MIDAS Summer Academy
July, 2025

Physics-Informed Neural Networks (PINNs)

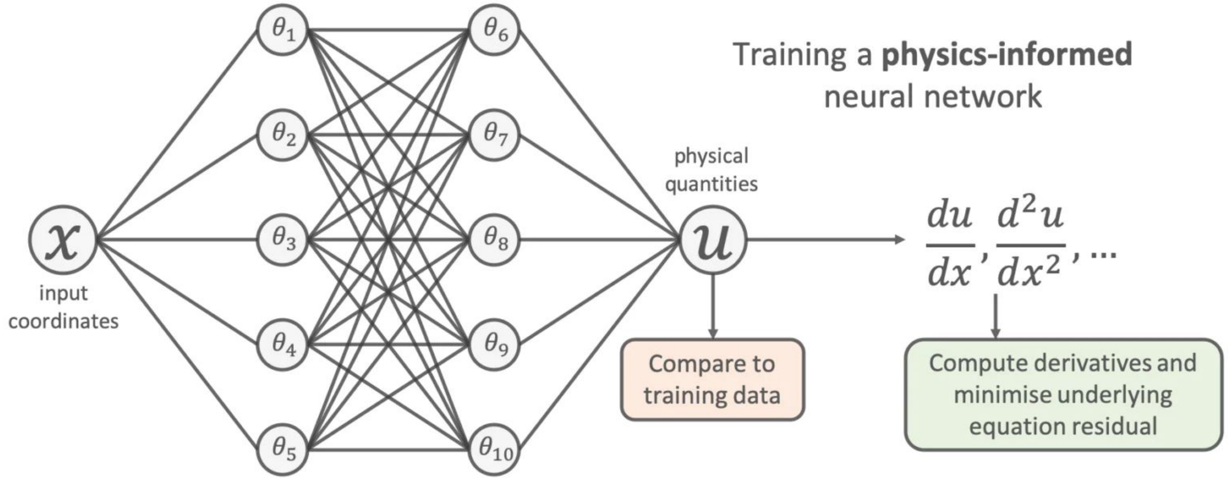
Yiluan Song

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

UNIVERSITY OF MICHIGAN

What are PINNs?

- A Physics-Informed Neural Network (**PINN**) is a type of **deep neural network** (**NN**) that **incorporates physical laws**, typically expressed as partial differential equations (PDEs), directly into the training process.
- Expected result: A perfectly trained neural network that fit the data while respecting physical laws

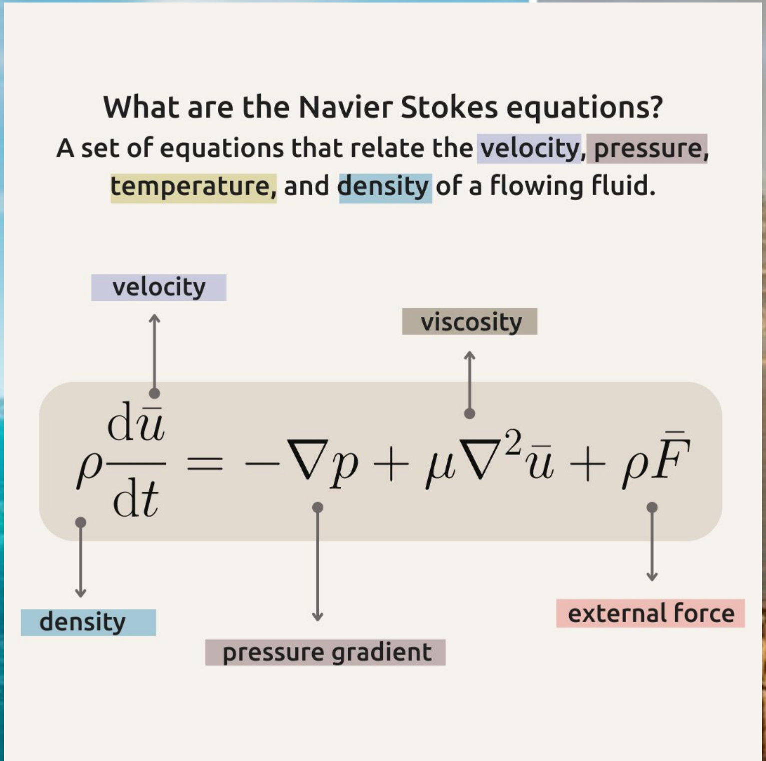


Ben Mosley

Why do we need PINNs?

Why do we need PINNs?

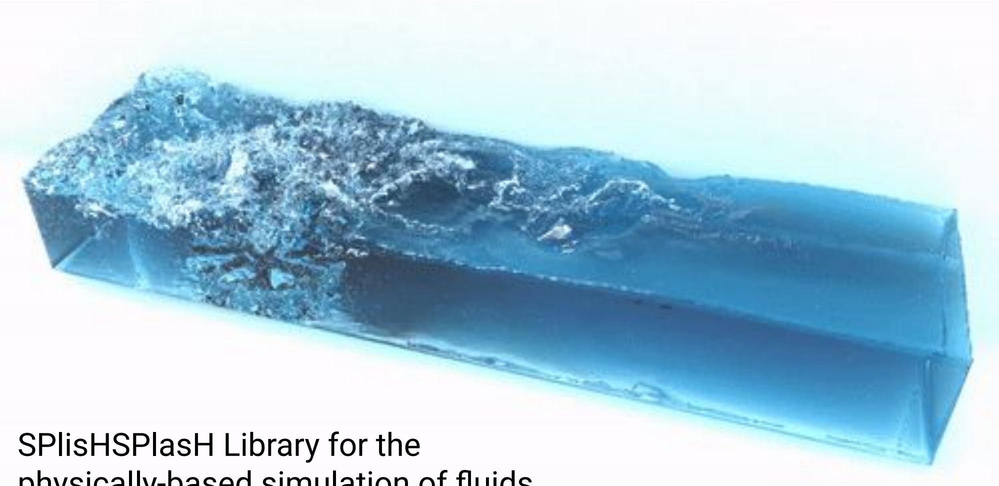
Why do we need PINNs?



- 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.

Challenges of numerical methods & Benefits of PINN

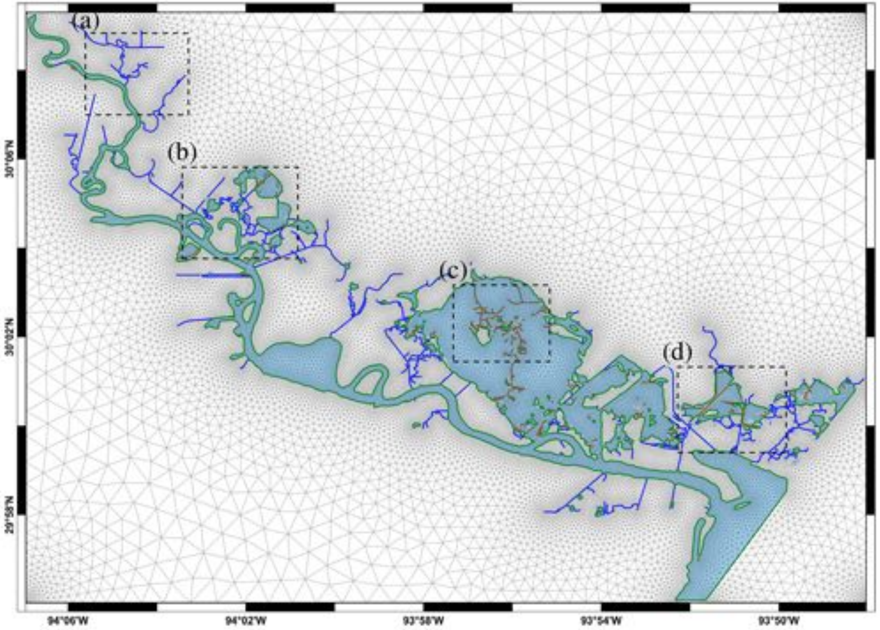
- High-dimensional problems may have impractical runtimes or memory requirements
  - e.g., simulating a 3D object over time
- PINN are often more efficient for solving high-dimensional PDEs



SPlisHSPlasH Library for the physically-based simulation of fluids

Challenges of numerical methods & Benefits of PINN

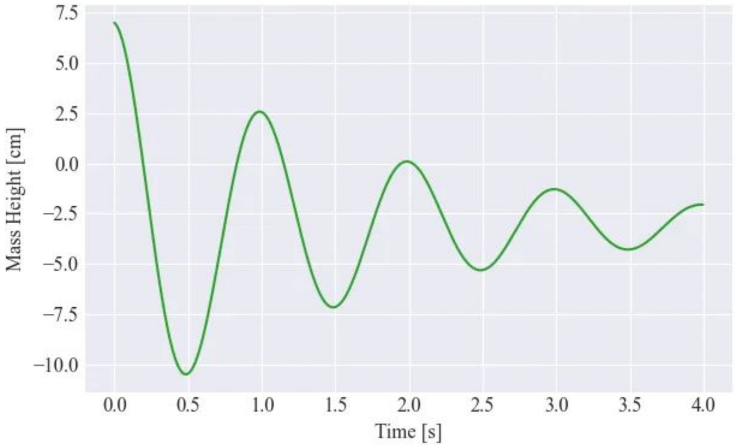
- Mesh generation for numerical methods remains complex
- PINN is mesh-free: no need for mesh/grid generation

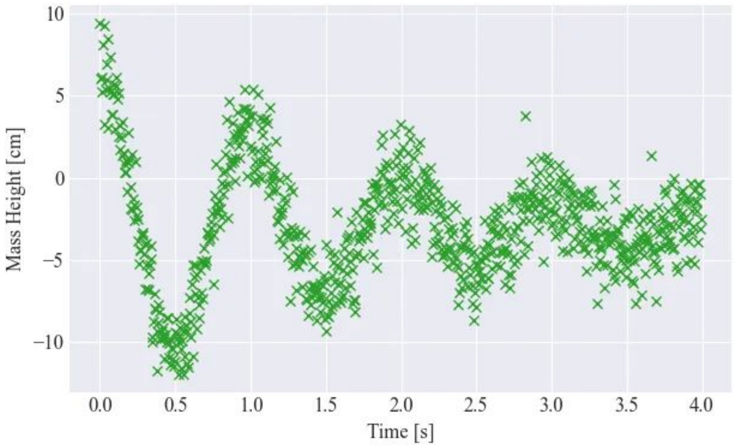


Kang & Kubatko (2024)

Challenges of numerical methods & Benefits of PINN

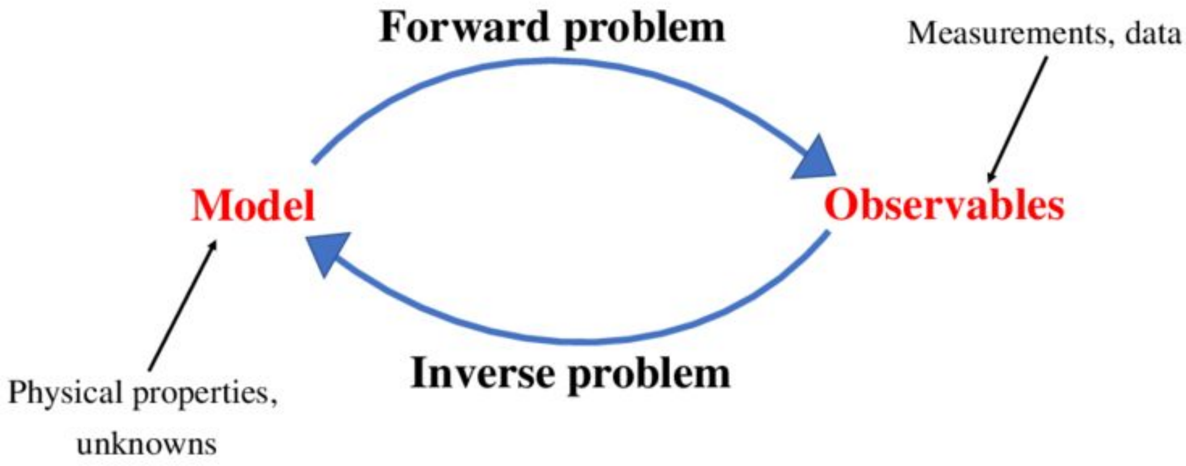
- Numerical method cannot seamlessly incorporate noisy observed data
- PINN can work with sparse or noisy data by relying on known physics
    - NN can be regularized to avoid fitting noise





Challenges of numerical methods & Benefits of PINN

- Solving inverse problems is often prohibitively expensive
- PINN can solve both forward and inverse problems using the same framework
  - No need to repeatedly solve the forward problem

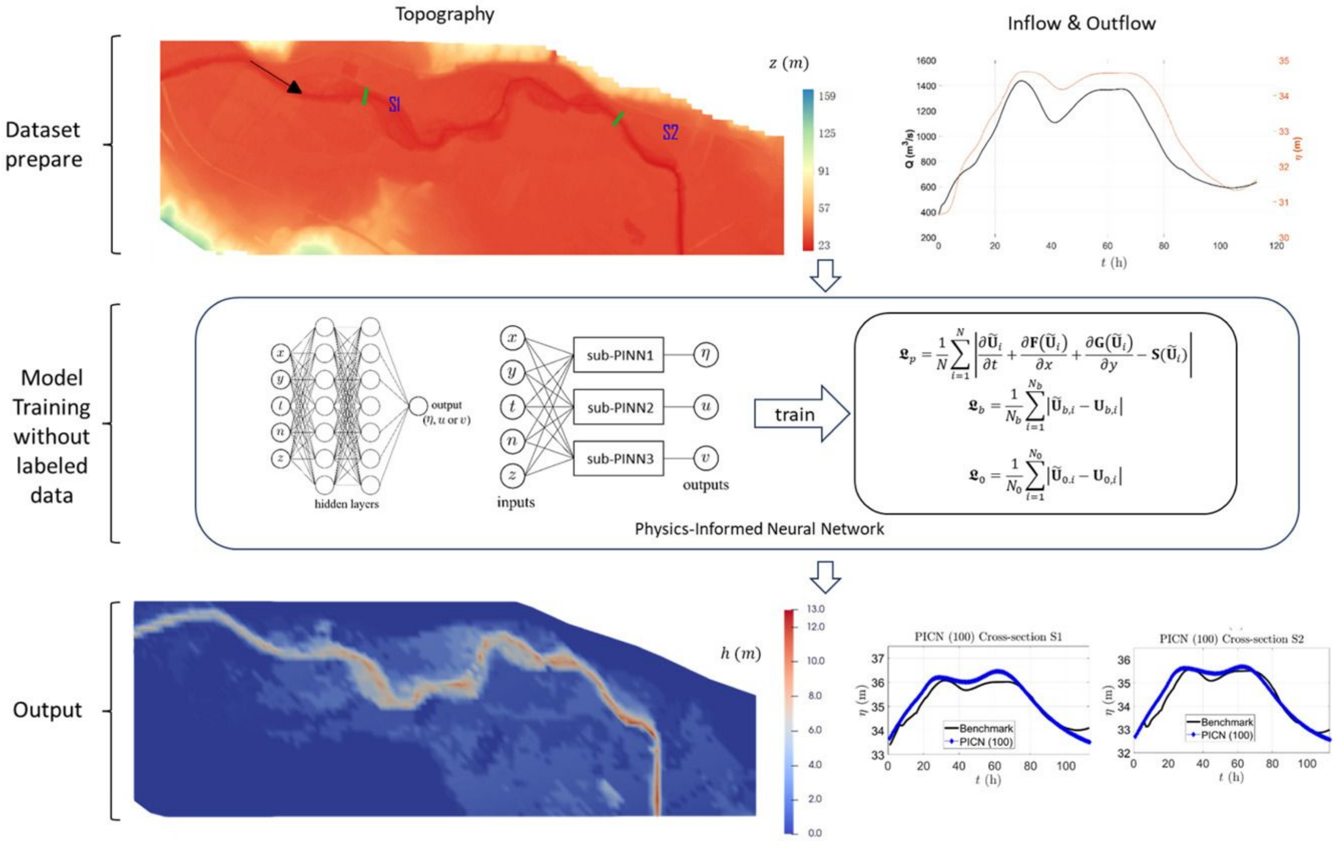


Al-Ibrahim (2021)

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.

Applications of PINNs: Hydrology



River flood simulation

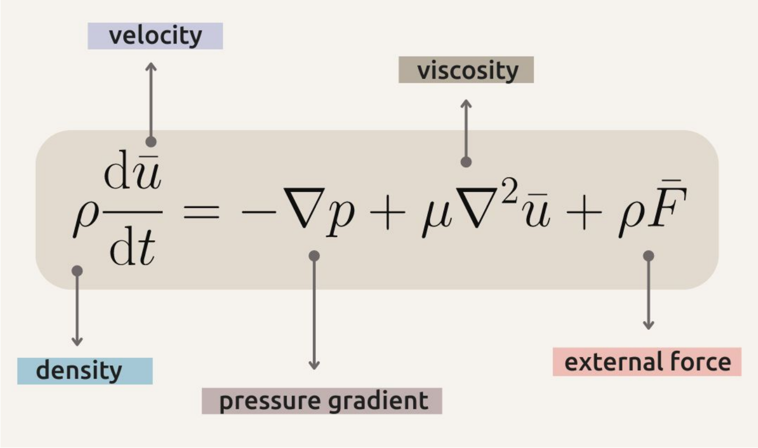
Qi et al. (2024)

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 a function **parameterized** by θ



$$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z} \right)$$

$$\nabla^2 \bar{u} = \begin{pmatrix} \nabla^2 u \\ \nabla^2 v \\ \nabla^2 w \end{pmatrix}$$

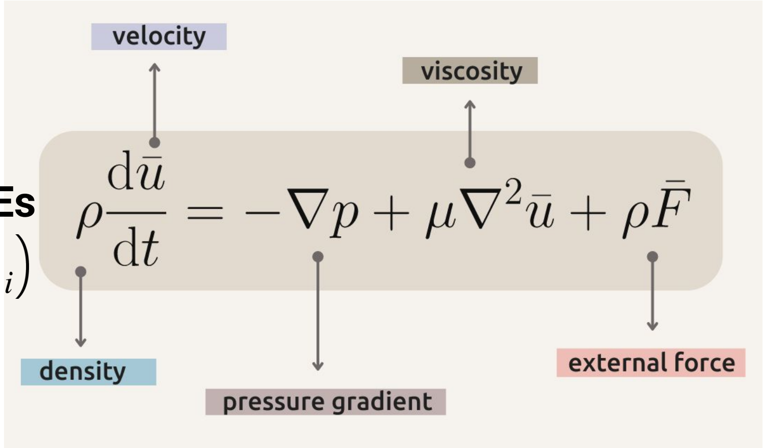
$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

General form of PDE

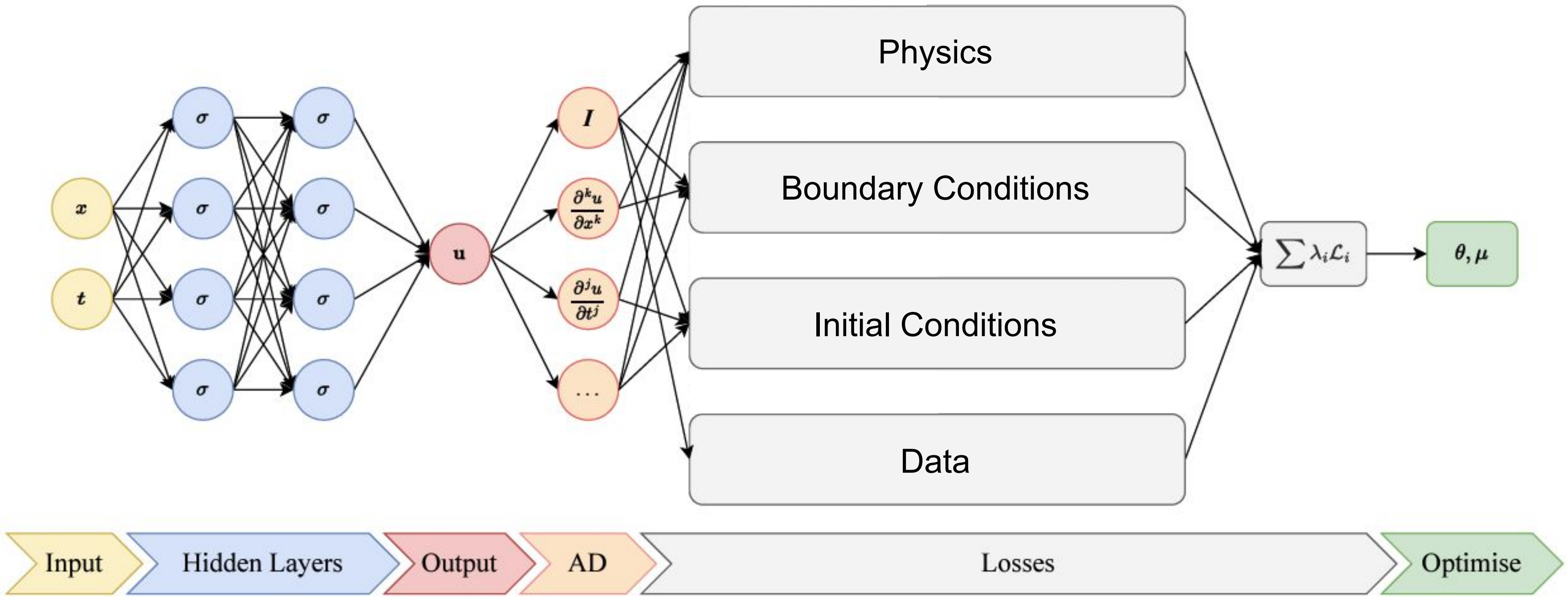
$$\frac{\partial u}{\partial t} = \mathcal{N}[u;\theta]$$

- $u = u(t, x)$ 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 a function **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**

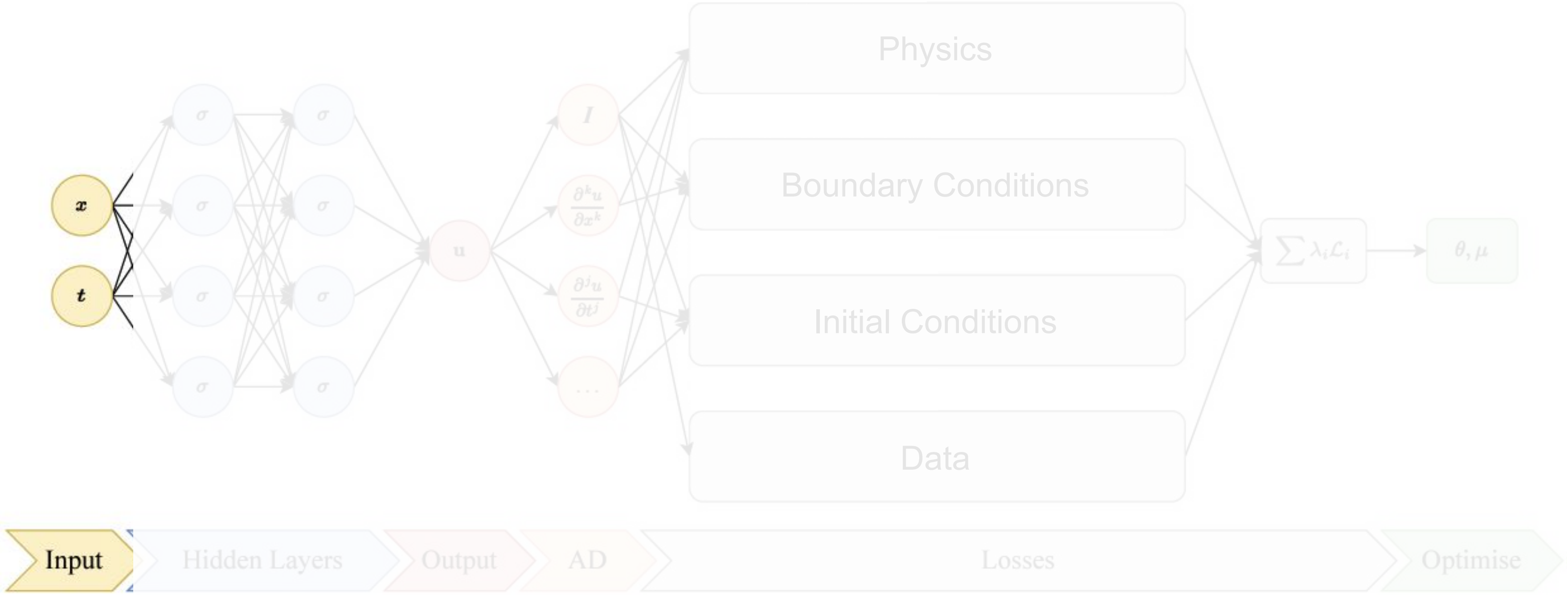


Workflow



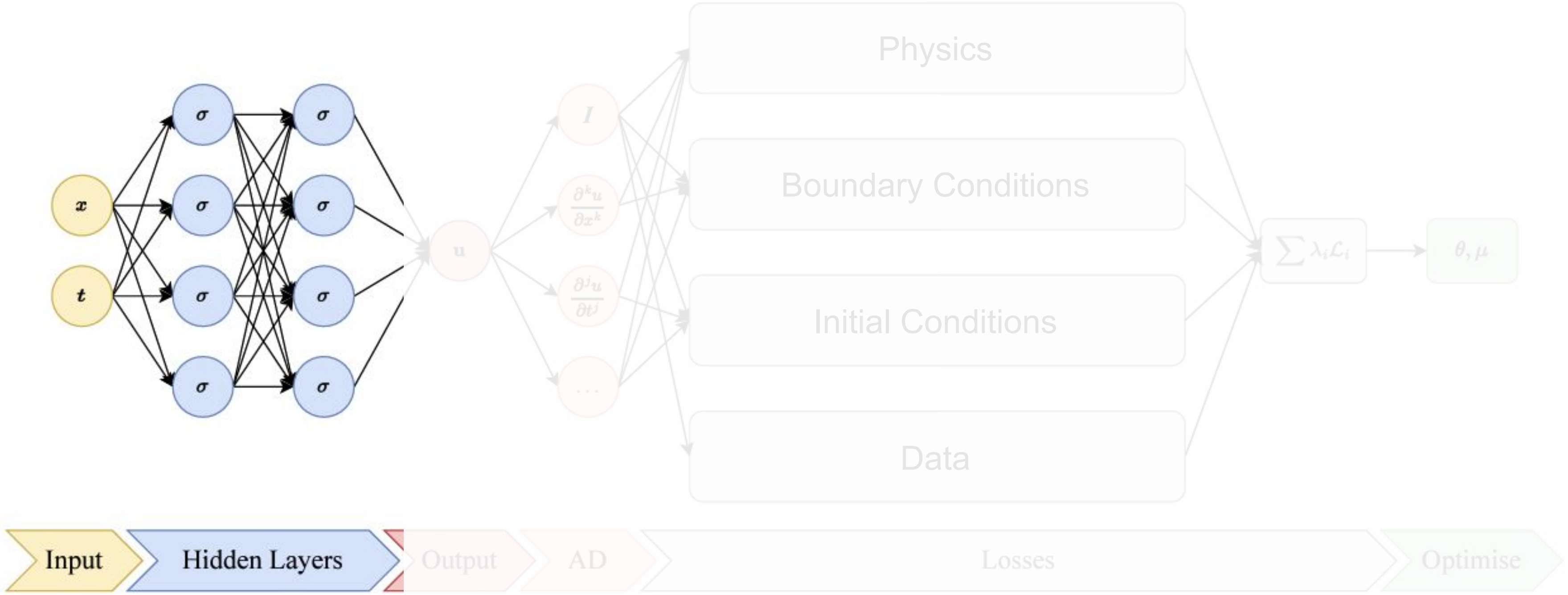
Bischof and Kraus

Workflow



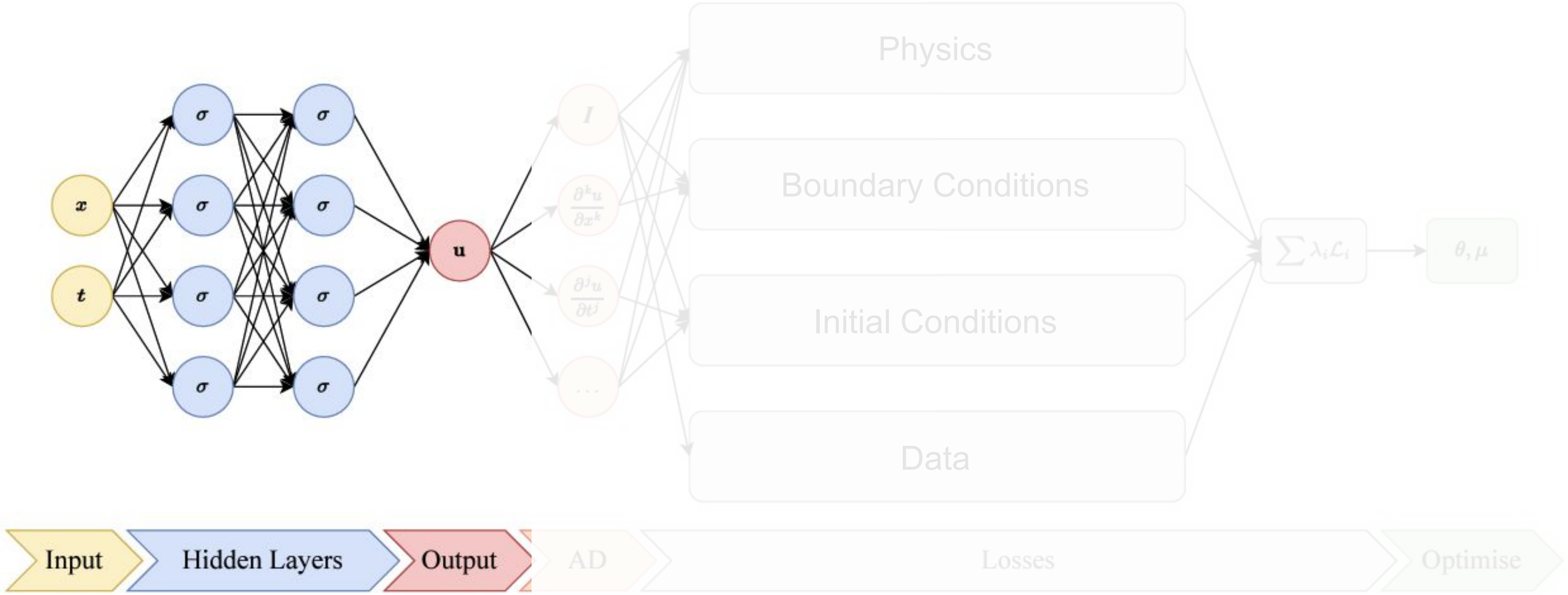
x
t
σ
u
I
$\frac{\partial^k u}{\partial x^k}$
$\frac{\partial^j u}{\partial t^j}$
$\dots$
Physics
Boundary Conditions
Initial Conditions
Data
$\sum \lambda_i \mathcal{L}_i$
θ, μ
Input
Hidden Layers
Output
AD
Losses
Optimise

Workflow



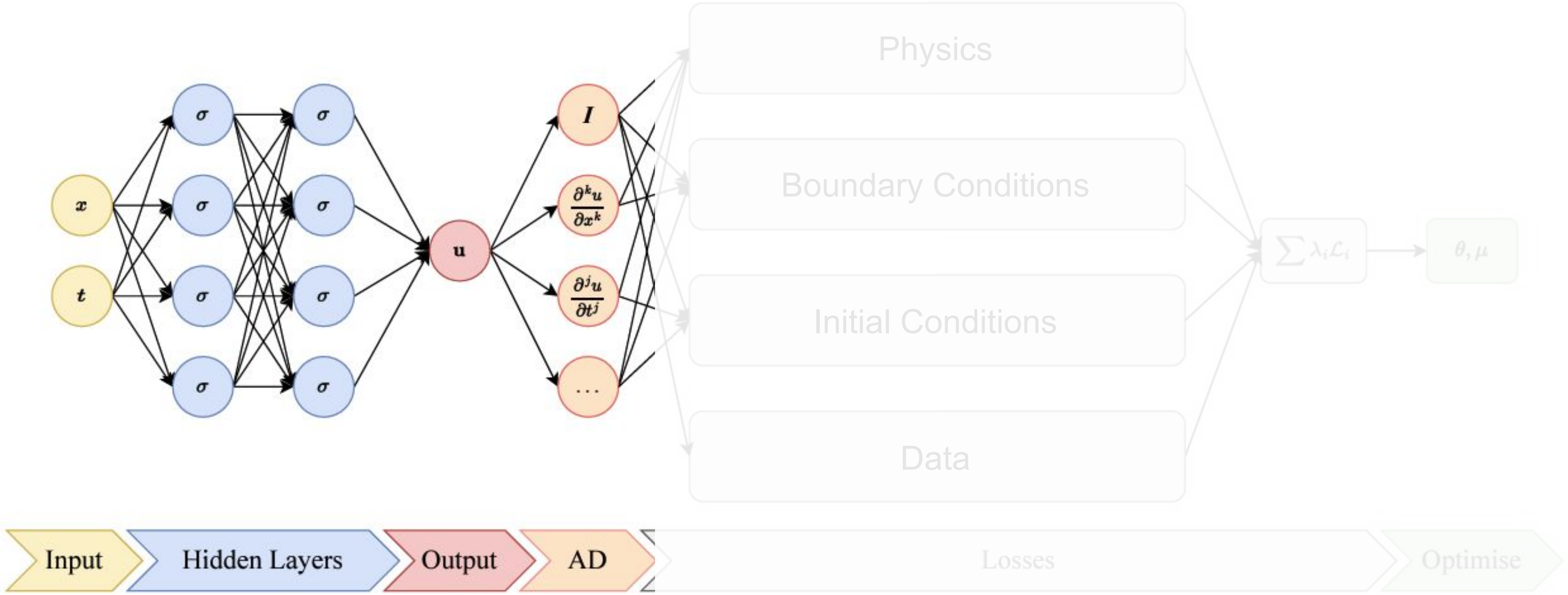
Bischof and Kraus

Workflow



Bischof and Kraus

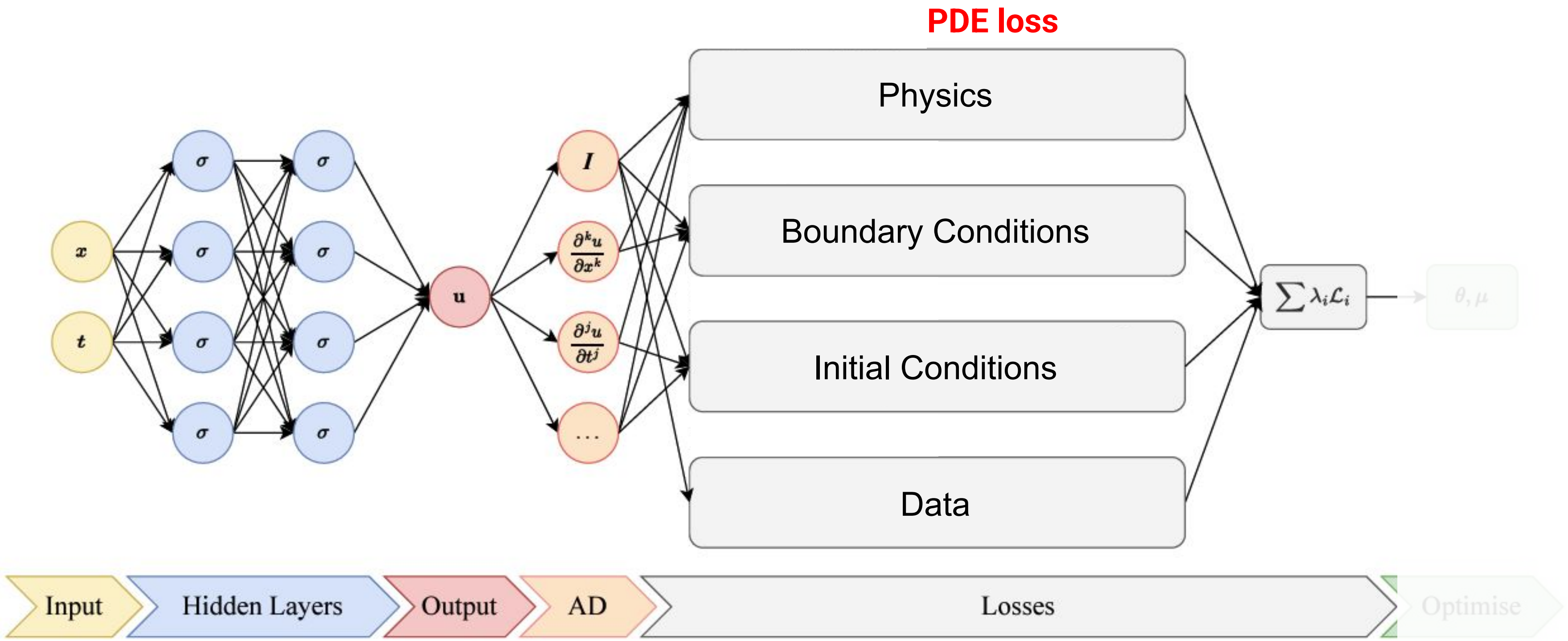
Workflow



Automatic differentiation

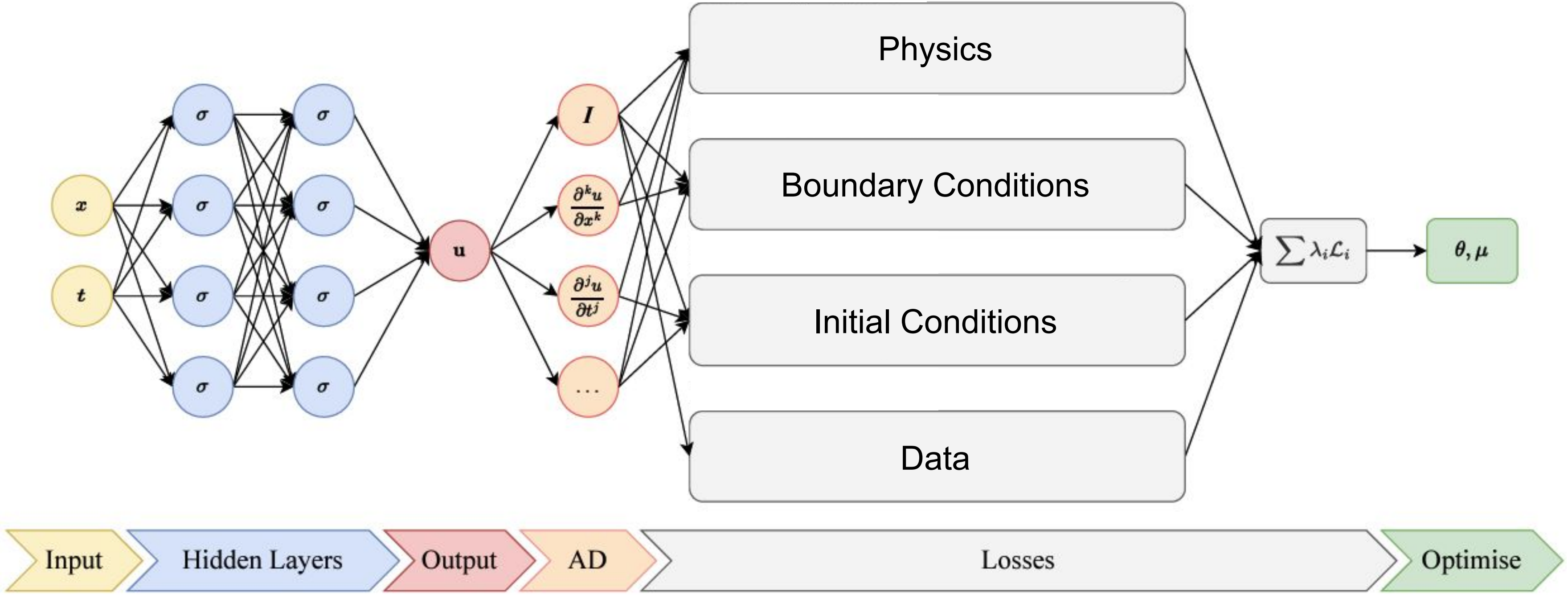
Bischof and Kraus

Workflow



Bischof and Kraus

Workflow



Bischof and Kraus

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

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

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

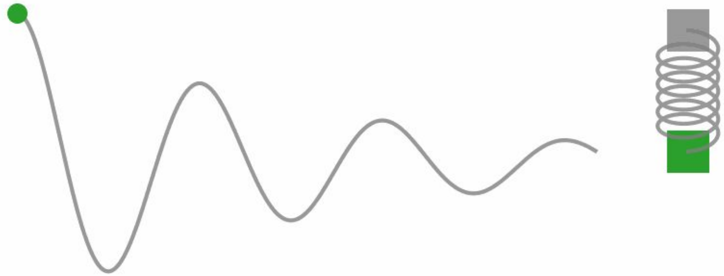
t - time (s) x- displacement from the equilibrium position (m)

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.

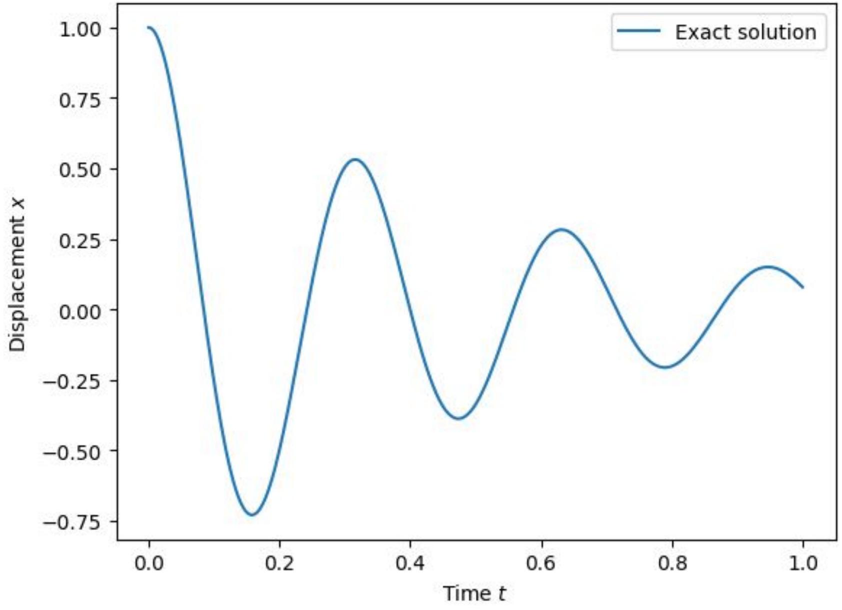


Analytical method

- Analytically, the problem has an exact solution

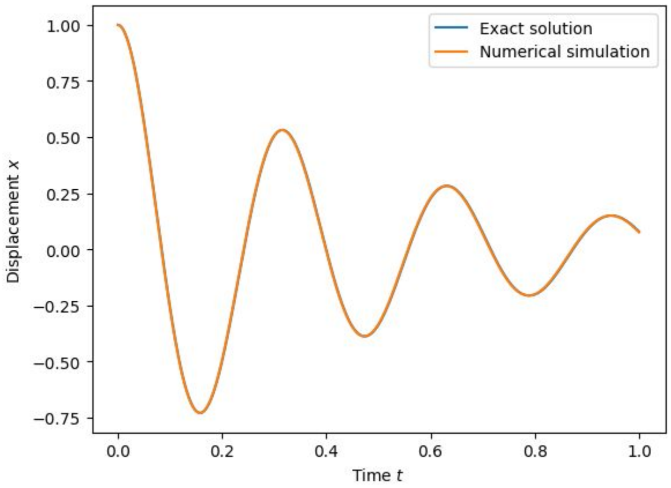
$$x(t) = e^{-\delta t}(2A\cos(\phi + \omega t)) \ , \qquad \text{with } \ \omega = \sqrt{\omega_0^2 - \delta^2}$$

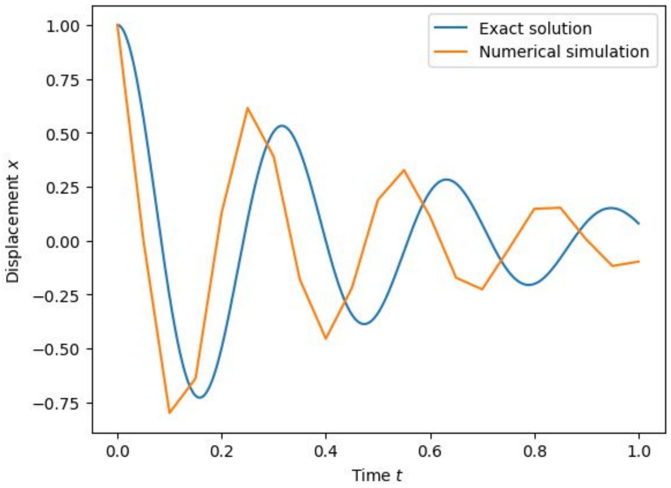
- But not all PDEs have exact solutions



Numerical method

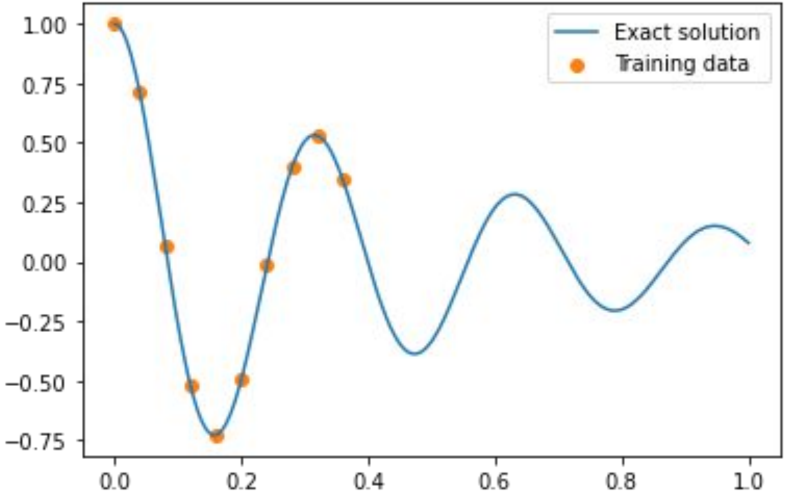
- Numerically, we can simulate changes at each time step
  - Calculate spring force and damping force
  - Calculate acceleration: `a = F/m`
  - Update velocity: `v = v + a*dt`
  - Update position: `x = x + v*dt`





Neural network (NN)

- The **data loss**, if available, measures how well the network fits any provided data points (e.g., experimental or observational data).



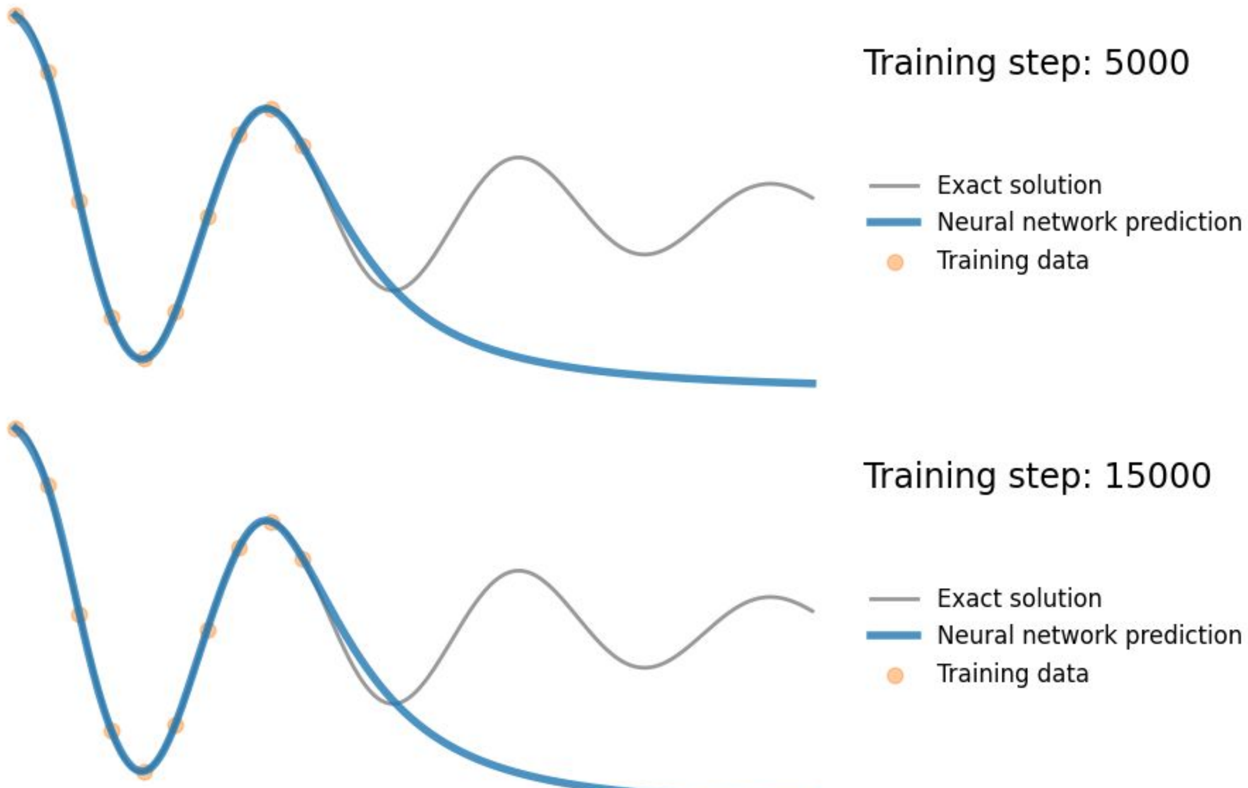
Step 1: Predict the solution at known data points

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

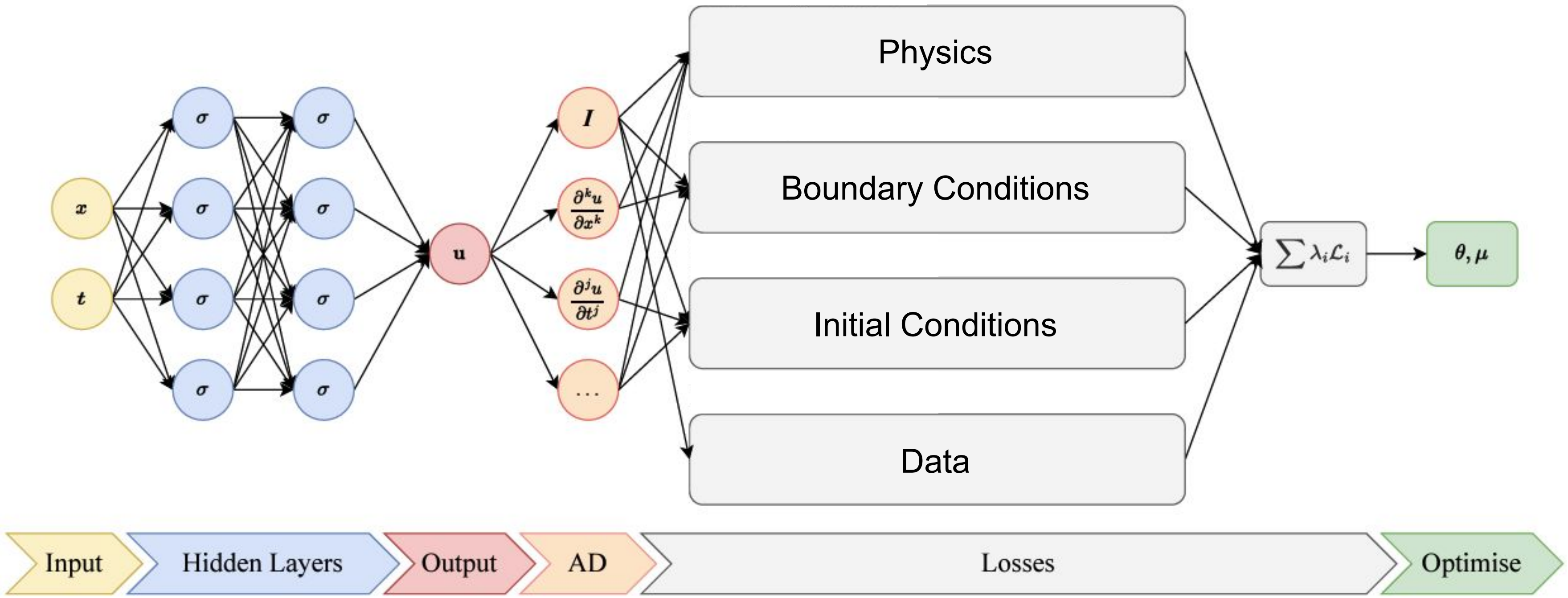
Step 2: Compute data residuals and data loss

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

Performance of NN



PINN



Bischof and Kraus

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

Loss Function: Physics loss

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

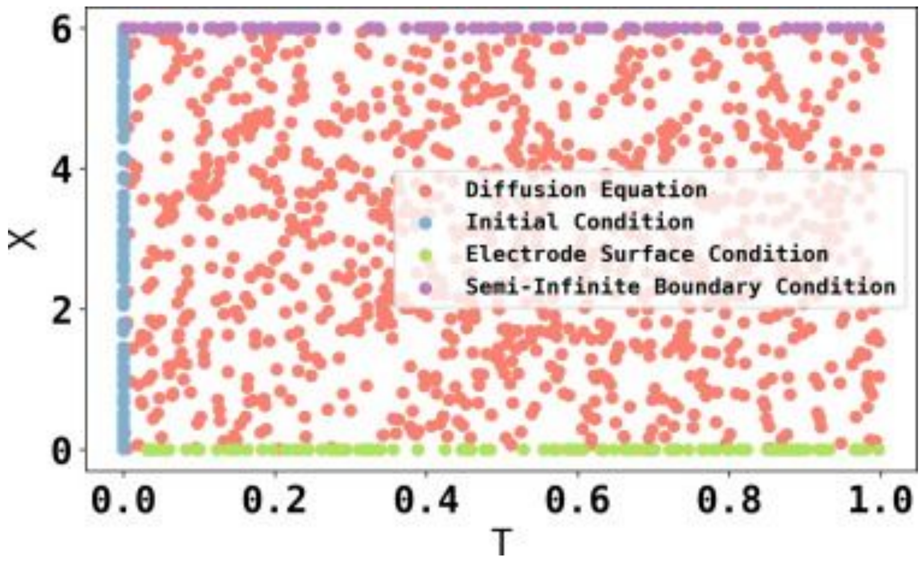
Step 1: Predict the solution from the neural network

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

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

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)
```

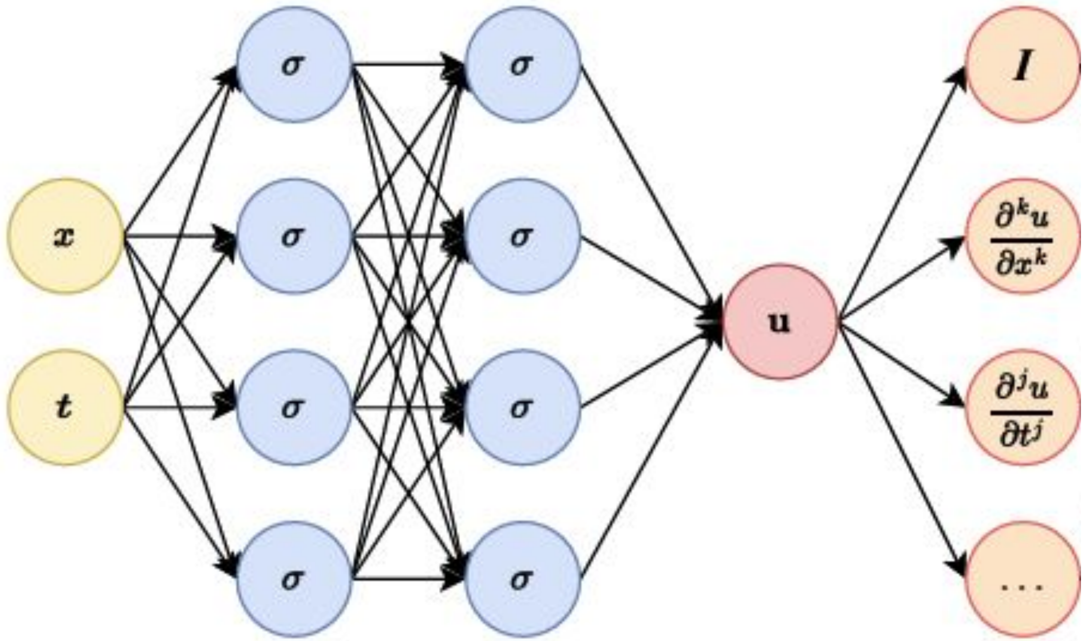
Loss Function: Physics loss

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

Step 2: Compute the derivatives with automatic differentiation

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

Bischof and Kraus

Loss Function: Physics loss

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

Step 2: Compute the derivatives with automatic differentiation

```
# compute dx/dt
dx = torch.autograd.grad(
    outputs=x_pred_physics,
    inputs=t_physics,
    grad_outputs=torch.ones_like(x_pred_physics),
    create_graph=True
)[0]
```

`create_graph=True`
to compute higher-order derivatives

Loss Function: Physics loss

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

Step 2: Compute the derivatives with automatic differentiation

```
# compute d²x/dt²
dx2 = torch.autograd.grad(
    outputs=dx,
    inputs=t_physics,
    grad_outputs=torch.ones_like(dx),
    create_graph=True
)[0]
```

Loss Function: Physics loss

- The **physics loss** ensures the network solution satisfies the governing physical equation (e.g., PDE or ODE).

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

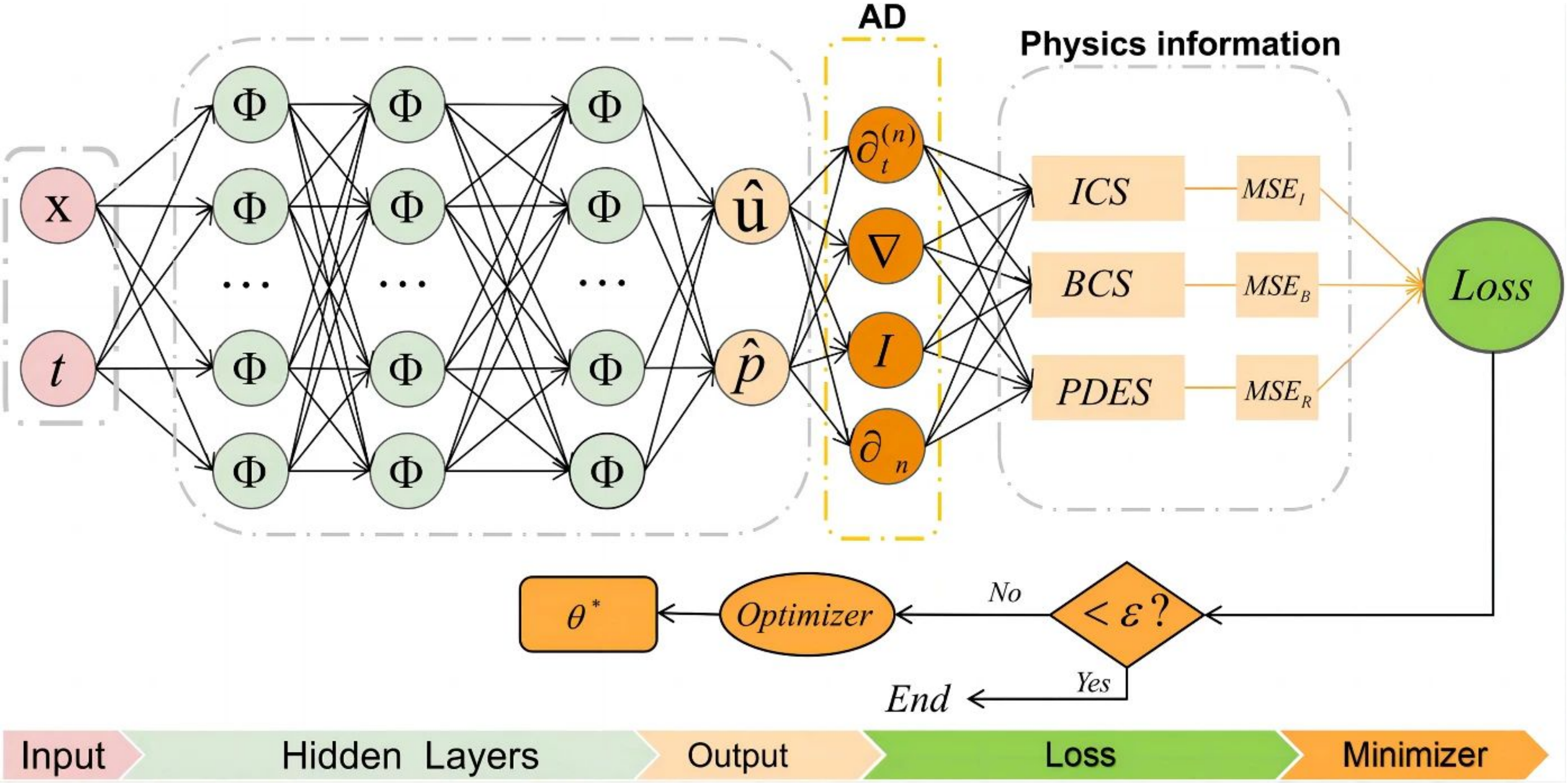
Step 3: Compute the physics residual and physics loss

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

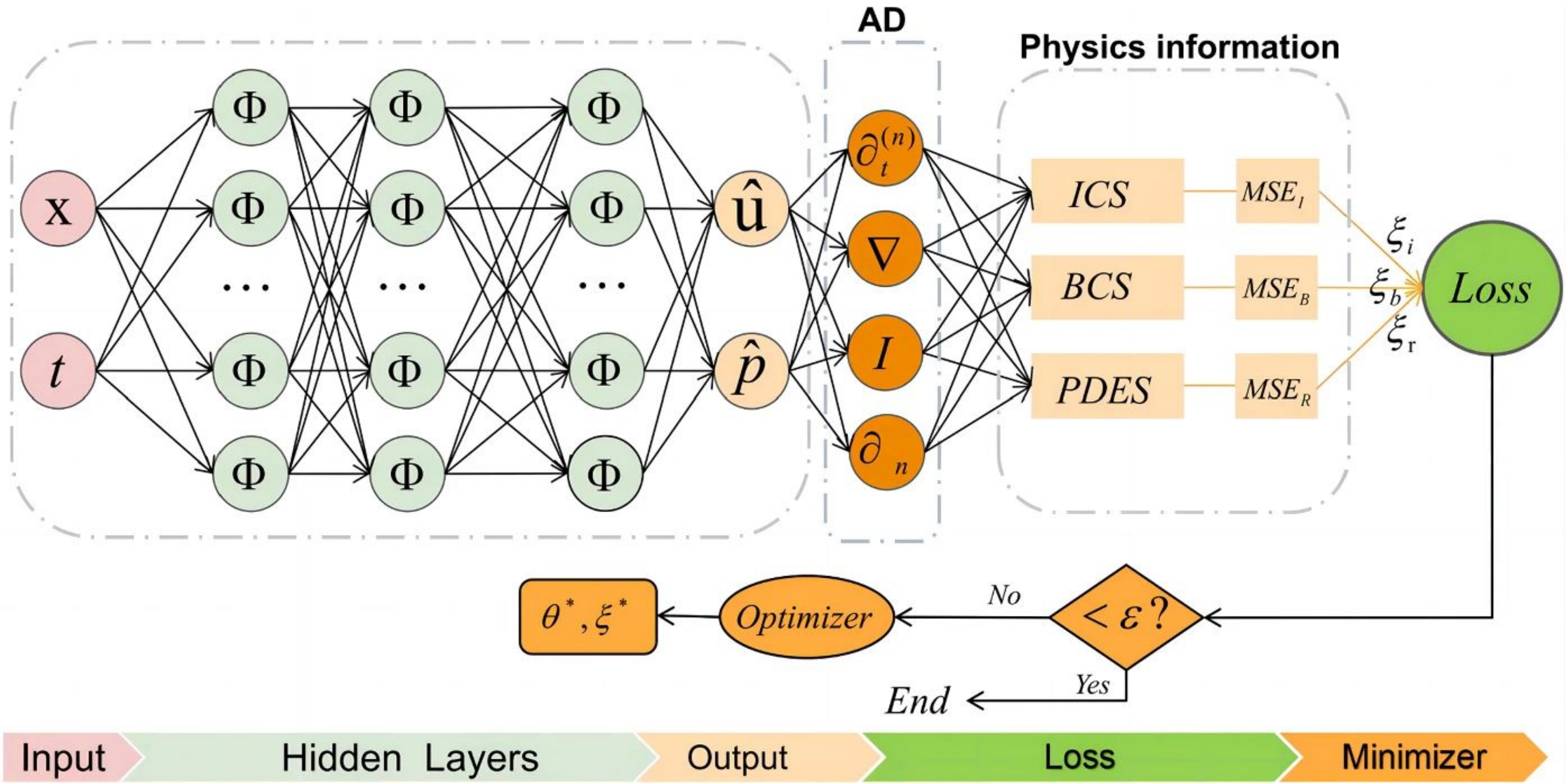
Loss term weighting



The architecture of the standard physics-informed neural networks

Wang et al. (2024)

Loss term weighting



The schematic diagram of the adaptive weighting physics-informed neural networks

Wang et al. (2024)

Loss Function: Initial condition loss

- The **initial condition loss** ensures the network solution satisfies the initial condition at time t_0, if available

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

In this case, the initial conditions are

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

Loss Function: Initial condition loss

- The **initial condition loss** ensures the network solution satisfies the initial condition at time t_0, if available

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

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

- The **initial condition loss** ensures the network solution satisfies the initial condition at time t_0, if available

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

Step 2: Compute the derivatives at initial condition

```
# compute dx/dt at t = 0
dx0 = torch.autograd.grad(
    outputs=x_pred_0,
    inputs=t_0,
    grad_outputs=torch.ones_like(x_pred_0),
    create_graph=True
)[0]
```

Loss Function: Initial condition loss

- The **initial condition loss** ensures the network solution satisfies the initial condition at time t_0, if available

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

Step 3: Compute the initial condition residuals and loss

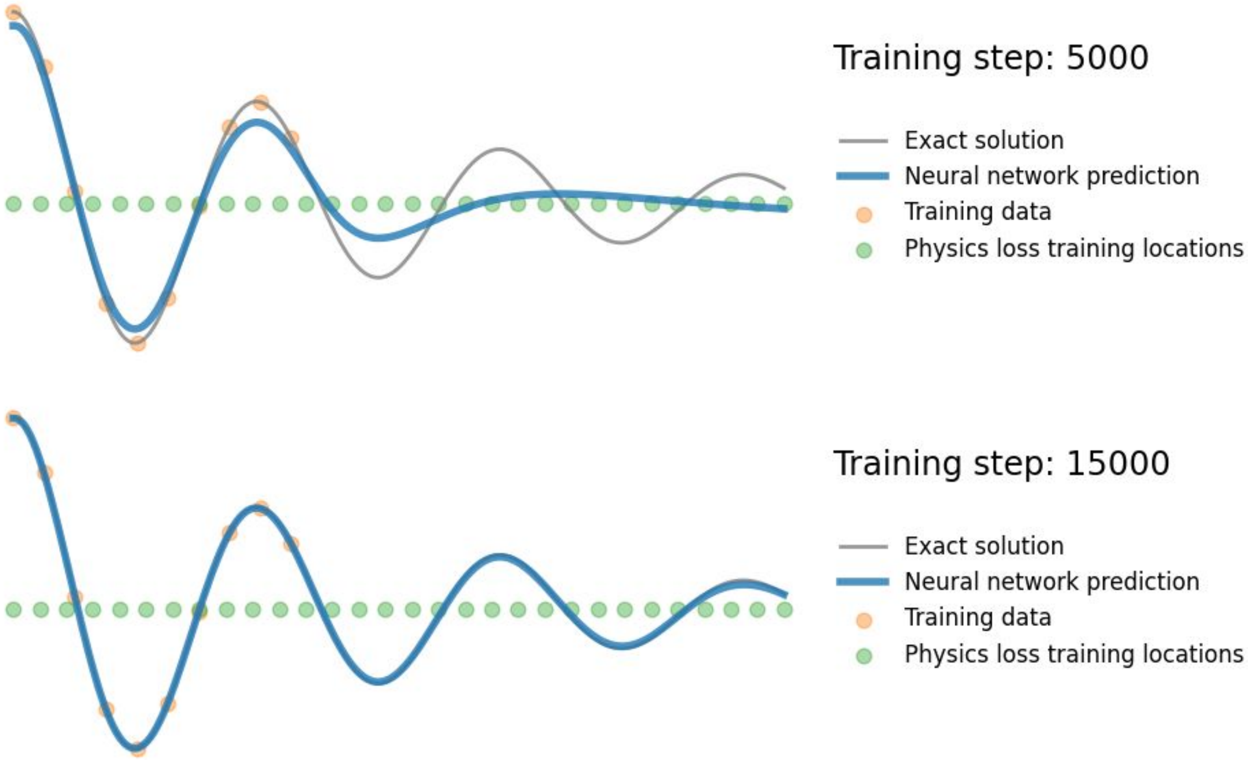
```
ic1 = x_pred_0 - 1
ic2 = dx0

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

Loss Function: Boundary condition loss

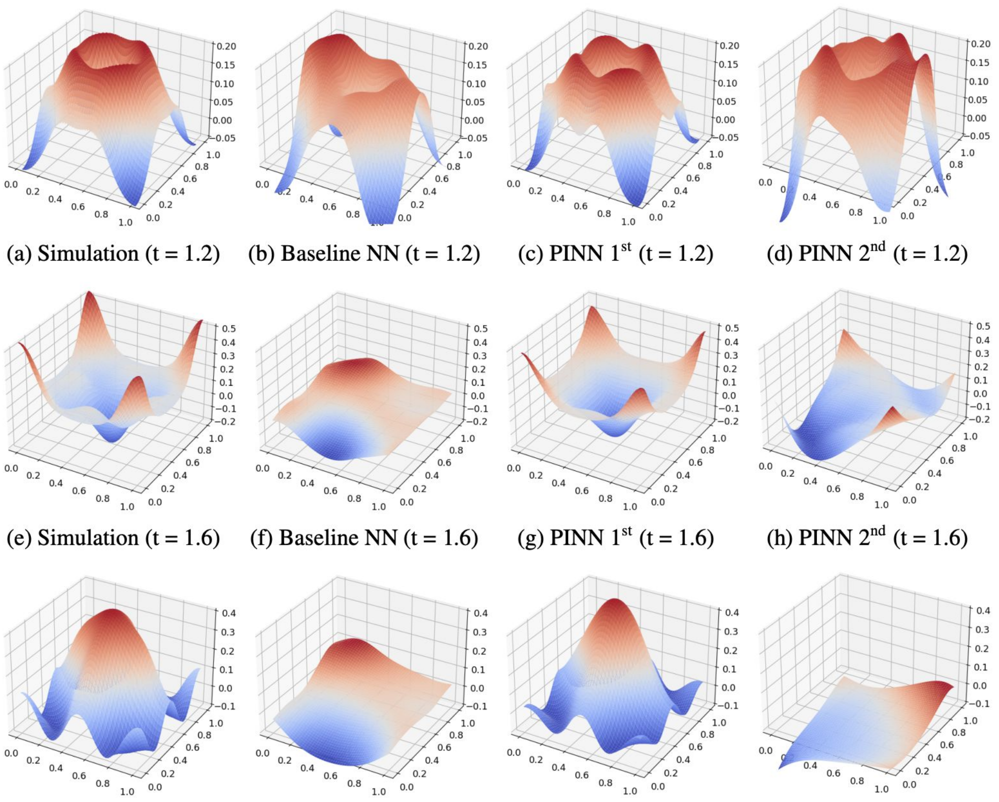
- The **boundary condition loss** enforces the boundary conditions at the spatial or temporal boundaries of the problem domain

- We don't have this term in this case study, but it's 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.

Evaluate performance with analytical solutions and observations



No guarantee in extrapolation



(a) Simulation (t = 1.2) (b) Baseline NN (t = 1.2) (c) PINN 1st (t = 1.2) (d) PINN 2nd (t = 1.2)

(e) Simulation (t = 1.6) (f) Baseline NN (t = 1.6) (g) PINN 1st (t = 1.6) (h) PINN 2nd (t = 1.6)

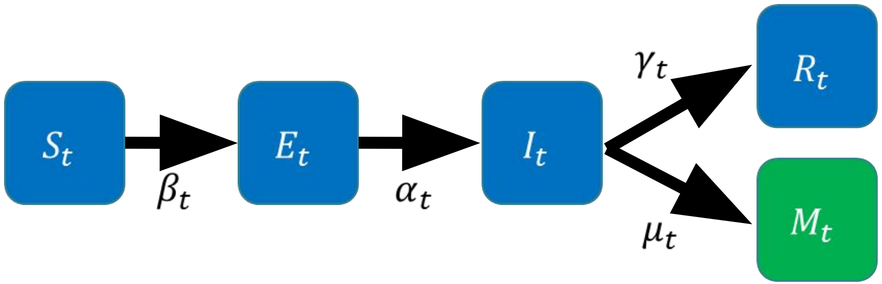
(i) Simulation (t = 2.0) (j) Baseline NN (t = 2.0) (k) PINN 1st (t = 2.0) (l) PINN 2nd (t = 2.0)

Davini et al. (2021)

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

[Rodríguez et al. AAAI 2023]

- Latent/unobserved ODE states
- Observed ODE states



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

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

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

where $\boldsymbol{s}_t = [S_t, E_t, I_t, R_t, M_t]^T \rightarrow$ ODE states — Unknown, except M_t
and $\Omega_t = [\beta_t, \alpha_t, \gamma_t, \mu_t]^T \rightarrow$ ODE parameters — Unknown

Workflow

1. Compute $\boldsymbol{s}_t$ via forward pass over neural network

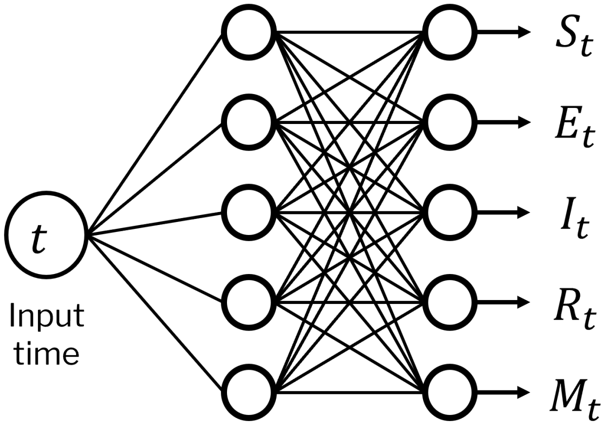
2. Compute $\frac{d\boldsymbol{s}_t}{dt}$ via automatic differentiation

3. Compute $f_{ODE}(\boldsymbol{s}_t, \boldsymbol{\Omega}_t)$ using NN outputs and ODE

4. Minimize residual of steps 2 and 3, and include ground truth data

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

5. Update both NN parameters and $\boldsymbol{\Omega}_t$

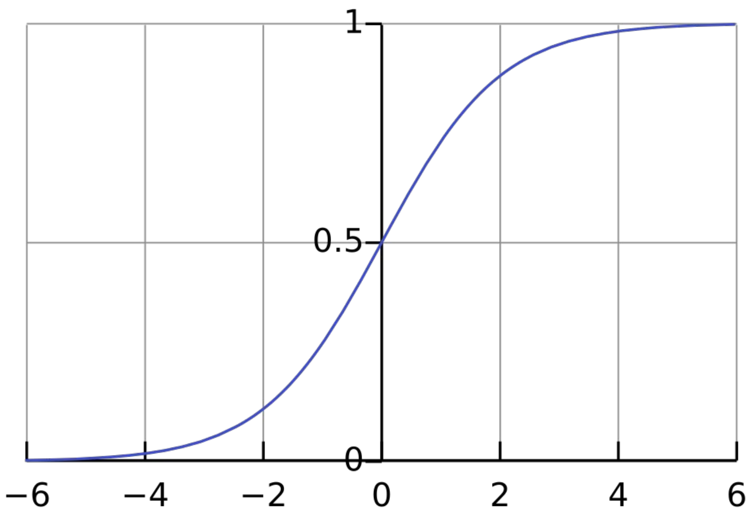


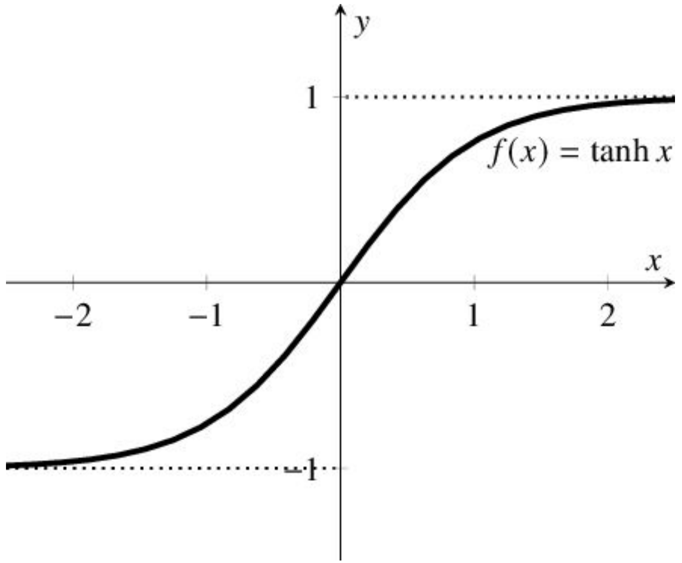
The methods do not change, we just need to make parameters learnable

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

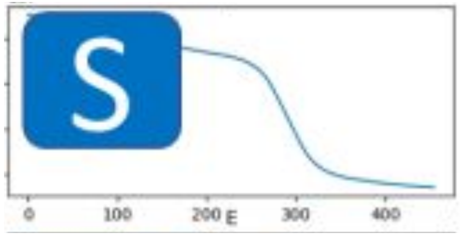
```
constrained_param = torch.sigmoid(self.raw_param)
```

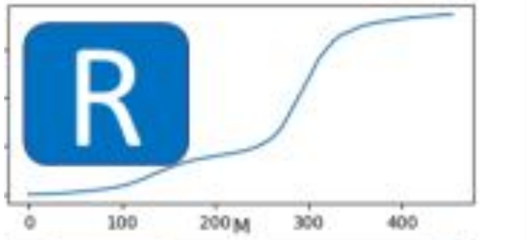




Constraining solution space w/ domain knowledge: monotonicity

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





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

Constraining solution space w/ domain knowledge: smoothness

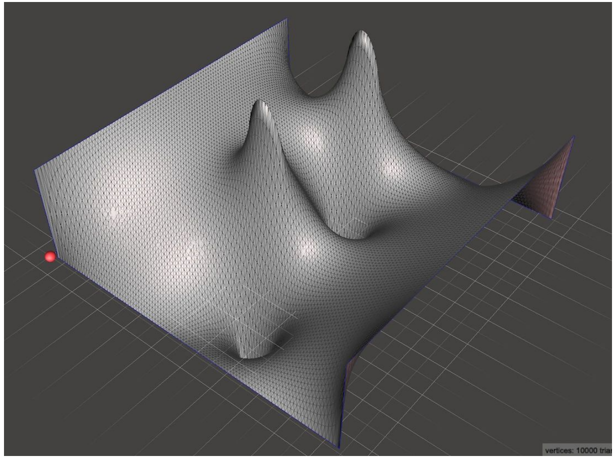
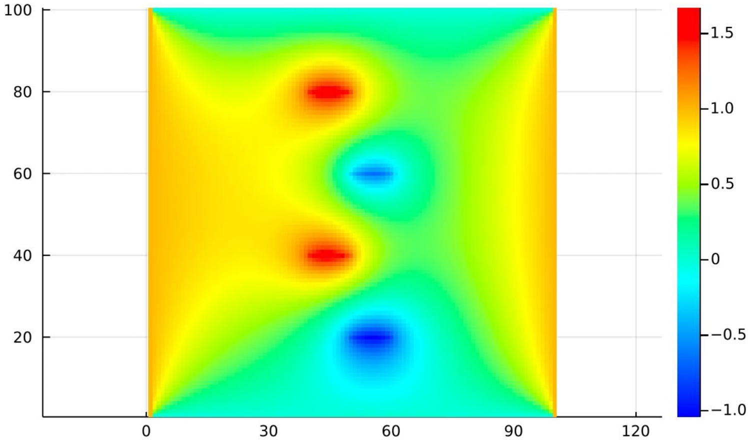
- Time-varying ODE parameters should change smoothly
  - Because **daily** epidemic dynamics change gradually
  - Thus, we should remove solutions for Ω_t that change abruptly
- Include a loss to incorporate this

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

Limitations, solutions, recent advancements

Limitation 1: Gradient imbalance

- Consider solving a Poisson equation (difference equation)

Matt Ferraro

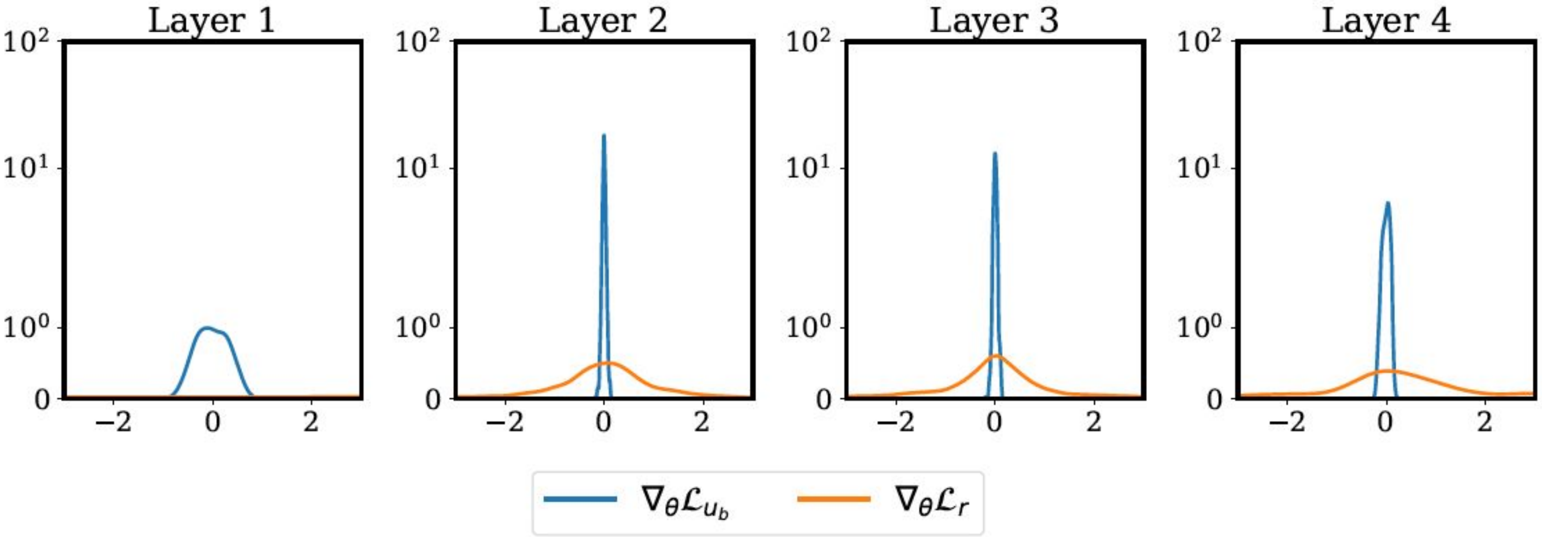
- We can obtain data for multiple values of C (a larger C means faster source-sink alteration), and for each we will train PINN of 4 layers using the following loss

$$\mathcal{L}(\theta) = \underbrace{\mathcal{L}_r(\theta)}_{\text{PDE loss}} + \underbrace{\mathcal{L}_{u_b}(\theta)}_{\text{BC loss}}$$

Wang et al., *SIAM J. Sci. Comput.* (2021)

Limitation 1: Gradient imbalance

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



Gradient of BC loss   Gradient of PDE loss

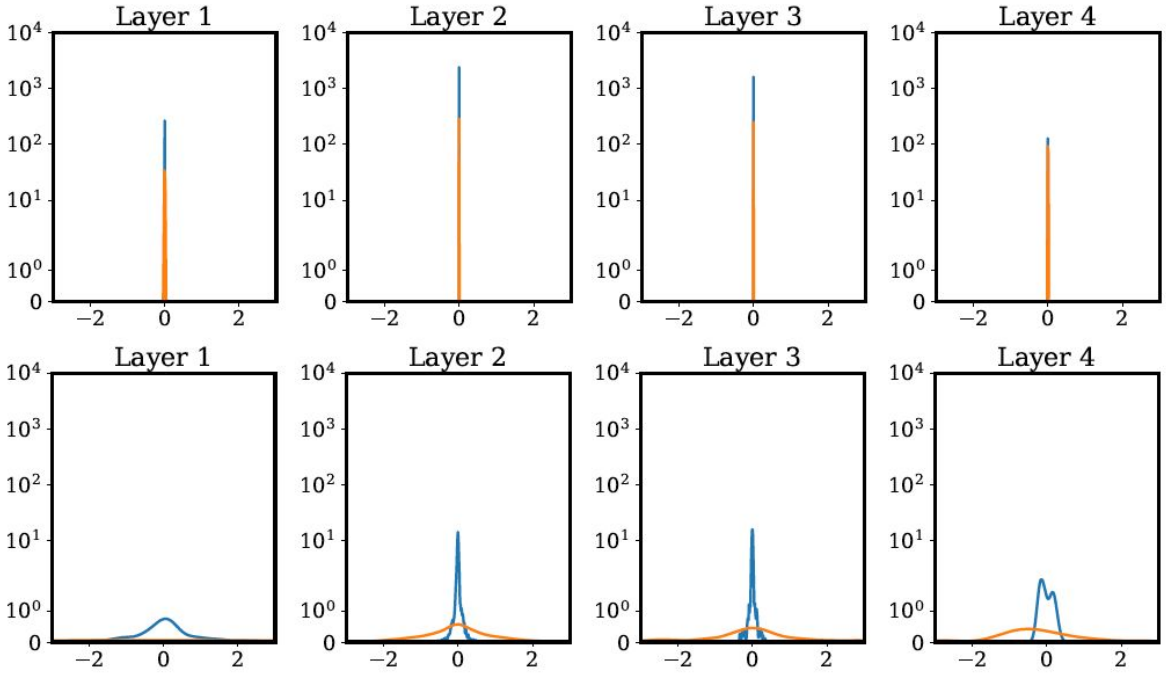
Wang et al., *SIAM J. Sci. Comput.* (2021)

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	$1.34e-04$	$4.65e-04$	$3.82e-03$	$1.30e-02$	$9.44e+00$

Wang et al., *SIAM J. Sci. Comput.* (2021)

Solution to limitation 1: Adaptive loss weights

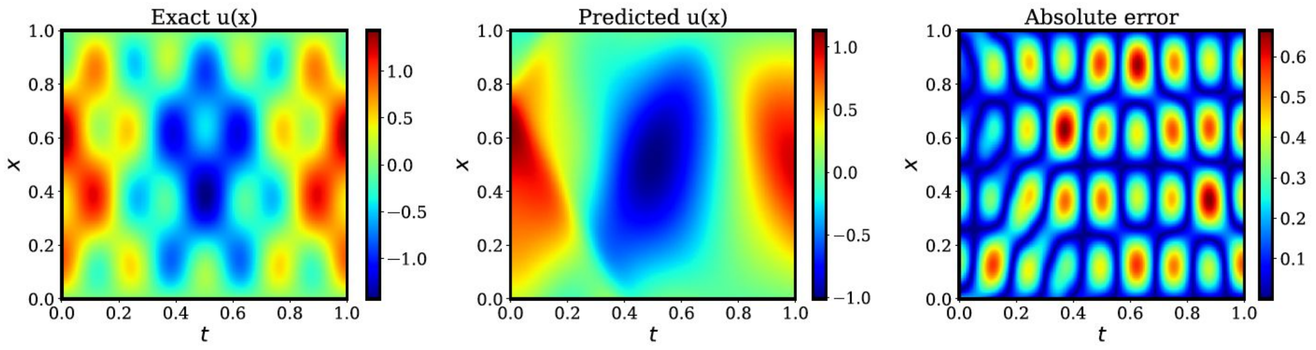
- Learning rate annealing algorithm using gradient statistics (Wang et al., SIAM J. Sci. Comput. 2021)
  - Intent: For any given loss term $\mathscr{L}_i$ (BC, IC, or data), find λ_i(weight) such that

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

  - To assign appropriate weight to each term in the loss function such that their gradients during back-propagation are similar in magnitude
  - Done at every iteration of the gradient descent loop or at a specified frequency
- We can also make the norm of gradients of each weighted loss equal to each other (Wang et al., *arXiv* 2023)

Limitation 2: Spectral bias

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

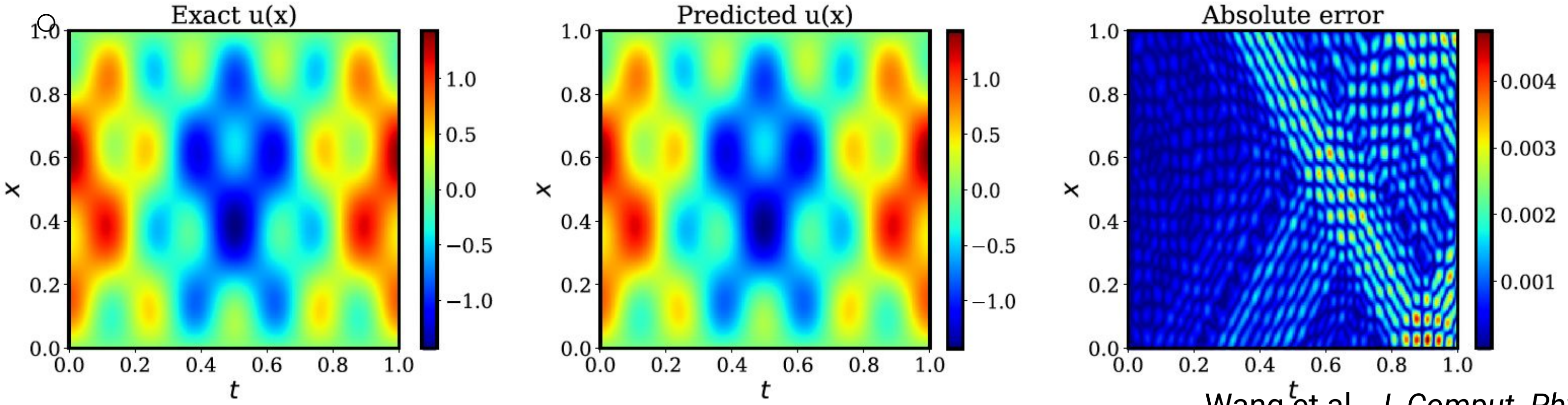


Misses several high-frequencies

Wang et al., *J. Comput. Phys.* (2022)

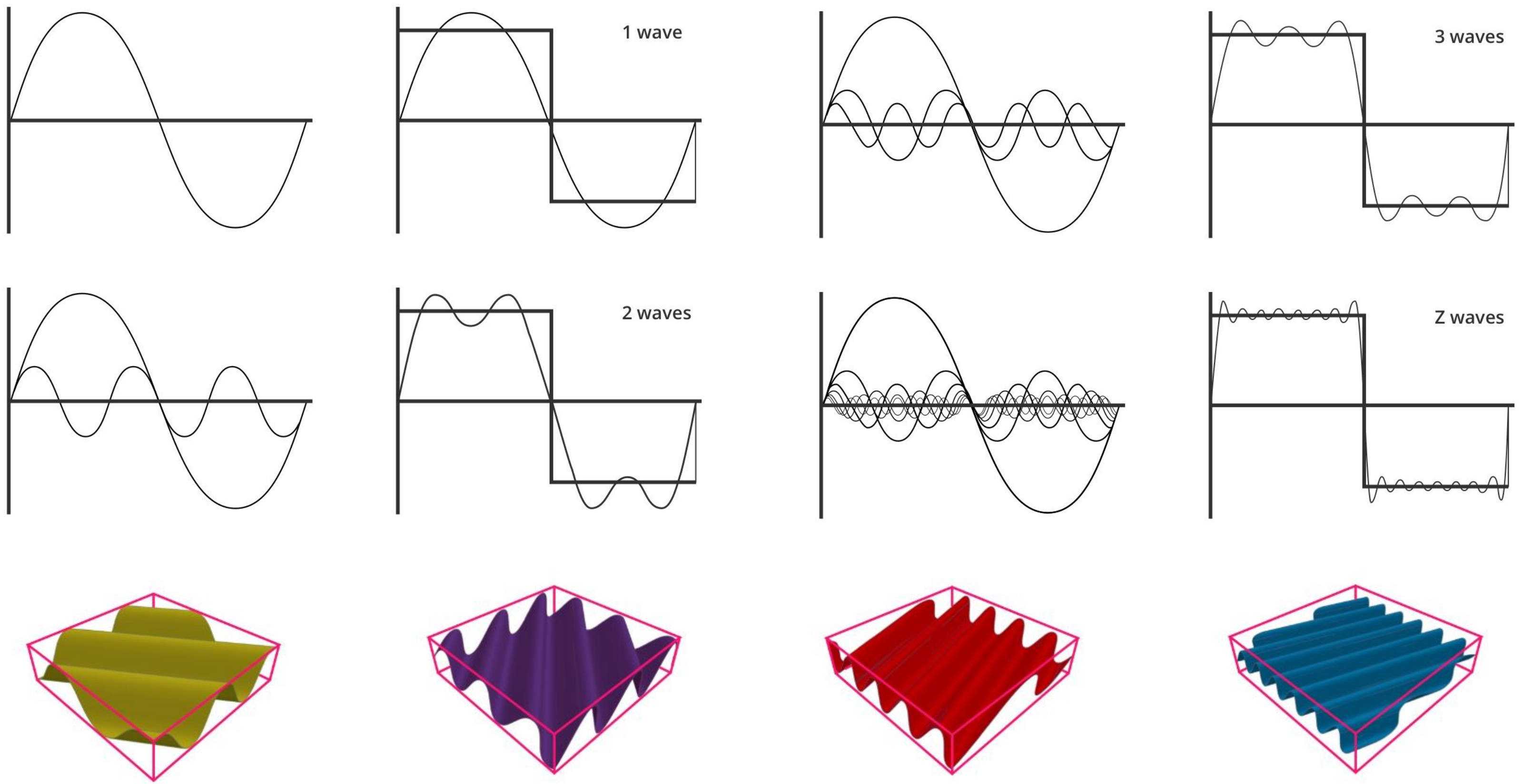
Solution to limitation 2: Adaptive learning rates

- 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
  -



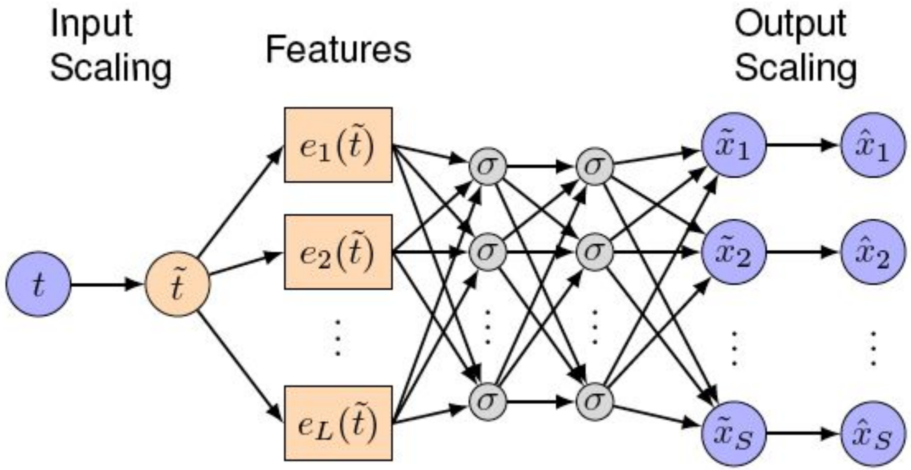
Wang et al., *J. Comput. Phys.* (2022)

Solution to limitation 2: Enriching frequencies via random Fourier features



Solution to limitation 3: Scaling and feature expansion

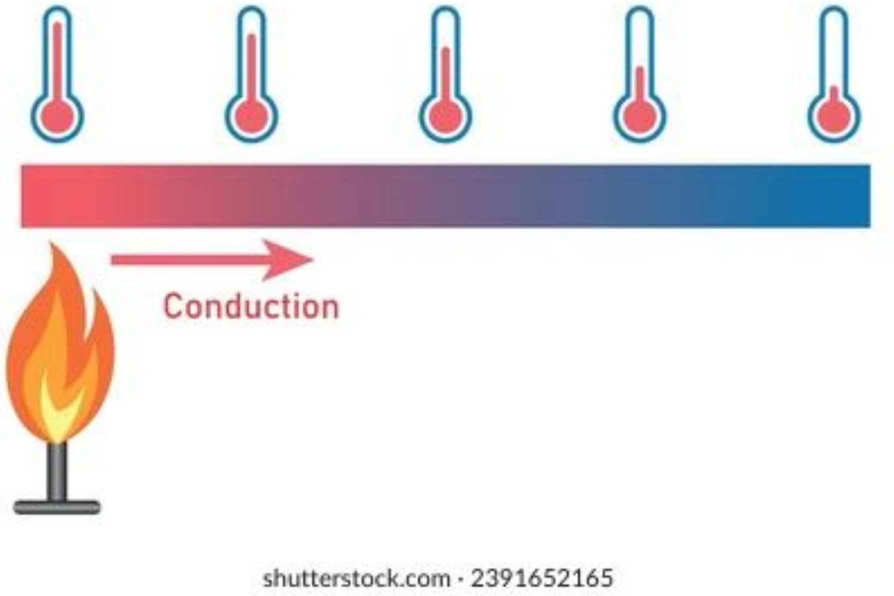
- Scaling: Normalize layer for inputs and outputs
- Feature expansion: add patterns relevant to the system, such as
  - Periodicity $\sin(kt)$
  - Fast decay e^{-kt}
  - Multiscale features



Yazdani et al. *PLoS Comp. Bio.* (2020)

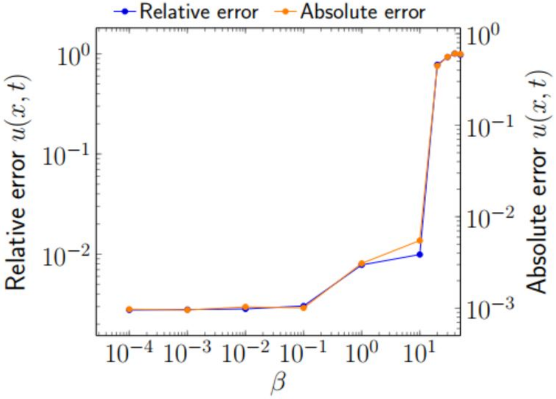
Limitation 3: Propagation failures

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



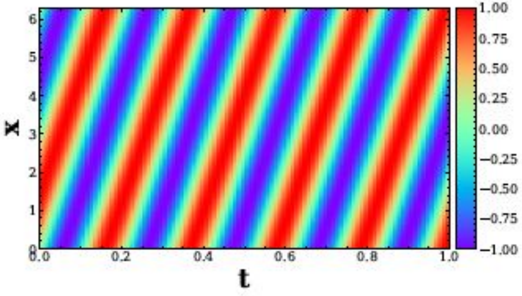
Limitation 3: Propagation failures

- 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.

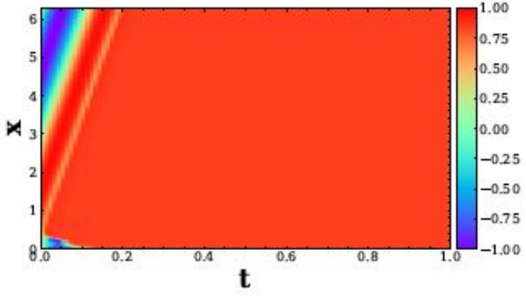


(a) *Error for different* β

- It becomes harder to train with larger β



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

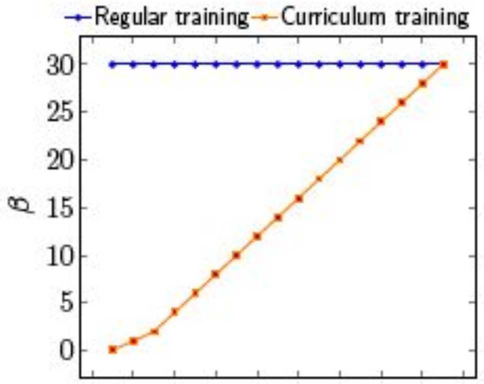


(c) *PINN solution for* $\beta = 30$

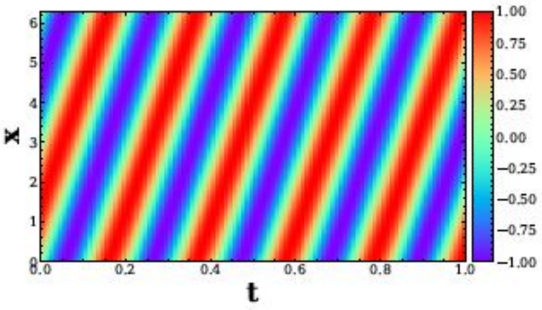
Krishnapriyan et al. *NeurIPS* (2021)

Solution to limitation 3: Curriculum regularization

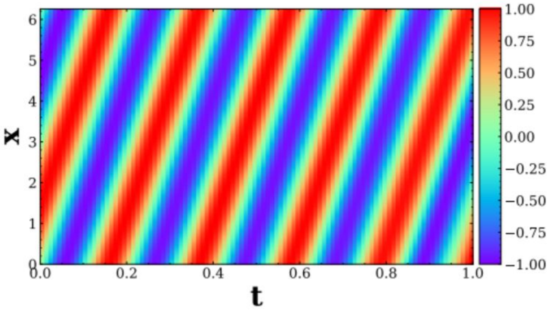
- Decompose the optimization task into a sequence of more manageable sub-tasks
- In this case, we warm start the NN training with easier instances of the PDE
- Objective: good initialization of NN weights



(a) *Curriculum regularization schematic*



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



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

Krishnapriyan et al. *NeurIPS* (2021)

Limitation 4: Inadequate inductive bias

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

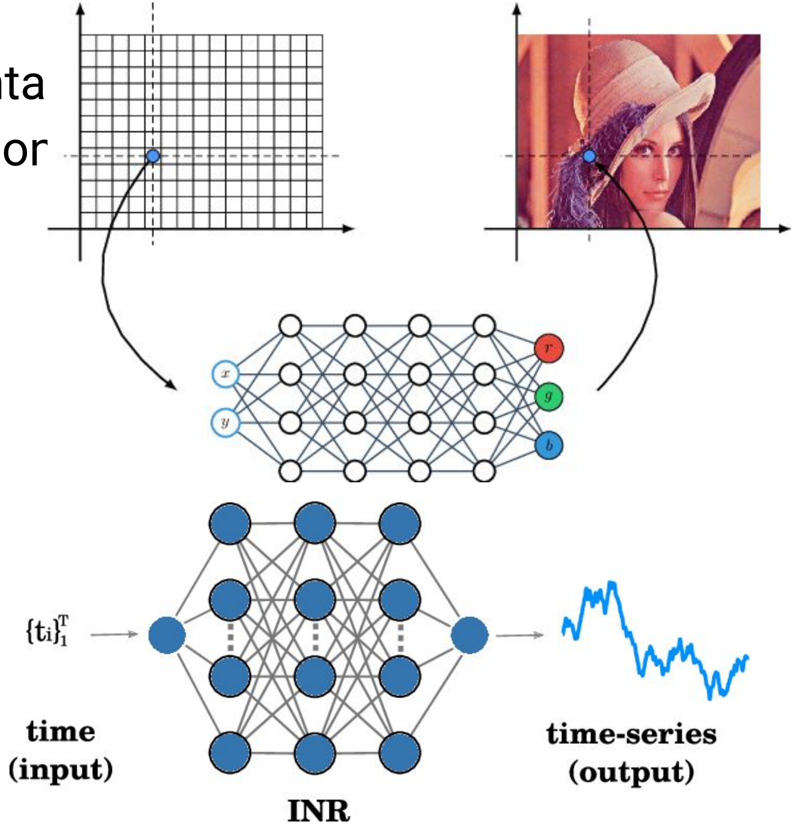
Component	Entities	Relations	Rel. inductive bias	Invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Nodes	Edges	Arbitrary	Node, edge permutations

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

Battaglia et al. *arXiv* (2018)

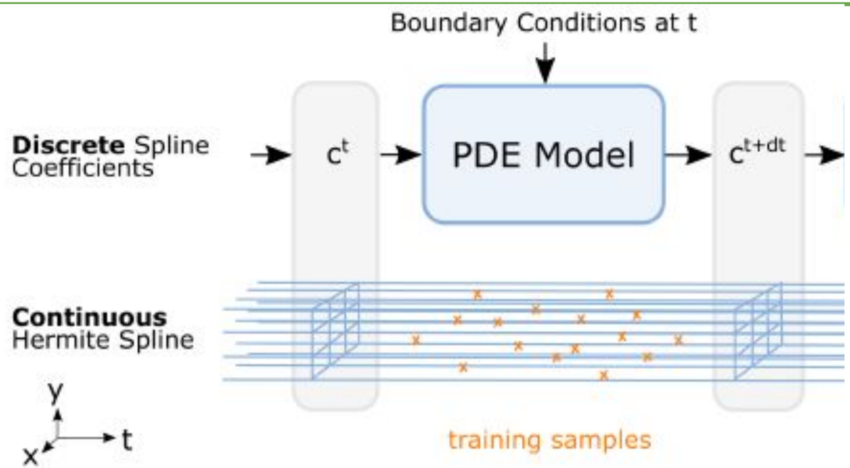
Limitation 4: Inadequate inductive bias

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

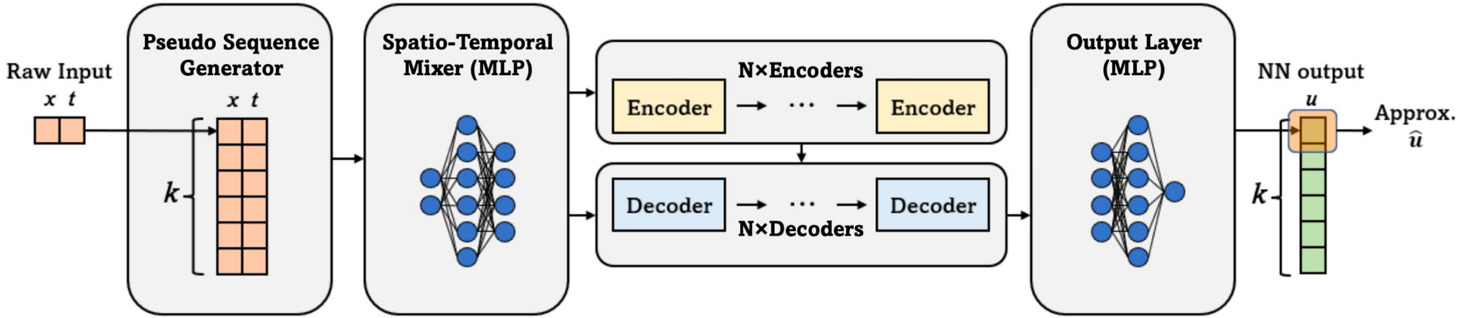


Solution to limitation 4: Connect to NN Architectures

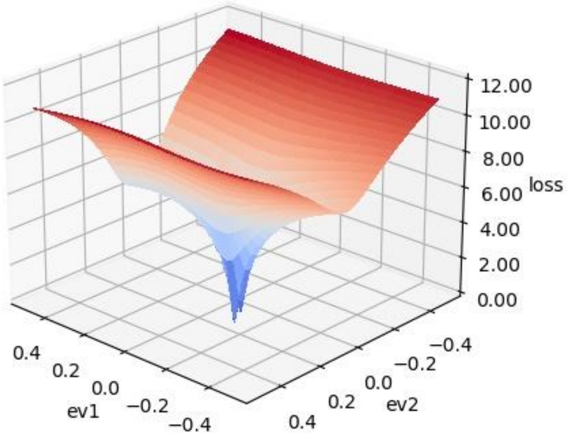
- PINNs + CNNs
  (Wandel et al., *AAAI* 2022)
  - Instead of having the CNN directly output the solution to the PDE, the CNN output spline coefficients on a discrete grid

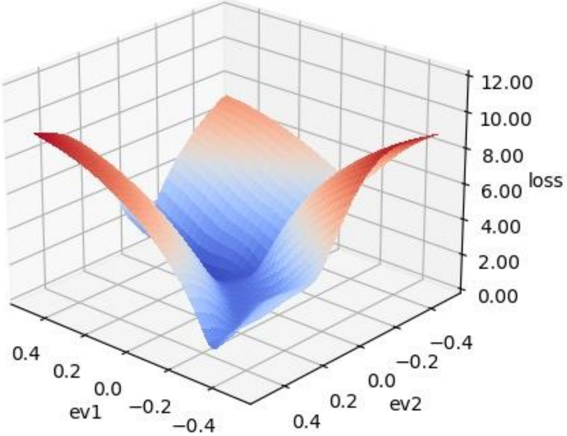


- PINNs + Transformers (Zhao et al., *ICLR* 2024)
  - Transforms point-wise inputs into pseudo sequences



Solution to limitation 4: Connect to NN Architectures





PINNs:

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

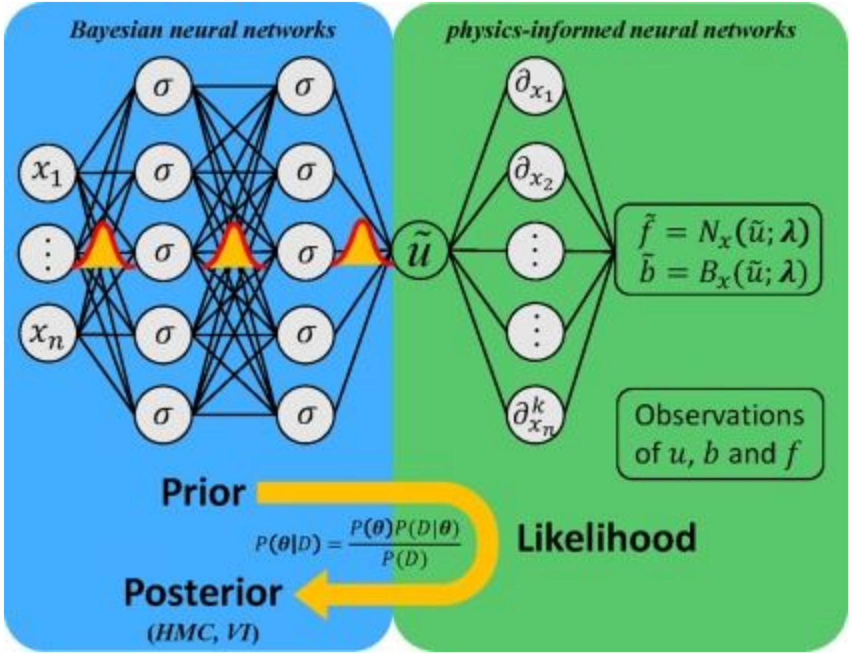
PINNs + Transformer:

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

Zhao et al., *ICLR* 2024

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.



Yang et al. (2021)

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

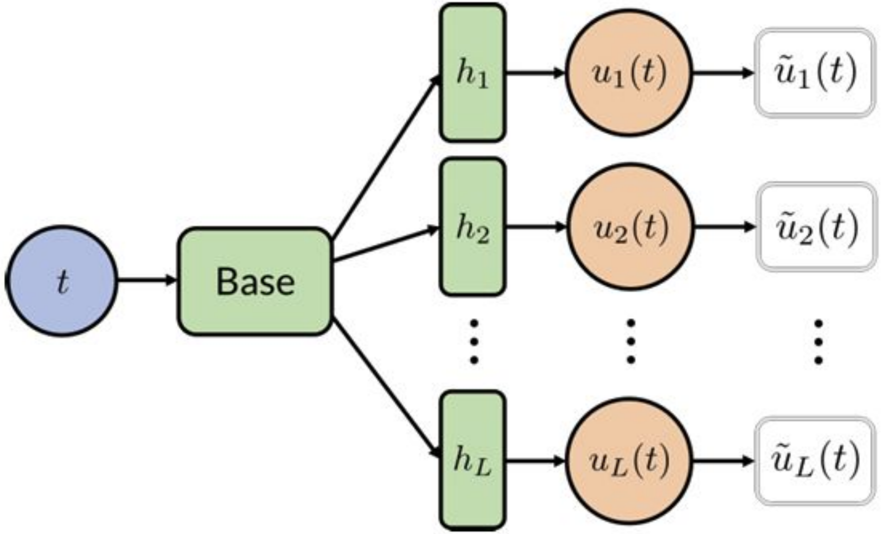
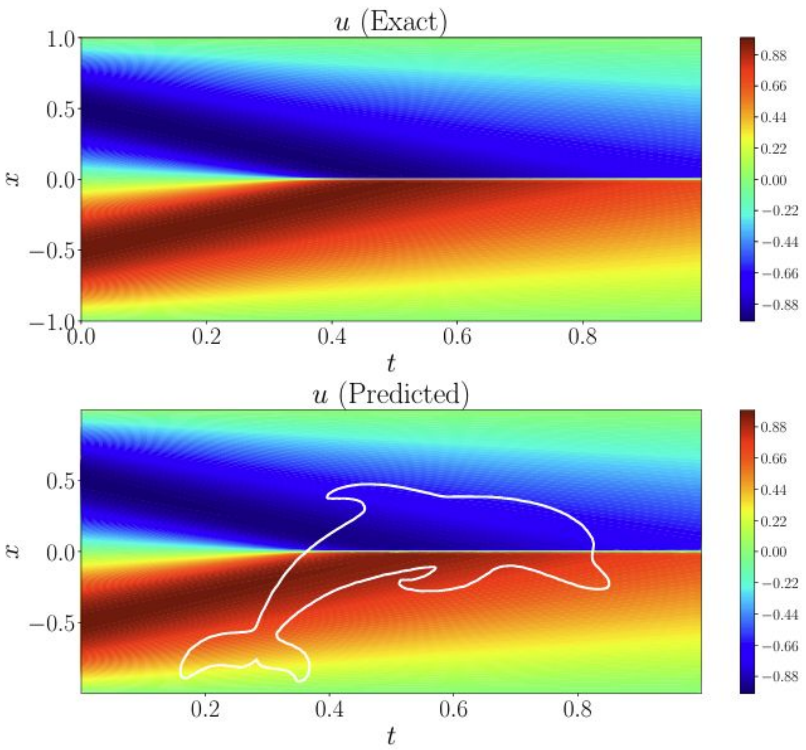


Figure 1: Multi-head PINN architecture. Each output head h_l is responsible for generating the solution to the l^{th} initial condition.

Pellegrin et al. (2022)

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



Jagtap & Karniadakis (2021)

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