: Given our initial condition, we have:

$$u(x, 0) = 2\text{sech}(x) , \text{ Initial Condition}$$

How to do the initial condition loss?

$$\mathcal{L}_{ic}(u_\theta) = \frac{1}{k_1} \sum_{i=1}^{k_1} |u_\theta(x_i, 0) - 2\text{sech}(x_i)|^2 .$$

Boundary condition terms

Recall, we had periodic boundary conditions:

$$u(-5,t) = u(5,t)\,, \qquad \text{Periodic Dirichlet BC}$$
$$u_x(-5,t) = u_x(5,t)\,, \qquad \text{Periodic Neumann BC}$$

We have *two* boundary conditions, so let's split our loss:

$$\mathcal{L}_{BC}(u_\theta) = \mathcal{L}_{\mathrm{D}}(u_\theta) + \mathcal{L}_N(u_\theta)\,.$$

Dirichlet Boundary Condition: Just enforce the boundaries are equal:

$$\mathcal{L}_D(u_\theta) = \frac{1}{k_2}\sum_{i=1}^{k_2} |u_\theta(-5,t_i) - u_\theta(5,t_i)|^2\,.$$

Neumann Boundary Condition: Use autodiff at the boundaries:

$$\mathcal{L}_N(u_\theta) = \frac{1}{k_2}\sum_{i=1}^{k_2} |\frac{\partial u_\theta}{\partial x}(-5,t_i) - \frac{\partial u_\theta}{\partial x}(5,t_i)|^2\,.$$

Last component: equation

How would you design the equation based loss?

Sample a series of points (in a smart way) (x_i, t_i) $i = 1, ..., k_3$. Then, using auto differentiation, we can compute:

$$\mathcal{L}_{eq}(u_\theta) = \frac{1}{k_3} \sum_{i=1}^{k_3} \left| -i \frac{\partial u_\theta}{\partial t}(x_i, t_i) - 0.5 \frac{\partial^2 u_\theta}{\partial x^2}(x_i, t_i) - |u_\theta(x_i, t_i)|^2 u_\theta(x_i, t_i) \right|^2$$

Putting it all together

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{data}}(u, u_\theta) + \lambda_2 \mathcal{L}_{IC}(u_\theta) + \lambda_3 \mathcal{L}_{BC}(u_\theta) + \lambda_4 \mathcal{L}_{eq}(u_\theta)\,.$$

Plugging in, we obtain:

$$\mathcal{L} = \underbrace{\frac{\lambda_1}{n_{\text{data}}} \sum_{i=1}^{n_{\text{data}}} |u(x_i, t_i) - u_\theta(x_i, t_i)|^2}_{\text{Data loss}} + \underbrace{\frac{\lambda_2}{k_1} \sum_{i=1}^{k_1} |u_\theta(x_i, 0) - 2\text{sech}(x_i)|^2}_{\text{Init. Cond. loss}}$$

$$+ \underbrace{\frac{\lambda_3}{k_2} \sum_{i=1}^{k_2} |u_\theta(-5, t_i) - u_\theta(5, t_i)|^2 + \frac{\lambda_3}{k_2} \sum_{i=1}^{k_2} |\frac{\partial u_\theta}{\partial x}(-5, t_i) - \frac{\partial u_\theta}{\partial x}(5, t_i)|^2}_{\text{Boundary cond. loss}}$$

$$+ \underbrace{\frac{\lambda_4}{k_3} \sum_{i=1}^{k_3} \left| -i\frac{\partial u_\theta}{\partial t}(x_i, t_i) - 0.5\frac{\partial^2 u_\theta}{\partial x^2}(x_i, t_i) - |u_\theta(x_i, t_i)|^2 u_\theta(x_i, t_i) \right|^2}_{\text{Equation loss}}$$

And the result is:

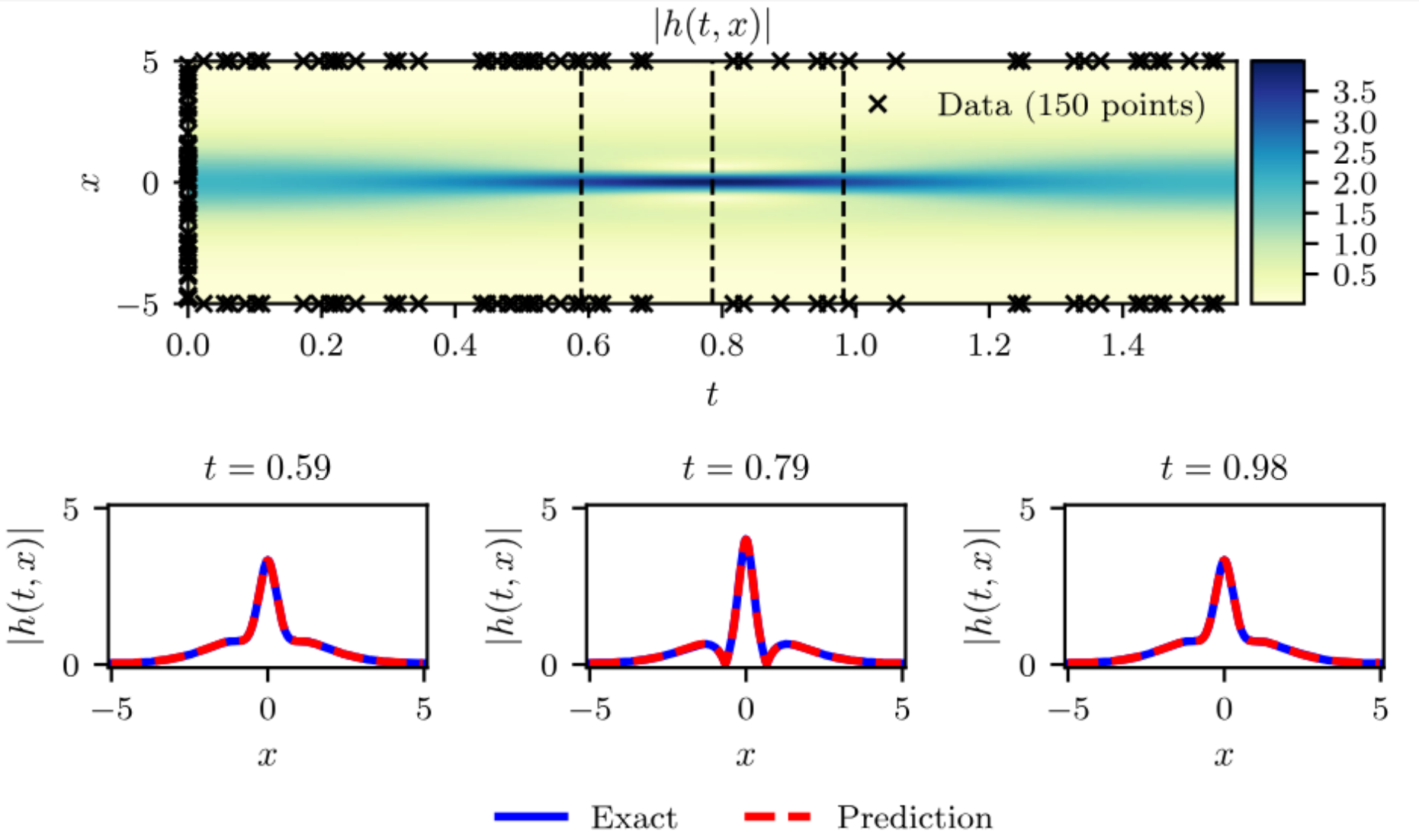


Figure: Top: u_θ for the Schrödinger equation. Bottom: Slices of solution at various timesteps.

Advantages of PINN loss

What advantages do you see?

- Flexibility of the framework - do we really need the data loss? Can we build semi-supervised learning methods out of this?
- Scalability - No need to retrain neural network when we want to query new points
- Inverse problems - If I am low on data, the PINN framework can be extremely valuable
- Soft regularization - do not need to design a network architecture to enforce my PDE
- Easily composable with other frameworks - Can you see how to combine PINNs and Neural ODE? PINNs and Neural Operators?

Disadvantages of PINN loss

What disadvantages do you see?

- Can be hard to train (More on this later). How to choose the weighting coefficients for λ is not obvious
- Computing the loss can be expensive if we need many points k_1, k_2, k_3 to evaluate at
- Struggles with challenging PDEs such as those with discontinuities or chaotic systems
- Soft regularization - Does not **guarantee** that my physical laws hold (of which they always should).
- **When NOT to use PINNs:** For well-posed forward PDEs with adequate mesh resolution, classical solvers outperform PINNs in both speed and accuracy. PINNs shine for *inverse problems*, *irregular geometries*, and *data assimilation*.

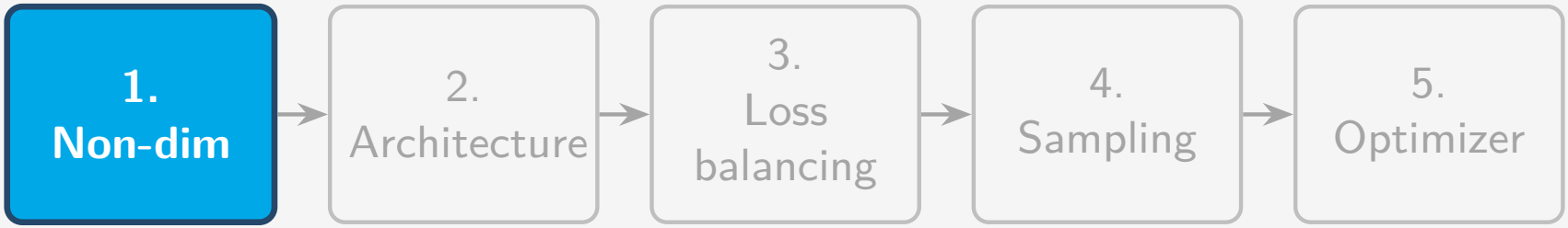
Let's talk about how to train PINNs

Many claim PINNs don't **really work**.

In practice, it turns out they are hard to train. We briefly are going to discuss:

- **Non-dimensionalization**: Modify the PDE such that the physical coefficients are scaled to be non-dimensionalized
- **Architecture design**: Initialization of parameters, weight factorization, activation functions, and mitigation of spectral bias.
- **Training pipeline**: How do we balance the different loss terms λ_i and curriculum learning.

Let's start with non-dimensionalization



Step 1 – Non-dimensionalize. Rescale x, t so coordinates are $\mathcal{O}(1)$, making the loss landscape more isotropic.

What is non-dimensionalization?

We rescale the PDE to normalize the units. Here's an example (Heat eqn.)

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \quad x \in [0, L], t \in [0, T],$$

with **zero boundary and initial conditions** and D as the diffusivity coefficient with units m^2/s.

- First, we define new variables called **dimensionless** variables

$$x' = \frac{x}{L}, \quad t' = \frac{t}{T}$$

- Compute derivatives via chain rule:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial t'}\frac{\partial t'}{\partial t} = \frac{\partial u}{\partial t'}\frac{1}{T},$$
$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial x'}\frac{\partial x'}{\partial x} = \frac{\partial u}{\partial x'}\frac{1}{L}, \quad \frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x'}\frac{\partial u}{\partial x}\frac{\partial x'}{\partial x} = \frac{\partial^2 u}{\partial x'^2}\frac{1}{L^2}$$

Non-dimensionalization continued:

Recall our PDE

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}\,, \quad x \in [0, L], t \in [0, T],$$

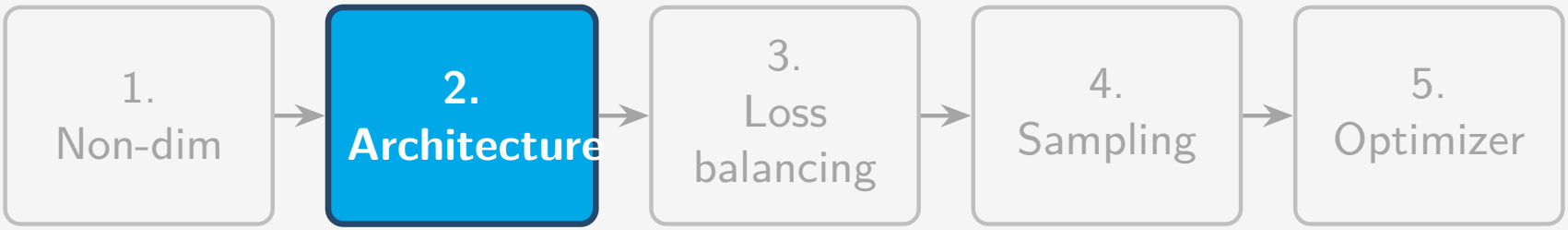
- Plug in derivatives: $\frac{\partial u}{\partial t'}\frac{1}{T} = D\frac{1}{L^2}\frac{\partial^2 u}{\partial x'^2}$
- Rescale $D^* = D \times \frac{T}{L^2}$ (dimensionless)

Yields

$$\frac{\partial u}{\partial t'} = D^*\frac{\partial^2 u}{\partial x'^2}\,, \quad t' \in [0, 1], x' \in [0, 1]\,.$$

Why did we do this?

Next: Feature tricks.

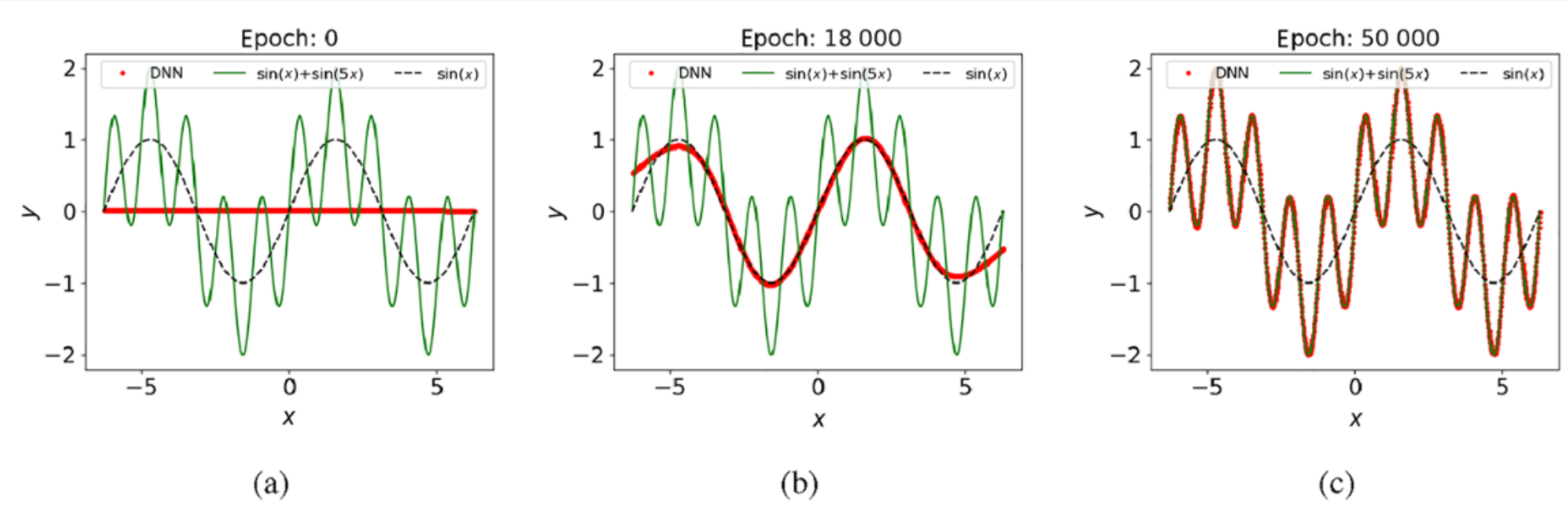


Step 2 – Architecture. Glorot init, smooth activations (tanh / GELU / SIREN), and Fourier feature encoding to mitigate spectral bias.

Architecture Design: Spectral bias

What is spectral bias

Answer: It is the model's tendency to learn the *low frequency modes* of the data before the high ones.



One way to prevent spectral bias

How to prevent spectral bias?

Answer: Pass the positional input data through an encoding. That is, let our input locations for $u_\theta(x)$ be given by the points $x_1, ..., x_n$. Then, we can pass our input data through the function $\gamma : \mathbb{R}^d \to \mathbb{R}^{2m}$

$$\gamma(x_i) = \begin{bmatrix} \cos(2\pi B x_i) \\ \sin(2\pi B x_i) \end{bmatrix}$$

where $B \in \mathbb{R}^{m \times d}$ is sampled from a Gaussian with mean 0 and variance σ^2 (user-tuned hyperparameter). (Paper link)

Why do you think this works?

Other architectural considerations

There are many other options to improve the performance of PINNs from an architecture standpoint.

- **Initialization:** Glorot initialization is recommended
- **Activation function:** Since ReLU is not C^1, we need smooth activations. In practice:
  - *tanh* – the default choice, smooth and bounded
  - *GELU / Swish* – work well for general problems
  - *SIREN* (sinusoidal, Paper link) – specifically helps with high-frequency targets
- **Weight factorization** – ensures the scales of the weights are properly normalized.

Next: Training Tricks.

Step 3 – Loss balancing. Reweight λ_i by gradient norms (moving average); weight the equation residual causally in time.

Training Considerations

How should we pick our weights λ_i?

Main approach (paper link): fix one loss term (typically the equation residual) at weight 1, then rescale the others so the gradients of each loss term are balanced:

$$\begin{aligned}
\hat{\lambda}_{\text{data}} &= \frac{\max_\theta \|\nabla_\theta \mathcal{L}_{eq}(\theta)\|}{\overline{\|\nabla_\theta \mathcal{L}_{\text{data}}(\theta)\|}} \\
\hat{\lambda}_{IC} &= \frac{\max_\theta \|\nabla_\theta \mathcal{L}_{eq}(\theta)\|}{\overline{\|\nabla_\theta \mathcal{L}_{IC}(\theta)\|}} \\
\hat{\lambda}_{BC} &= \frac{\max_\theta \|\nabla_\theta \mathcal{L}_{eq}(\theta)\|}{\overline{\|\nabla_\theta \mathcal{L}_{BC}(\theta)\|}} \\
\hat{\lambda}_{eq} &= 1
\end{aligned}$$

where max is taken over the parameters at the current iterate and $\overline{(\cdot)}$ denotes the mean. Note: the normalization $\sum \lambda_i = 1$ from earlier can be dropped here – what matters is the *relative* scaling.

Choosing λ_i

But, loss functions are noisy? Could this cause numerical stability?

Answer: Yes, so we use a moving average.

$$\lambda_{\text{new}} = \alpha\lambda_{\text{old}} + (1-\alpha)\hat{\lambda}_{\text{new}}$$

where $\alpha \in [0, 1]$ is a hyperparameter chosen by the user. Typically α is chosen close to 1 (e.g. 0.9 or 0.99) so λ changes slowly.

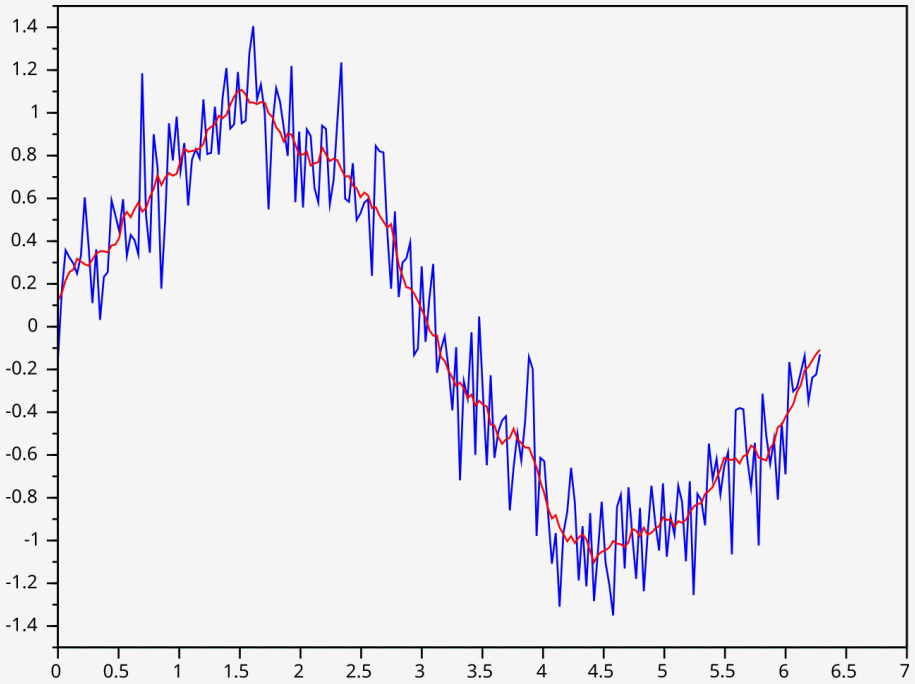


Figure: Red line: Moving average. Blue line: Loss function.

A second trick....

Main challenge: Temporal dependency

PINNs struggle when the PDE is asked to be solved over long time ranges. Think $t \in [0, 100]$ as a "long time". How to overcome this?

Capitalize on PDE structure by temporally splitting the domain.

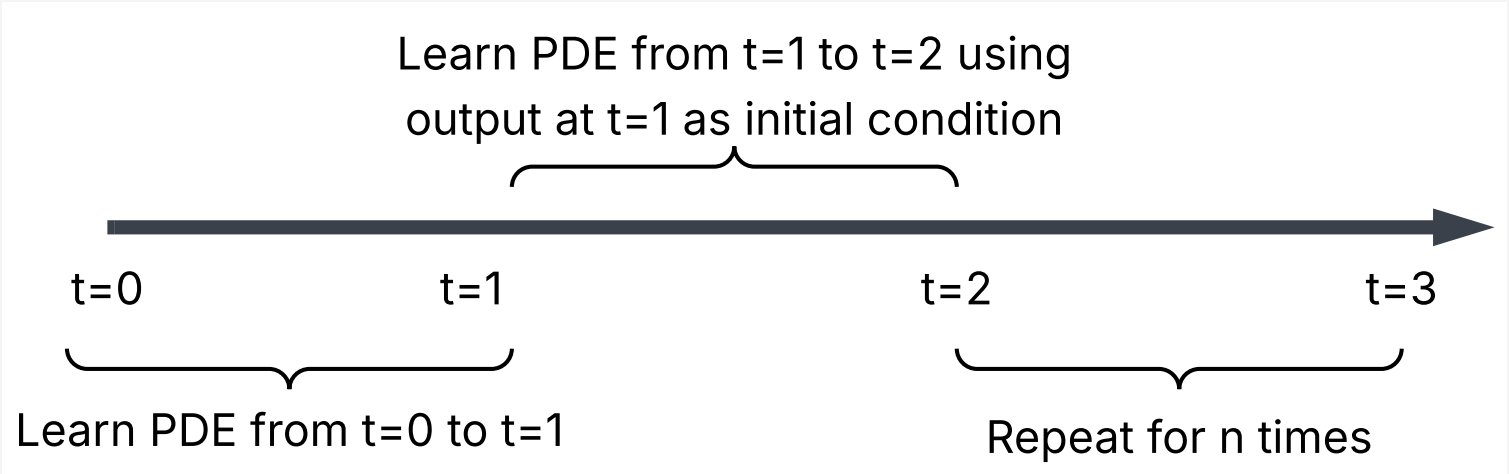


Figure: Curriculum learning approach to PDEs

Causal training

Problem with vanilla PINNs: the equation loss treats all collocation points equally, so the network can "cheat" by fitting late-time residuals before earlier ones are learned – violating the *causal* structure of the PDE.
Idea: (paper link) bin collocation points by time $t_1 < t_2 < \ldots < t_{N_t}$ and weight each bin's residual so a bin only matters *once the earlier ones are small*:

$$\mathcal{L}_{eq}(u_\theta) = \frac{1}{N_t} \sum_{i=1}^{N_t} w_i \, \mathcal{L}_{eq}(t_i; u_\theta) \,, \quad w_i = \exp\left(-\epsilon \sum_{j=1}^{i-1} \mathcal{L}_{eq}(t_j; u_\theta) \right) .$$

- If early-time residuals are large $\Rightarrow$ $w_i \approx 0$ for later times $\Rightarrow$ network focuses on early times first.
- As early-time residuals shrink $\Rightarrow$ $w_i \to 1$ for later times $\Rightarrow$ later times are gradually "unlocked".
- **Hyperparameter:** causality strength ϵ (typically 10–100).
- Continuous version of the curriculum-learning idea – no manual time-domain splitting needed.

Choosing collocation points: why it matters

The equation loss is a Monte Carlo estimate:

$$\mathcal{L}_{eq}(u_\theta) = \frac{1}{k_3}\sum_{i=1}^{k_3} |\mathcal{R}(u_\theta)(x_i, t_i)|^2 \approx \int_\Omega |\mathcal{R}(u_\theta)(x,t)|^2 \, dx\, dt\,.$$

- **Uniform sampling** is the default but wastes points in regions where the residual is already small.
- For PDEs with sharp features, residuals concentrate in small regions $\Rightarrow$ uniform sampling under-resolves them.

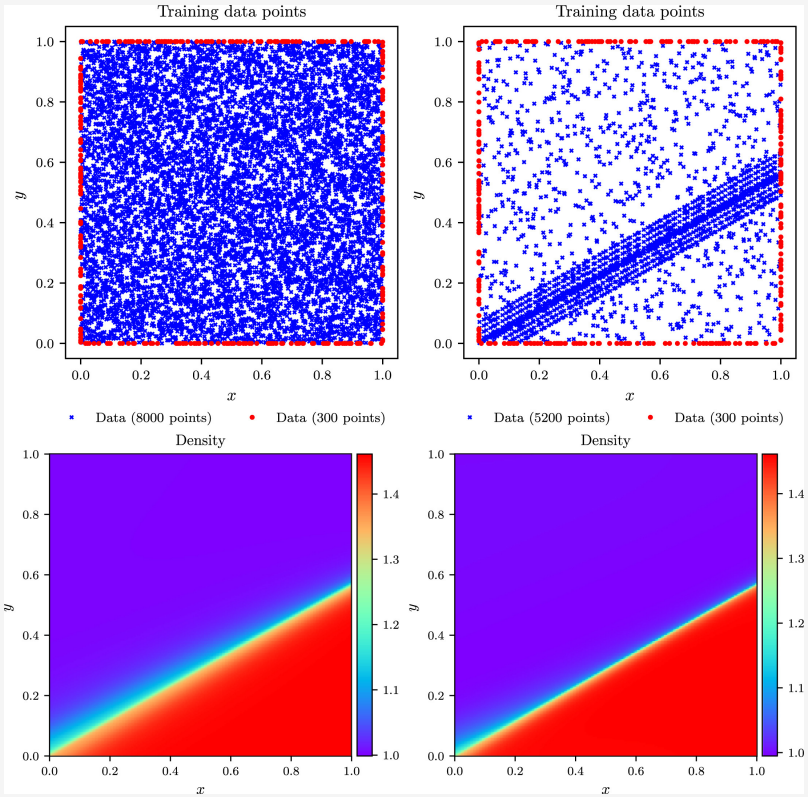
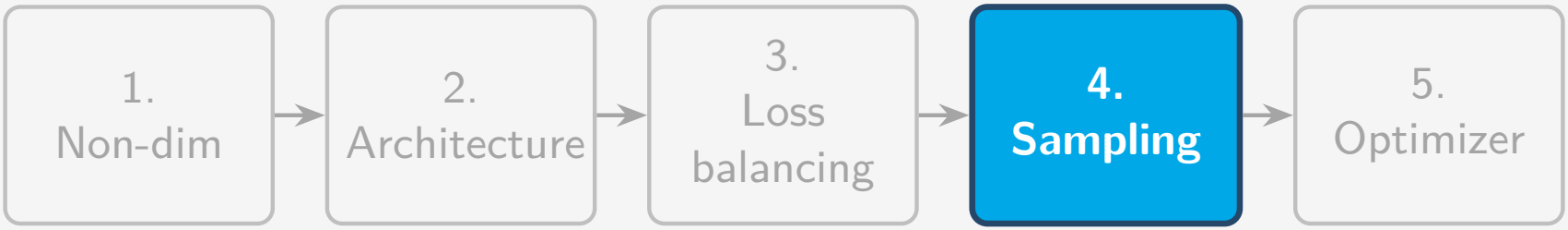


Figure: Uniform (left) vs. targeted (right) sampling on a shock-wave problem.

Sampling tricks...



Step 4 – Adaptive sampling. Replace uniform collocation with RAR – add new points where the residual is currently largest.

Residual-based Adaptive Refinement (RAR)

Idea : (paper link)periodically add collocation points *where the residual is currently largest*.

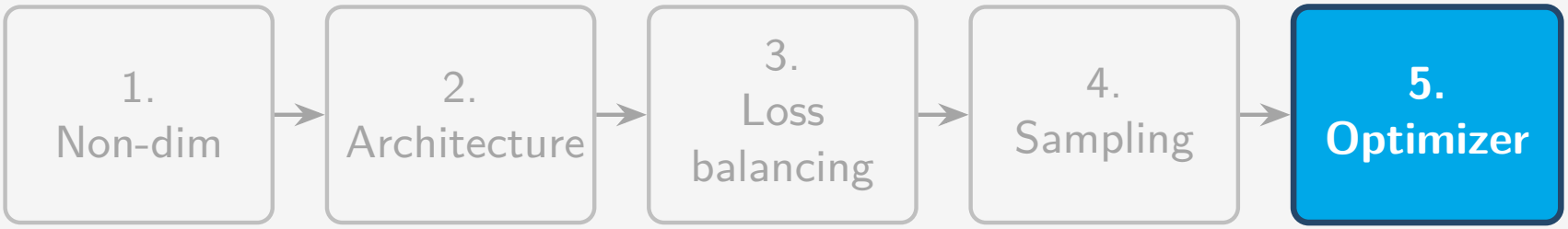
Algorithm:

1. Train PINN for N steps on current collocation set $\mathcal{S}$.
2. Sample a large candidate pool $\mathcal{C}$ ($\sim$10k points) uniformly from Ω.
3. Evaluate $|\mathcal{R}(u_\theta)(x_i, t_i)|$ for every $(x_i, t_i) \in \mathcal{C}$.
4. Add the top-m highest-residual points to $\mathcal{S}$.
5. Repeat.

Why it works: the network "self-discovers" which regions are hard, without any prior physical knowledge of where shocks or boundary layers live.

Variants: RAR-D (distribution-based, samples proportional to residual rather than top-k), RAD (resample entire set each iteration), causal sampling (bias toward early times).

Optimizer tricks...



Step 5 – Optimizer. Adam for exploration, then switch to L-BFGS to exploit curvature near the minimum.

Optimizer tuning: A quick overview

Adam – first-order, adaptive

- Momentum + adaptive step size:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\,\nabla_\theta \mathcal{L}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\,(\nabla_\theta \mathcal{L})^2$$
$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

- Cheap per step, robust to noisy gradients
- Good at *exploring* non-convex landscapes

L-BFGS – approximate second-order

- Quasi-Newton: approximates the inverse Hessian $\tilde{H}_t^{-1}$ from recent gradients
- Update uses curvature:

$$\theta_{t+1} = \theta_t - \alpha_t \, \tilde{H}_t^{-1} \, \nabla_\theta \mathcal{L}$$

- "Limited-memory": stores only the last $\sim$10 updates, not the full $\tilde{H}^{-1}$
- Line search picks α_t each step
- Expensive per step; needs full-batch gradients (deterministic)

Adam warmstart + BFGS fine tune

Stage 1 – Adam (explore)

- $\sim$10k–50k steps, learning rate $\alpha \sim 10^{-3}$
- Goal: get θ into a good basin (don't chase precision)

Stage 2 – L-BFGS (refine)

- *Reuse the same* θ; swap optimizer, keep the loss
- $\sim$500–2000 iterations, strong-Wolfe line search
- Goal: drive residual down 1–3 orders of magnitude

Why this order? L-BFGS from a random init is useless. The Hessian approximation needs a smooth region to be meaningful. Adam warm starts. L-BFGS is also expensive. (See other second order optimizers Soap and Muon)

Summary on how to train PINNs

- **Non-dimensionalize** the PDE
- **Architecture:** Glorot initialization, smooth activation (tanh/GELU/SIREN), weight factorization
- **Fourier encoding** for positional inputs to mitigate spectral bias
- **Loss balancing:** reweigh λ_i via gradient norms + moving average
- **Time dependencies:** curriculum learning or causal weighting
- **Optimizer:** Adam (exploration) $\rightarrow$ L-BFGS (refinement)
- **Sampling:** adaptive (RAR) instead of uniform when residuals concentrate

Do you see why people say these are hard to train?

On top of typical hyperparameters, training PINNs introduces (at minimum) 4+ extra hyperparameters: Fourier-feature variance σ, moving-average rate α, causality strength ϵ, and RAR refinement frequency/count.

Ablation Example: Viscous Burgers' Equation

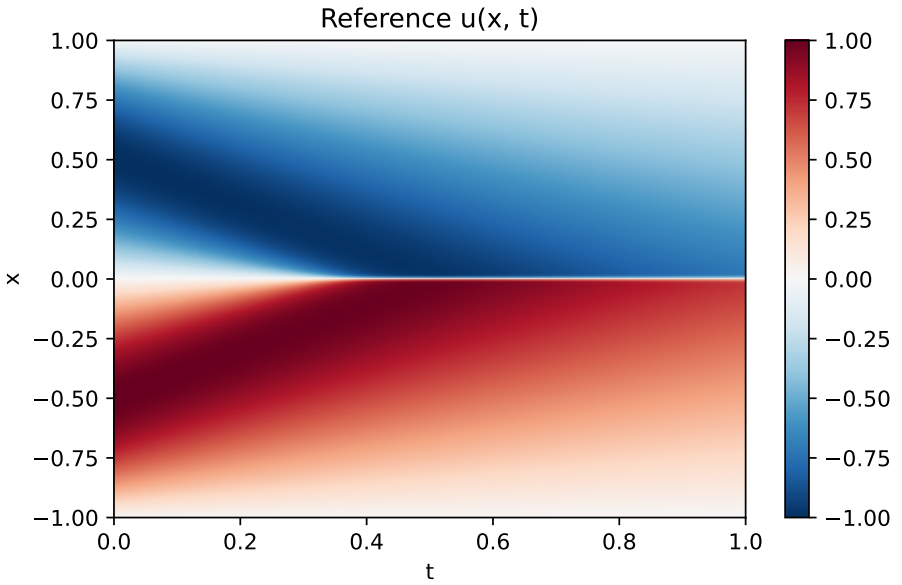
The classical PINN benchmark:

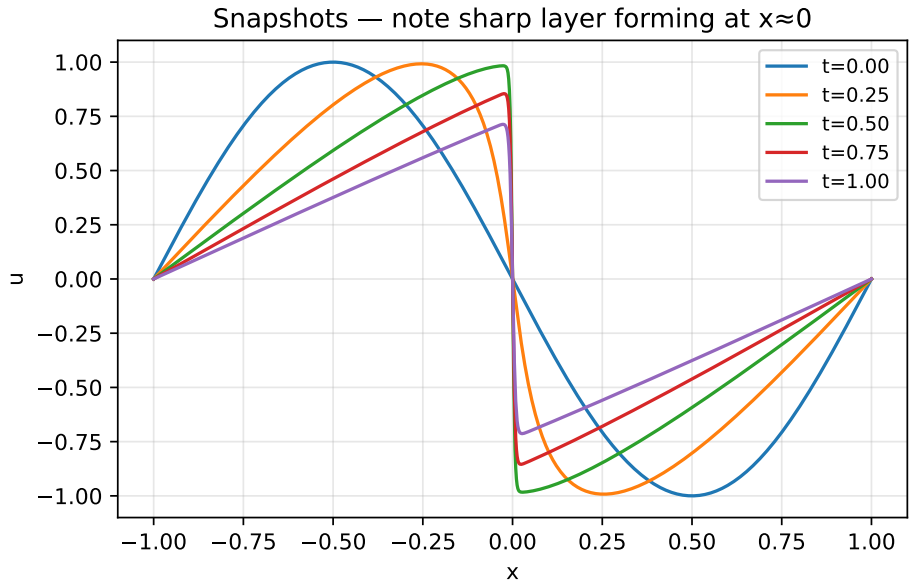
$$
\begin{aligned}
u_t + u\,u_x &= \nu\,u_{xx}\,, && x \in (-1,1),\ t \in (0,1),\ \nu = 0.01/\pi \\
u(x,0) &= -\sin(\pi x)\,, && \text{Initial Condition} \\
u(\pm 1,t) &= 0\,, && \text{Dirichlet Boundary Conditions}
\end{aligned}
$$

Why this is hard:

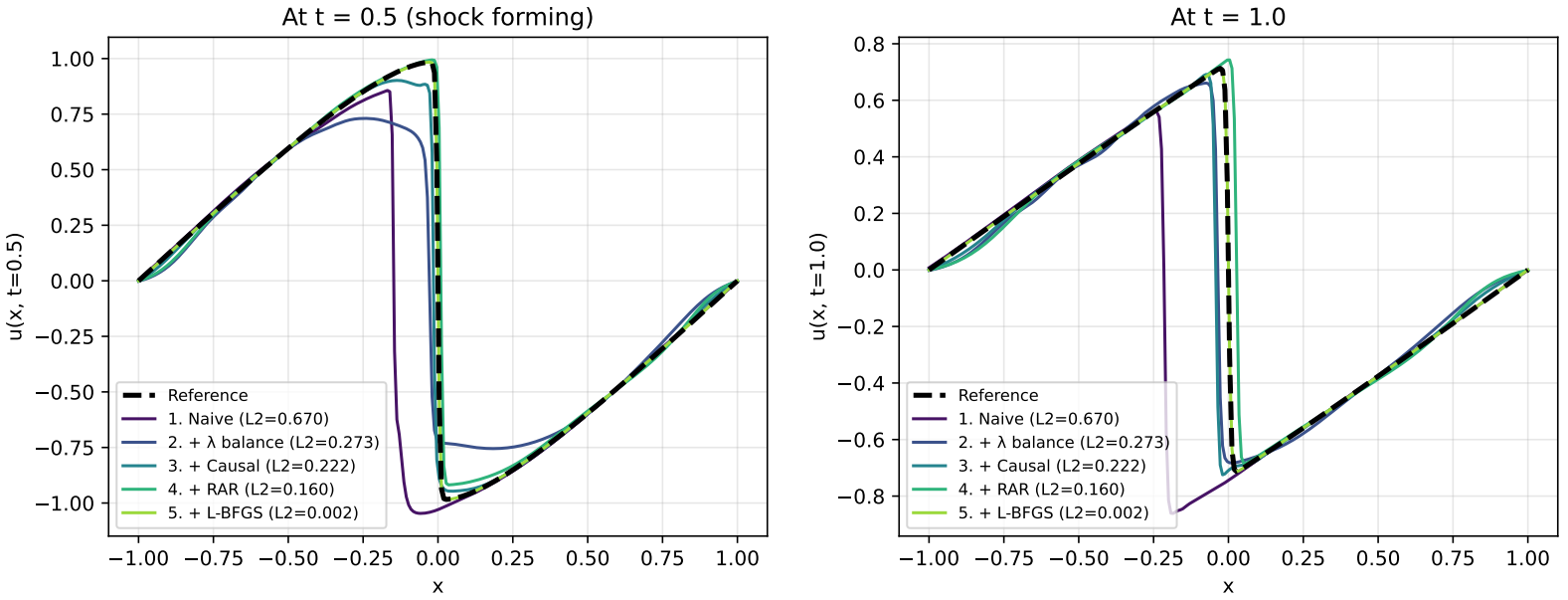
- Nonlinear advection ($u\,u_x$) steepens the wave over time
- A sharp internal layer forms at $x \approx 0$ near $t \approx 0.5$ (gradient exceeds 100)
- Diffusion is tiny relative to advection ($\nu \approx 0.003$)

Example of true solution



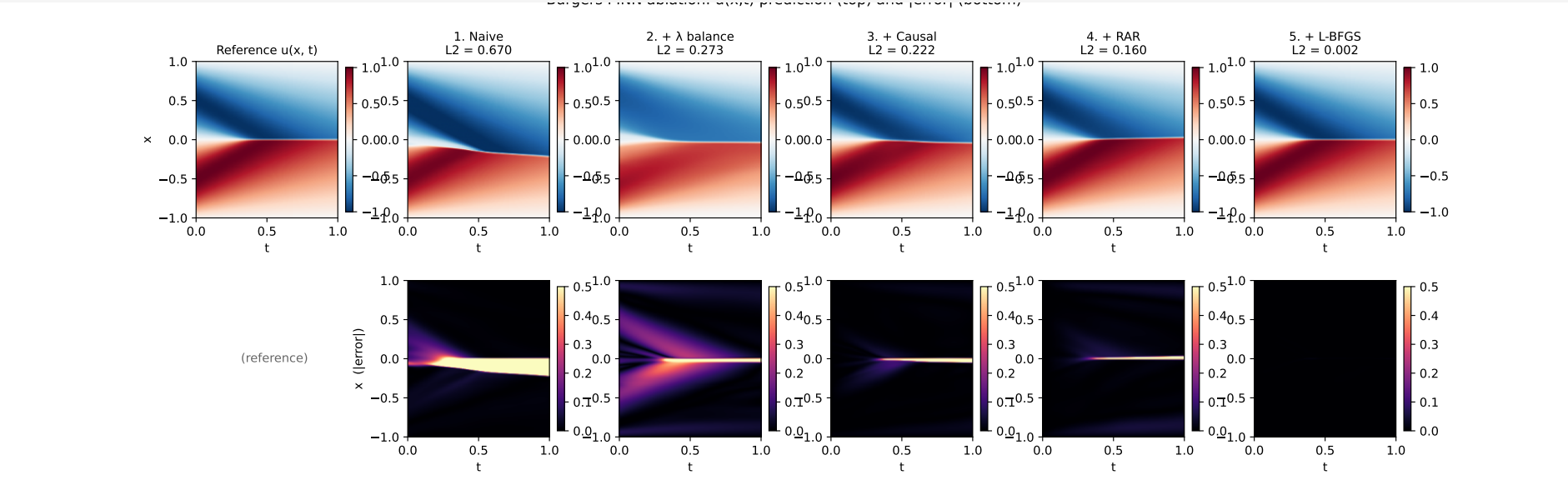


Comparison of various training methods

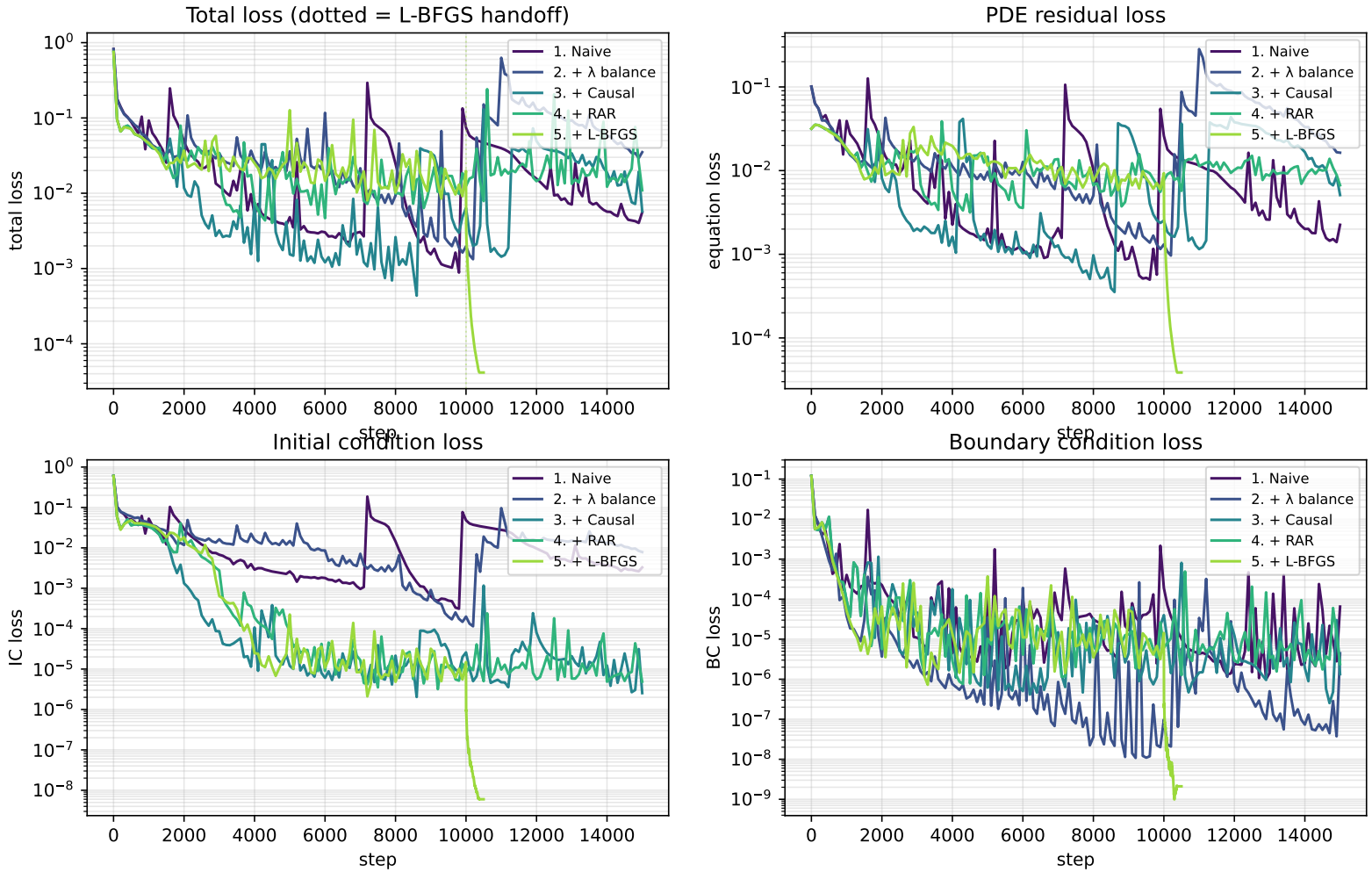


Configuration	**rel. L^2**	**time (s)**	**collocation pts**
1. Naive	0.6696	78.0	2000
2. + λ balance	0.2734	80.3	2000
3. + Causal	0.2219	574.0	2000
4. + RAR	0.1598	560.2	3400
5. + L-BFGS	**0.0016**	**844.7**	**2800**

Comparison of various training methods



Example of Loss functions



Hard vs. Soft constraints

Soft constraints (what we've been doing): Add IC/BC violations to the loss. Doesn't *guarantee* physics holds.
Hard constraints: Bake the IC/BC into the architecture so they hold by construction. Example for an IC $u(x, 0) = g(x)$:

$$u_\theta(x, t) = g(x) + t \cdot \mathrm{NN}_\theta(x, t)\,.$$

At $t = 0$: $u_\theta(x, 0) = g(x)$ exactly, regardless of network weights.

- Pros: Physics is enforced *exactly*, no λ_{IC} to tune.
- Cons: Can be hard to design for complex geometries/BCs. Restricts expressivity of the network.

Applications of PINNs: Inverse problems

Inverse problems: I have a low amount of data, say k sensor measurements and I want to reconstruct the PDE *and/or its unknown coefficients* from these measurements.
Key trick: Treat the unknown PDE coefficients (e.g., diffusivity D, Reynolds number Re) as *learnable parameters* and optimize them jointly with the network weights θ:

$$\min_{\theta, D} \quad \mathcal{L}_{\text{data}}(u, u_\theta) + \lambda \mathcal{L}_{eq}(u_\theta; D)\,.$$

Can you think of any real-world examples?

- Earthquake epicenter detection
- MRI/CT Scans are common inverse problems
- Acoustic reconstruction
- Image generation - Creating 3D images from 2D measurements (NeRFs technically solve an inverse problem)

Thermal design in GPUs (Navier Stokes)

$$\frac{\partial \theta}{\partial t} + (\boldsymbol{v} \cdot \nabla)\theta = \frac{1}{\mathrm{Pe}} \nabla^2 \theta$$
$$\frac{\partial \boldsymbol{v}}{\partial t} + (\boldsymbol{v} \cdot \nabla)\boldsymbol{v} = -\nabla p + \frac{1}{\mathrm{Re}} \nabla^2 \boldsymbol{v} + \mathrm{Ri}\theta\,, \quad \nabla \cdot \boldsymbol{v} = 0\,.$$

θ is temperature, $\boldsymbol{v} = (v_1, v_2)^T$ is the velocity field, p is the pressure field.

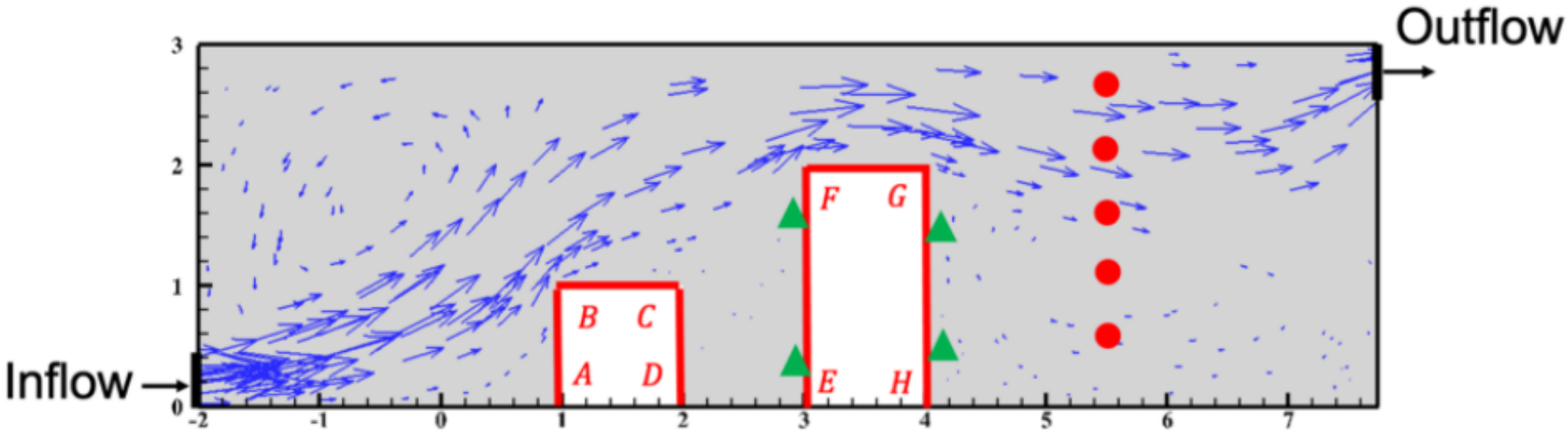
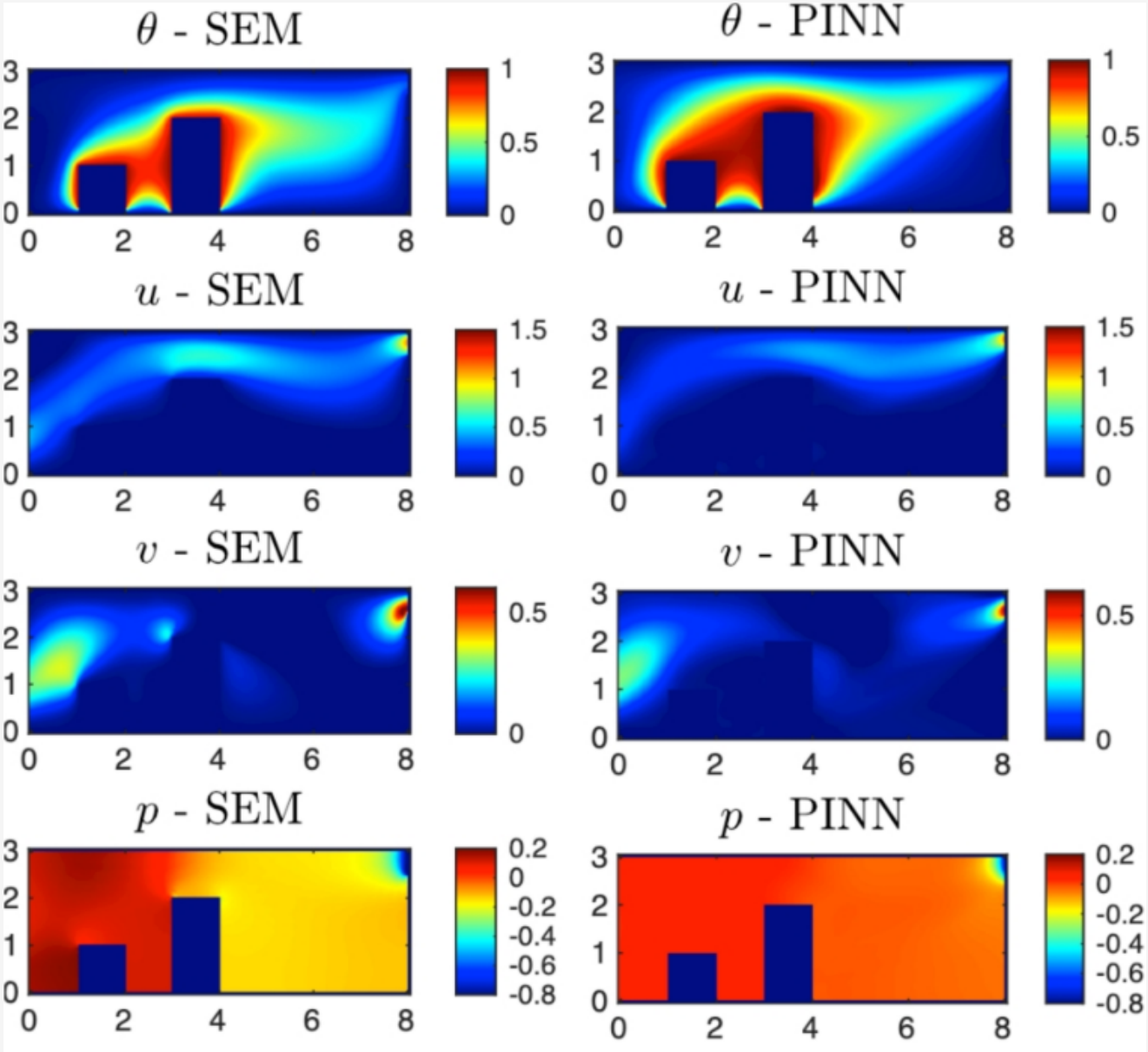


Figure: Blue represents the airflow $\boldsymbol{v}$. There are two obstacles outlined in red that effect the flow. We have measurements for temperature at the green arrows and velocity at the red circles and we want to know the flows everywhere.

Question: From our sensor measurements, reconstruct the PDE with a PINN



MRI Inference: Arterial flow reconstruction

Consider the problem of blood flowing through a junction:

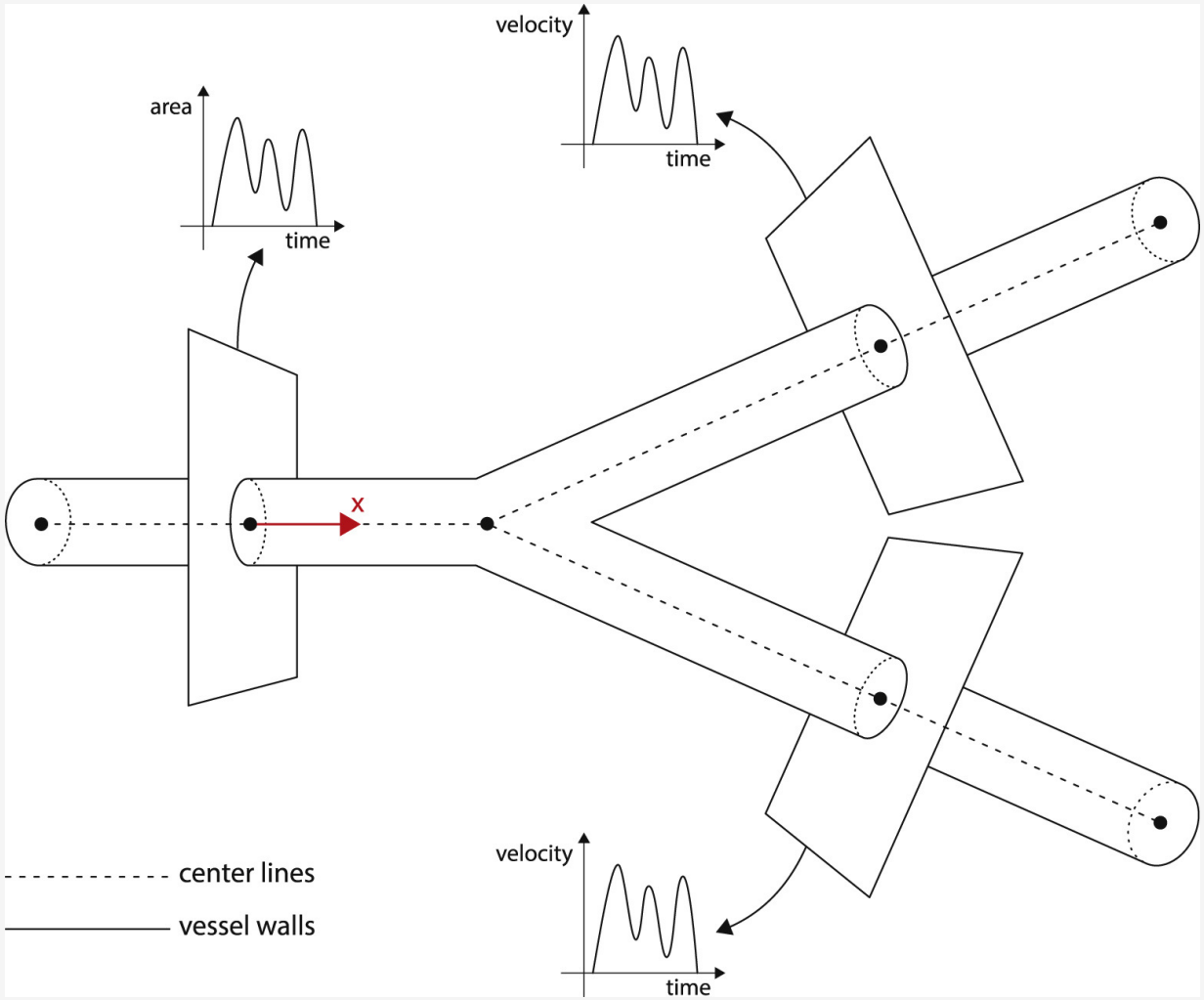


Figure: Blood flows from the left and is split over an artery with two sensor measurements on each side. Goal: Predict the velocity and pressure.

MRI Inference: Data + Sensor placement

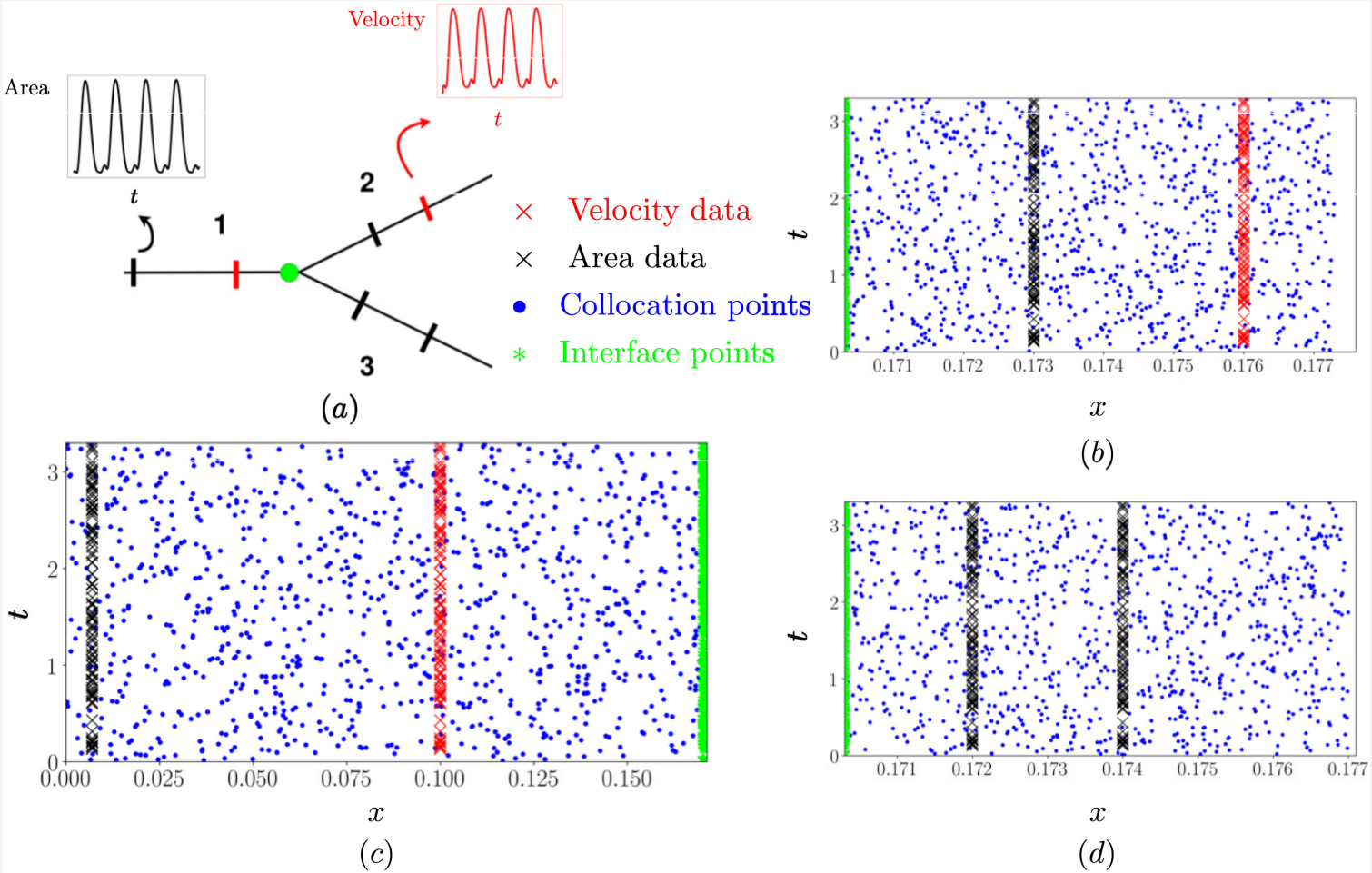


Figure: (a) Schematic of artery. (b) Data for position 2. (c) Data for position 1. (d) Data for position 3.

MRI Inference: Pressure and velocity results

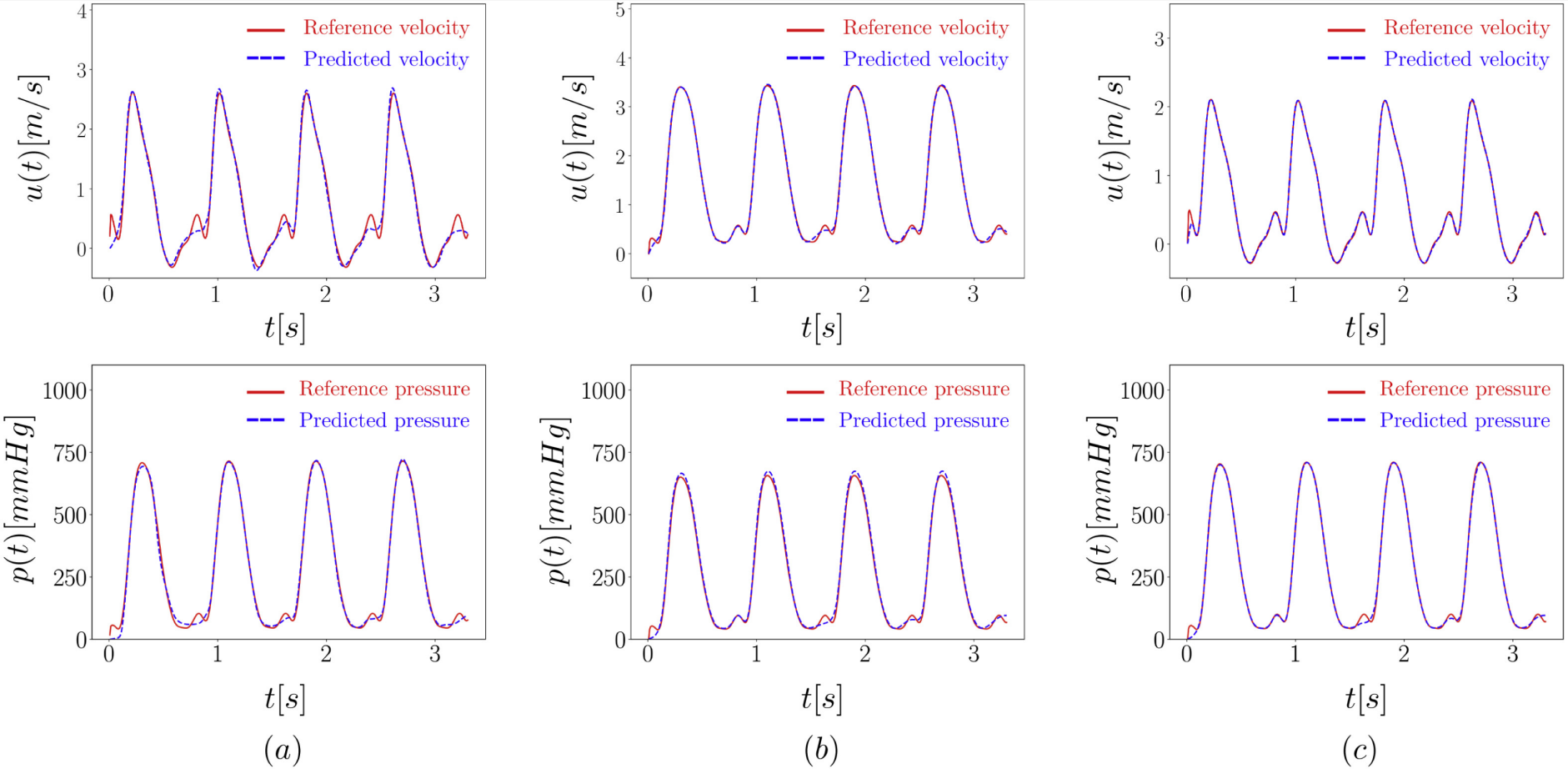


Figure: (a) The pressure and velocity at point 1. (b) The pressure and velocity at point 2. (c) The pressure and velocity at point 3.

Future directions of PINNs:

- Training still is not easy - better methodologies, architectures, etc.
- **Applications** - still many opportunities here
- Combine PINNs with other methodologies such as neural operators, Neural ODEs, transformers etc.
- Soft regularization still yields poor performance - hard constraints?