

MIDAS Summer Academy
July, 2025

Physics-Informed Neural Networks (PINNs)

Yiluan Song

Schmidt AI in Science Postdoctoral Fellow
Michigan Institute of Data & AI in Society
Institute for Global Change Biology
University of Michigan
songyl@umich.edu



Why do we need
PINNs?

Why do we need PINNs?



Why do we need PINNs?

What are the Navier Stokes equations?
A set of equations that relate the **velocity**, **pressure**,
temperature, and **density** of a flowing fluid.

$$\rho \frac{d\bar{u}}{dt} = -\nabla p + \mu \nabla^2 \bar{u} + \rho \bar{F}$$

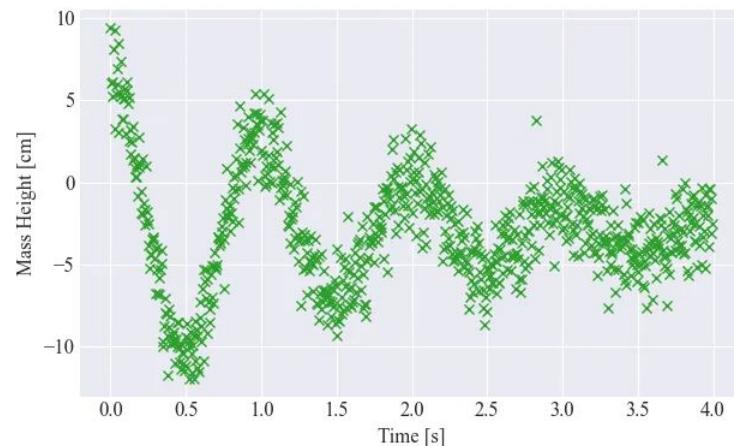
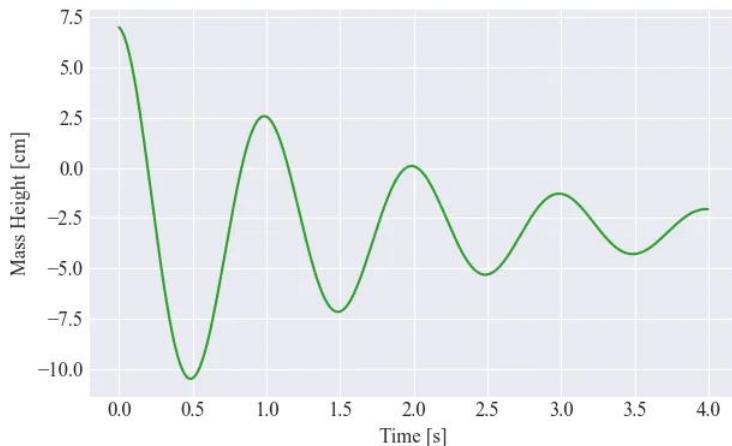
Diagram illustrating the Navier-Stokes equation components:

- velocity
- density
- viscosity
- pressure gradient
- external force

- When water flows in a game, it often follows Navier-Stokes equations – a partial differential equation (**PDE**) system describing fluid dynamics
- These PDEs are too complex to solve **analytically** in real time.
- The game engine uses **numerical** methods to approximate how water behaves frame-by-frame.
 - Solve PDEs at grids
 - Track water parcels

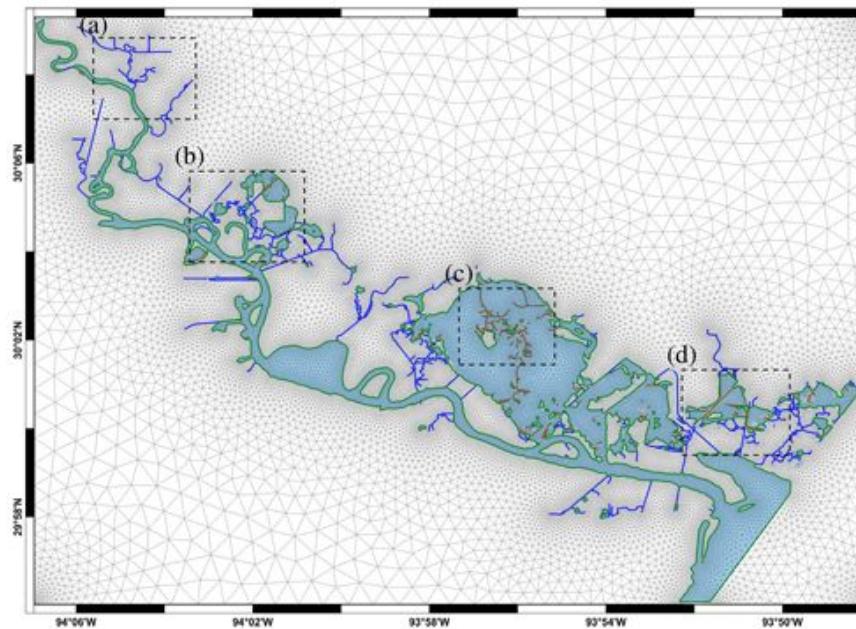
Challenges of numerical methods

- Cannot seamlessly incorporate noisy observed data



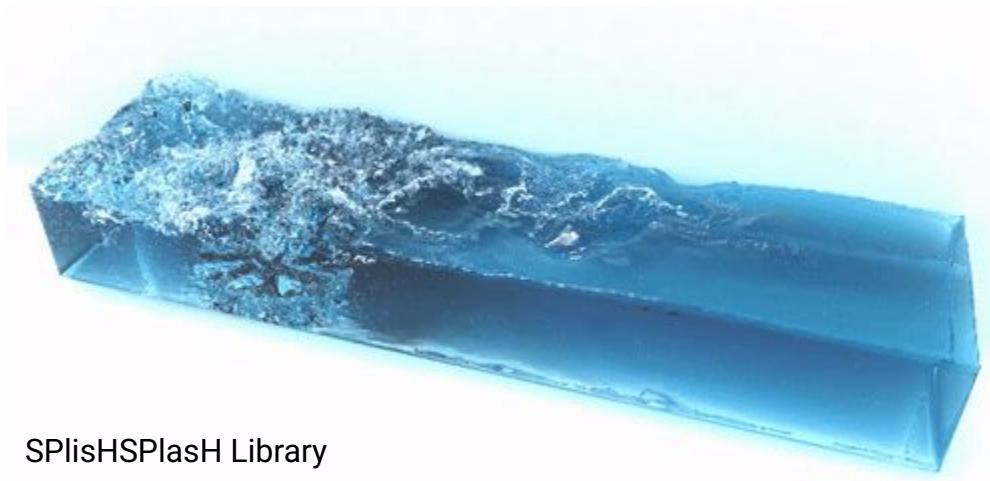
Challenges of numerical methods

- Mesh generation remains complex
 - When we need to solve PDEs at grids



Challenges of numerical methods

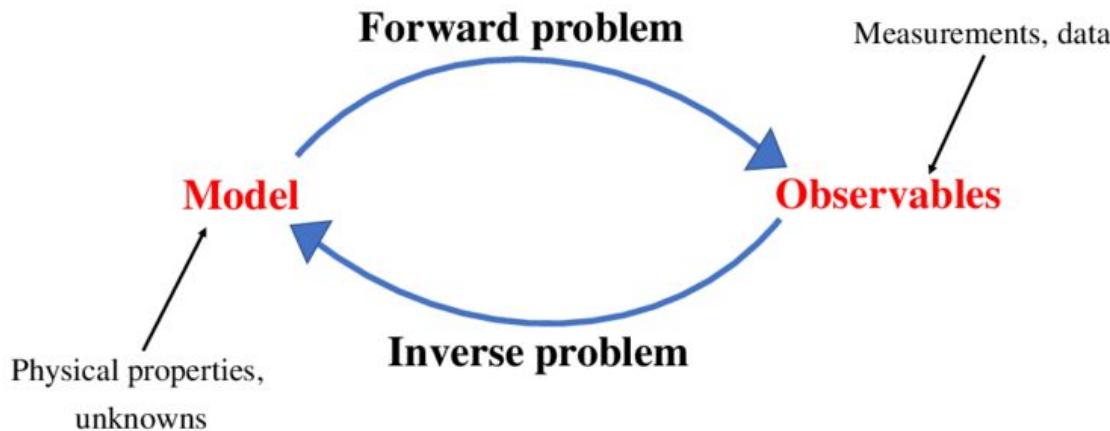
- High-dimensional problems may have impractical runtimes or memory requirements
 - e.g., simulating a 3D object over time



SPlisHSPlasH Library

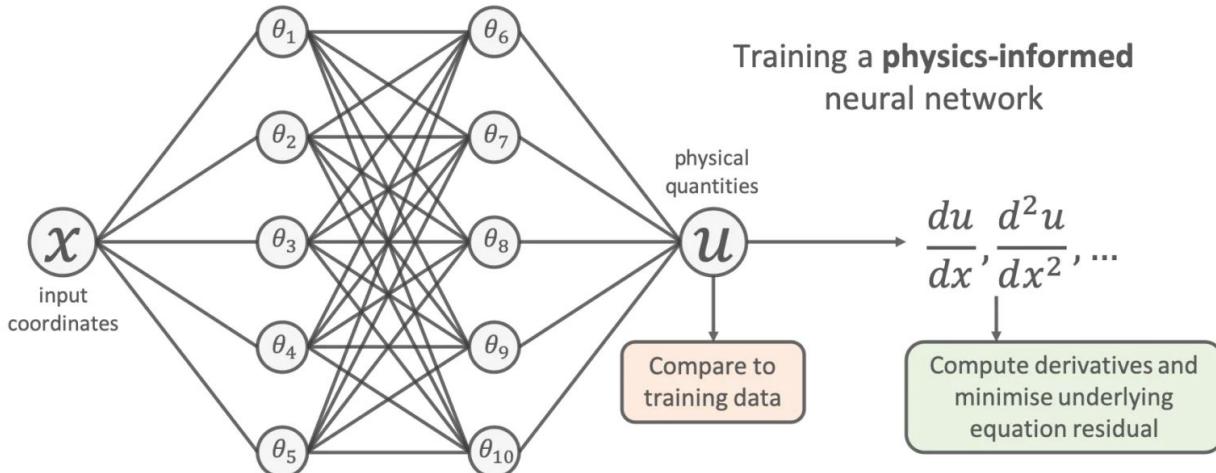
Challenges of numerical methods

- Solving inverse problems is often prohibitively expensive
 - **Forward problem:** You know the system and the input, and you want to compute the output.
 - **Inverse problem:** You observe the output, and try to infer unknown inputs or system parameters that produced it.



What are PINNs?

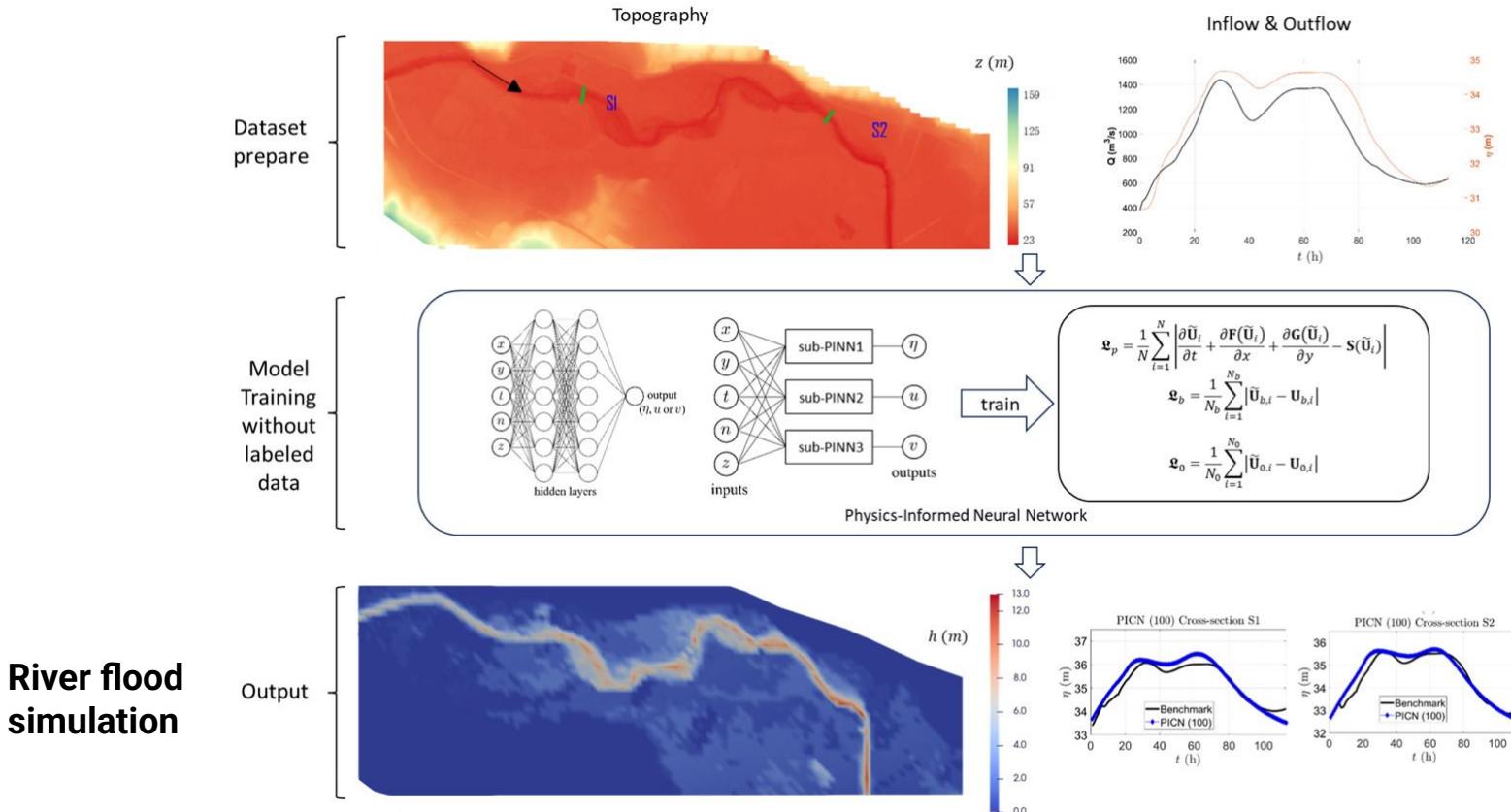
- A Physics-Informed Neural Network (**PINN**) is a type of **deep neural network (NN)** that solves or learns from PDEs by **embedding physical laws** directly into the training process.
- Expected result: A perfectly trained neural network that fit the data while respecting physical laws



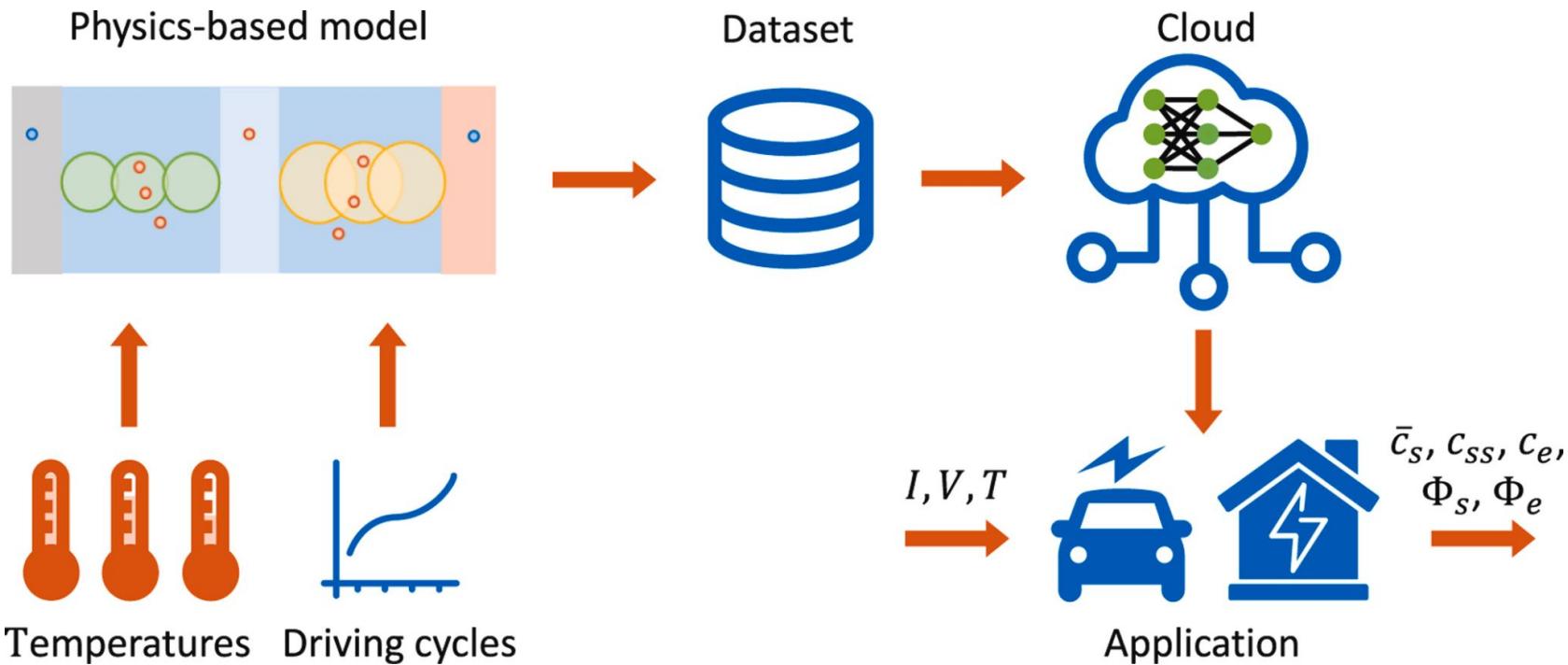
Benefits of PINNs

- Can work with sparse or noisy data by relying on known physics
 - NN can be regularized to avoid fitting noise
- Mesh-free: no need for mesh/grid generation
- Better handling of high-dimensional data
- No need to repeatedly solve the forward problem
 - Can solve both forward and inverse problems using the same framework

Applications of PINNs: Hydrology



Applications of PINNs: Electrochemistry



State estimation in lithium-ion batteries

Applications of PINNs: Robotics

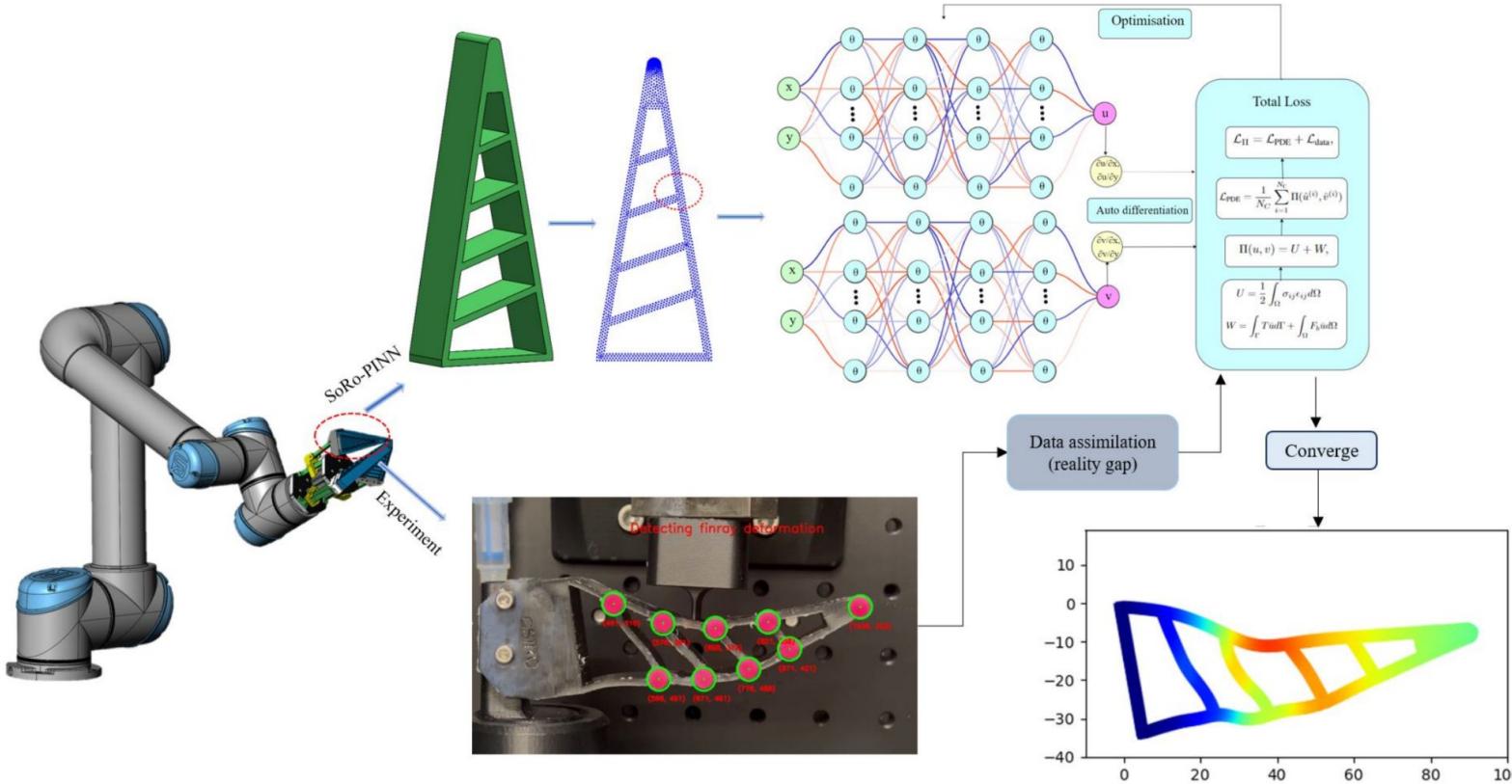
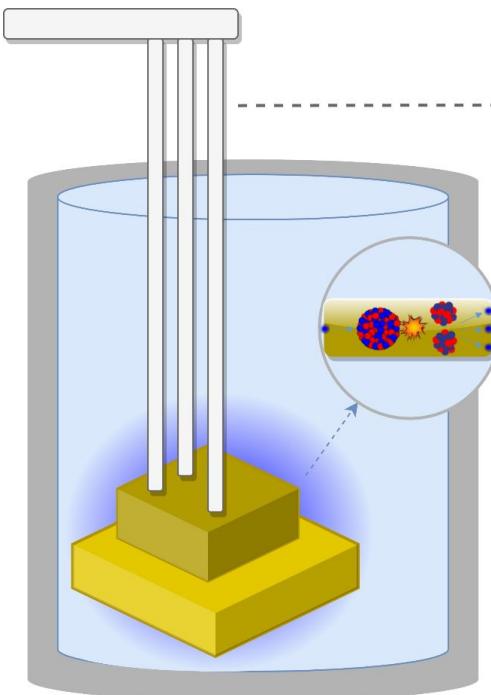
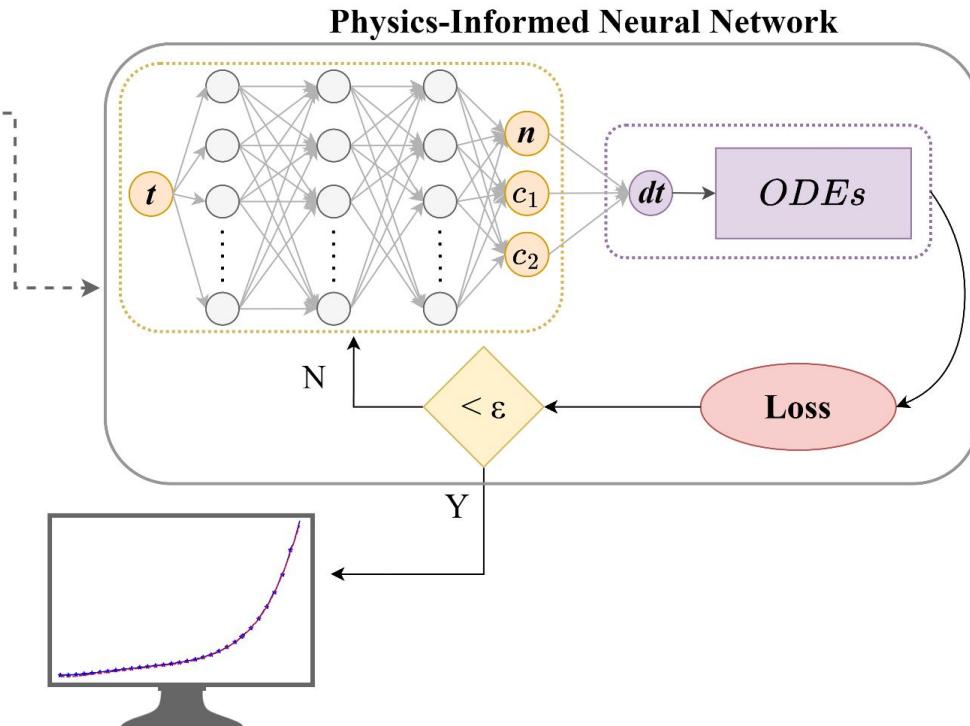


Fig. 1: The framework of PINN-Ray to model the deformation of soft robotic finger with sim-to-real treatment.

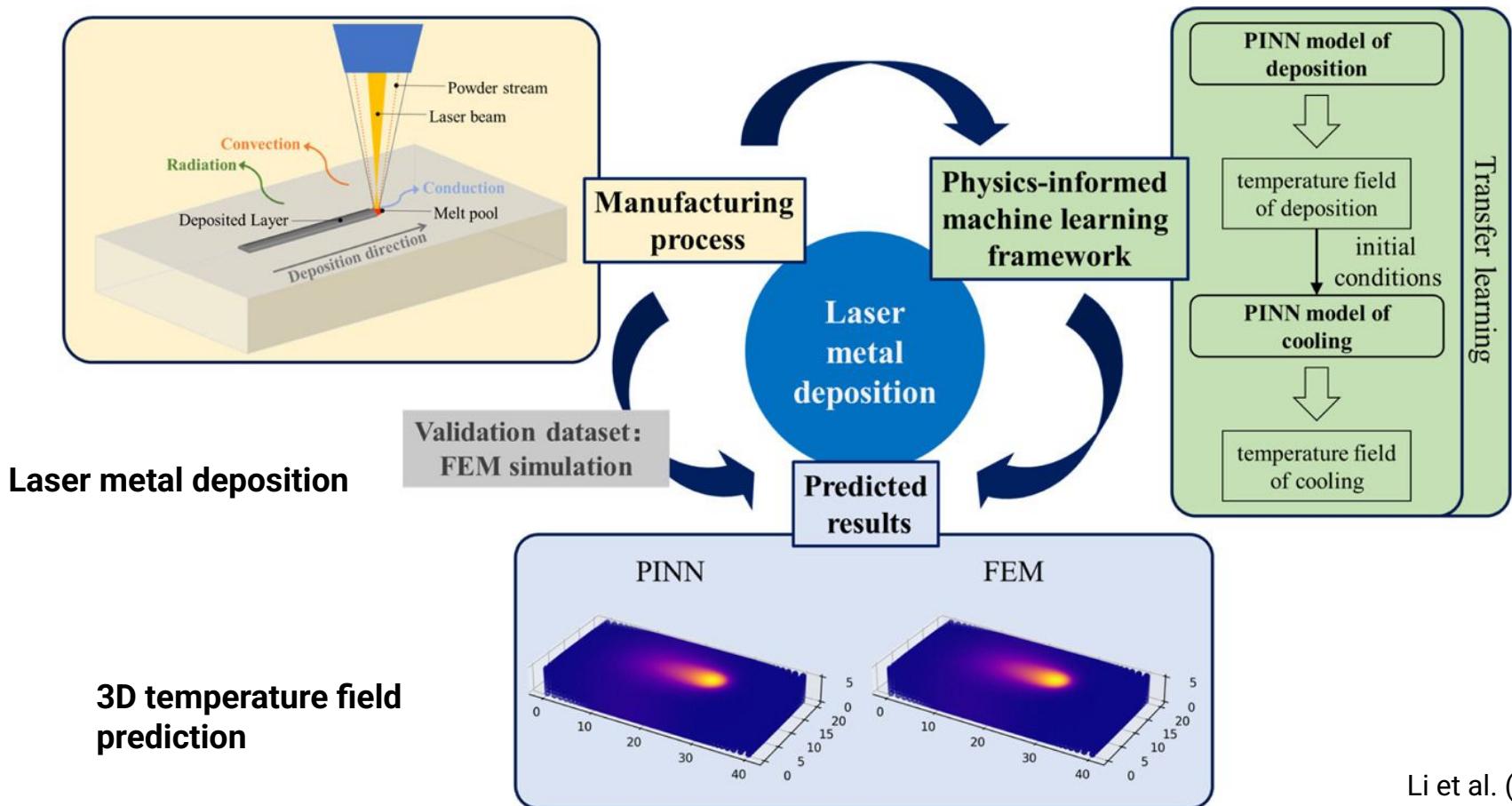
Applications of PINNs: Nuclear engineering



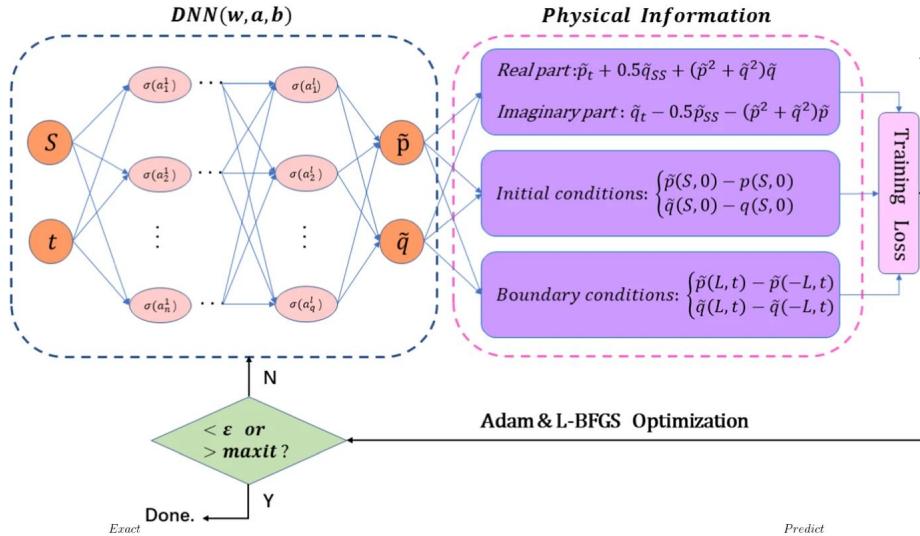
Nuclear reactor monitoring



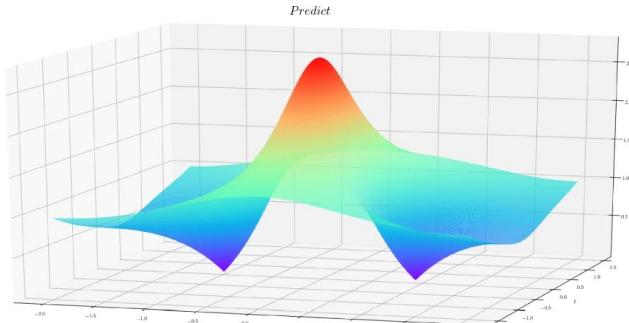
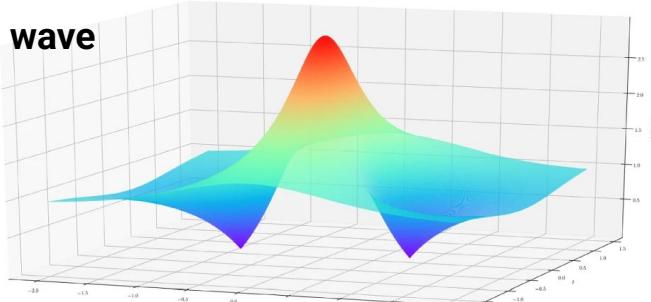
Applications of PINNs: Manufacturing



Applications of PINNs: Finance



An option pricing model



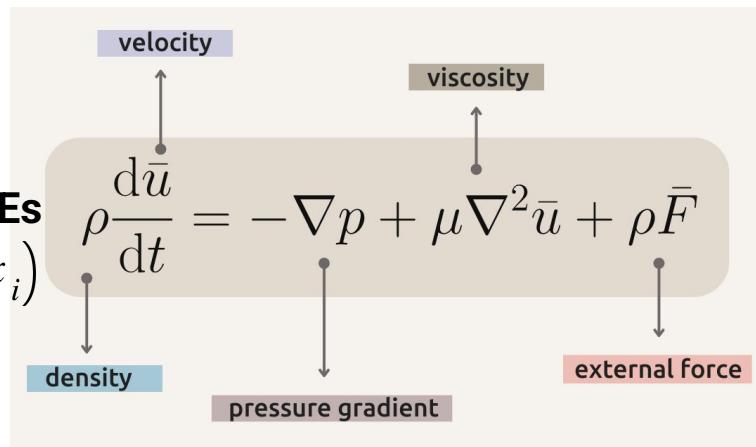
Bai et al. (2022)

How do we train PINNs?

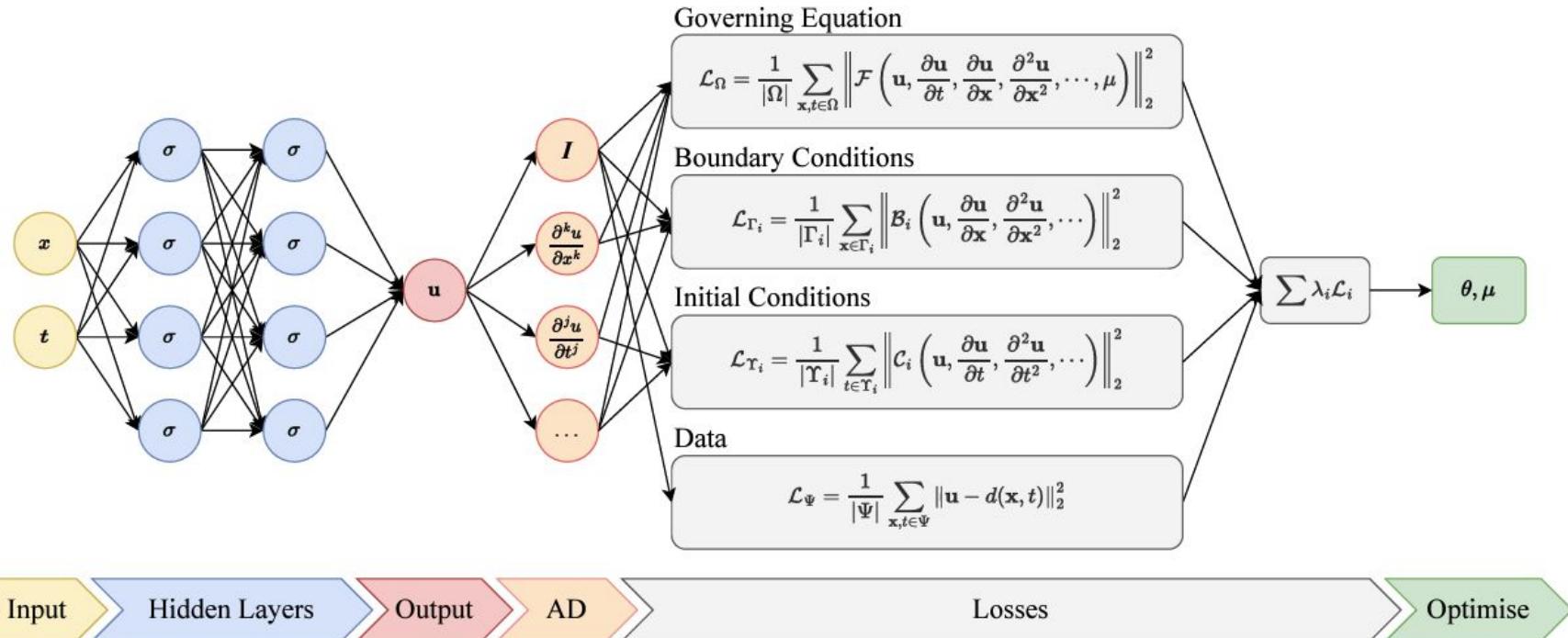
General form of PDE

$$\frac{\partial u}{\partial t} = \mathcal{N}[u; \theta]$$

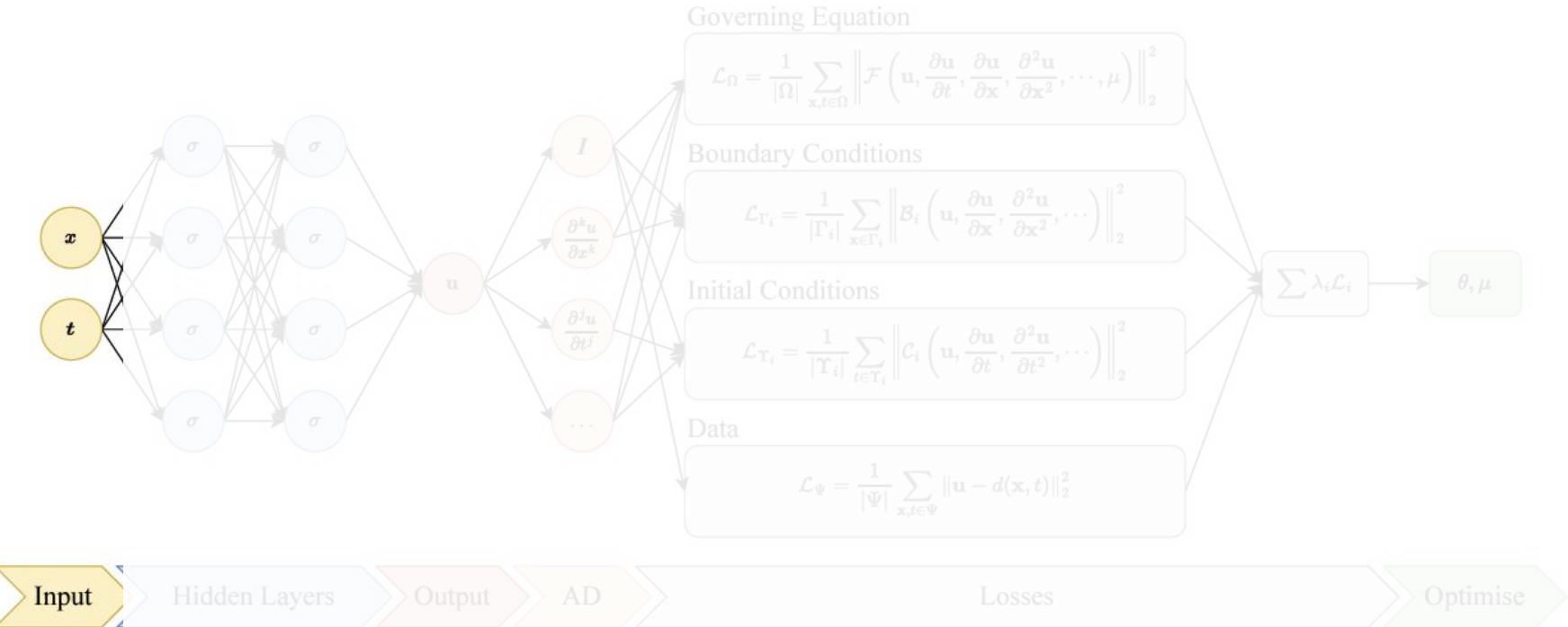
- $u = u(t, x)$ denotes the latent solution (t - time, x - space)
 - A **latent solution** is a solution to a PDE that is **implicitly** represented by a neural network, rather than explicitly computed at discrete points.
- \mathcal{N} is an operator **parameterized** by θ
- Given θ , what is $u(t, x)$?
 - Forward problem, inference, or **solving the PDEs**
- Find θ that best describes observations $u(t_i, x_i)$
 - Inverse problem, learning, or **system identification**



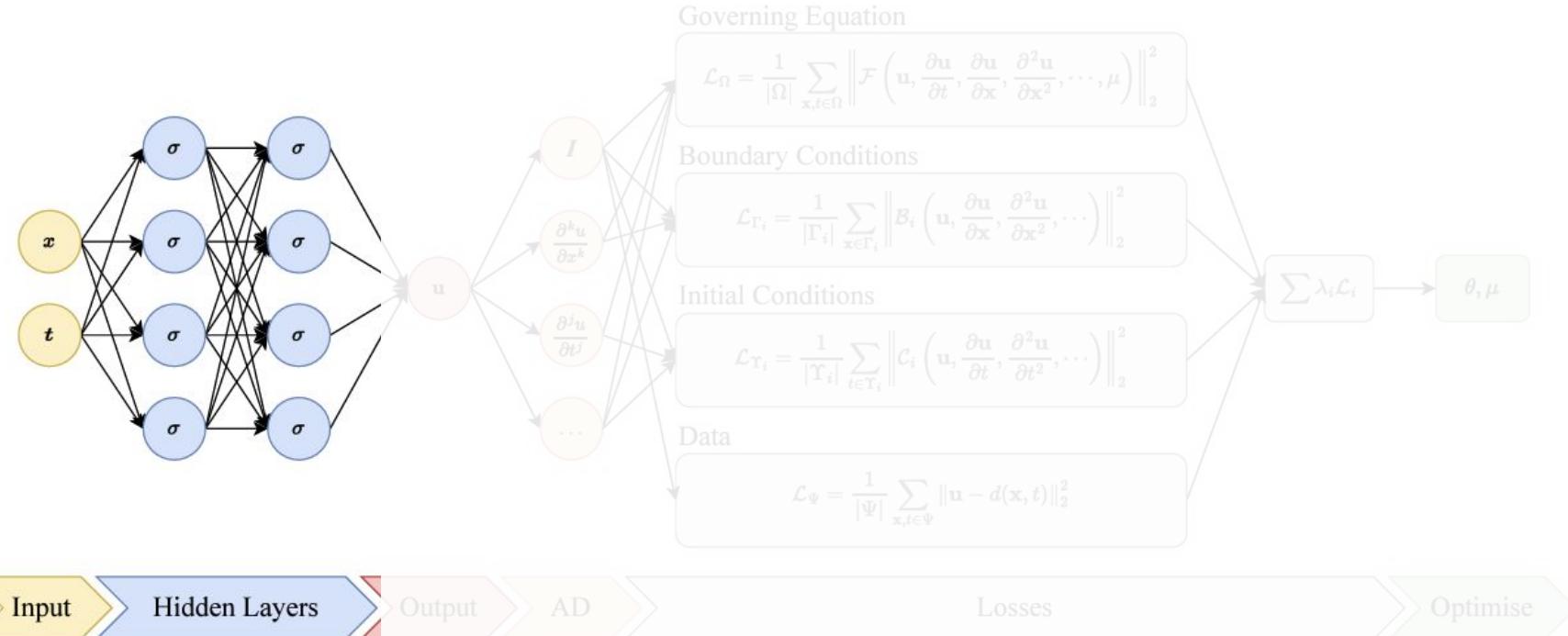
Workflow



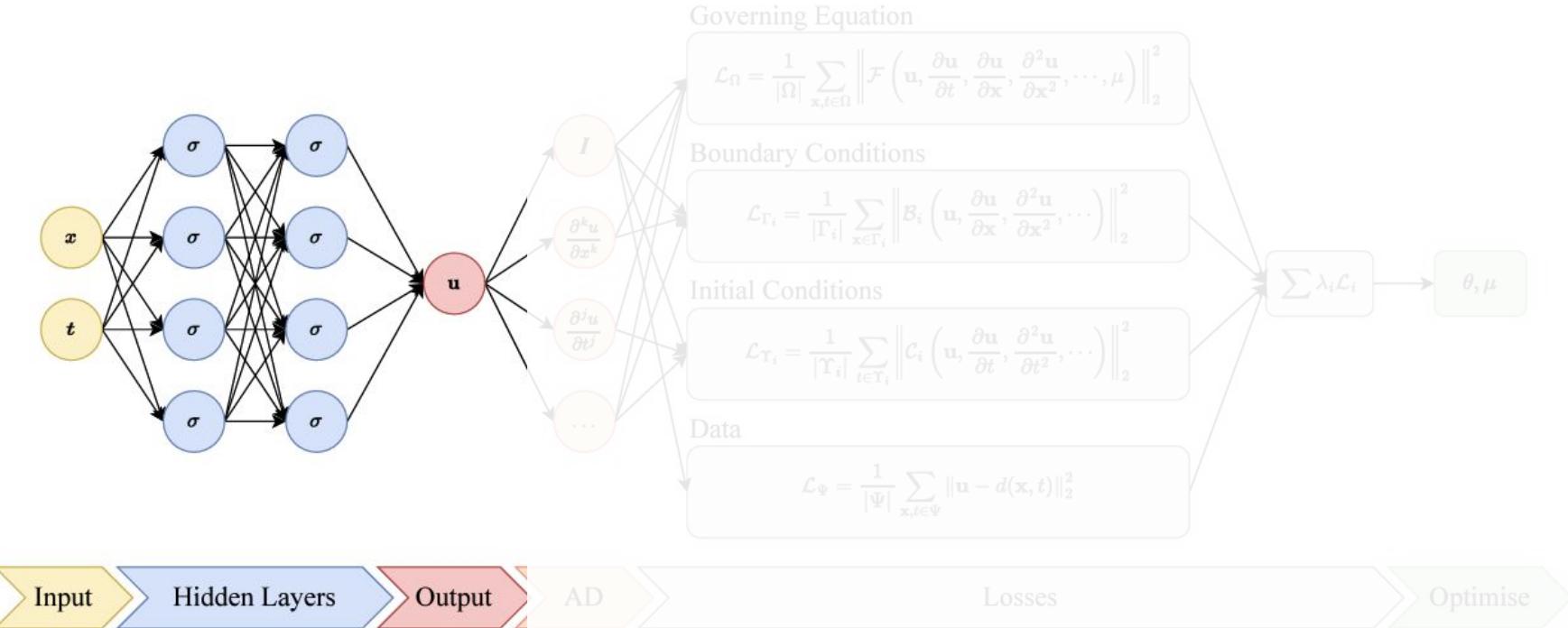
Workflow



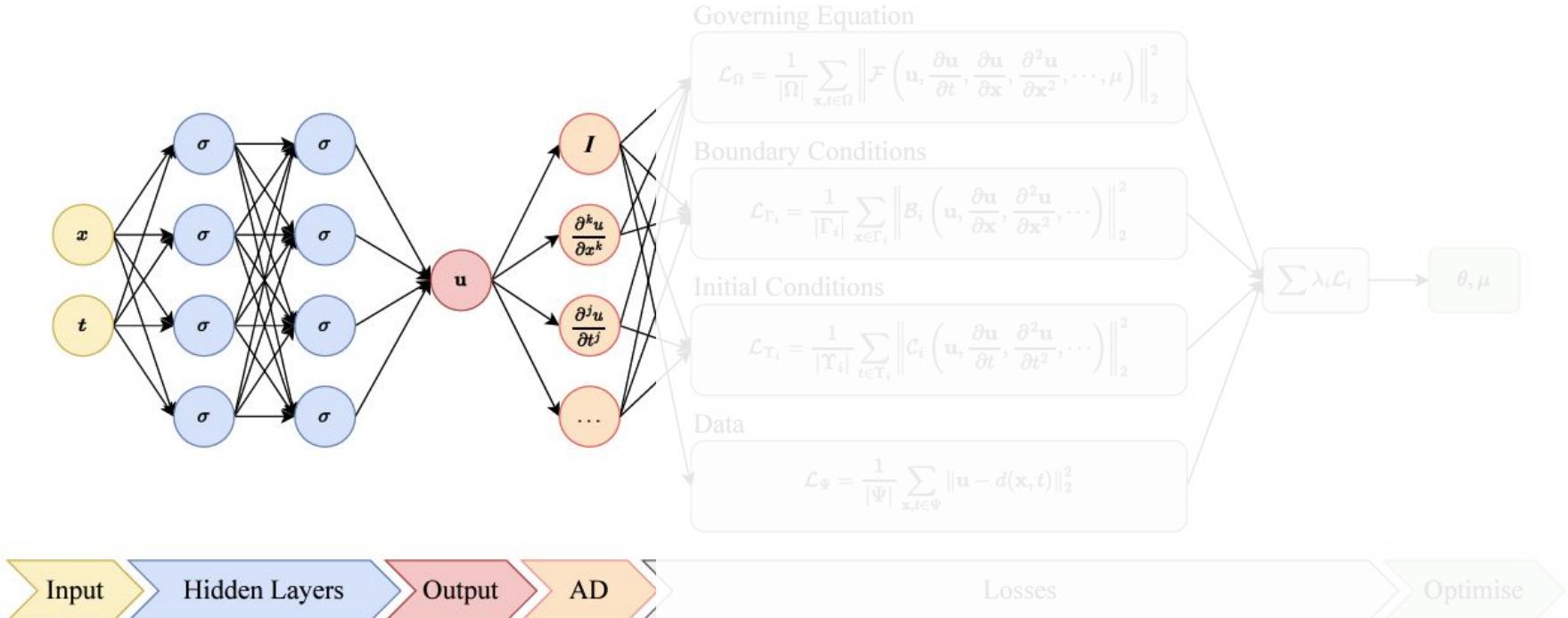
Workflow



Workflow



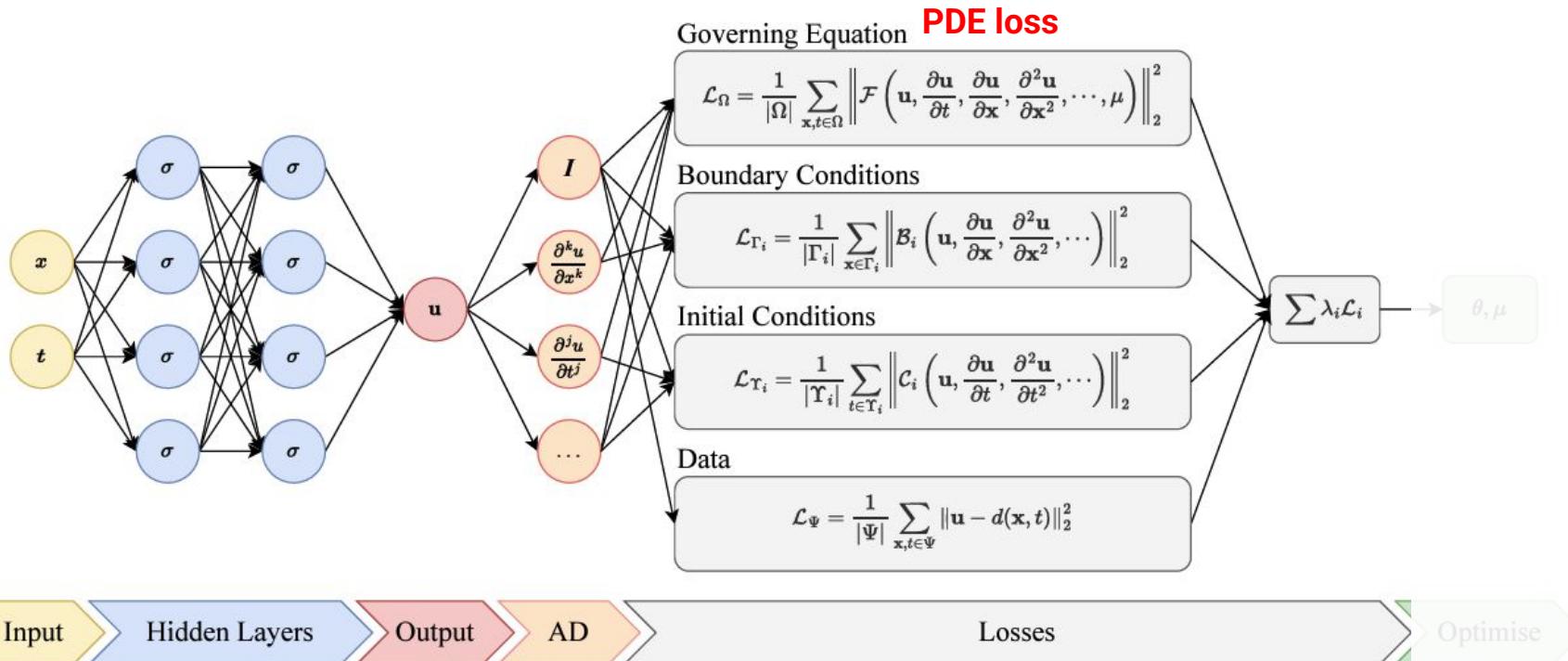
Workflow



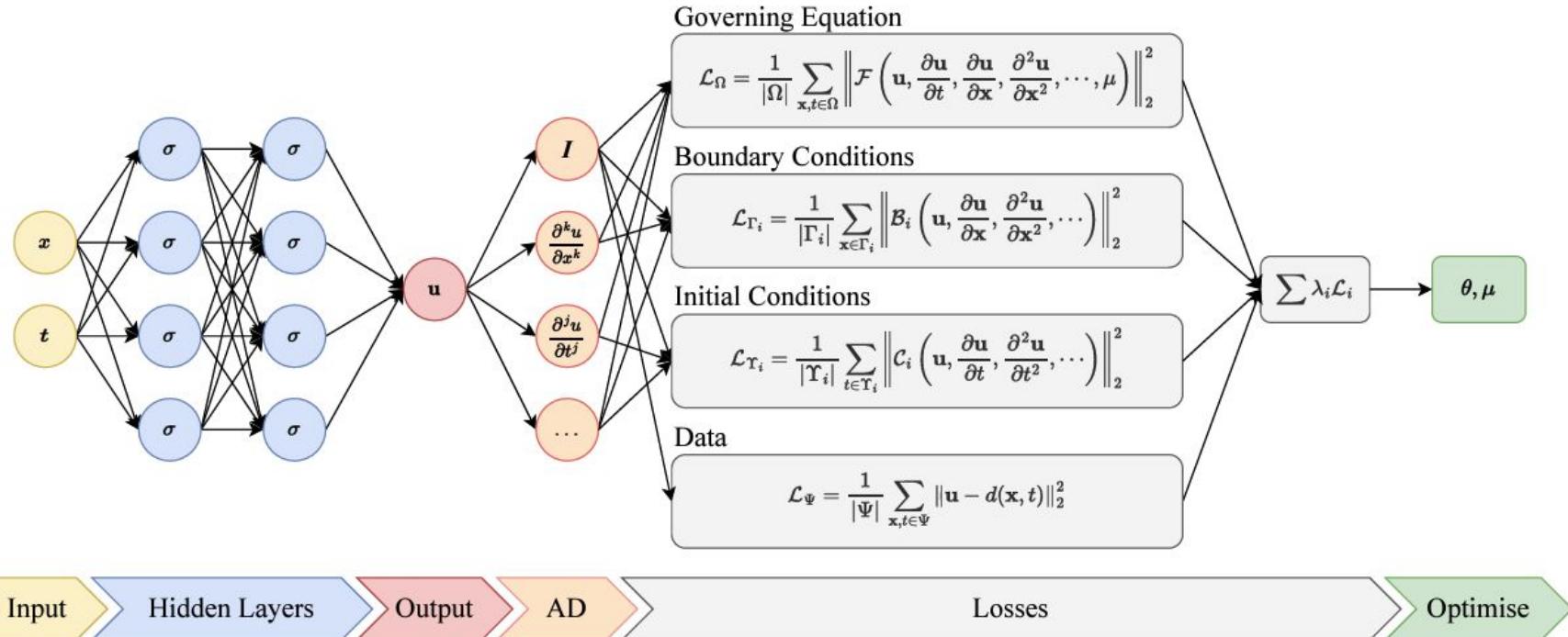
Automatic
differentiation

Bischof and Kraus

Workflow



Workflow



Loss Function

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{physics}} + \mathcal{L}_{\text{BC}} + \mathcal{L}_{\text{IC}} + \mathcal{L}_{\text{data}}$$

Where:

- $\mathcal{L}_{\text{physics}}$: The **Physics Loss**, which ensures the network solution satisfies the governing physical equation (e.g., the residual of a PDE or ODE).

$$\mathcal{L}_{\text{physics}} = \text{mean} (\text{residual of PDE}^2)$$

- \mathcal{L}_{BC} : The **Boundary Condition Loss**, which enforces the boundary conditions at the spatial or temporal boundaries of the problem domain.

$$\mathcal{L}_{\text{BC}} = \text{mean} (u_{\text{pred}}(x_{\text{boundary}}) - u_{\text{boundary}})^2$$

- \mathcal{L}_{IC} : The **Initial Condition Loss**, which ensures the solution satisfies the initial condition at time t_0 (if applicable).

$$\mathcal{L}_{\text{IC}} = \text{mean} (u_{\text{pred}}(x, t_0) - u_{\text{initial}})^2$$

- $\mathcal{L}_{\text{data}}$: The **Data Loss**, if available, measures how well the network fits any provided data points (e.g., experimental or observational data).

$$\mathcal{L}_{\text{data}} = \text{mean} (u_{\text{pred}}(x_{\text{data}}, t_{\text{data}}) - u_{\text{data}})^2$$

Showing MSE loss here but
feel free to use others

Case Study 1: 1D Harmonic Oscillator (a forward problem)

The example problem we solve here is a 1D damped harmonic oscillator

$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

t - time (s)

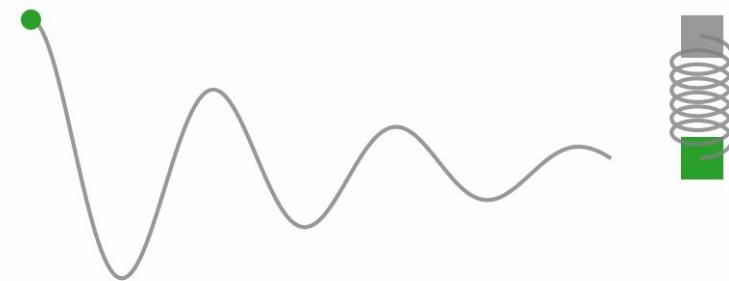
x - displacement from the equilibrium position (m)

m - mass (kg) It determines the inertia of the system, i.e., how much it resists changes in motion.

μ - damping coefficient (kg/s) It represents the strength of the damping force, which can arise from friction, air resistance, or any resistive force.

k - spring constant / stiffness (N/m) It characterizes the restoring force.

An Ordinary Differential Equation (**ODE**) involves derivatives with respect to one independent variable, often time. It can be considered a special case of PDE, which involves derivatives with respect to multiple independent variables.



Loss Function: Physics loss

- $\mathcal{L}_{\text{physics}}$: The **Physics Loss**, which ensures the network solution satisfies the governing physical equation (e.g., the residual of a PDE or ODE).

$$\mathcal{L}_{\text{physics}} = \text{mean}(\text{residual of PDE}^2)$$

$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

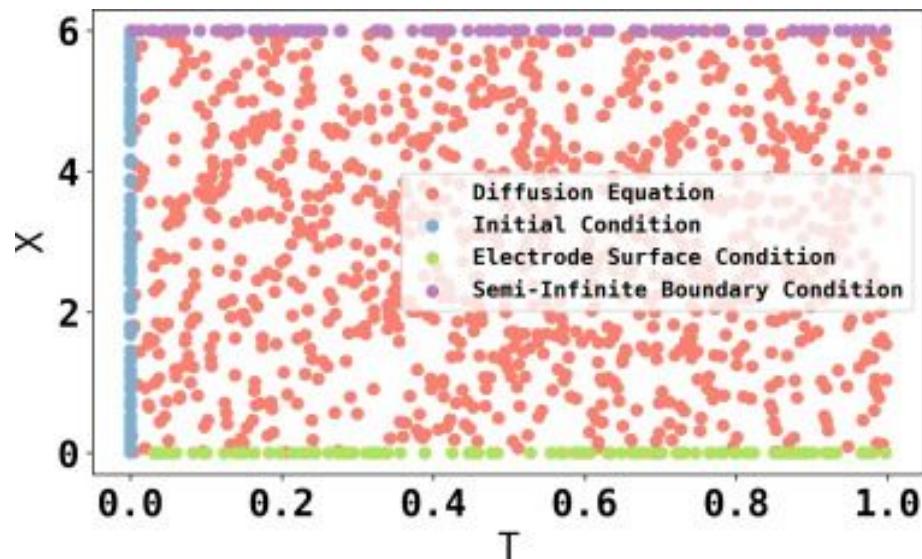
Step 1: Predict the solution from the neural network

What independent variables do I use to predict the solutions?

Collocation points

Collocation points

- **Collocation points** are points in the domain where we evaluate the residual of the PDE.
- The more points you have, the more accurately the model can represent the physical behavior.
- Types
 - Interior points
 - Boundary points
 - Initial condition points
- Generation
 - Uniform sampling
 - Random sampling
 - Adaptive sampling



Loss Function: Physics loss

- $\mathcal{L}_{\text{physics}}$: The **Physics Loss**, which ensures the network solution satisfies the governing physical equation (e.g., the residual of a PDE or ODE).

$$\mathcal{L}_{\text{physics}} = \text{mean}(\text{residual of PDE}^2)$$

$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

Step 1: Predict the solution from the neural network

What independent variables do I use to predict the solutions?

Collocation points

```
t_physics =  
torch.linspace(0, 1, 30).view(-1, 1).requires_grad_(True)  
  
x_pred_physics = model(t_physics)
```

Loss Function: Physics loss

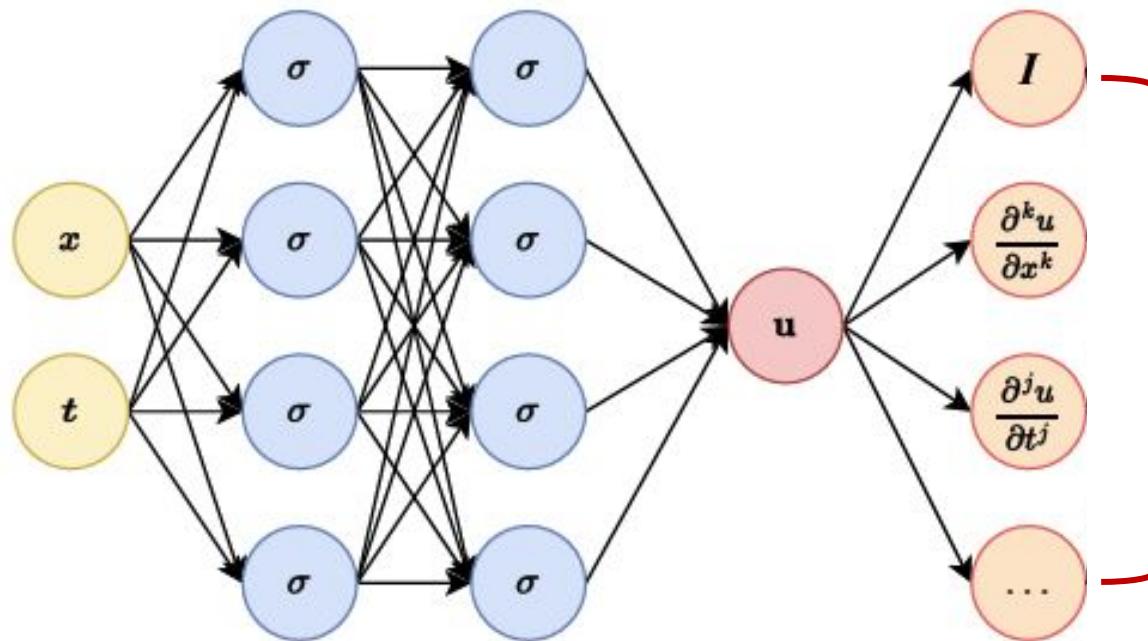
- $\mathcal{L}_{\text{physics}}$: The **Physics Loss**, which ensures the network solution satisfies the governing physical equation (e.g., the residual of a PDE or ODE).

$$\mathcal{L}_{\text{physics}} = \text{mean}(\text{residual of PDE}^2)$$
$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

Step 2: Compute the derivatives with **automatic differentiation**

```
# compute dx/dt  
  
dx = torch.autograd.grad(x_pred_physics, t_physics,  
torch.ones_like(x_pred_physics), create_graph=True)[0]  
  
# compute d^2x/dt^2      create_graph=True to compute higher-order derivatives  
  
dx2 = torch.autograd.grad(dx, t_physics,  
torch.ones_like(dx), create_graph=True)[0]
```

Automatic differentiation



Automatic differentiation is a technique that automatically computes exact derivatives of functions (like neural network outputs) with respect to their inputs by applying the chain rule efficiently through the computation graph

Via automatic
differentiation
Over NN

Loss Function: Physics loss

- $\mathcal{L}_{\text{physics}}$: The **Physics Loss**, which ensures the network solution satisfies the governing physical equation (e.g., the residual of a PDE or ODE).

$$\mathcal{L}_{\text{physics}} = \text{mean}(\text{residual of PDE}^2)$$
$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

Step 3: Compute the physics residual and physics loss

They should be zero if the model solves the physical equation perfectly

```
physics = m*dx2 + mu*dx + k*x_pred_physics
```

```
loss_physics = (1e-4)*torch.mean(physics**2)
```

Weight

Loss Function: Initial condition loss

- \mathcal{L}_{IC} : The **Initial Condition Loss**, which ensures the solution satisfies the initial condition at time t_0 (if applicable).

$$\mathcal{L}_{\text{IC}} = \text{mean} (u_{\text{pred}}(x, t_0) - u_{\text{initial}})^2$$

In this case, the initial conditions are

$$x(0) = 1, \frac{dx}{dt}(0) = 0$$

- 1) The displacement is 1,
- 2) The acceleration is 0.

Loss Function: Initial condition loss

- \mathcal{L}_{IC} : The **Initial Condition Loss**, which ensures the solution satisfies the initial condition at time t_0 (if applicable).

$$\mathcal{L}_{\text{IC}} = \text{mean} (u_{\text{pred}}(x, t_0) - u_{\text{initial}})^2$$

$$x(0) = 1, \frac{dx}{dt} = 0$$

Step 1: Predict the solution at initial condition

```
t_0 = torch.tensor([[0.0]], requires_grad=True)
```

```
x_pred_0 = model(t_0)
```

Loss Function: Initial condition loss

- \mathcal{L}_{IC} : The **Initial Condition Loss**, which ensures the solution satisfies the initial condition at time t_0 (if applicable).

$$\mathcal{L}_{\text{IC}} = \text{mean} (u_{\text{pred}}(x, t_0) - u_{\text{initial}})^2$$

$$x(0) = 1, \frac{dx}{dt} = 0$$

Step 2: Compute the derivatives at initial condition

```
# compute dx/dt at t=0
```

```
dx0 = torch.autograd.grad(x_pred_0, t_0,  
torch.ones_like(x_pred_0), create_graph=True)[0]
```

Loss Function: Initial condition loss

- \mathcal{L}_{IC} : The **Initial Condition Loss**, which ensures the solution satisfies the initial condition at time t_0 (if applicable).

$$\mathcal{L}_{\text{IC}} = \text{mean} (u_{\text{pred}}(x, t_0) - u_{\text{initial}})^2$$

$$x(0) = 1, \frac{dx}{dt} = 0$$

Step 3: Compute the initial condition residuals and loss

```
ic1 = x_pred_0 - 1
```

```
ic2 = dx0
```

```
loss_ic = (1e-4) * (ic1**2 + ic2**2)
```

Loss Function: Boundary condition loss

- \mathcal{L}_{BC} : The **Boundary Condition Loss**, which enforces the boundary conditions at the spatial or temporal boundaries of the problem domain.

$$\mathcal{L}_{BC} = \text{mean} (u_{\text{pred}}(x_{\text{boundary}}) - u_{\text{boundary}})^2$$

We don't have this term in this case study, but its implementation is similar to that of initial condition loss.

Instead of specifying $t=0$, we examine the solutions and derivatives at boundaries of x and t .

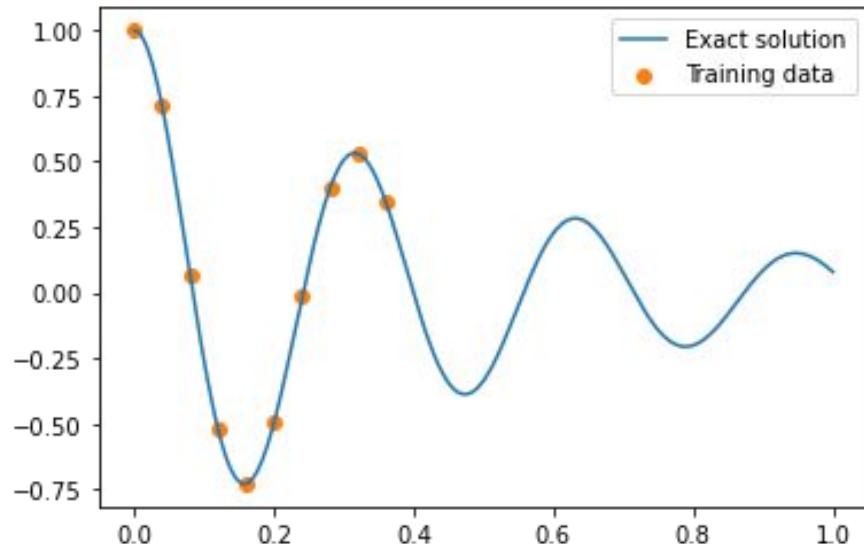
Loss Function: Data loss

- $\mathcal{L}_{\text{data}}$: The **Data Loss**, if available, measures how well the network fits any provided data points (e.g., experimental or observational data).

$$\mathcal{L}_{\text{data}} = \text{mean} (u_{\text{pred}}(x_{\text{data}}, t_{\text{data}}) - u_{\text{data}})^2$$

Step 1: Predict the solution at known data points

```
x_pred_data = model(t_data)
```



Loss Function: Data loss

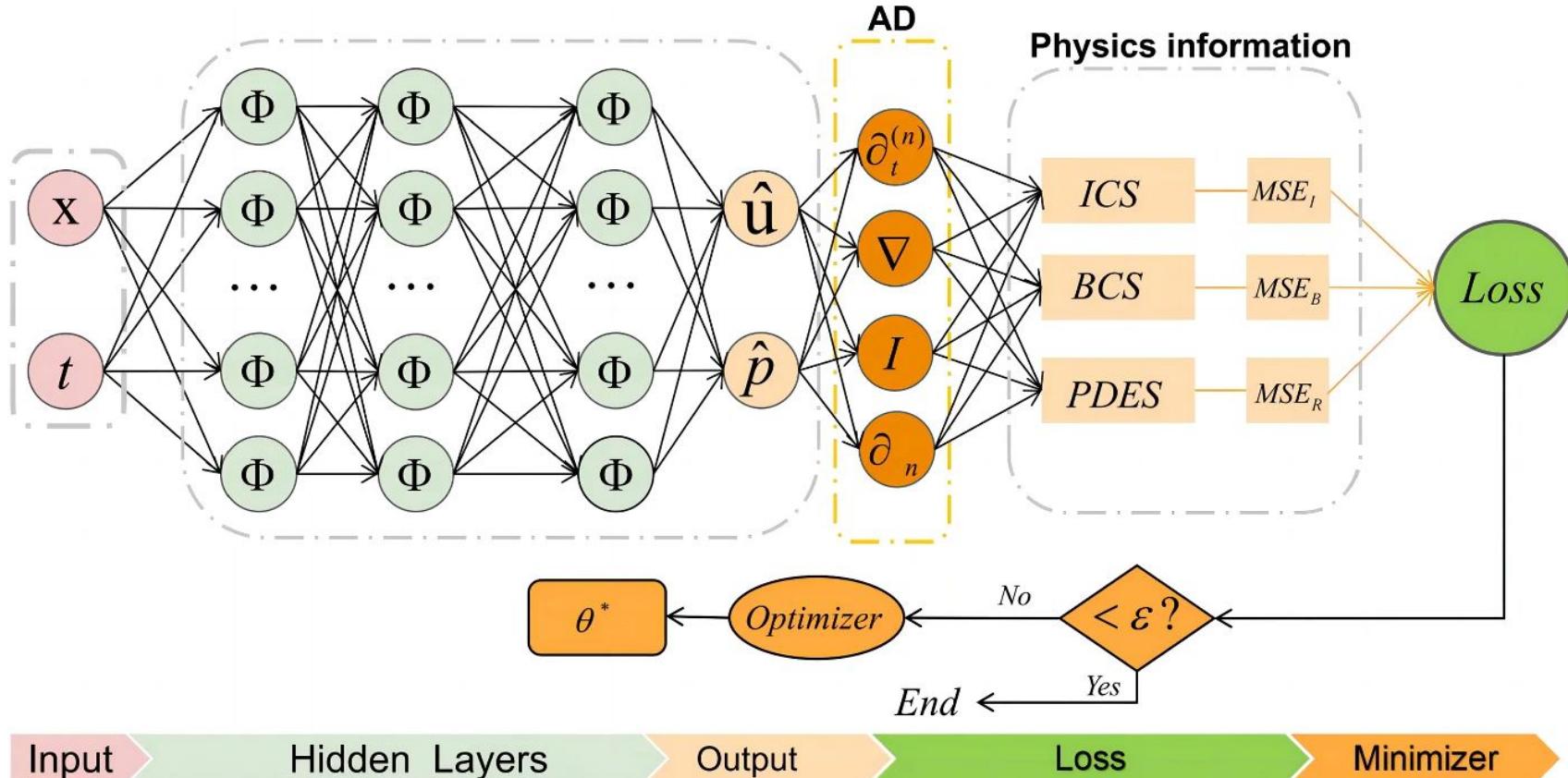
- $\mathcal{L}_{\text{data}}$: The **Data Loss**, if available, measures how well the network fits any provided data points (e.g., experimental or observational data).

$$\mathcal{L}_{\text{data}} = \text{mean} (u_{\text{pred}}(x_{\text{data}}, t_{\text{data}}) - u_{\text{data}})^2$$

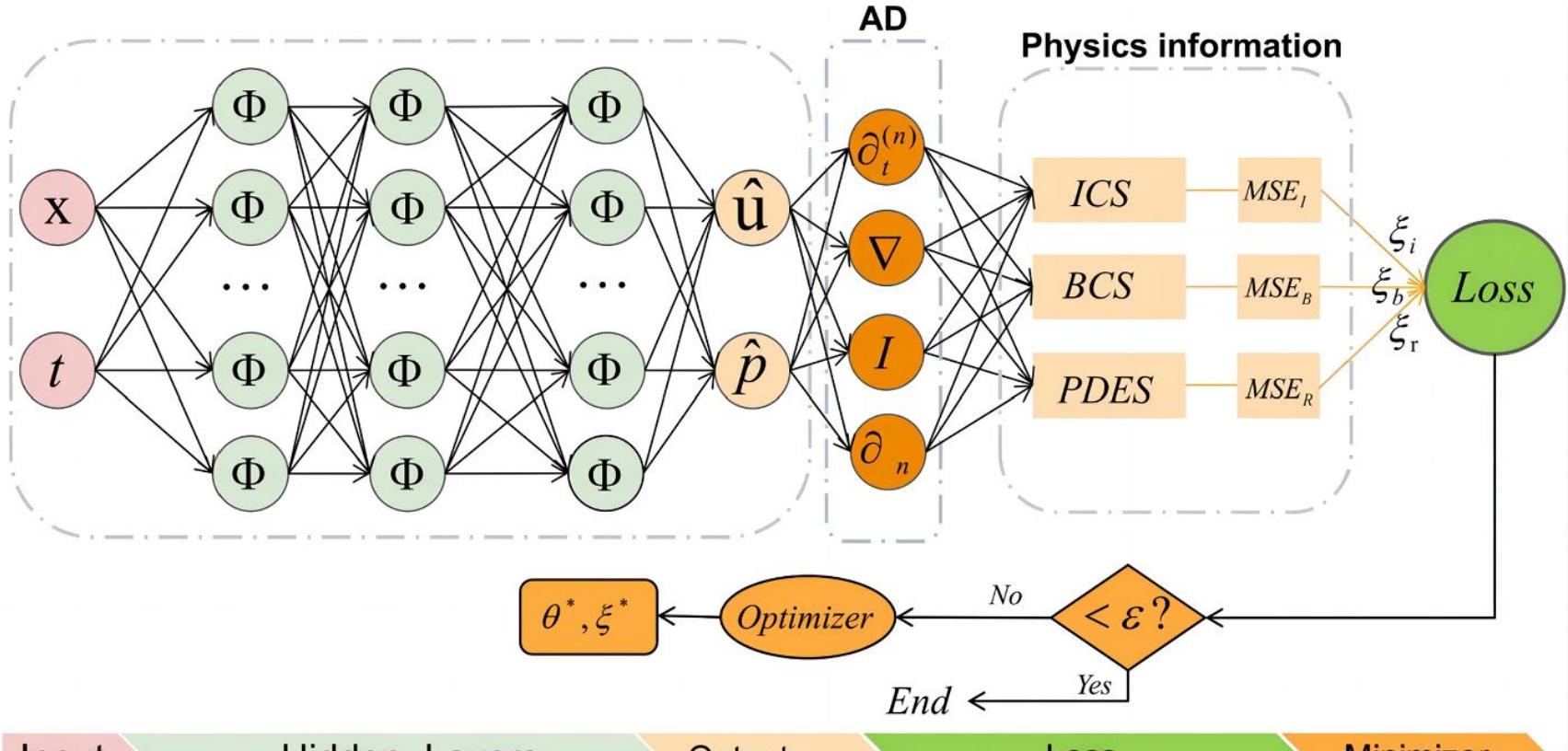
Step 2: Compute data residuals and data loss

```
loss_data = torch.mean((x_pred_data-y_data)**2)
```

Loss term weighting



Loss term weighting



Input

Hidden Layers

Output

Loss

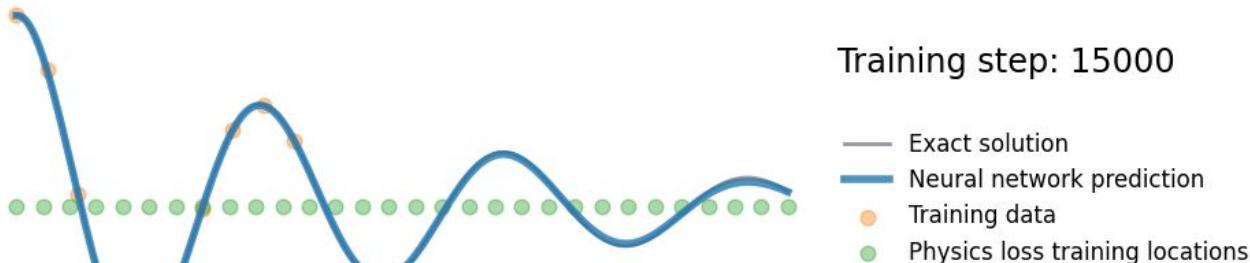
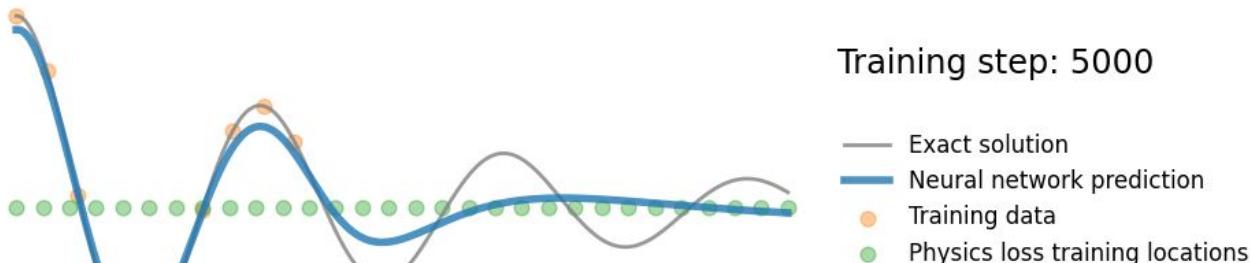
Minimizer

Neural network setup

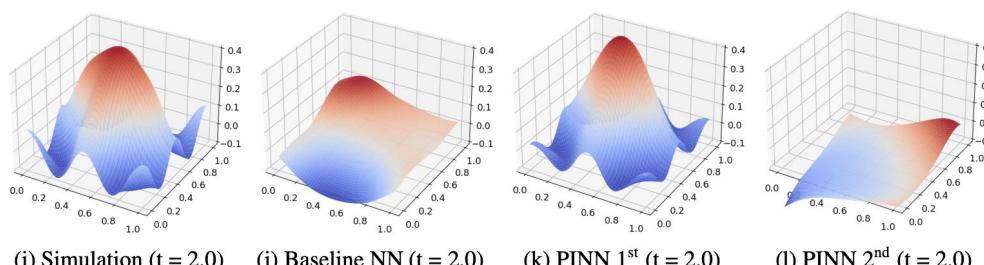
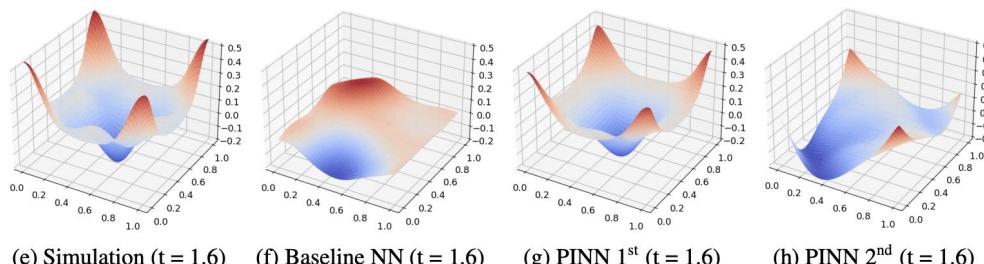
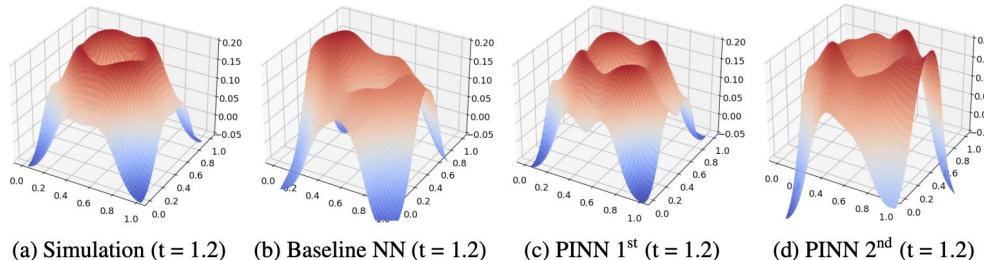
The setup of the neural network is the similar

- **Input:** Temporal & spatial coordinates t, x
- **Output:** Approximated latent solution $u(t, x)$
- **Architecture:** typically shallow (2–4 hidden layers) and moderately wide (20–100 neurons/layer)
- No dropout or batch norm
 - These can interfere with derivative computations and physics consistency
- **Activation functions:** Smooth activations preferred
 - e.g., \tanh , \sin , softplus , but often not ReLU
 - Smoothness is crucial: we differentiate the NN output to compute residuals
- **Optimizer:** Start with Adam, then switch to L-BFGS

Evaluate performance with analytical solutions and observations

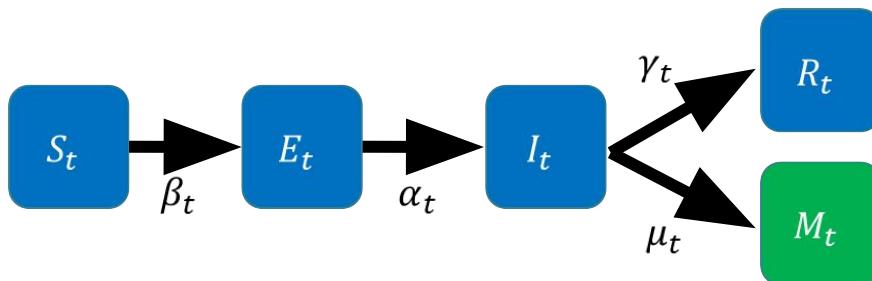


No guarantee in extrapolation



Case study 2: SEIRM Epidemiological Model (an inverse problem)

- Latent/unobserved ODE states
- Observed ODE states



[Rodríguez et al. AAAI 2023]

ODE States (at time t)

\$S_t\$: No. of susceptible

\$E_t\$: No. of exposed

\$I_t\$: No. of infectious

\$R_t\$: No. of recovered

\$M_t\$: No. of deaths

$$\frac{dS_t}{dt} = -\beta_t \frac{S_t I_t}{N} \quad \frac{dE}{dt} = \beta_t \frac{S_t I_t}{N} - \alpha_t E_t$$

$$\frac{dI_t}{dt} = \alpha_t E_t - \gamma_t I_t - \mu_t I_t \quad \frac{dR_t}{dt} = \gamma_t I_t \quad \frac{dM_t}{dt} = \mu_t I_t$$

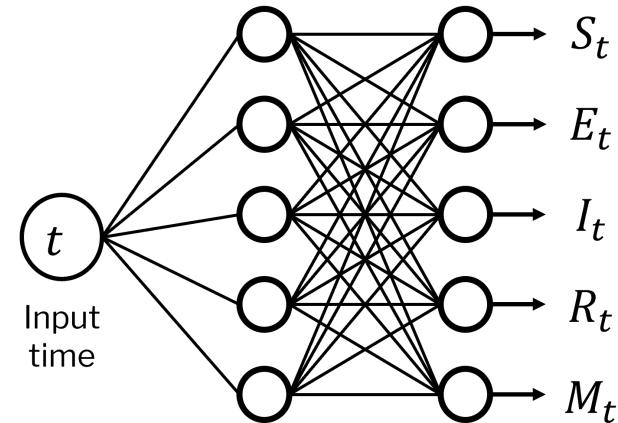
where $s_t = [S_t, E_t, I_t, R_t, M_t]^T \rightarrow$ ODE states
and $\Omega_t = [\beta_t, \alpha_t, \gamma_t, \mu_t]^T \rightarrow$ ODE parameters

Unknown, except M_t
Unknown

Workflow

1. Compute s_t via forward pass over neural network
2. Compute $\frac{ds_t}{dt}$ via automatic differentiation
3. Compute $f_{ODE}(s_t, \Omega_t)$ using NN outputs and ODE
4. Minimize residual of steps 2 and 3, and include ground truth data

$$\min \frac{1}{N+1} \sum_{t=t_0}^{t_N} \left[\frac{ds_t}{dt} - f_{ODE}(s_t, \Omega_t) \right]^2 + \frac{1}{N+1} \sum_{t=t_0}^{t_N} \left[\hat{M}_t - M_t \right]^2$$

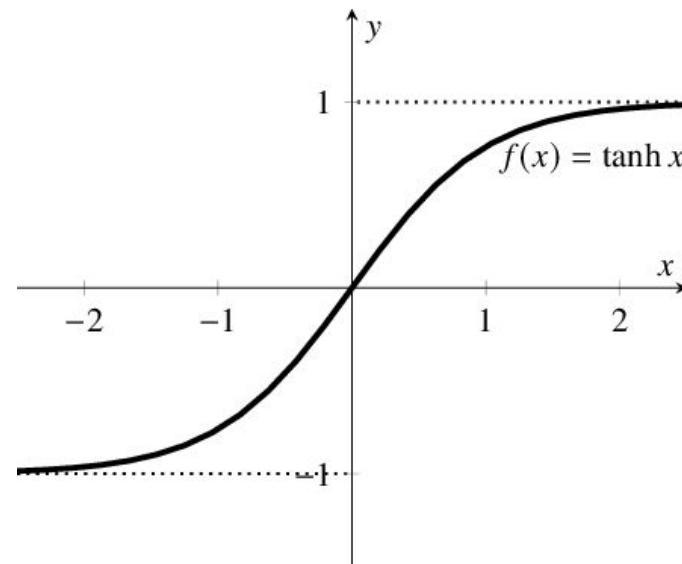
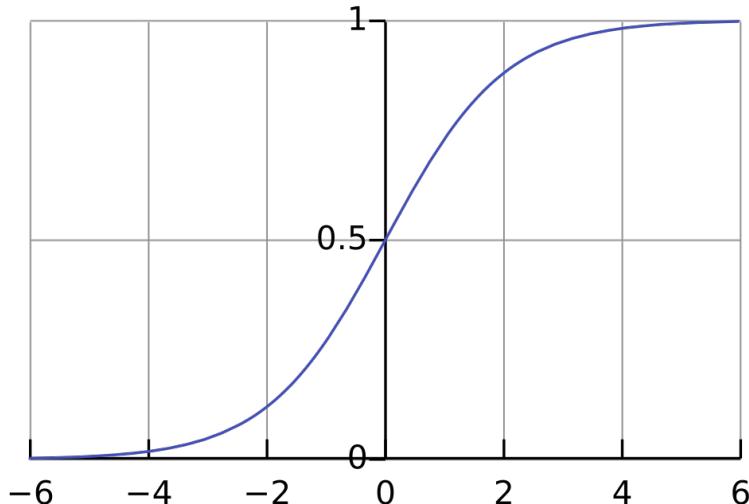


5. Update both NN parameters and Ω_t

The methods do not change, we just need to make parameters learnable

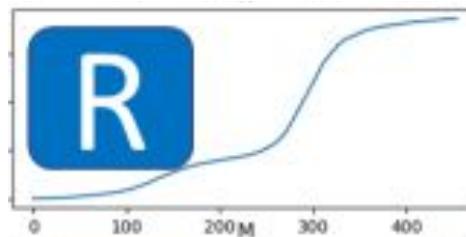
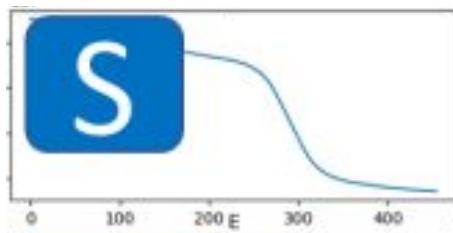
Issue in learning PDE / ODE parameters

- ODE parameters Ω_t need to be within $[0,1]$ but gradient descent is not designed to work with constraints
- Solution: add a sigmoid or tanh layer



Constraining solution space w/ domain knowledge: monotonicity

- Low observability → ill-posed problem!
- We know some variables are monotonic wrt time: we can construct losses to impose this
- For example, we know susceptible and recovered are monotonically decreasing and increasing, respectively



$$\mathcal{L}^{Mono} = \frac{1}{N+1} \left(\sum_{t=t_0}^{t_N} \left[\frac{dS_t}{dt} \text{ReLU}\left(\frac{dS_t}{dt}\right) \right] + \sum_{t=t_0}^{t_N} \left[-1 \frac{dR_t}{dt} \text{ReLU}\left(-\frac{dR_t}{dt}\right) \right] \right)$$

Constraining solution space w/ domain knowledge: smoothness

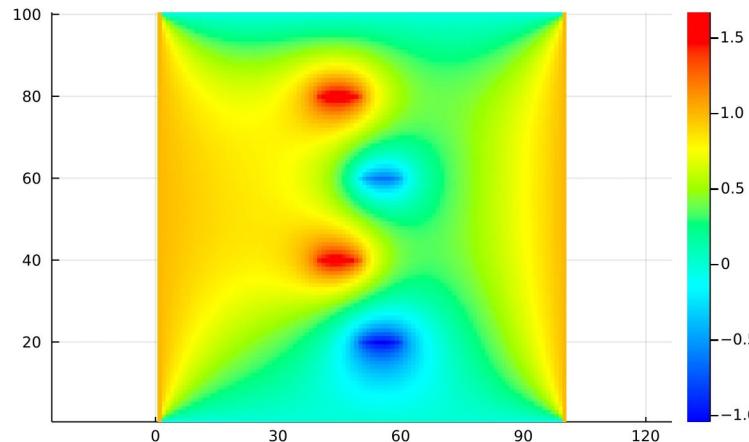
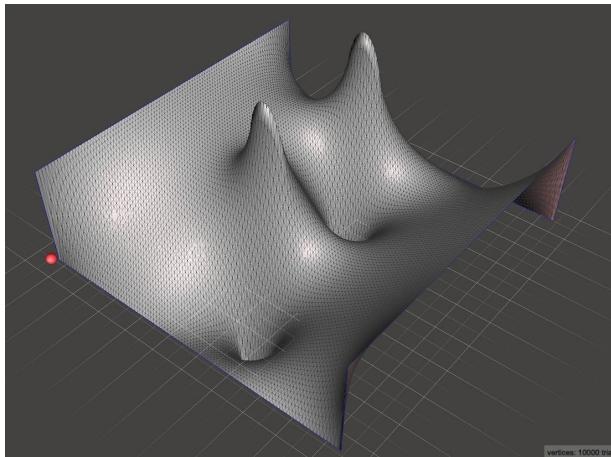
- Time-varying ODE parameters should change smoothly
 - Because **daily** epidemic dynamics change gradually
 - Thus, we should remove solutions for Ω_t that change abruptly
- Include a loss to incorporate this

$$\mathcal{L}^{Param} = \frac{1}{N+1} \sum_{t=t_0}^{t_N} [\Omega_{t+1} - \Omega_t]^2$$

Limitations, solutions,
recent advancements

Limitation 1: Gradient imbalance

- Consider solving a Poisson equation (difference equation)



Matt Ferraro

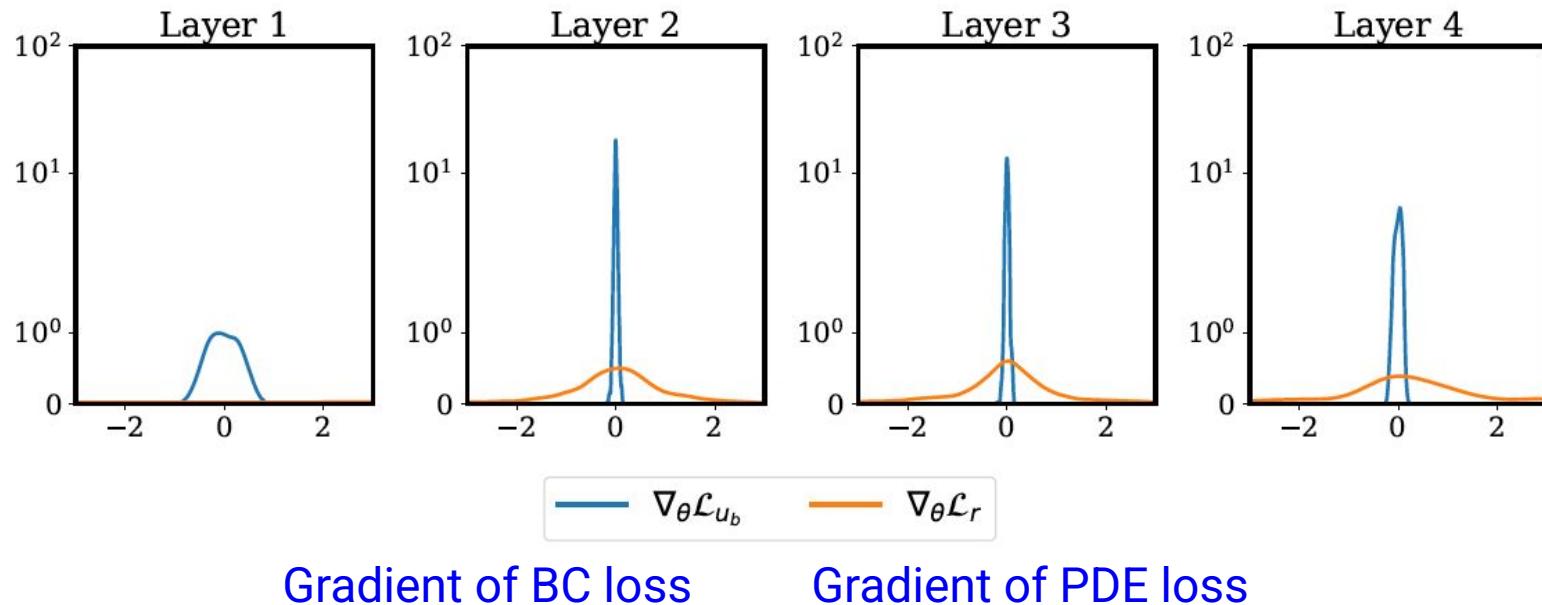
- We can obtain data for multiple values of C, and for each we will train PINN of 4 layers using the following loss

$$\mathcal{L}(\theta) = \mathcal{L}_r(\theta) + \mathcal{L}_{u_b}(\theta)$$

PDE loss BC loss

Limitation 1: Gradient imbalance

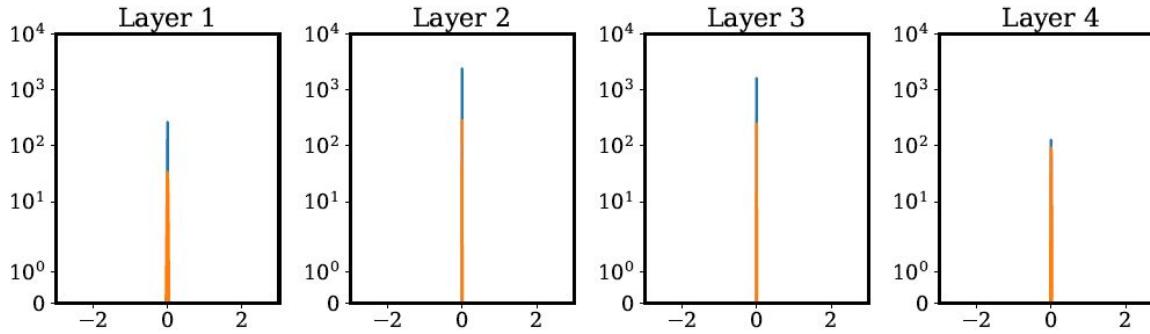
- Gradients of PDE residual loss generally dominate the gradients of the others
- Histogram of gradients (notice the imbalance)



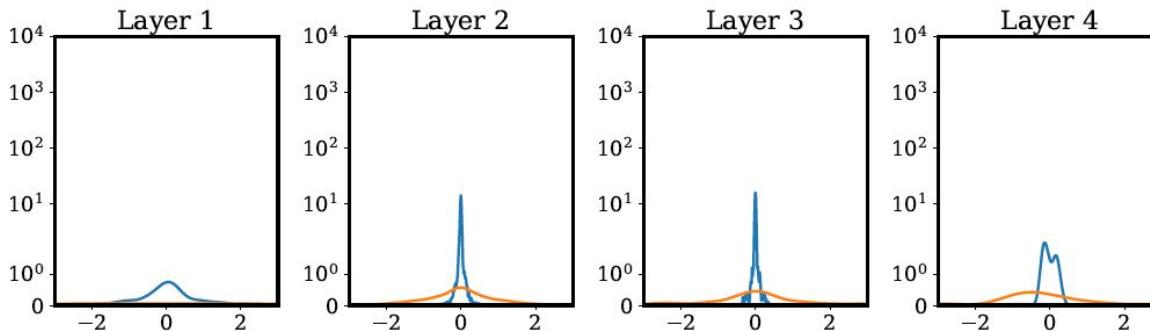
Limitation 1: Gradient imbalance

- Larger the gradient imbalance, larger the error

C=1



C=4



	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$
Rel. L^2 error	$1.34e - 04$	$4.65e - 04$	$3.82e - 03$	$1.30e - 02$	$9.44e + 00$

Solution to limitation 1: Adaptive loss weights

- Loss balancing via inverse of losses (Lu et al., *SIAM Rev.* 2021)
- Learning rate annealing algorithm using gradient statistics (Wang et al., *SIAM J. Sci. Comput.* 2021)
 - Intent: For any given loss term \mathcal{L}_i (BC, IC, or data), find λ_i (weight) such that

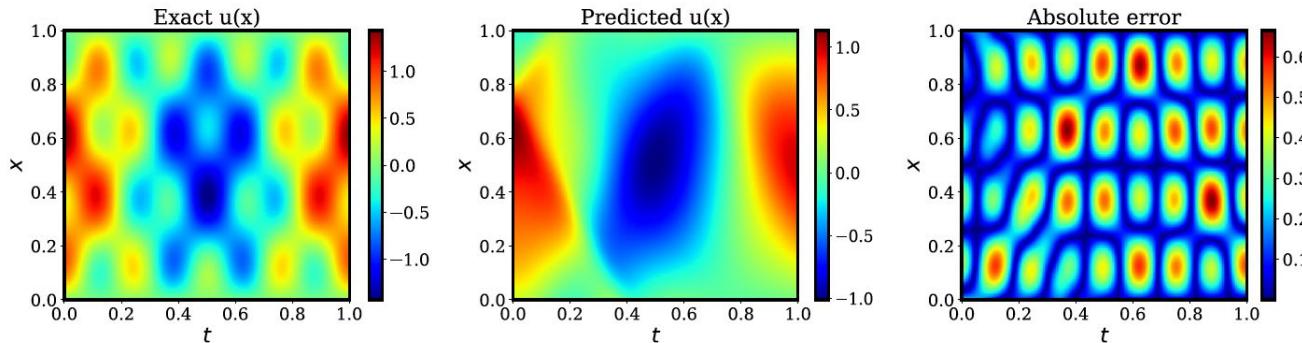
$$\lambda_i \overline{|\nabla_{\theta} \mathcal{L}_i(\theta)|} = \max_{\theta_n} \{ |\nabla_{\theta} \mathcal{L}_r(\theta_n)| \},$$

i.e., find weight such that weight * mean of loss = largest residual loss

- We can make the norm of gradients of each weighted loss equal to each other (Wang et al., *arXiv* 2023)
 - This prevents being biased towards minimizing certain terms during training

Limitation 2: Spectral bias

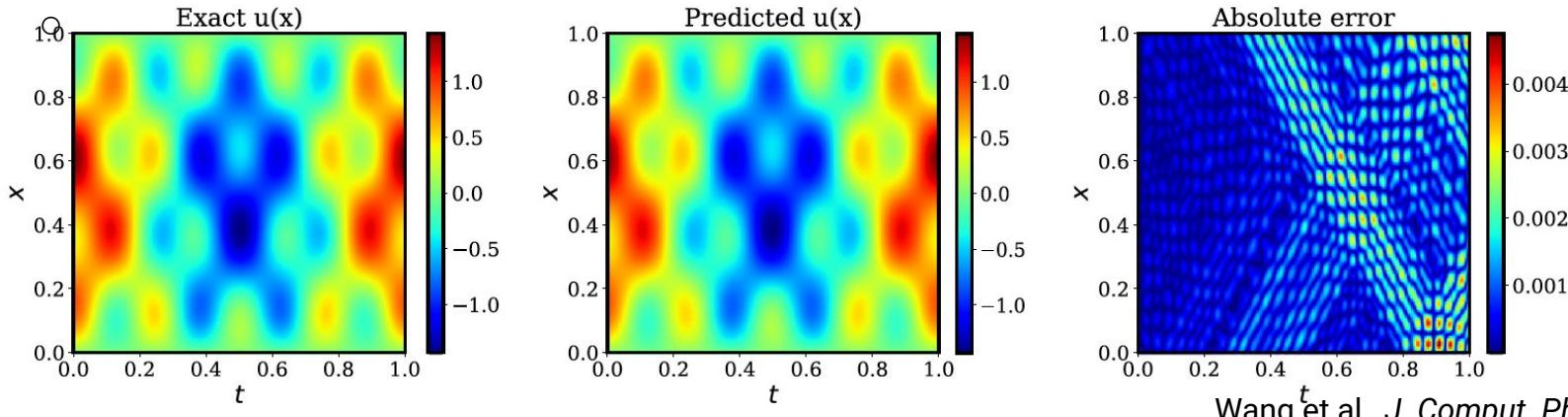
- Several studies pointed out that fully-connected NNs have difficulties in learning high-frequency functions



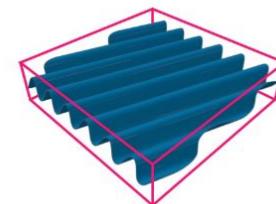
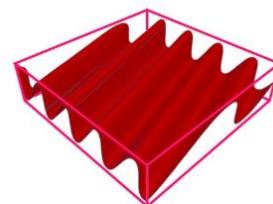
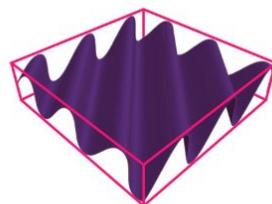
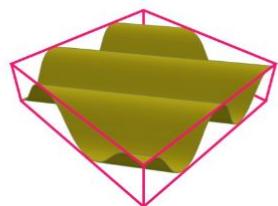
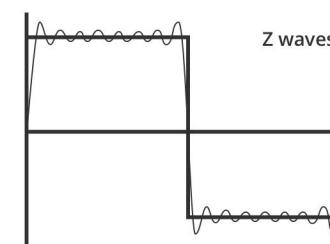
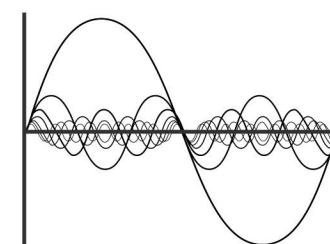
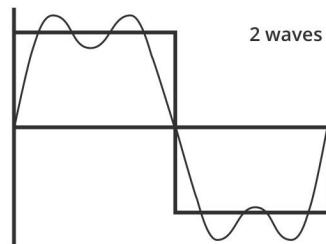
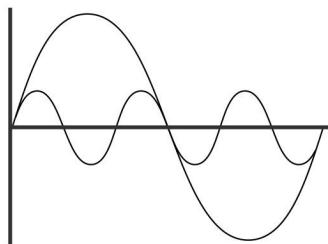
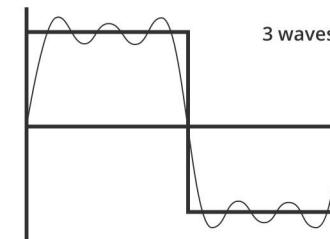
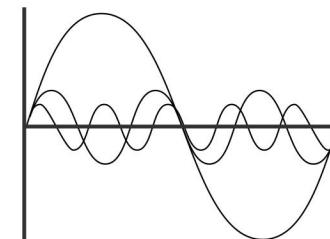
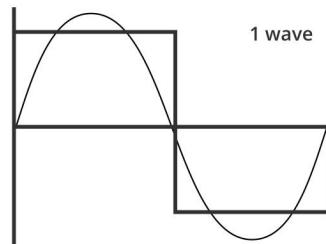
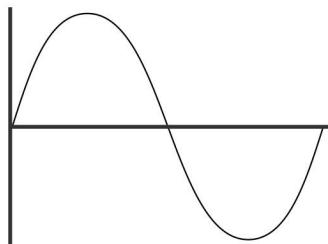
Misses several
high-frequencies

Solution to limitation 2: Adaptive learning rates

- The Neural Tangent Kernel (NTK) theory
 - Key idea: infinitely wide NNs are almost linear
 - The NTK tells us how each parameter update affects the function output
 - The eigenvalues of the NTK indicate how quickly different modes (or regions) of the function are being learned
- This insight can be used to adapt learning rates
 - Training points or components in slow-learning regions are given higher weight



Solution to limitation 2: Enriching frequencies via random Fourier features



Solution to limitation 2: Enriching frequencies via random Fourier features

- Random Fourier features can make coordinate-based MLPs represent higher frequency details and thus better suited for modeling functions in low dimensions.

$$\Gamma(v) = [\cos(2\pi \mathbf{B}v), \sin(2\pi \mathbf{B}v)]^T$$

$\mathbf{B} \in \mathbb{R}^{d \times 1}$ is sampled from $\mathcal{N}(0, \sigma^2)$

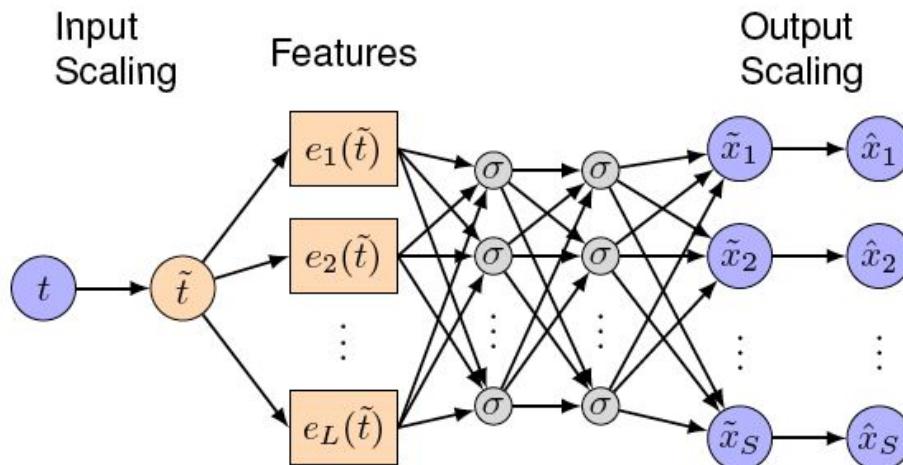
where v is input, e.g., x and t

σ is a user-specified hyper-parameter

\mathbf{B} control the frequencies in the Fourier features

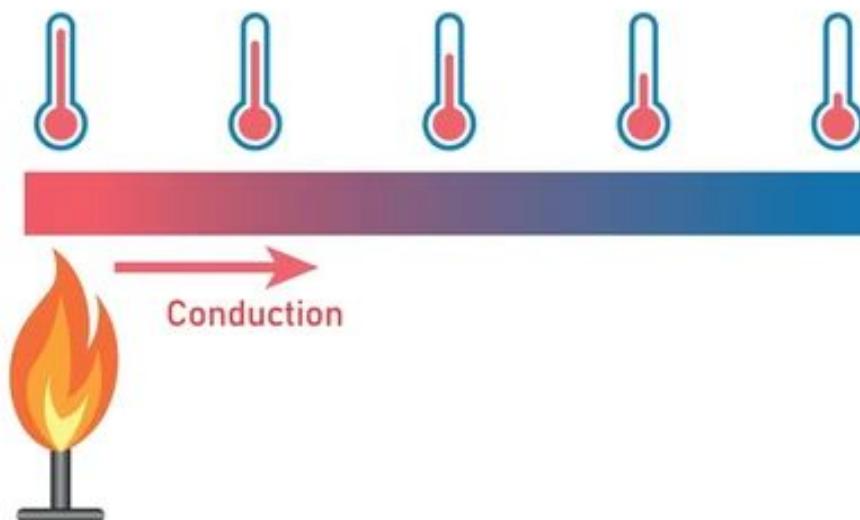
Solution to limitation 3: Scaling and feature expansion

- Scaling: Normalize layer for inputs and outputs
- Feature expansion: add patterns relevant to the system, such as
 - Periodicity $\sin(kt)$
 - Fast decay e^{-kt}
 - Multiscale features



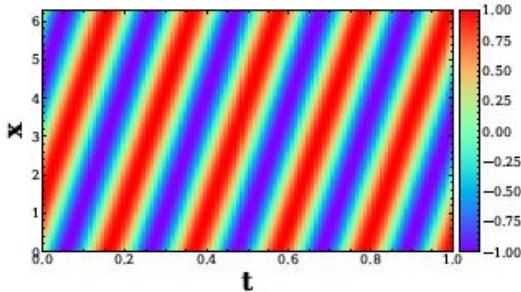
Limitation 3: Propagation failures

- PINNs relies on successful “propagation” of solution from initial and/or boundary condition points to interior points

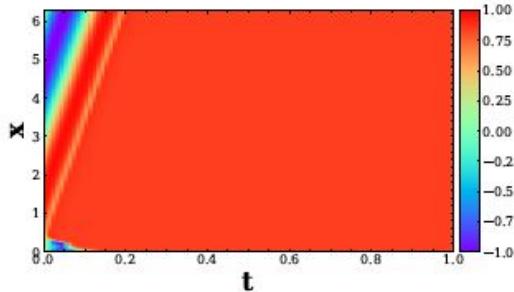


Limitation 3: Propagation failures

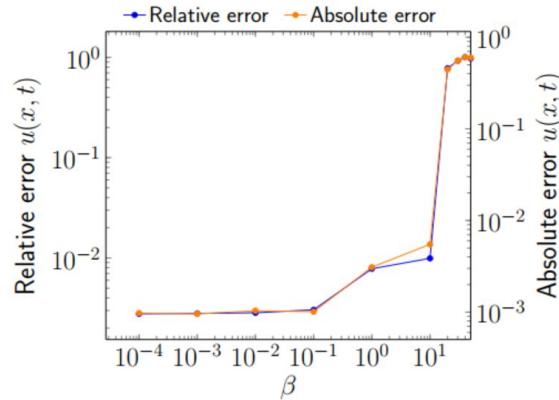
- Example in 1D convection problem
 - β is convection coefficient
- The PINN has difficulty predicting the solution past a certain time step but can fit the boundary conditions.



(b) Exact solution for $\beta = 30$



(c) PINN solution for $\beta = 30$

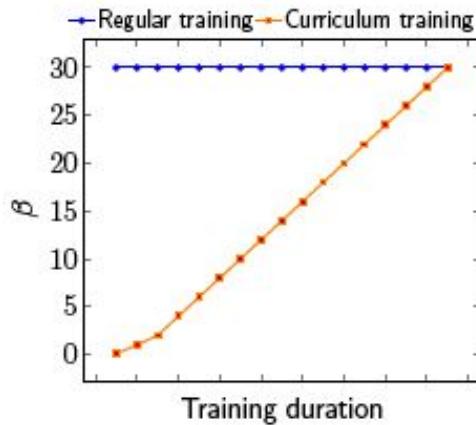


(a) Error for different β

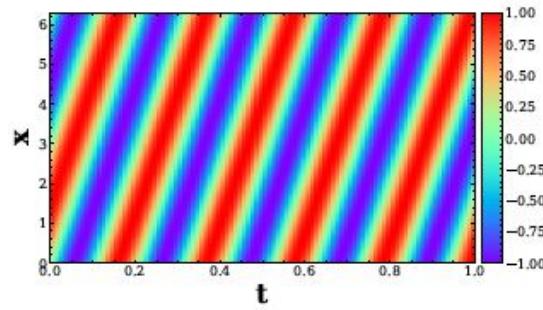
- It becomes harder to train with larger β

Solution to limitation 3: Curriculum regularization

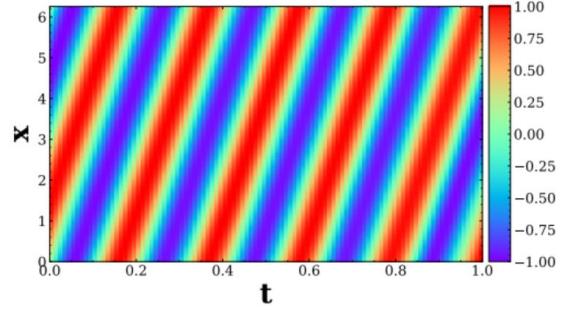
- Decompose the optimization task into a sequence of more manageable sub-tasks
- In this case, we warm start the NN training with easier instances of the PDE
- Objective: good initialization of NN weights



(a) Curriculum regularization schematic



(b) Exact solution for $\beta = 30$



(c) Curriculum training PINN solution for $\beta = 30$

Limitation 4: Inadequate inductive bias

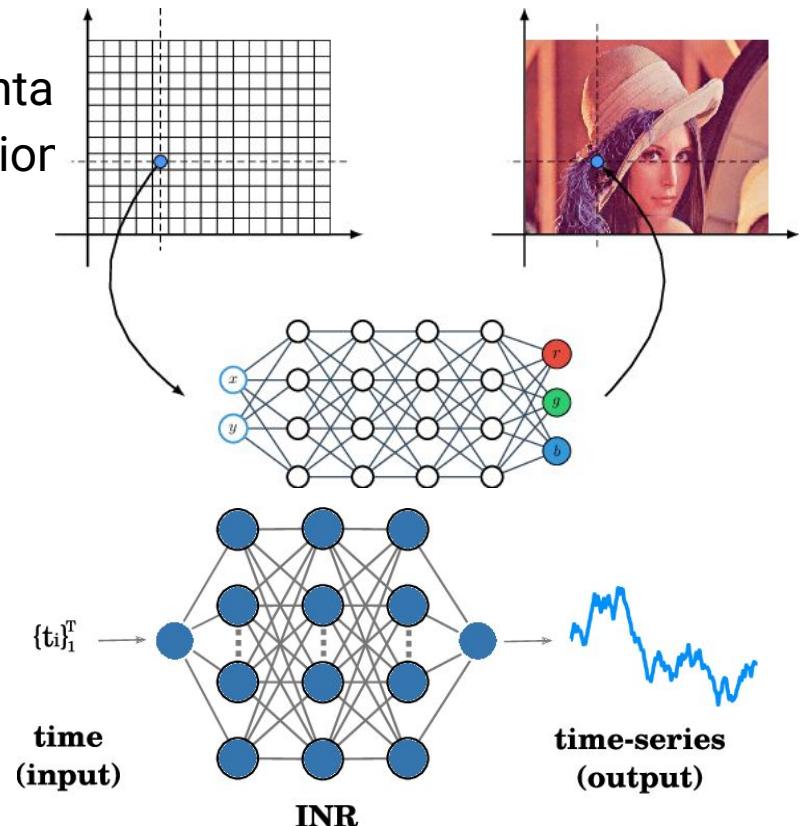
- Relational inductive bias: **inductive biases** which impose constraints on relationships and interactions among entities in a learning process

Component	Entities	Relations	Rel. inductive bias	Invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Nodes	Edges	Arbitrary	Node, edge permutations

Table 1: Various relational inductive biases in standard deep learning components. See also Section 2.

Limitation 4: Inadequate inductive bias

- In traditional PINNs, we do not take advantage of the relationships/structure of the solution space
- These NN are known as implicit neural representations (INR):
 - take only coordinate or time as input

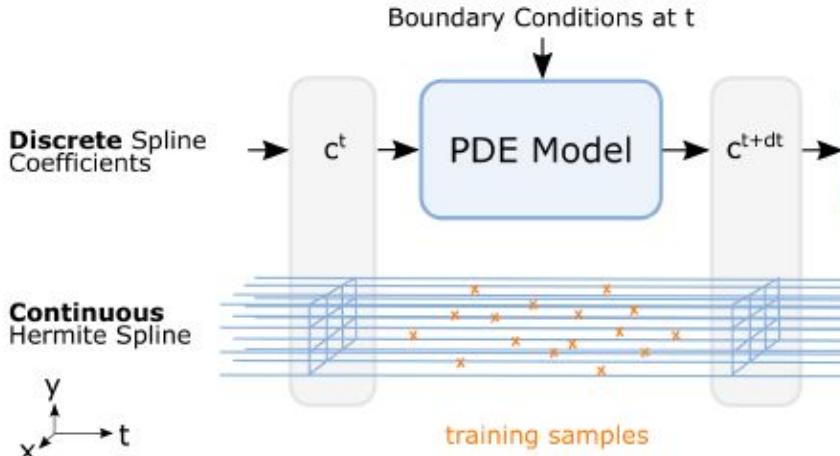


Solution to limitation 4: Connect to NN Architectures

- PINNs + CNNs

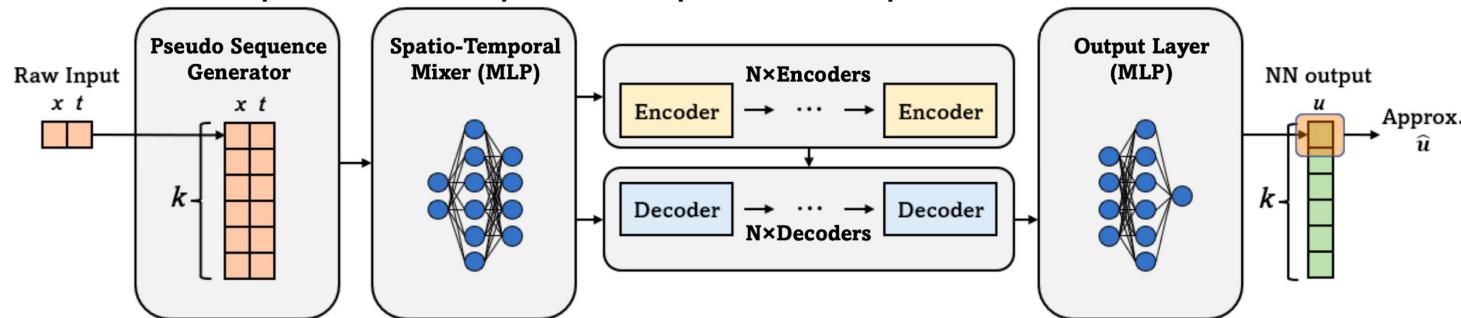
(Wandel et al., AAAI 2022)

- Instead of having the CNN directly output the solution to the PDE, the CNN output spline coefficients on a discrete grid

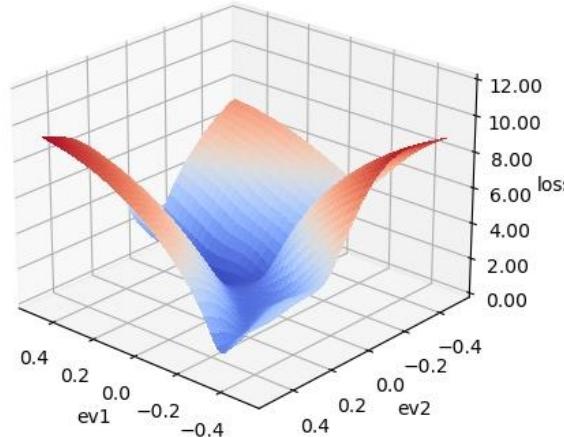
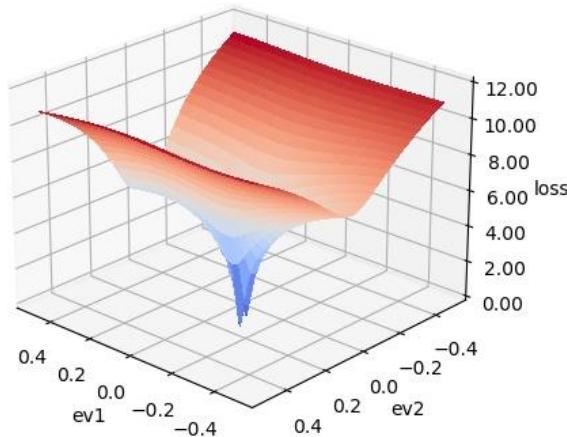


- PINNs + Transformers (Zhao et al., ICLR 2024)

- Transforms point-wise inputs into pseudo sequences



Solution to limitation 4: Connect to NN Architectures



PINNs:

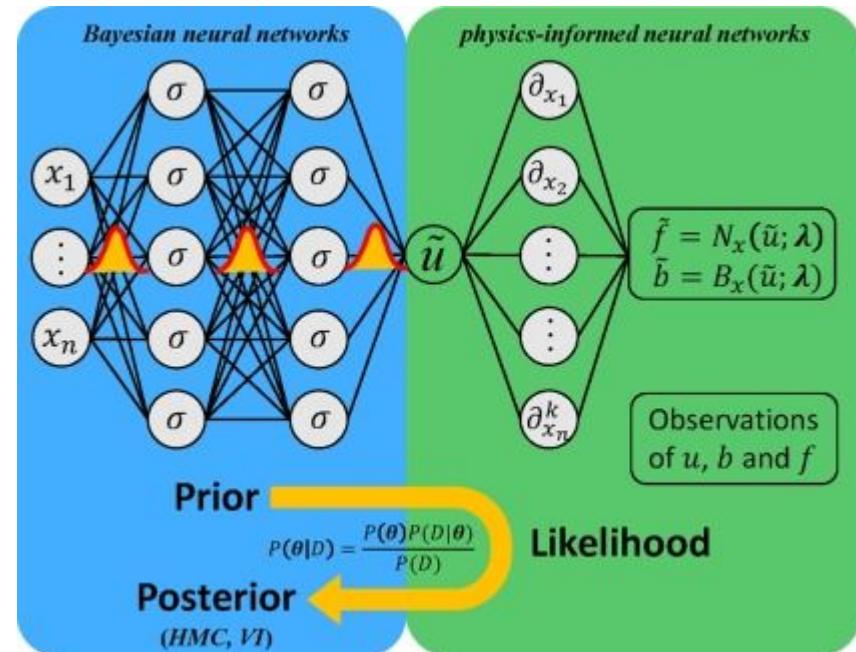
- Rough loss landscape
- Multiple local minima
- Hard to optimize

PINNs + Transformer:

- Smoother loss landscape
- Single local minima
- Easier to optimize

Recent advancement: Bayesian PINNs

- **Bayesian PINN** integrates uncertainty quantification and more domain knowledge into PINNs.
- In traditional PINNs, the network weights are deterministic (i.e., fixed values after training).
- In Bayesian PINNs, we treat the weights as distributions to model uncertainty in the solution.



Recent advancement: Transfer Learning with PINNs

- **Pretrain** a base PINN with a representative configuration.
- **Fine-tune** this model with new but related configurations using fewer training iterations.
- Multi-head architecture

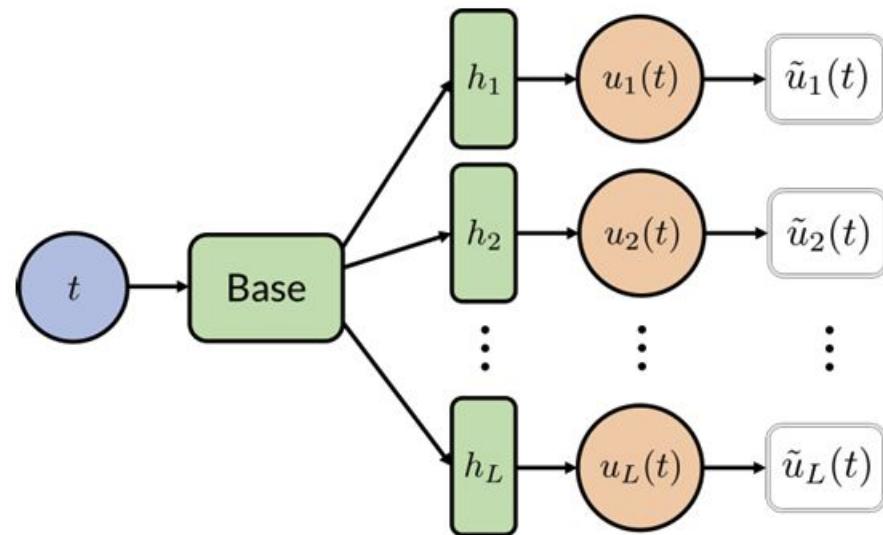
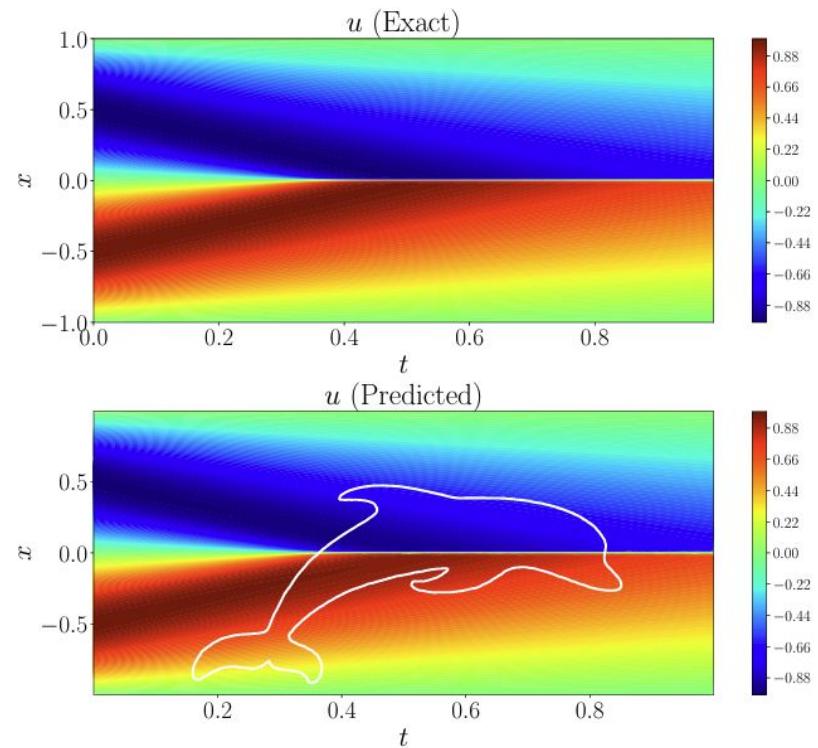


Figure 1: Multi-head PINN architecture. Each output head h_l is responsible for generating the solution to the l^{th} initial condition.

Recent advancement: eXtended Physics-Informed Neural Networks (XPINNs)

- Split the physical domain into overlapping or non-overlapping subdomains
- Train a separate PINN in each subdomain
- Enforce interface conditions to ensure continuity across subdomains
- Parallel training across subdomains
→ faster computation



Summary

- **Pros:** PINN is simple yet powerful idea; it can be implemented in a few lines of code for any PDE.
- **Cons:** In several applications you will faces and will need multiple additions to make it work.
- Good news: active community working on multiple tools
- NOT a replacement for classical partial differential equations solving methods

More tips on training PINNs

[Wang et al. arXiv 2023]

AN EXPERT'S GUIDE TO TRAINING PHYSICS-INFORMED NEURAL NETWORKS

Sifan Wang

Graduate Group in Applied Mathematics
and Computational Science
University of Pennsylvania
Philadelphia, PA 19104
sifanw@sas.upenn.edu

Shyam Sankaran

Department of Mechanical Engineering
and Applied Mechanics
University of Pennsylvania
Philadelphia, PA 19104
shyamss@seas.upenn.edu

Hanwen Wang

Graduate Group in Applied Mathematics
and Computational Science
University of Pennsylvania
Philadelphia, PA 19104
wangh19@sas.upenn.edu

Paris Perdikaris

Department of Mechanical Engineering
and Applied Mechanics
University of Pennsylvania
Philadelphia, PA 19104
pgp@seas.upenn.edu