Physics-Informed Neural Networks

Scientific Machine Learning

Alena Kopaničáková

ENSEEIHT / Toulouse INP

Spring 2026

Outline

Motivation: Why PINNs?

Limitations of Classical PDE Solvers

Classical methods (FDM, FEM, spectral): Mature, reliable, backed by theory

But they struggle in several important regimes:

1. **High-dimensional PDEs:** Meshes need $\mathcal{O}(h^{-d})$ points. For $d = 100$ (Black-Scholes, Hamilton-Jacobi-Bellman, Schrödinger equation), $h = 0.1$: 10^{100} dofs
2. **Complex / moving geometries:** Remeshing is expensive and error-prone
3. **Parametric PDE families:** Engineering design: same PDE, thousands of parameter values - classical solvers treat each instance independently
4. **Inverse problems:** Recovering PDE coefficients or initial conditions from sparse, noisy observations - not naturally expressed in the classical framework
5. **Equation discovery:** Measurements exist but are sparse and PDE is unknown

The Core Idea of PINNs

Fundamental observation

If u solves the PDE, the PDE residual is zero everywhere in Ω

- PINNs turn this into a loss function

Approach:

- Approximate u by a neural network $\hat{u}_\theta$
- Train θ to minimize the PDE residual at a finite set of **collocation points**
- The PDE provides an *infinite supply of pseudo-data* ($y_i = 0$), supplementing true physical observations
- The optimization landscape is shaped by *both data and physics simultaneously*

Connection to supervised learning

- Collocation points $\{(\boldsymbol{x}_i^r, 0)\}$ play the role of training data
- Labels $y_i = 0$ are generated by physics, not measurement
- Feature map $\boldsymbol{x} \mapsto \mathcal{L}[\hat{u}_\theta](\boldsymbol{x})$ involves automatic differentiation

PINNs for Forward Problems

General PDE Setup

Consider a boundary value problem on a bounded domain $\Omega \subset \mathbb{R}^d$:

$$\begin{cases} \mathcal{L}[u](\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ \mathcal{B}[u](\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \partial\Omega \end{cases}$$

- $\mathcal{L}$: (possibly nonlinear) differential operator
- $\mathcal{B}$: boundary operator - Dirichlet ($u|_{\partial\Omega} = g$), Neumann ($\partial_n u|_{\partial\Omega} = g$), or mixed
- Time-dependent: replace Ω by $Q_T = \Omega \times (0, T]$, add initial condition $u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x})$

Three running examples

1. **Poisson (elliptic):** $-\Delta u = f$ on Ω, $u = 0$ on $\partial\Omega$ (Prototypical linear elliptic problem)
2. **Heat (parabolic):** $\partial_t u - \nu \Delta u = 0$ on $\Omega \times (0, T]$ (Governs diffusion, thermal conduction)
3. **Burgers (nonlinear):** $\partial_t u + u\,\partial_x u - \nu\,\partial_{xx} u = 0$ (Develops near-discontinuous shock for small ν)

The Continuous PINN Objective

Measure how well $\hat{u}_\theta$ satisfies the PDE by integrating the squared residual:

$$\mathcal{R}(\theta) = \underbrace{\gamma_r \int_\Omega \left|\mathcal{L}[\hat{u}_\theta](\boldsymbol{x}) - f(\boldsymbol{x})\right|^2 \rho_r(\boldsymbol{x})\, d\boldsymbol{x}}_{\mathcal{R}_r(\theta)\text{: interior residual}} + \underbrace{\gamma_b \int_{\partial\Omega} \left|\mathcal{B}[\hat{u}_\theta](\boldsymbol{x}) - g(\boldsymbol{x})\right|^2 \rho_b(\boldsymbol{x})\, dS}_{\mathcal{R}_b(\theta)\text{: boundary residual}},$$

where ρ_r, ρ_b are probability densities (typically uniform) and $\gamma_r, \gamma_b > 0$ are loss weights

Theorem (Equivalence with the PDE)

$$\mathcal{R}(\theta) = 0 \iff \mathcal{L}[\hat{u}_\theta](\boldsymbol{x}) = f(\boldsymbol{x}) \;\; \textit{a.e. in } \Omega \;\; \textit{and} \;\; \mathcal{B}[\hat{u}_\theta](\boldsymbol{x}) = g(\boldsymbol{x}) \;\; \textit{a.e. on } \partial\Omega$$

The global minimizer of $\mathcal{R}$ is any network that exactly solves the PDE

Numerical Quadrature in 1D

Goal: Approximate $I = \int_a^b f(x)\,dx$ by $Q_n[f] = \sum_{i=1}^n w_i f(x_i)$
- Choice of **nodes** x_i and **weights** w_i defines the numerical integration (quadrature) rule

Newton-Cotes (uniform nodes)

Uniform spacing $h = \frac{b-a}{n}$:

- **Midpoint:** $w_i = h$, error $\mathcal{O}(h^2)$
- **Trapezoidal:** $w_1 = w_n = \frac{h}{2}$, rest h, error $\mathcal{O}(h^2)$
- **Simpson:** alternating $\frac{h}{3}, \frac{4h}{3}$, error $\mathcal{O}(h^4)$

Accuracy limited by smoothness of f

Gauss-Legendre (n nodes on $[-1, 1]$)

Optimal nodes *and* weights: $x_i =$ roots of P_n,
$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$

- Exact for deg $\leq 2n-1$
- **Spectral** convergence for smooth f
- Non-uniform: denser near ± 1
- Map to $[a, b]$: $x \mapsto \frac{a+b}{2} + \frac{b-a}{2}x$

Why it matters for PINNs

The PINN loss $\mathcal{L}_r(\theta) = \int_\Omega \mathcal{R}^2\,d\boldsymbol{x}$ must be discretised - the quadrature rule determines **how many collocation points** are needed and **how fast the error decays**

Gauss-Legendre Quadrature: Derivation

Goal: Choose nodes x_i and weights w_i on $[-1,1]$ so that $Q_n[f] = \int_{-1}^{1} f\,dx$ is exact for as high a polynomial degree as possible

Counting degrees of freedom

$2n$ free parameters (n nodes + n weights) $\Rightarrow$ target exactness for deg $\leq 2n-1$

Key idea: Orthogonal polynomials

Any $p \in \mathcal{P}_{2n-1}$ splits as $p = q\,P_n + r$ ($\deg r < n$, P_n = Legendre poly.). Since $P_n \perp \mathcal{P}_{n-1}$ on $[-1,1]$: $\int_{-1}^{1} p\,dx = \int_{-1}^{1} r\,dx$

Set $x_i =$ **roots of** P_n: then $P_n(x_i) = 0 \Rightarrow p(x_i) = r(x_i)$, so $\sum_i w_i p(x_i) = \sum_i w_i r(x_i)$

Fix weights: $w_i = \int_{-1}^{1} \ell_i(x)\,dx$, $\ell_i = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$ $\Rightarrow$ exact for $r \Rightarrow$ exact for all $p \in \mathcal{P}_{2n-1}$.

Result

$$x_i = \text{roots of } P_n, \qquad w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}, \qquad \left|\int_{-1}^{1} f\,dx - Q_n[f]\right| = \frac{2^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} f^{(2n)}(\xi)$$

Gauss-Legendre: 3-Point Example in 1D

On $[-1, 1]$, the 3-point GL rule uses roots of $P_3(x) = \frac{1}{2}(5x^3 - 3x)$:

$$x_1 = -\sqrt{\tfrac{3}{5}} \approx -0.7746, \quad x_2 = 0, \quad x_3 = +\sqrt{\tfrac{3}{5}} \approx 0.7746$$

$$w_1 = \tfrac{5}{9} \approx 0.5556, \quad w_2 = \tfrac{8}{9} \approx 0.8889, \quad w_3 = \tfrac{5}{9} \approx 0.5556$$

Exact for all $p \in \mathcal{P}_5$ (degree $\leq 2 \cdot 3 - 1 = 5$)

Example: $\int_{-1}^{1} e^x \, dx$

Exact value: $e - e^{-1} \approx 2.35040$.

GL3 approximation:

$$\begin{aligned} Q_3[e^x] &= \tfrac{5}{9}\, e^{-0.7746} + \tfrac{8}{9}\, e^{0} + \tfrac{5}{9}\, e^{0.7746} \\ &= \tfrac{5}{9}(0.4607) + \tfrac{8}{9}(1) + \tfrac{5}{9}(2.1699) \\ &= 0.2559 + 0.8889 + 1.2055 \\ &\approx 2.35040 \end{aligned}$$

Error $< 10^{-5}$ with only 3 function evaluations!

Comparison with uniform rules (3 pts)

Rule	Result	Error
Midpoint	$2\,e^0 = 2.000$	0.350
Trapezoidal	$e^{-1} + e^1 \approx 2.086$	0.264
Simpson	≈ 2.3504	$\approx 5 \times 10^{-5}$
GL3	≈ 2.35040	$< 10^{-5}$

GL3 matches Simpson accuracy but uses **optimal** (non-uniform) nodes - and beats it for non-polynomial f

Method 1: Gauss-Legendre Quadrature

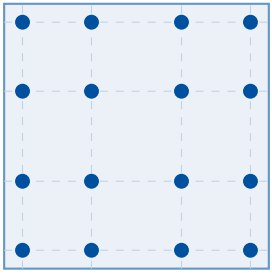
Idea: Place n nodes per axis at Gauss-Legendre positions (roots of Legendre polynomials on $[-1, 1]$, scaled to Ω); assign weights w_i, then

$$\int_0^1 f(x)\,dx \approx \sum_{i=1}^{n} w_i\, f(x_i)$$

Exact for all polynomials of degree $\leq 2n-1$; $N_r = n^d$ nodes in d dimensions

Key properties

- **Error:** $\mathcal{O}(N_r^{-2p/d})$ for C^{2p} integrands (p smoothness orders per axis; $n = N_r^{1/d}$ nodes per axis)
- **Exponential** convergence for analytic f
- Nodes are **non-uniformly spaced**: denser near boundaries
- **Curse of dimensionality:** n^d nodes - impractical for $d > 4$



$N_r = 16$ total

Method 2: Monte Carlo

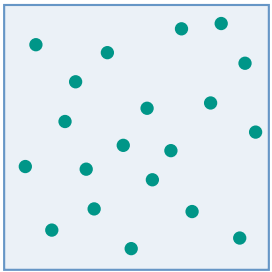
Idea: Draw N_r points i.i.d. from ρ_r (e.g. uniform on Ω) and average the integrand:

$$\int_\Omega f\,\rho_r\,d\boldsymbol{x} \;\approx\; \frac{1}{N_r}\sum_{i=1}^{N_r} f(\boldsymbol{x}_i), \quad \boldsymbol{x}_i \overset{\text{i.i.d.}}{\sim} \rho_r$$

Error $= \sigma_f/\sqrt{N_r}$ by the Central Limit Theorem, where $\sigma_f^2 = \mathrm{Var}_{\rho_r}[f]$

Key properties

- **Error:** $\mathcal{O}(N_r^{-1/2})$ - **dimension-free**
- No regularity assumption on f
- Stochastic: different runs give different results
- Original PINN (Raissi et al. 2019) uses uniform sampling
- Variance can be large if f has sharp features



$N_r = 20$ i.i.d. uniform samples

Method 3: Quasi-Monte Carlo (Halton/Sobol Sequences)

Idea: Replace i.i.d. samples with a *low-discrepancy sequence*(**Halton/Sobol**)
For example, Halton uses base-p_j Van der Corput sequences per dimension - error $= \mathcal{O}\big((\log N_r)^d / N_r\big)$ (Koksma-Hlawka).

Properties: Better than MC for $d \leq 10$; deterministic; use Sobol' for large d

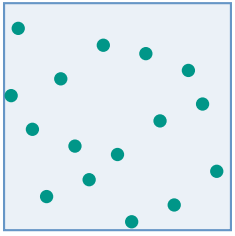
Python (`scipy.stats.qmc`)

```python
from scipy.stats.qmc import Halton, Sobol

# Halton in [0,1]^d
sampler = Halton(d=2, scramble=False)
pts = sampler.random(n=1000)  # (1000, 2)

# Sobol' (better for large d)
sampler = Sobol(d=2, scramble=True)
pts = sampler.random_base2(m=10) # 2^10 pts

# Scale to [a, b]^d
x_r = a + (b - a) * pts
```



$N_r = 16$ Halton (bases 2, 3)

Method 4: Latin Hypercube Sampling (LHS)

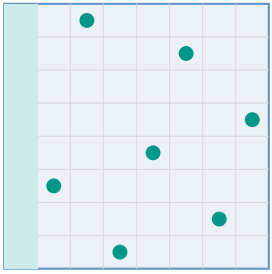
Idea: Divide each axis into N_r equal strips; place exactly *one point per strip in every dimension* (stratified sampling)

Construction (in $d = 2$):

1. Divide $[0, 1]$ into N_r intervals of width $1/N_r$
2. Choose a random permutation π of $\{1, \ldots, N_r\}$ for each axis
3. Set $x_i^{(j)} = (\pi_j(i) - U_i^{(j)})/N_r$, $U_i^{(j)} \sim \mathrm{Uniform}(0, 1)$

Key properties

- **Error:** $\mathcal{O}(N_r^{-1/2-1/d})$ - better than MC
- Each axis is stratified: no two points share a strip
- Low variance for smooth functions (each axis is well-covered)
- **Deterministic** after fixing the permutation



$N_r = 8$: one point per row and column

Method 5: Importance Sampling / RAR

Idea: Sample from a distribution $\rho_r(\boldsymbol{x}) \propto |r(\boldsymbol{x})|$ that concentrates points where the residual is large:

$$\int_\Omega r(\boldsymbol{x})\, d\boldsymbol{x} = \int_\Omega \frac{r(\boldsymbol{x})}{\rho_r(\boldsymbol{x})}\, \rho_r(\boldsymbol{x})\, d\boldsymbol{x} \approx \frac{1}{N_r} \sum_i \frac{r(\boldsymbol{x}_i)}{\rho_r(\boldsymbol{x}_i)}$$

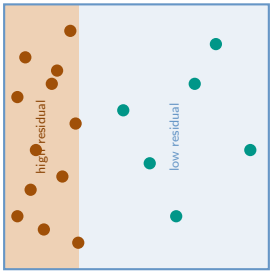
Optimal $\rho_r^\star \propto |r|$ makes the estimator variance **zero**

RAR (residual-based adaptive refinement):

1. Train K steps on current points
2. Evaluate $|r(\boldsymbol{x})|$ on a candidate grid
3. Add ΔN new points $\propto |r(\boldsymbol{x})|$; repeat

Key properties

- Same $\mathcal{O}(N_r^{-1/2})$ rate, but **much smaller constant**
- Essential for boundary layers / sharp gradients
- RAR is a greedy, tractable approximation to optimal IS



Points $\propto |r(\boldsymbol{x})|$: dense where residual is large

Discretizing the PINN Integral: Sampling Strategies

The interior integral $I = \int_\Omega r(\boldsymbol{x})\, \rho_r(\boldsymbol{x})\, d\boldsymbol{x}$ must be approximated numerically - N_r total points ($n = N_r^{1/d}$ per axis)

Method	**Error at fixed N_r**	**How d enters**	**Stoch.?**
Gauss (tensor)	$\mathcal{O}(N_r^{-2p/d})$	exponent $\frac{2p}{d} \to 0$	No
Monte Carlo	$\mathcal{O}(N_r^{-1/2})$	does not enter	Yes
Quasi-MC	$\mathcal{O}(N_r^{-1}(\log N_r)^d)$	prefactor $(\log N_r)^d \to \infty$	No
LHS	$\mathcal{O}(N_r^{-1/2-1/d})$	exponent $\to \frac{1}{2}$	No
Importance sampling	$\mathcal{O}(N_r^{-1/2})$, low const.	does not enter	Yes

- **Gauss:** Exponentially accurate for smooth integrands, but needs n^d nodes - only feasible for $d \leq 4$
- **MC:** Error $\mathcal{O}(N_r^{-1/2})$ regardless of d - dimension-free. Original PINN (Raissi et al. 2019) uses uniform MC
- **Importance sampling:** Sample $\rho_r \propto |r(\boldsymbol{x})|$; concentrates points where the residual is large
- **Integration error = generalization gap**: $\sup_\phi |\mathcal{R}(\phi) - \widehat{\mathcal{L}}(\phi)|$

The PINN Loss (Monte Carlo Discretization)

Applying MC discretization yields the **empirical PINN loss**:

$$\widehat{\mathcal{L}}(\theta) = \underbrace{\frac{\gamma_r}{N_r}\sum_{i=1}^{N_r} \left|\mathcal{L}[\hat{u}_\theta](\boldsymbol{x}_i^r) - f(\boldsymbol{x}_i^r)\right|^2}_{\widehat{\mathcal{L}}_r(\theta)} + \underbrace{\frac{\gamma_b}{N_b}\sum_{j=1}^{N_b} \left|\mathcal{B}[\hat{u}_\theta](\boldsymbol{x}_j^b) - g(\boldsymbol{x}_j^b)\right|^2}_{\widehat{\mathcal{L}}_b(\theta)}$$

For time-dependent problems on $Q_T = \Omega \times (0, T]$ and problems with data, also add:

$$\widehat{\mathcal{L}}_0(\theta) = \frac{\gamma_0}{N_0}\sum_{k=1}^{N_0} \left|\hat{u}_\theta(\boldsymbol{x}_k^0, 0) - u_0(\boldsymbol{x}_k^0)\right|^2, \qquad \widehat{\mathcal{L}}_d(\theta) = \frac{\gamma_d}{N_d}\sum_{k=1}^{N_d} \left|\hat{u}_\theta(\boldsymbol{x}_k^d) - y_k^d\right|^2$$

Total: $\widehat{\mathcal{L}} = \widehat{\mathcal{L}}_r + \widehat{\mathcal{L}}_b + \widehat{\mathcal{L}}_0 + \widehat{\mathcal{L}}_d$ with weights $(\gamma_r, \gamma_b, \gamma_0, \gamma_d)$ to be tuned

Collocation points as pseudo-data

Pairs $\{(\boldsymbol{x}_i^r, 0)\}$ are training examples with label $y_i = 0$: the PDE dictates that the residual must vanish everywhere. Labels are *free* (generated by physics), N_r controls integration accuracy, not statistical uncertainty

Automatic Differentiation for PDE Residuals

How $\mathcal{L}[\hat{u}_\theta](x)$ is evaluated

- $\hat{u}_\theta$ is a DNN prediction - a composition of elementary operations (affine maps + activations)
- All spatial derivatives $\partial_{x_i}\hat{u}_\theta$, $\partial^2_{x_i x_j}\hat{u}_\theta$, etc. are computed **exactly** via automatic differentiation (AD)
- AD applies the chain rule symbolically through the computational graph
- Second-order derivatives: apply AD a second time through the first-derivative graph
- The same reverse-mode pass computes both the PDE residual *and* $\nabla_\theta \widehat{\mathcal{L}}(\theta)$ for the optimizer - frameworks: JAX, PyTorch, TensorFlow

Activation function requirement

ReLU: $\Delta\hat{u}_\theta = 0$ a.e. - unsuitable for 2nd-order PDEs!

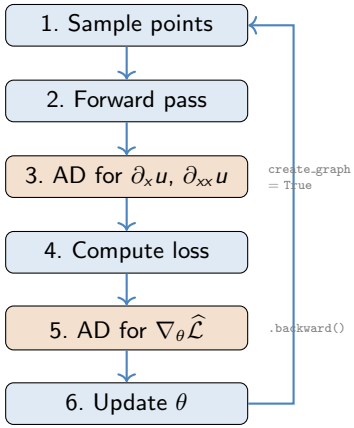
Use: tanh, GELU, Swish, or SIREN
(all smooth to all orders)

Activation	**Higher derivatives**
tanh	Smooth all orders
ReLU	Zero beyond order 1
GELU	Smooth all orders
Swish $t\sigma(t)$	Smooth all orders

PINN Training Loop

1. **Sample** boundary and physics collocation points $\{x_i^r\} \subset \Omega$, $\{x_j^b\} \subset \partial\Omega$
2. **Forward pass:** compute network outputs $\hat{u}_\theta(x_i^r)$, $\hat{u}_\theta(x_j^b)$
3. **Compute spatial derivatives** $\partial_x \hat{u}_\theta$, $\partial_{xx} \hat{u}_\theta$, ... via *recursive* autodiff applied to the network's computational graph $\leftarrow$ *extends the graph*
4. **Evaluate loss** $\widehat{\mathcal{L}}(\theta) = \widehat{\mathcal{L}}_r + \widehat{\mathcal{L}}_b$
5. **Backpropagate:** apply autodiff on the *extended* graph to compute $\nabla_\theta \widehat{\mathcal{L}}(\theta)$ $\leftarrow$ *differentiates through step 3*
6. **Update:** $\theta \leftarrow \theta - \alpha \nabla_\theta \widehat{\mathcal{L}}(\theta)$

Autodiff is applied **twice**: once to get the PDE residual (steps 2-3), and once more to get the gradient w.r.t. θ (step 5). PyTorch requires `create_graph=True` in step 3 so that the extended graph survives for step 5.



Cost of Computing Higher-Order Derivatives

Let C_f = cost of one forward pass ($\propto P$ parameters, L layers)

Quantity	Method	Compute	Memory
$u_\theta(\boldsymbol{x})$	Forward pass	C_f	$\mathcal{O}(L)$
$\nabla_\theta \widehat{\mathcal{L}}$	Reverse AD	$\mathcal{O}(C_f)$	$\mathcal{O}(L)$
$\partial_{x_i} u_\theta$	Reverse AD	$\mathcal{O}(C_f)$	Graph retained
$\partial_{x_i x_j} u_\theta$	2nd reverse AD	$\mathcal{O}(d\, C_f)$	$\mathcal{O}(L^2)$
Δu_θ	Diagonal of Hessian	$\mathcal{O}(d\, C_f)$	$\mathcal{O}(L^2)$

Why 2nd-order AD is expensive

`create_graph=True` keeps the gradient graph alive. The 2nd backward pass differentiates *through* it. Memory: $\mathcal{O}(L^2)$ vs. $\mathcal{O}(L)$ for standard backprop- main bottleneck for deep networks or large N_r.

Scaling with d and PDE order k

Each extra AD pass costs $\sim d\times$ more than standard backprop:
- cost $\sim d^{k-1}\, C_f$ per collocation point, e.g.,
- d=3, k=2 (Poisson): $\sim 3\times$
- d=10, k=2: $\sim 10\times$
- k=4 (biharmonic): $\sim d^3\times$

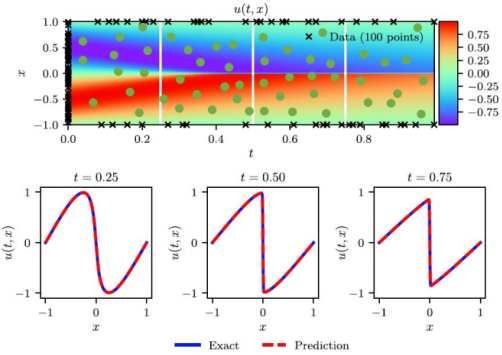
Example: Burgers Equation PINN (PyTorch) - Space-time discretization

Problem: $\partial_t u + u\,\partial_x u - \nu\,\partial_{xx} u = 0$ on $[-1,1] \times [0,1]$, $u(\pm 1, t) = 0$, $u(x,0) = -\sin(\pi x)$, $\nu = 0.01/\pi$
Network: 4-8 hidden layers, 20-50 neurons, tanh activations Typical: $N_r = 10\,000$, $N_b = N_0 = 200$

```python
def burgers_loss(model, x_r, t_r, x_b, t_b, x_0, nu=0.01/math.pi):
    # Interior residual: r = u_t + u*u_x - nu*u_xx
    xt_r = torch.cat([x_r, t_r], dim=1).requires_grad_(True)
    u    = model(xt_r)
    du   = torch.autograd.grad(u.sum(), xt_r, create_graph=True)[0]
    u_x, u_t = du[:, 0:1], du[:, 1:2]
    u_xx = torch.autograd.grad(u_x.sum(), xt_r, create_graph=True)[0][:, 0:1]
    loss_r = ((u_t + u * u_x - nu * u_xx)**2).mean()
    # Boundary: u(-1,t) = 0 and u(+1,t) = 0
    loss_b = (model(torch.cat([x_b, t_b], dim=1))**2).mean()
    # Initial condition: u(x,0) = -sin(pi*x)
    xt_0   = torch.cat([x_0, torch.zeros_like(x_0)], dim=1)
    loss_0 = ((model(xt_0) + torch.sin(math.pi * x_0))**2).mean()
    return loss_r + loss_b + loss_0
```

Note, `create_graph=True`: keeps the AD graph alive so `loss.backward()` can differentiate through the residual w.r.t. θ

Burgers PINN: **Space-Time** Approach (Raissi et al., 2019)



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

One network $u_\theta(x,t)$ on the full domain $[-1,1]\times[0,1]$. Circles = interior collocation points $(x_i,t_i)\sim \mathrm{Uniform}(Q_T)$; crosses = boundary/IC data. Net: 8 layers $\times$ 20 neurons, tanh; N_r=10 000, N_b=N_0=100; relative L^2 error $\approx 1.6\times10^{-3}$.

Discrete time-stepping: Classical Time-Stepping for ODEs

$$\text{IVP: } \frac{du}{dt} = f(t,u),\ u(t_0) = u_0$$

Discretise: $t_n = t_0 + n\Delta t$, advance $u^n \approx u(t_n)$

Forward (Explicit) Euler

$$u^{n+1} = u^n + \Delta t\, f(t_n, u^n)$$

- Order 1: $\mathcal{O}(\Delta t)$
- Explicit - u^{n+1} computed directly
- Conditionally stable ($\Delta t \leq \Delta t_{\max}$)

Classical RK4

$$k_1 = f(t_n, u^n), \quad k_2 = f\left(t_n + \tfrac{\Delta t}{2}, u^n + \tfrac{\Delta t}{2} k_1\right)$$

$$k_3 = f\left(t_n + \tfrac{\Delta t}{2}, u^n + \tfrac{\Delta t}{2} k_2\right), \quad k_4 = f(t_{n+1}, u^n + \Delta t\, k_3)$$

$$u^{n+1} = u^n + \tfrac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Order 4: $\mathcal{O}(\Delta t^4)$, explicit, conditionally stable

Key idea for the discrete-time PINN approach

Each scheme gives **explicit targets** u^{n+1} from $u^n \Rightarrow$ Train a network at t_{n+1} by regression against those targets

Discrete-Time Approach (1/2): Forward Euler PINN

Setting: PDE $\partial_t u = \mathcal{N}[u]$ on Ω
Use one network $u_\theta^n(x)$ per time slot

Step 1 - Compute targets (no unknowns)

Given trained u_θ^n, sample $\{x_k\} \subset \Omega$ and set:

$$y_k = u_\theta^n(x_k) + \Delta t\, \mathcal{N}[u_\theta^n](x_k),$$

where $\mathcal{N}[u_\theta^n](x_k)$ uses a forward pass + AD.
Targets y_k are **fixed numbers** once u_θ^n is known.

Step 2 - Pure regression for u_θ^{n+1}

$$\min_\theta \frac{1}{N_x} \sum_k \left(u_\theta^{n+1}(x_k) - y_k\right)^2$$

Repeat for every time step.

Algorithm

1. Sample $\{x_k\} \subset \Omega$
2. Eval $u_\theta^n(x_k)$, $\mathcal{N}[u_\theta^n](x_k)$ via AD
3. Form targets y_k
4. Minimise regression loss $\Rightarrow u_\theta^{n+1}$
5. Advance: $n \leftarrow n+1$, repeat

Limitations

- Order 1 in time
- CFL: $\Delta t = \mathcal{O}(h)$
- Errors accumulate
- One optimisation per step

Discrete-Time Approach (2/2): Boundary Conditions and Extensions

Adding boundary conditions

Augment the loss at each step with a boundary term:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{FE}}(\theta) + \gamma_b \frac{1}{N_b} \sum_{j=1}^{N_b} \left(u_\theta^{n+1}(x_j^b) - g(x_j^b, t_{n+1}) \right)^2,$$

where $\{x_j^b\} \subset \partial\Omega$ and g is the boundary data. BCs can also be enforced *exactly* via a distance-function ansatz.

Comparison: FE vs. continuous-time PINN

	Discrete (FE)	**Continuous**
Networks	One per step	One for all t
Targets	Explicit y_k	Residual $= 0$
∂_t via AD?	No	Yes
Stability	CFL limited	Δt free
Time accuracy	$\mathcal{O}(\Delta t)$	Spectral

Extensions

- Replace FE by **RK4**: compute $k_1, \ldots, k_4$ via AD on u_θ^n, assemble target, same regression step - fourth-order accuracy
- **Implicit** methods (IRK): u^{n+1} appears inside the RHS; requires coupled optimisation, but unconditionally stable
- Adaptive Δt strategies

Hard Enforcement of Boundary Conditions

Soft: $\widehat{\mathcal{L}}_b$ penalizes BC violations - approximate, requires tuning γ_b
Hard: Build BCs into the architecture via a *distance-function ansatz*

Homogeneous Dirichlet BCs ($u = 0$ on $\partial\Omega$)

$\hat{u}_\theta(\boldsymbol{x}) = d(\boldsymbol{x})\,\phi_\theta(\boldsymbol{x})$, where $d = 0$ on $\partial\Omega$, $d > 0$ in Ω. BCs satisfied exactly for **all** θ; no boundary loss needed.
Examples: $d(x) = x(1-x)$ on $[0,1]$; $d(x,y) = x(1-x)y(1-y)$ on $[0,1]^2$

Non-homogeneous Dirichlet ($u = g$ on $\partial\Omega$)

Construct lifting $\tilde{g}$ extending g into Ω; then $\hat{u}_\theta(\boldsymbol{x}) = \tilde{g}(\boldsymbol{x}) + d(\boldsymbol{x})\,\phi_\theta(\boldsymbol{x})$

	Soft	**Hard**
BC satisfaction	Approximate	Exact (machine precision)
Hyperparameters	γ_b must be tuned	None for BCs
Loss landscape	Multi-objective	Single-objective
Generality	Any BC type	Easiest for Dirichlet
Complex domains	Straightforward	Requires $d(\boldsymbol{x})$

Error Analysis

Three-Term Error Decomposition

Theorem (Error Decomposition for PINNs)

Under a coercivity estimate for $\mathcal{L}$:

$$\|\hat{u}_\theta - u^\star\|_{H^1(\Omega)}^2 \lesssim \underbrace{\inf_{\phi \in \hat{\mathcal{F}}_{m,L}} \|\phi - u^\star\|_{H^1}^2}_{\text{Approximation error}} + \underbrace{\sup_{\phi \in \hat{\mathcal{F}}_{m,L}} \left|\mathcal{R}(\phi) - \widehat{\mathcal{L}}(\phi)\right|}_{\text{Generalization gap}} + \underbrace{\widehat{\mathcal{L}}(\hat{u}_\theta) - \inf_{\phi} \widehat{\mathcal{L}}(\phi)}_{\text{Optimization error}}$$

- **Approximation error:** How well $\hat{\mathcal{F}}_{m,L}$ can represent $u^\star$. Controlled by width m, depth L, and PDE data regularity k.
- **Generalization gap:** Discrepancy between $\mathcal{R}$ and $\widehat{\mathcal{L}}$ - the integration error
- **Optimization error:** Failure to find the global minimizer of $\widehat{\mathcal{L}}$ (non-convex; least understood; never exactly zero in practice)

Why coercivity is the key bridge?

Coercivity ($\|v\|_{H^1} \leq C(\|\mathcal{L}[v]\|_{L^2} + \|\mathcal{B}[v]\|_{L^2})$) converts $\mathcal{R}(\theta) \to 0$ into $\|\hat{u}_\theta - u^\star\|_{H^1} \to 0$
Without it, minimizing the loss does not imply convergence to $u^\star$

Generalization Error and the Role of Quadrature

Generalization gap: Population $\mathcal{R}(\phi) = \int_\Omega |\mathcal{L}[\phi] - f|^2 \, d\boldsymbol{x}$ vs. empirical $\widehat{\mathcal{L}}(\phi) = \frac{1}{N_r} \sum_i |\mathcal{L}[\phi](x_i) - f(x_i)|^2$
Two separate analyses bound the same gap, depending on how $\{x_i\}$ are chosen

Random points (MC): Rademacher complexity

Points drawn i.i.d. $\Rightarrow$ probabilistic tool. Pointwise CLT gives $\mathcal{O}(N_r^{-1/2})$ for fixed ϕ, but optimizer picks ϕ *after* seeing $\{x_i\}$, so need *uniform* bound:

$$\sup_\phi |\mathcal{R}(\phi) - \widehat{\mathcal{L}}(\phi)| \leq 2 \underbrace{\mathfrak{R}_{N_r}}_{\lesssim m^2 L^2/\sqrt{N_r}} + C\sqrt{\frac{\log(1/\delta)}{N_r}}.$$

Gap rate: $\mathcal{O}(m^2 L^2/\sqrt{N_r})$. Condition: $m^2 L^2/N_r \to 0$ (Shin et al., 2020).

Deterministic points (GL/QMC): quadrature error

Points fixed, not random $\Rightarrow$ Rademacher does **not apply**.
Gap bounded directly by numerical integration error of $\mathcal{R}$:

Rule	Gap rate
Monte Carlo	$\mathcal{O}(N_r^{-1/2})$
Quasi-MC	$\mathcal{O}(N_r^{-1}(\log N_r)^d)$
Gauss-Legendre	$\mathcal{O}(N_r^{-2p/d})$

GL is **spectral** for smooth residuals - different bound, same gap.

Takeaway

Two frameworks, same gap:
1.) Rademacher (MC/random) and quadrature error (GL/deterministic)
2.) GL closes the gap with **exponentially fewer points** than MC when the residual is smooth

Approximation Error: Yarotsky's Theorem

Data regularity: PDE has C^k coefficients, $f \in H^k(\Omega) \xrightarrow{\text{elliptic reg.}} u^\star \in H^{k+2}(\Omega)$

Approximating in H^1 costs one smoothness order, giving effective order $s = k + 1$

Theorem (Yarotsky 2017 - deep ReLU networks)

Let $f \in W^{s,\infty}([0,1]^d)$ with $\|f\|_{W^{s,\infty}} \leq 1$. For every $\varepsilon \in (0, \frac{1}{2}]$ there exists a ReLU network $\hat{f}_\theta$ with

$$\text{depth} \leq C_{s,d} \log \tfrac{1}{\varepsilon}, \qquad \text{nonzero weights} \leq C_{s,d}\, \varepsilon^{-d/s} \log \tfrac{1}{\varepsilon}, \qquad \|\hat{f}_\theta - f\|_{L^\infty} \leq \varepsilon.$$

The weight bound is optimal *up to* log*: no network with fewer than $c_{s,d}\, \varepsilon^{-d/s}$ weights can approximate all $f \in W^{s,\infty}$ to accuracy ε.*

Applied to PINNs with $s = k + 1$: $P = \mathcal{O}\left(\varepsilon^{-d/(k+1)} \log \frac{1}{\varepsilon}\right)$ parameters achieve $\|\hat{u}_\theta - u^\star\|_{H^1} \leq \varepsilon$

Key takeaways

- Smoother data ($k \uparrow$) $\Rightarrow$ fewer parameters
- Higher $d \Rightarrow$ exponentially more (curse of dim.)
- Shocks / corners: k drops, rates deteriorate
- Depth $\mathcal{O}(\log 1/\varepsilon)$ sufficient - depth helps

Why ReLU if PINNs use tanh?

ReLU: $\partial_{xx} \hat{u} = 0$ a.e. - unusable for 2nd-order PDEs. However, $\varepsilon^{-d/s}$ is an *information-theoretic* limit of $W^{s,\infty}$, not of the activation. *Tanh*/sin achieve the **same rate** and are often more parameter-efficient on smooth targets.

Optimization for PINNs

The Optimization Landscape

The PINN loss is a **finite-sum / ERM problem**:

$$\min_{\theta \in \mathbb{R}^n} \widehat{\mathcal{L}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} f_i(\theta), \quad m = N_r + N_b$$

In PINNs, $f_i(\theta) = |\mathcal{L}[\hat{u}_\theta](\boldsymbol{x}_i) - f(\boldsymbol{x}_i)|^2$ involves **spatial derivatives** of $\hat{u}_\theta$ - unlike standard regression which depends only on network *outputs*

Three structural consequences

1. **Smoothness requirements:** Smooth activations are mandatory (see AD slide)
2. **Mini-batch variance:** Nearby collocation points share information through $\mathcal{L}$ $\implies$ adding more points in the same region reduces variance far less than in i.i.d. settings
3. **Ill-conditioned loss landscape:** The differential operator shapes the Hessian through the network Jacobian - condition numbers grow polynomially with solution frequency

Optimization Methods for PINNs

General update: $\theta_{k+1} = \theta_k - \alpha_k M_k^{-1} \nabla f(\theta_k)$. Choice of M_k determines the optimizer.

1. First-order (Gradient Descent, $M_k = I$)

$\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k)$
Low cost per step; many iterations needed

2. Second-order (Newton / L-BFGS)

$M_k = H(\theta_k)$
Fast local convergence; $\mathcal{O}(n^3)$ per step for full Newton
L-BFGS: quasi-Newton; stores $\sim$20 gradient vectors; popular in PINNs (full-batch)

3. Adaptive gradient methods (Adam)

Diagonal M_k from running 1st + 2nd moment estimates of gradients
Mini-batch compatible; partially corrects ill-conditioning

Common practice in PINNs

Run Adam for 10 000-50 000 steps, then switch to L-BFGS for better convergence.

BFGS and L-BFGS: Derivation

Goal: Approximate $H(\theta_{k+1})$ cheaply from gradient differences only

Step 1 - Secant condition: Let $s_k = \theta_{k+1} - \theta_k$, $y_k = \nabla f(\theta_{k+1}) - \nabla f(\theta_k)$. Require $B_{k+1}s_k = y_k$: the approximation must match observed curvature.

Step 2 - BFGS rank-2 update (closest SPD matrix to B_k satisfying secant):

$$B_{k+1} = B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k}$$

Step 3 - Invert via Sherman-Morrison ($p_k = 1/y_k^\top s_k$):

$$B_{k+1}^{-1} = (I - p_k s_k y_k^\top)\, B_k^{-1}\, (I - p_k y_k s_k^\top) + p_k s_k s_k^\top$$

Cost: $\mathcal{O}(n^2)$ per step instead of $\mathcal{O}(n^3)$ for full Newton

Step 4 - L-BFGS: Store only the last S pairs $\{(s_j, y_j)\}_{j=k-S}^{k}$. Compute $B_{k+1}^{-1}g$ via *two-loop recursion* in $\mathcal{O}(Sn)$ without forming B_{k+1}^{-1}. Typical $S = 20$; memory $\mathcal{O}(Sn)$ vs. $\mathcal{O}(n^2)$ for full BFGS

Neural Tangent Kernel: From Parameters to Functions

Definition (Empirical NTK)

For a network $h_\theta : \Omega \to \mathbb{R}$, the **Neural Tangent Kernel** is

$$K_\theta(x, x') \;=\; \nabla_\theta h_\theta(x)^\top \nabla_\theta h_\theta(x')$$

Key insight: Gradient descent in parameter space

$$\dot{\theta}(t) = -\nabla_\theta f\big(h_\theta\big)$$

induces a *kernel gradient flow* in function space:

$$\boxed{\dot{h}_\theta(x) = -\int_\Omega K_\theta(x, x')\, \frac{\delta f}{\delta h(x')}\, dx'}$$

- The NTK determines *which directions in function space* GD can explore
- Eigenfunctions of K_θ with large eigenvalues are learned fastest

Discrete NTK and Convergence

From Jacobian to kernel: Let $J_k \in \mathbb{R}^{n \times m}$ be the Jacobian of the network outputs w.r.t. parameters at step k: $[J_k]_{ij} = \frac{\partial h_\theta(x_i)}{\partial \theta_j}$

Then the discrete parameter update $\theta_{k+1} = \theta_k - \alpha\, J_k^\top \nabla_{\mathsf{h}} L$ gives

$$\mathsf{h}_{k+1} \;\approx\; \mathsf{h}_k + J_k\,(\theta_{k+1} - \theta_k) \;=\; \mathsf{h}_k - \alpha\, \underbrace{J_k\, J_k^\top}_{m\,\Theta_k}\, \nabla_{\mathsf{h}} L(\mathsf{h}_k)$$

This defines the **empirical kernel matrix** $\Theta_k = \frac{1}{m}\, J_k\, J_k^\top \in \mathbb{R}^{n \times n}$

Squared loss: $L = \frac{1}{2}\|\mathsf{h} - \mathsf{y}\|^2$:

$$\mathsf{h}_{k+1} \;\approx\; \mathsf{h}_k - \alpha\, \Theta_k\,(\mathsf{h}_k - \mathsf{y})$$

Convergence (quadratic regime): Let $H = \frac{1}{m} J^\top J$ (same nonzero spectrum as Θ). Then

$$\|\theta_{k+1} - \theta^\star\| \;\le\; \frac{\kappa(H) - 1}{\kappa(H) + 1}\, \|\theta_k - \theta^\star\|$$

$\Longrightarrow$ Convergence is fast when $\kappa(H)$ is small; it stalls if the problem is ill-conditioned ($\kappa(H)$ is large)

NTK for PINNs: Residual-Based Kernel

In standard regression, the Jacobian J_k differentiates the *network output* $h_\theta(x_i)$

In **PINNs**, the loss is defined on **residuals**, **not on the output directly**:

$$r_{\mathrm{pde}}(\theta) = \mathcal{N}[h_\theta](x_i^\Omega), \qquad r_{\mathrm{bc}}(\theta) = h_\theta(x_j^{\partial\Omega}) - g_j$$

The Jacobian is therefore taken *w.r.t. these residuals*:

$$J_k \;=\; \begin{bmatrix} \partial r_{\mathrm{pde}}/\partial\theta \\ \partial r_{\mathrm{bc}}/\partial\theta \end{bmatrix}$$

Note: $\partial r_{\mathrm{pde}}/\partial\theta$ involves differentiating through the operator $\mathcal{N}$ (e.g. ∇^2), so its rows are *structurally different* from the standard output Jacobian

The resulting kernel matrix inherits a **block structure**:

$$\Theta_k = \tfrac{1}{m} J_k J_k^\top \;=\; \begin{bmatrix} \boldsymbol{K}_{\mathrm{rr}} & \boldsymbol{K}_{\mathrm{rb}} \\ \boldsymbol{K}_{\mathrm{br}} & \boldsymbol{K}_{\mathrm{bb}} \end{bmatrix}$$

The **block spectrum** of Θ_k determines which residual components (PDE vs. BC) converge first - a key diagnostic for training pathologies in PINNs

Spectral Bias

Mode-wise error decay: $\Theta = Q\Lambda Q^\top$, $e_0 = \sum_i c_i q_i$

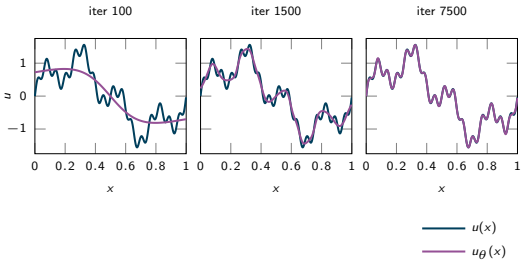
$$\|e_k\|^2 = \sum_i c_i^2 (1 - \alpha\lambda_i)^{2k}$$

Large $\lambda_i \Rightarrow$ fast decay; small $\lambda_i \Rightarrow$ decay ≈ 1

Spectral bias (Rahaman 2019; Wang 2021)

DNNs learn **low-frequency** components first
High-frequency modes: small NTK eigenvalues
$\Rightarrow$ Extremely slow learning

$u(x) = \sin(2\pi x) + 0.5\sin(8\pi x) + 0.2\sin(32\pi x)$



Consequences:

- Sharp features / boundary layers: hard to learn
- Fourier embeddings & SIREN mitigate this

NTK and Differential Operators: Frequency Scaling

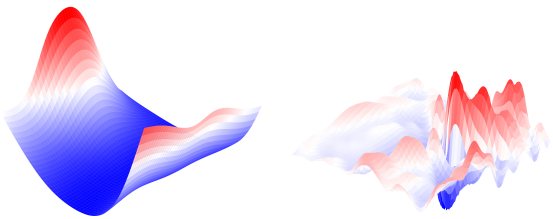
Why PINNs are harder to optimize:

Fourier mode $e^{i\omega x}$ in h_θ: applying p derivatives gives ω^p:

$$\partial_x^p h_\theta \sim \omega^p \;\Rightarrow\; H(\theta) \sim \omega^{2p}$$

$$\kappa(H) \approx \left(\frac{\omega_{\max}}{\omega_{\min}}\right)^{2p}$$

Loss landscape: data-driven (left) vs. PINN (right)



2D projection of the same DNN architecture. PINN landscape is markedly more anisotropic.

Key consequence

For a p-th order PDE: condition number scales as $(\omega_{\max}/\omega_{\min})^{2p}$
High-freq modes $\Rightarrow$ steep ravines; low-freq $\Rightarrow$ flat directions

Remedies for Spectral Bias

Fourier Feature Embeddings (Tancik et al., 2020)

Encode input before the network: $\gamma(\boldsymbol{x}) = [\cos(2\pi \boldsymbol{B}\boldsymbol{x}), \sin(2\pi \boldsymbol{B}\boldsymbol{x})]^\top$, $\boldsymbol{B}_{ij} \sim \mathcal{N}(0, \sigma_{\text{ff}}^2)$
Scale σ_{ff} tunes frequency content; yields substantially more uniform NTK spectrum

Multi-frequency Architectures (Wang et al., 2022)

Run K parallel branches, each with a different Fourier scale σ_k, then merge:

$$\hat{u}_\theta(\boldsymbol{x}) = \boldsymbol{W}_{\text{out}} \left[\mathrm{MLP}_1(\gamma_{\sigma_1}(\boldsymbol{x})); \cdots ; \mathrm{MLP}_K(\gamma_{\sigma_K}(\boldsymbol{x}))\right], \quad \sigma_1 < \cdots < \sigma_K.$$

Branch k acts as a specialist at scale $\sim 1/\sigma_k$ (multigrid analogy)
A lighter variant replaces each layer by a learned blend of two feature gates U, V at different scales, adaptively mixing low- and high-frequency content

Preconditioned / Adaptive Methods

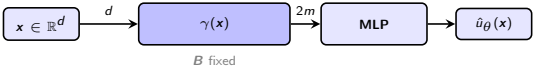
Adaptive kernel: $\Theta_k^{(M_k)} = \frac{1}{m} J_k M_k^{-1} J_k^\top$
Well-chosen M_k **flattens the spectrum**: $\kappa(\Theta_k^{(M_k)}) \ll \kappa(\Theta_k)$

Fourier Feature Embeddings: Architecture & NTK Effect

Architecture: Prepend a fixed random map before the MLP:

$$\gamma(\boldsymbol{x}) = \left[\cos(2\pi \boldsymbol{B}\boldsymbol{x}),\ \sin(2\pi \boldsymbol{B}\boldsymbol{x})\right]^\top \in \mathbb{R}^{2m}$$

$\boldsymbol{B} \in \mathbb{R}^{m \times d}$, $B_{ij} \sim \mathcal{N}(0, \sigma_{\text{ff}}^2)$ - **fixed, not trained**

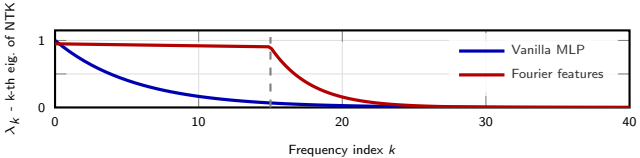


NTK effect:

$$k_{\text{FF}}(\boldsymbol{x}, \boldsymbol{x}') = k_{\text{MLP}}(\gamma(\boldsymbol{x}), \gamma(\boldsymbol{x}'))$$

- γ injects sinusoids up to $\sim \sigma_{\text{ff}}$
- NTK eigenvalues **flat** for $\omega \leq \sigma_{\text{ff}}$
- $\kappa = \mathcal{O}(1)$ vs. $\mathcal{O}(e^{c\omega})$ for plain MLP

Tuning: $\sigma_{\text{ff}} \approx$ highest freq in PDE
Too small $\rightarrow$ high freq stagnate
Too large $\rightarrow$ fits noise



Multi-Scale Architectures (Wang et al., 2022)

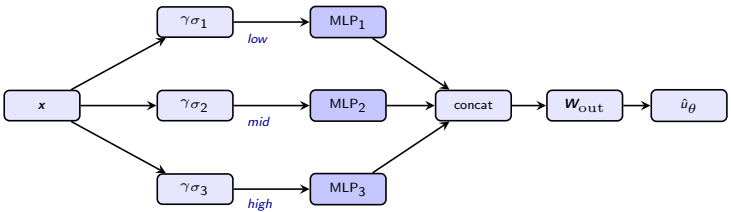
Idea: K parallel branches, each at a different Fourier scale:

$$\hat{u}_\theta(\boldsymbol{x}) = \boldsymbol{W}_{\text{out}}\left[\text{MLP}_1(\gamma_{\sigma_1}(\boldsymbol{x})); \cdots ; \text{MLP}_K(\gamma_{\sigma_K}(\boldsymbol{x}))\right]$$

$\sigma_1 < \sigma_2 < \cdots < \sigma_K$ (e.g. $\sigma_k = 2^{k-1}$).

Why it works:

- Branch k is a specialist at scale $\sim 1/\sigma_k$
- No frequency competition for gradient budget
- Analogue of **multigrid**: each level resolves one scale



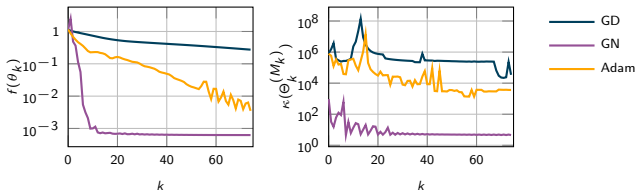
Kernels of Adaptive Methods

Every adaptive/second-order method can be viewed as kernel gradient descent with

$$\Theta_k^{(M_k)} = \frac{1}{m} J_k M_k^{-1} J_k^\top$$

- **GD** ($M_k = I$): most ill-conditioned; slowest
- **Adam** (M_k diagonal): partial spectrum flattening
- **Gauss-Newton** ($M_k = J_k^\top J_k + \beta I$): smallest κ; fastest convergence

Loss $f(\theta_k)$ and condition number $\kappa(\Theta_k^{(M_k)})$ over iterations



Smaller $\kappa(\Theta_k^{(M_k)}) \Rightarrow$ more uniform decay across all error modes

Gradient Pathology: the NTK View

PINN loss $\widehat{\mathcal{L}} = \gamma_r \widehat{\mathcal{L}}_r + \gamma_b \widehat{\mathcal{L}}_b \Rightarrow$ gradient flow governed by a **composite kernel**:

$$\Theta^{(\gamma)} = \begin{pmatrix} \gamma_r \boldsymbol{K}_{rr} & 0 \\ 0 & \gamma_b \boldsymbol{K}_{bb} \end{pmatrix}, \qquad \kappa\big(\Theta^{(\gamma)}\big) = \frac{\max\big(\gamma_r \lambda^{(r)}_{\max},\ \gamma_b \lambda^{(b)}_{\max}\big)}{\min\big(\gamma_r \lambda^{(r)}_{\min},\ \gamma_b \lambda^{(b)}_{\min}\big)}$$

The problem

$\boldsymbol{K}_{rr}$ (PDE residual): differential operator *suppresses* eigenvalues ($\lambda^{(r)}$ small)
$\boldsymbol{K}_{bb}$ (boundary): pointwise evaluation retains large eigenvalues ($\lambda^{(b)}$ large)
With $\gamma_r{=}\gamma_b{=}1$: $\kappa(\Theta^{(\gamma)}) \approx \lambda^{(b)}_{\max}/\lambda^{(r)}_{\min} \gg 1$

Consequence (from spectral bias): each error mode decays as $(1 - \alpha\lambda_i)^k$

- Boundary modes ($\lambda \sim \lambda^{(b)}_{\max}$): converge rapidly to ≈ 0
- Interior / PDE modes ($\lambda \sim \lambda^{(r)}_{\min}$): barely move
- $\Rightarrow$ **Gradient pathology**: $\widehat{\mathcal{L}}_b \to 0$ while $\widehat{\mathcal{L}}_r$ stagnates

Goal of weighting: reduce $\kappa(\Theta^{(\gamma)})$ by reshaping the composite spectrum

Per-Block Weighting Strategies

One weight per loss component - targets the **inter-block** spectral gap

1. NTK-based weighting (Wang et al., 2022)

Set $\gamma_b/\gamma_r = \lambda_{\max}^{(r)}/\lambda_{\max}^{(b)} \;\Rightarrow\; \gamma_r\lambda_{\max}^{(r)} = \gamma_b\lambda_{\max}^{(b)}$ (tops aligned)

Kernel effect: inter-block gap eliminated; $\kappa(\Theta^{(\gamma)}) = \max\big(\kappa(\boldsymbol{K}_{rr}),\, \kappa(\boldsymbol{K}_{bb})\big)$

Cost: $\mathcal{O}(P^2N)$ per NTK recomputation (every $T = 100$-1000 steps)

2. GradNorm (Chen et al., 2018)

Equalize $\|\gamma_i \nabla_\theta \widehat{\mathcal{L}}_i\|$ across losses via auxiliary optimization

Proxy for spectral balancing: $\|\gamma_i \nabla_\theta \widehat{\mathcal{L}}_i\|^2 = \gamma_i^2\, \boldsymbol{e}_i^\top \boldsymbol{K}_{ii}\, \boldsymbol{e}_i$

Kernel effect: balances effective spectral radii weighted by current errors

Cheaper ($\mathcal{O}(N)$); history-aware via relative training speed r_i

Per-Point Weighting Strategies

One weight per collocation point - reduces condition number κ **within-block**

3. Self-adaptive weights (McClenny & Braga-Neto, 2023)

Minimax formulation: $\min_\theta \max_{\boldsymbol{\gamma}\geq 0} \sum_i \gamma_i |r_i|^2$

Weighted kernel: $\boldsymbol{K}_{rr}^{(\Gamma)} = \frac{1}{N_r} \sum_i \gamma_i \boldsymbol{j}_i \boldsymbol{j}_i^\top$

Kernel effect: Adversarial γ_i amplifies rank-one terms at high-residual points
$\Rightarrow$ boosts smallest eigenvalues of $\boldsymbol{K}_{rr}^{(\Gamma)}$, reducing $\kappa(\boldsymbol{K}_{rr}^{(\Gamma)})$

4. Spatially adaptive weighting

Closed-form weights: $\xi_j = |r_j|^\beta \big/ \big(\frac{1}{N_r} \sum_k |r_k|^\beta\big), \quad \beta > 0$

Kernel effect: Same mechanism - redistributes spectral mass from well-learned to under-learned modes
Deterministic; no adversarial optimization

"Soft" version of adaptive sampling (reweights existing points instead of adding new ones)

Weighting Strategies: Numerical Example

Setup: $\boldsymbol{K}_{rr}$ eigenvalues $\{0.1, 0.04, 0.02, 0.01\}$ (κ=10), $\boldsymbol{K}_{bb}$ eigenvalues $\{10, 5\}$ (κ=2)

Case 1: Uniform (γ_r=γ_b=1)

Composite: $\{10, 5, 0.1, 0.04, 0.02, 0.01\}$

$\kappa(\Theta^{(\gamma)}) = 10/0.01 = 1000$

Contraction factor: $999/1001 \approx 0.998$

$\sim$1150 iters for 10$\times$ error reduction

Case 2: NTK-based (γ_b=0.01)

Composite: $\{0.1, 0.1, 0.05, 0.04, 0.02, 0.01\}$

$\kappa(\Theta^{(\gamma)}) = 0.1/0.01 = 10$

Contraction: $9/11 \approx 0.818$

$\sim$11 iters for 10$\times$ error reduction

Case 3: + Self-adaptive

Adversarial γ_i upweights high-residual points

(e.g., γ_3=3, γ_4=5 at hardest points)

Within-block: $\{0.1, 0.04, 0.06, 0.05\}$

$\kappa(\boldsymbol{K}_{rr}^{(\Gamma)}) \approx 2.5$ (from 10)

Composite $\kappa \approx 2.5$

Strategy	κ	Targeted spectrum
Uniform	1000	-
NTK-based	10	inter-block
+Self-adaptive	2.5	+within-block

Sampling Strategies for Training

Collocation points in PINNs are *chosen by the user*; this choice directly shapes the optimization landscape.

Mini-batch vs. full-batch

- **Full-batch:** Stable gradients, compatible with L-BFGS; expensive per step
- **Mini-batch SGD:** Cheap, but PINN residuals at nearby points are correlated through $\mathcal{L}$ - gradient variance is higher than in standard regression

Adaptive collocation point selection

Repeat: (1) train K steps; (2) evaluate $|r(\boldsymbol{x})|$ on a candidate grid; (3) add ΔN new points sampled $\propto |r(\boldsymbol{x})|$
Concentrates budget near boundary layers and sharp gradients

Practical recommendations

- Use full-batch with L-BFGS for final fine convergence
- Apply RAR if boundary layers or sharp features are present
- Monitor $\widehat{\mathcal{L}}_r$ and $\widehat{\mathcal{L}}_b$ separately to detect imbalance

Weak-Form and Variational PINNs

From Strong Form to Weak Form

Strong-form PINN: Evaluates $\mathcal{L}[\hat{u}_\theta]$ pointwise - requires 2nd-order derivatives of $\hat{u}_\theta$ (two AD passes); worsens NTK conditioning for high frequencies

Weak form (Poisson: $-\Delta u = f$, $u|_{\partial\Omega} = 0$): Multiply by $v \in H_0^1(\Omega)$ and integrate by parts:

$$-\int_\Omega (\Delta u)\, v\, d\boldsymbol{x} = \int_\Omega \nabla u \cdot \nabla v\, d\boldsymbol{x} - \underbrace{\int_{\partial\Omega} (\nabla u \cdot \boldsymbol{n})\, v\, dS}_{=0 \text{ since } v|_{\partial\Omega}=0} = \int_\Omega f\, v\, d\boldsymbol{x}$$

Weak (variational) form: Find $u \in H_0^1(\Omega)$ s.t. $a(u, v) := \int_\Omega \nabla u \cdot \nabla v\, d\boldsymbol{x} = \int_\Omega f\, v\, d\boldsymbol{x}\ \forall v$

Two key advantages:

1. **Reduced derivative order:** u and v need only *first* derivatives (one AD pass, not two)
2. **Coercivity / energy structure:** $a(v, v) = \|\nabla v\|_{L^2}^2 \geq C\|v\|_{H^1}^2$ by Poincaré's inequality $\Rightarrow$ direct variational approach (Deep Ritz)

Deep Ritz Method

Because bilinear form a is symmetric and coercive, the solution minimizes the **Dirichlet energy**:
$J[v] = \frac{1}{2}\int_\Omega |\nabla v|^2 \, d\boldsymbol{x} - \int_\Omega f\, v \, d\boldsymbol{x}$
Deep Ritz (E & Yu, 2018): substitute $\hat{u}_\theta$ for v; minimize $J[\hat{u}_\theta]$ over θ
Empirical loss (MC + hard BCs, $\hat{u}_\theta = x(1-x)\phi_\theta$):

$$\widehat{\mathcal{L}}_{\text{Ritz}}(\theta) = \frac{|\Omega|}{N_r} \sum_{i=1}^{N_r} \left[\tfrac{1}{2} |\nabla_{\boldsymbol{x}} \hat{u}_\theta(\boldsymbol{x}_i)|^2 - f(\boldsymbol{x}_i)\, \hat{u}_\theta(\boldsymbol{x}_i) \right]$$

Only **first-order** AD needed (one pass, not two)! By **Céa's lemma**:

$$\|\hat{u}_\theta - u^\star\|_{H^1} \leq \frac{1}{\alpha} \inf_{\phi \in \hat{\mathcal{F}}_{m,L}} \|\phi - u^\star\|_{H^1}$$

Error bounded purely by approximation quality - not by the optimizer

Connection to Galerkin FEM

Both minimize $J[v]$ over a finite-dimensional subspace of $H_0^1(\Omega)$. In FEM: fixed local piecewise-polynomial basis. In Deep Ritz: adaptive global network class. Céa's lemma applies to both with the same constant $1/\alpha$.

PyTorch: Strong PINN vs. Deep Ritz

Same problem: $-u''(x) = \pi^2 \sin(\pi x)$ on $(0,1)$, $u(0)=u(1)=0$, exact $u^\star = \sin(\pi x)$.
Same network: $\hat{u}_\theta(x) = x(1-x)\,\phi_\theta(x)$ (hard BCs).

Strong-form PINN loss

```
def pinn_loss(model, x_r):
    x = x_r.requires_grad_(True)
    u    = model(x)
    u_x  = autograd.grad(u.sum(), x,
               create_graph=True)[0]
    u_xx = autograd.grad(u_x.sum(), x,
               create_graph=True)[0]
    res = u_xx + f(x)        # -u''=f
    return (res**2).mean()
```

Two autograd passes ($u' \to u''$).
Residual: $|u''_\theta(x) + f(x)|^2$.

```
# Shared setup
model = PoissonNet(width=32, depth=3)
x_r   = torch.rand(500, 1)
opt   = Adam(model.parameters(), 1e-3)

for epoch in range(5000):
    opt.zero_grad()
    loss = pinn_loss(model, x_r)  # or
    # loss = deep_ritz_loss(model, x_r)
    loss.backward(); opt.step()
```

Deep Ritz loss

```
def deep_ritz_loss(model, x_r):
    x = x_r.requires_grad_(True)
    u   = model(x)
    u_x = autograd.grad(u.sum(), x,
              create_graph=True)[0]

    energy = 0.5*u_x**2 - f(x)*u
    return energy.mean()
```

One autograd pass (u' only).
Energy: $\frac{1}{2}|u'_\theta|^2 - f\,u_\theta$.

Key difference

PINN: Penalises residual pointwise, needs u''
Ritz: Minimises energy, needs u' only

Same network, same data, same optimizer - only the **loss function** changes

Strong PINN vs. Deep Ritz: Comparison

	Strong PINN	**Deep Ritz**
Derivative order in loss	2nd (u'')	1st (u')
Autograd passes	2	1
Cost per iteration	Higher	Lower
Loss meaning	Residual MSE	Energy $J[\hat{u}_\theta]$
Requires energy func.?	No	Yes (symm., coercive)
Non-symmetric PDE?	Yes	No

When to use Deep Ritz

- Elliptic, symmetric, coercive PDEs
- Large domains: cheaper 1st-order AD
- Want quasi-optimality (Céa's lemma)

When to use Strong PINN

- General PDEs (non-symmetric, time-dep.)
- No energy functional available
- Mixed BCs via soft penalty

Inverse Problems with PINNs

Inverse Problems: Overview and Types

Forward: PDE fully specified $\Rightarrow$ Find u
Inverse: Some aspects of the problem are unknown $\implies$ infer from sparse, noisy observations of u

Three principal types:

1. **Parameter identification:** Unknown coefficient λ in $\mathcal{L}_\lambda[u] = f$, e.g.,
   - Diffusivity in $-\nabla \cdot (\lambda \nabla u) = f$ (subsurface flow, EIT)
   - Wave speed $c(x)$ in $u_{tt} - c^2 \Delta u = 0$ (seismic inversion)
   - Reaction rate λ in $-\Delta u + \lambda u = f$
2. **Source identification:** Unknown forcing f partially/fully unknown
   - Pollutant source in advection-diffusion from downstream measurements
   - Heat source distribution from surface temperature measurements
3. **Initial/boundary condition recovery**
   - Often **severely ill-posed**, e.g., backward heat equation $u_t = -\Delta u$ is exponentially unstable

Why PINNs are natural for inverse problems

Classical methods: Two stages (forward solver + outer loop over λ)
PINNs: **Single joint optimization** - no explicit forward solver, meshing, or sensitivity analysis

Classical Approach: The Adjoint Method

Goal: Find λ minimising $J(\lambda) = \frac{1}{2}\sum_k |u(\lambda; \boldsymbol{x}_k^d) - y_k^d|^2 + \mu\,\Omega_{\text{reg}}(\lambda)$, where $u(\lambda;\cdot)$ solves $\mathcal{L}_\lambda[u] = f$

Problem: Need $\nabla_\lambda J$. Naïve: perturb each component $\Rightarrow$ $\dim(\lambda)$ forward PDE solves per gradient

Solution: Introduce Lagrangian with adjoint variable p:

$$L(u, \lambda, p) = J(u, \lambda) + \langle p,\, \mathcal{L}_\lambda[u] - f \rangle_{L^2}$$

Stationarity conditions give three equations:

1. **Forward:** $\mathcal{L}_\lambda[u] = f$ (Solve for u given current λ)
2. **Adjoint:** $\mathcal{L}_\lambda^*[p] = -\nabla_u J$ (Backward PDE; one extra solve)
3. **Gradient:** $\nabla_\lambda J = \mu \nabla_\lambda \Omega_{\text{reg}} + \langle p,\, \partial_\lambda \mathcal{L}_\lambda[u] \rangle$

Cost: **2 PDE solves** per gradient (forward + adjoint), regardless of $\dim(\lambda)$

Limitation - and PINN motivation

- Requires a **PDE solver** + its **adjoint implementation** (often non-trivial)
- Mesh generation for complex geometries
- Separate code paths for forward and inverse problems

PINNs collapse this into a **single joint optimisation** - no solver, no adjoint, no mesh needed

Joint State-Parameter Formulation using PINNs

Two sets of trainable parameters:
θ (approximating u) and ψ (encoding λ_ψ - scalar, vector, or second network)
Joint inverse PINN loss:

$$\widehat{\mathcal{L}}_{\text{inv}}(\theta,\psi) = \underbrace{\frac{\gamma_r}{N_r}\sum_i \left|\mathcal{L}_{\lambda_\psi}[\hat{u}_\theta](\boldsymbol{x}_i) - f(\boldsymbol{x}_i)\right|^2}_{\text{physics}} + \underbrace{\frac{\gamma_d}{N_d}\sum_k \left|\hat{u}_\theta(\boldsymbol{x}_k^d) - y_k^d\right|^2}_{\text{data}} + \underbrace{\frac{\gamma_b}{N_b}\sum_j \left|\mathcal{B}[\hat{u}_\theta](\boldsymbol{x}_j^b) - g_j\right|^2}_{\text{BCs}}$$

- $\widehat{\mathcal{L}}_r$: physics prior - propagates information beyond measurement locations
- $\widehat{\mathcal{L}}_d$: data anchor - without it, any λ_ψ satisfying $\mathcal{L}_{\lambda_\psi}[u] = f$ would minimize $\widehat{\mathcal{L}}_r$
- Weight selection under Gaussian noise σ^2: $\gamma_d/\gamma_r \propto 1/\sigma^2$

Positivity constraint on λ: Parameterize as $\lambda_\psi = e^\psi$ (single trainable scalar $\psi \in \mathbb{R}$)
For spatially varying $\lambda(x)$: Use a second MLP $\lambda_\psi(x) = e^{\phi_\psi(x)}$ with H^1 regularization

Identifiability and Ill-Posedness

Identifiability (definition)

λ is identifiable from observations $\{u(\boldsymbol{x}_k^d)\}$ if no two distinct $\lambda \neq \lambda'$ produce the same observed values

Structural conditions affecting identifiability:

- **Spatial coverage:** Measurement points must cover the region where λ varies
- **Measurements vs. unknowns:** Need far more observations than parameters (due to noise)
- **Sensitivity:** Small Jacobian $\partial u/\partial \lambda$ at measurement points $\Rightarrow$ effectively unobservable

Hadamard's conditions: existence, uniqueness, stability (most inverse problems violate at least one (typically stability))

Implicit regularization

Architectural bias: MLP truncates high frequencies
Early stopping $\approx$ Tikhonov regularization

Explicit regularization

$\widehat{\mathcal{L}}_{\text{reg}} = \widehat{\mathcal{L}}_{\text{inv}} + \mu\, \Omega_{\text{reg}}(\lambda_\psi)$:
$\|\lambda_\psi\|_{H^1}^2$ (smooth), $\|\lambda_\psi\|_{L^1}$ (sparse)

Error Structure in the Inverse Setting

The three-term decomposition extends to the inverse setting with additional coupling between state and parameter errors:

- **Approximation error:** Two hypothesis classes involved - $\hat{u}_\theta \in \hat{\mathcal{F}}_{m,L}$ and $\lambda_\psi \in \hat{\mathcal{G}}_{m',L'}$. Approximation error in λ_ψ is amplified by the *ill-posedness constant* κ_{inv}:

$$\|\lambda_\psi - \lambda^\star\| \lesssim \kappa_{\text{inv}} \left(\inf_{\phi \in \hat{\mathcal{F}}} \|\phi - u^\star\| + \inf_{\nu \in \hat{\mathcal{G}}} \|\nu - \lambda^\star\| \right)$$

  For ill-posed problems, κ_{inv} can be exponentially large

- **Generalization gap:** Dominated by the data term. Effective complexity $\sim \min(N_r, N_d)$. Adding more collocation points beyond $\mathcal{O}(N_d)$ yields diminishing returns

- **Optimization error:** Coupling of $\hat{u}_\theta$ and λ_ψ through $\mathcal{L}_{\lambda_\psi}[\hat{u}_\theta]$ creates non-separable, non-convex landscape. Local minima more prevalent than in forward setting. Good initialization (e.g. forward solve with nominal λ_0) significantly helps

Worked Example: Scalar λ (PyTorch)

Problem: $-\lambda\, u''(x) = \pi^2 \sin(\pi x)$ on $(0,1)$, $u(0){=}u(1){=}0$, $\lambda^\star{=}1$ unknown

Ansatz: $\hat{u}_\theta(x) = x(1{-}x)\,\phi_\theta(x)$ (hard BCs), $\lambda = e^{\psi}$ (positivity via unconstrained $\psi \in \mathbb{R}$)

```
phi     = MLP(width=32, depth=3)          # state network
log_lam = nn.Parameter(torch.tensor(0.)) # scalar: log(lambda)

def inv_pinn_loss(phi, log_lam, x_r, x_d, y_d):
    lam = torch.exp(log_lam)              # lambda > 0 always
    x   = x_r.clone().requires_grad_(True)
    u   = x * (1-x) * phi(x)              # hard BCs
    u_x  = torch.autograd.grad(u.sum(), x, create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x.sum(), x, create_graph=True)[0]
    res  = -lam * u_xx - f(x)             # PDE residual
    loss_r = (res**2).mean()
    loss_d = ((x*(1-x)*phi(x_d) - y_d)**2).mean()   # data misfit
    return gamma_r * loss_r + gamma_d * loss_d

optimizer = torch.optim.Adam([*phi.parameters(), log_lam], lr=1e-3)
```

Key points:

- Gradient w.r.t. ψ flows through e^{ψ} and u''_θ - single backward pass
- `create_graph=True`: second-order AD for u''

Observations

- Halving N_d: doubles parameter error
- $\gamma_d \gg \gamma_r$ + sparse data: overfits noise
- Monitor $\widehat{\mathcal{L}}_r$, $\widehat{\mathcal{L}}_d$ separately

From Scalar λ to Field $\lambda(x)$: What Changes?

When $\lambda(x)$ is spatially varying, the PDE becomes $-(\lambda(x)\,u'(x))' = f(x)$ and we need three changes:

Scalar $\lambda = e^{\psi}$

```
# one trainable scalar
log_lam = nn.Parameter(
            torch.tensor(0.))
lam = torch.exp(log_lam)

# residual: -lam * u'' - f
res = -lam * u_xx - f(x)
```

Field $\lambda(x) = e^{\phi_\psi(x)}$

```
# second MLP for log-conductivity
phi_lam = MLP(width=32, depth=3)
lam = torch.exp(phi_lam(x))

# residual: -(lam(x)*u')' - f
flux   = lam * u_x
flux_x = autograd.grad(flux.sum(),
           x, create_graph=True)[0]
res    = -flux_x - f(x)
```

Three key differences:

1. **Parameterisation:** Scalar $\psi \to$ MLP $\phi_\psi(x)$ (Positivity still via $\exp(\cdot)$)
2. **Residual:** $-\lambda u'' \to -(\lambda(x)u')'$ (Differentiate the *flux* $\lambda(x)u'(x)$, not just u)
3. **Regularisation:** Add H^1 penalty on λ to prevent overfitting:

```
# H^1 regularisation on lambda(x)
lam_x = autograd.grad(lam.sum(),
          x, create_graph=True)[0]
loss_reg = (lam_x**2).mean()

loss = (gamma_r * loss_r
      + gamma_d * loss_d
      + mu * loss_reg)
```

Optimizer

`Adam([*phi.parameters(), *phi_lam.parameters()], lr=1e-3)` - both networks trained jointly

Equation Discovery (SINDy)

Equation Discovery: Problem Statement

Parameter identification: PDE form known, only coefficients unknown
Equation discovery: Given data, identify the **full PDE from scratch**
Observe u at N_d space-time locations with noise $y_k = u(\boldsymbol{x}_k, t_k) + \varepsilon_k$
Hypothesize: $u_t = \mathcal{N}[u, u_x, u_{xx}, u^2, u\,u_x, \ldots]$
Goal: Which terms? What coefficients?

	Parameter identification	**Equation discovery**
PDE form known?	Yes	No
Unknowns	Scalar/field λ	Structure + coefficients
Solution space	Continuous	Combinatorial (which terms?)
Regularization	Tikhonov, L^2	Sparsity (L^1, thresholding)
Key challenge	Ill-posedness	Model selection / identifiability

Foundational method

SINDy (Brunton, Proctor, Kutz, 2016) - Sparse Identification of Nonlinear Dynamics
Originally for ODEs; extended to PDEs by Rudy et al. (2017)

SINDy: Three-Step Pipeline

1. **Derivative estimation** from noisy data:
   Central FD: $u_x \approx (u_{i+1} - u_{i-1})/(2h)$, error $\mathcal{O}(\sigma/h + h^2)$.
   Second derivatives: error $\mathcal{O}(\sigma/h^2 + h^2)$ - *noise amplified by* $1/h^k$.
   Optimal spacing: $h^\star \sim \sigma^{1/(2k)}$; min. error $\mathcal{O}(\sigma^{1-1/(2k)})$ ($= 10\%$ for $k = 2$, $\sigma = 1\%$, regardless of grid spacing!)

2. **Library construction:** Candidate terms $\varphi_1, \ldots, \varphi_p$; assemble: $[\boldsymbol{\Phi}]_{n,j} = \varphi_j(u(\boldsymbol{x}_n), u_x(\boldsymbol{x}_n), \ldots)$.

$$\boldsymbol{\Phi} = [1 \mid u \mid u^2 \mid u^3 \mid u_x \mid u_{xx} \mid u\,u_x \mid \ \ldots] \in \mathbb{R}^{N_d \times p}$$

3. **Regression:** Hypothesis: $u_t \approx \boldsymbol{\Phi}\boldsymbol{\xi}$ with sparse $\boldsymbol{\xi}$
   - **OLS (Ordinary least-squares):** gives dense $\hat{\boldsymbol{\xi}}$
   - **LASSO:** $\min_{\boldsymbol{\xi}} \frac{1}{2N_d} \|u_t - \boldsymbol{\Phi}\boldsymbol{\xi}\|_2^2 + \mu \|\boldsymbol{\xi}\|_1$
   - **STLS (Tresholding):** (i) OLS init; (ii) zero out $|\hat{\xi}_j| < \tau$; (iii) re-solve OLS on active set; repeat until stable

SINDy Example: Discovering Burgers' Equation

Setup:
Eq. $u_t + u\,u_x = \nu\,u_{xx}$, $\nu = 0.1$
Data: $n_x = 256$ spatial, $n_t = 100$ time points ($N_d = 25\,600$), $\sigma = 1\%$

Library ($p = 8$): $\Theta = [1 \mid u \mid u^2 \mid u^3 \mid u_x \mid u_{xx} \mid u\,u_x \mid u^2 u_x]$
STLS execution ($\tau = 0.05$):

- **Iter 0 (OLS):** Dense solution - true terms visible (u_{xx}: 0.097; $u\,u_x$: -0.994) but all 8 are nonzero
- **Iter 1:** 6 terms with $|\hat{\xi}_j| < 0.05$ eliminated; active set $= \{u_{xx}, u\,u_x\}$; re-solve on 2 terms $\Rightarrow$ (0.1004, -1.0021)
- **Iter 2:** both exceed threshold; converged

Recovered: $u_t = 0.1004\,u_{xx} - 1.0021\,u\,u_x$ (errors: 0.4% on ν, 0.2% on advection)

σ	$\hat{\xi}_{u_{xx}}$	$\hat{\xi}_{uu_x}$	False positives	$\left\|\hat{\boldsymbol{\xi}} - \boldsymbol{\xi}^\star\right\|_2$
10^{-3}	0.1000	-1.0001	0	0.0001
10^{-2}	0.1004	-1.0021	0	0.0023
10^{-1}	0.089	-0.941	2	0.078

SINDy: Identifiability

Identifiability conditions for SINDy:

- **Library completeness:** True PDE must lie in the column span of Φ
  If there are missing terms $\Rightarrow$ SINDy finds the best sparse approximation, which may be wrong
- **Library incoherence (RIP):** Columns of Φ not nearly collinear u_x and $u\,u_x$ can be highly correlated for certain solutions - genuine identifiability failure
- **Experimental design:** Choose initial condition to excite multiple scales; reduces column correlations

PINN-Based Equation Discovery: Direct Approach

Idea: Include unknown PDE coefficients $\boldsymbol{\xi}$ as *trainable parameters* alongside network weights θ

Joint loss (Raissi et al., 2019)

Parameterize PDE: $u_t = \sum_{j=1}^{p} \xi_j \, \varphi_j(u, u_x, u_{xx}, \ldots)$

$$\widehat{\mathcal{L}}(\theta, \boldsymbol{\xi}) = \underbrace{\frac{1}{N_r} \sum_i \left| (\hat{u}_\theta)_t - \textstyle\sum_j \xi_j \, \varphi_j(\hat{u}_\theta, \ldots) \right|^2}_{\text{PDE residual (unknown coefficients)}} + \underbrace{\frac{1}{N_d} \sum_k |\hat{u}_\theta - y_k|^2}_{\text{data fidelity}} + \underbrace{\mu \, \|\boldsymbol{\xi}\|_1}_{\text{sparsity}}$$

Optimize jointly over $(\theta, \boldsymbol{\xi})$ by gradient descent

Advantages

- Single optimization loop
- PDE structure regularizes $\hat{u}_\theta$
- AD replaces finite-difference derivatives

Challenges

- Non-convex in $(\theta, \boldsymbol{\xi})$ jointly
- Derivatives of $\hat{u}_\theta$ unstable early on
- L^1 penalty: proximal methods needed

Alternating Discovery: Phases 1 & 2

Idea: Decouple into phases and then alternate between state estimation and equation discovery

Phase 1 - State estimation (non-convex in θ, no physics)

$$\mathcal{L}_{\text{data}}(\theta) = \frac{1}{N_d} \sum_k |\hat{u}_\theta(\boldsymbol{x}_k, t_k) - y_k|^2$$

Fit smooth interpolant $\hat{u}_\theta$; compute $(\hat{u}_\theta)_t, (\hat{u}_\theta)_x, (\hat{u}_\theta)_{xx}$ via AD (no FD, no noise amplification)

Phase 2 - Equation discovery (**convex** in $\boldsymbol{\xi}$)

Freeze θ. Build library $\boldsymbol{\Phi}$ and target $(\hat{u}_\theta)_t$ from the network ($\boldsymbol{\Phi}$ is built using automatic differentiation)

$$\mathcal{L}_{\text{SINDy}}(\boldsymbol{\xi}) = \frac{1}{2N_r} \|(\hat{u}_\theta)_t - \boldsymbol{\Phi}\,\boldsymbol{\xi}\|_2^2 + \mu \|\boldsymbol{\xi}\|_1$$

Same LASSO as classical SINDy - full toolkit (STLS, cross-validation) applies

Running Phase 1 $\rightarrow$ Phase 2 only *once* is simplest, but errors in the state estimate propagate directly into the discovered equation

Solution: Add a refinement phase and iterate (next slide)

Alternating Discovery: Phase 3 & Iteration

Phase 3 - Physics-informed refinement (non-convex in θ)

Use discovered equation ($\hat{\boldsymbol{\xi}}$, active set $\mathcal{S}$) as a **physics regularizer**:

$$\mathcal{L}_{\text{refine}}(\theta) = \underbrace{\frac{1}{N_d}\sum_k |\hat{u}_\theta - y_k|^2}_{\text{data fidelity}} + \gamma_r \underbrace{\frac{1}{N_r}\sum_i \left|(\hat{u}_\theta)_t - \sum_{j\in\mathcal{S}} \hat{\xi}_j\, \varphi_j(\hat{u}_\theta, \ldots)\right|^2}_{\text{PDE residual with discovered equation}}$$

Two improvements from Phase 3:

- **Inter-data extrapolation:** PDE residual provides signal where no data exists
- **Derivative accuracy:** Physics-consistent $\hat{u}_\theta \Rightarrow$ better library in the next Phase 2

Full algorithm: Iterate Phase 1 $\rightarrow$ Phase 2 $\rightarrow$ Phase 3 $\rightarrow$ Phase 2 $\rightarrow$ Phase 3 $\rightarrow \cdots$ until active set $\mathcal{S}$ and coefficients $\hat{\boldsymbol{\xi}}$ stabilize (typically 2-5 outer iterations)

Alternating Discovery: Summary

Phase	Loss	Optimizes	Convex?	Physics used
1 (state est.)	$\mathcal{L}_{\text{data}}$	θ	No	None
2 (discovery)	$\mathcal{L}_{\text{SINDy}}$	$\boldsymbol{\xi}$	**Yes**	Library $\boldsymbol{\Phi}$
3 (refinement)	$\mathcal{L}_{\text{refine}}$	θ	No	Discovered PDE
Joint (direct)	$\widehat{\mathcal{L}}(\theta, \boldsymbol{\xi})$	$(\theta, \boldsymbol{\xi})$	No	Library $\boldsymbol{\Phi}$

Advantages of alternating

- Phase 2 is **convex** with guarantees
- Modular: any sparse regression method
- Progressive refinement loop

Limitations

- Outer loop convergence not guaranteed
- Phase 1 errors propagate to Phase 2
- Typically 2-5 outer iterations suffice

Example (Burgers) - same as slide 66:
Iter 1 $\rightarrow$ LASSO finds $\hat{\boldsymbol{\xi}}^{(1)} = (0, 0, 0, 0, 0, 0.08, -0.92, 0.03)$ (spurious last-term: $u^2\, u_{xx}$)
Iter 2 $\rightarrow$ PDE-refined surrogate eliminates spurious term: $\hat{\boldsymbol{\xi}}^{(2)} = (0, 0, 0, 0, 0, 0, 0.10, -1.00, 0)$

Limitations and When to Use PINNs

Known Limitations of PINNs

1. **Curse of dimensionality (revisited):** PINNs avoid mesh $\mathcal{O}(h^{-d})$ cost, but need $N_r = \mathcal{O}(\varepsilon^{-d/s})$ points for $W^{s,\infty}$ solutions $\implies$ *same exponent*
   Genuine advantage only when: effective dimension is low, PDE is separable, or $s \gg d$

2. **Stiff / multi-scale problems:** Spectral bias: PINN fits the smooth functions first, stagnates for high-freq. components

3. **Low accuracy vs. classical solvers ($d \leq 3$):** PINNs achieve $\approx 10^{-3}$-10^{-5} relative error, while high-order FEM/spectral: 10^{-8}-10^{-12}
   Root causes: (a) optimization error - local minima, loss plateaus; (b) approximation-optimization coupling - wider networks $\neq$ better solution

4. **Failure on non-smooth problems:** Strong-form L^2 residual is wrong for weak solutions
   PINNs may converge to a **non-entropic** weak solution for hyperbolic conservation laws
   Long-time integration: errors accumulate (use sequential / causal PINNs)
   Non-unique solutions: no mechanism to select the physical branch

When to Use PINNs vs. Classical Methods

Setting	Prefer PINN	Prefer FEM/FDM
Dimension d	$d \geq 10$	$d \leq 3$
Solution regularity	Smooth ($s \gg d$)	Low regularity, shocks
Domain geometry	Complex / no mesh	Simple / structured
Inverse problem	Yes	Harder to formulate
Many parameter instances	Yes (amortized)	Expensive (per-instance)
Rigorous error bounds needed	No	Yes
Forward accuracy ($d \leq 3$)	10^{-3}-10^{-5}	10^{-8}-10^{-12}
Conservation laws with shocks	Risky	Standard (upwinding)
Safety-critical applications	Not yet	Established

PINNs in 2026: A powerful niche tool

Best for: high-dimensional, smooth, parametric, or inverse problems
Most productive: **alongside** classical solvers:
- PINNs for fast approximation / UQ
- FEM for ground-truth verification