Neural Networks and Partial Differential Equations

AI for Mathematics (General Elective) — Week 14

7 July 2026

The curse of dimensionality

Partial differential equations

A **partial differential equation (PDE)** relates an unknown function of several variables to its partial derivatives. Two problems are asked:

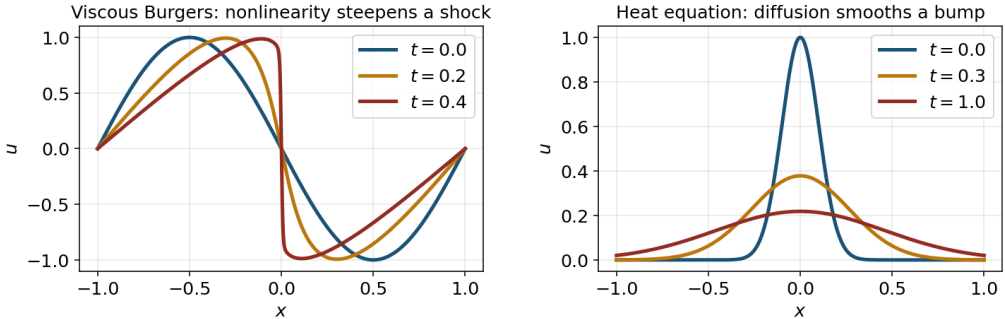
Forward the equation and its coefficients are known; find u.

Inverse some measurements of u are known, but a coefficient, a source term, or the equation itself is not; find the missing piece.

Three running examples for the lecture:

- **Poisson** $-\Delta u = f$ on $[0,1]$ with $u(0) = u(1) = 0$ — the simplest steady-state PDE. Δ is the Laplacian, the sum of second derivatives.
- **Viscous Burgers** $u_t + u\,u_x = \nu\,u_{xx}$. The nonlinear term $u\,u_x$ steepens the profile into a near-shock; the viscosity $\nu\,u_{xx}$ smooths it. The standard neural-solver benchmark.
- **Black–Scholes / Hamilton–Jacobi–Bellman (HJB)** in d dimensions — option pricing on d assets, optimal control with d state variables. Genuinely high-dimensional.

What the solutions look like: shock and diffusion



Two behaviors that recur all lecture. **Left:** viscous Burgers ($u_t + u\,u_x = \nu\,u_{xx}$, the running benchmark). The nonlinear term steepens a smooth profile into a near-shock; the small viscosity keeps it finite. **Right:** the heat equation ($u_t = \Delta u$). Diffusion smooths every bump. Real PDEs mix the two. The sharp features on the left are what strain a solver.

Classical solvers and the curse of dimensionality

Classical methods discretize the domain and solve a large algebraic system:

- **finite-difference method (FDM)** — derivatives by difference quotients on a grid;
- **finite-element method (FEM)** — local basis functions on a mesh;
- **spectral methods** — global basis functions, very accurate for smooth solutions.

All place about N points per spatial dimension, hence N^d unknowns in d dimensions — the **curse of dimensionality**. For $d \leq 3$ these methods are fast and accurate; for d in the tens or hundreds they are impossible.

Two settings where a network helps. **High dimension**: no grid, so not automatically N^d. **Inverse problems**: sparse or noisy data. For smooth low-dimensional forward problems, classical methods stay faster and more accurate.

The curse in numbers

For basket Black–Scholes (d assets) or the Schrödinger equation ($3k$ dimensions for k electrons), d runs into the tens or hundreds. A grid with N points per axis has N^d nodes:

dimension d	nodes at $N = 10$	nodes at $N = 100$
1	10	10^2
2	10^2	10^4
3	10^3	10^6
10	10^{10}	10^{20}
100	10^{100}	10^{200}

At $d = 100$ (last row), the node count 10^{100} exceeds the $\approx 10^{80}$ atoms in the observable universe. No finer grid and no faster computer changes this; the obstruction is the exponent d. Monte-Carlo methods replace the grid by M random samples, with error $\propto 1/\sqrt{M}$, *independent of* d.

What makes a PDE hard

Four independent sources of difficulty, each defeating a different classical method:

High dimension the curse above; grids die.

Nonlinearity terms multiply the unknown (Burgers' $u\,u_x$); the algebraic system is nonlinear and solutions can form shocks.

Sharp features shocks, boundary layers, high-frequency oscillation; a uniform grid must be fine everywhere to resolve them.

Irregular geometry a complicated domain is expensive to mesh.

A real problem may combine several. Neural methods target high dimension and mesh-free nonlinearity. Sharp features remain hard.

Solving one PDE with a network

Automatic differentiation

A PDE is a statement about derivatives of u. We write $u_\theta(x,t)$ for a neural network with trainable weights θ used to approximate the PDE solution. This notation is used throughout the lecture. To check whether u_θ satisfies the PDE one needs u_x, u_{xx}, u_t, and so on. **Automatic differentiation (AD)** computes them *exactly*. Because u_θ is a known composition of elementary operations, the chain rule is applied mechanically through its computational graph.

- Not finite differences (no truncation error) and not symbolic algebra (no expression blow-up).
- The same machinery that trains a network (derivatives with respect to the weights) here also gives derivatives with respect to the *inputs*.

Cheap exact automatic differentiation is the enabling technology. The founding idea is from 1998. The field became active around 2017, when the differentiation libraries (TensorFlow, PyTorch, JAX) matured.

Writing the solution as a network

A PDE solution $u(x,t)$ is a function: given any point in the domain, return the field value there.

Classical: discretize the domain; solve a linear system for u at N grid points. The solution is N numbers.

Neural: represent u as a network $u_\theta(x,t)$ — plug in the coordinates, get the field value out. Find θ by training. After training, $u_\theta(x_0,t_0)$ — one forward pass — gives the solution at any point.

Lagaris, Likas, Fotiadis (1998) — the first paper to do this for ODEs and PDEs. Write $u(x) = A(x) + F(x)\,N_\theta(x)$: A satisfies the boundary conditions exactly, F vanishes on the boundary. Train at fixed **collocation points** (locations where the PDE must hold) to minimize the residual. Predated deep learning and modern automatic differentiation; revived in 2019.

Physics-informed neural networks (Raissi, Perdikaris, Karniadakis, 2019)

"Physics-informed": the known PDE is written into the training loss as a residual penalty. No labeled solution data are needed. Let u_θ be a **multilayer perceptron (MLP)**: inputs (x, t), scalar output u, smooth activation $\tanh$ (so the derivatives the PDE needs exist).

For a PDE $\mathcal{N}[u] = 0$ (Burgers: $\mathcal{N}[u] = u_t + u\,u_x - \nu\,u_{xx}$), define the **residual** $r_\theta = \mathcal{N}[u_\theta]$, every derivative by automatic differentiation. The loss is

$$L(\theta) = \underbrace{\tfrac{1}{N_r}\textstyle\sum_i r_\theta(x_i, t_i)^2}_{\text{PDE residual}} + \lambda_b \underbrace{\tfrac{1}{N_b}\textstyle\sum_j (u_\theta - u_{\mathrm{BC}})^2}_{\text{boundary}} + \lambda_0 \underbrace{\tfrac{1}{N_0}\textstyle\sum_k (u_\theta - u_{\mathrm{IC}})^2}_{\text{initial}}.$$

(N_r: number of interior collocation points; N_b: boundary points; N_0: initial-condition points.) The boundary condition (BC) and initial condition (IC) enter as penalties — **soft constraints**, enforced only approximately, with weights λ_b, λ_0.

PINN: input, output, objective

A physics-informed neural network (PINN):

Input: the PDE operator, the domain, the boundary and initial conditions, and collocation points. For a forward problem *no solution data are needed* — the equation supplies the supervision.

Output: a trained network u_θ approximating the solution continuously over the whole domain.

Objective: minimize the PDE residual plus the boundary/initial mismatch, by Adam (an adaptive-learning-rate gradient method) then L-BFGS (a curvature-using quasi-Newton method for the final sharp convergence).

Benchmark. Burgers with $\nu = 0.01/\pi$, initial condition $u(0, x) = -\sin(\pi x)$ on $[-1, 1]$, zero boundary values. A steep front forms near $x = 0$ at $t \approx 0.4$. The PINN recovers the solution, front included, to a relative L^2 error on the order of 10^{-3} against a high-resolution reference.

The PINN network for Burgers: architecture

The trial function $u_\theta(x,t)$ for the Burgers benchmark is a feed-forward network:

Input: $(x,t) \in \mathbb{R}^2$ — one space coordinate, one time value.

Hidden layers: four layers of 64 neurons each; $\tanh$ activation. $\tanh$ is differentiable to all orders — the PDE residual needs u_x, u_{xx}, and u_t via automatic differentiation.

Output: $u_\theta(x,t) \in \mathbb{R}$ — the approximate velocity at (x,t).

Parameters: $12{,}737$ weights and biases.

A forward pass at any (x,t) gives the approximate solution. The domain $[-1,1] \times [0,1]$ is never discretized into a grid; collocation points are sampled randomly from it during training.

A worked example: a PINN for the Poisson problem

Choose $u^*(x) = \sin(\pi x)$ and derive the forcing term: $f(x) = -u^{*\prime\prime}(x) = \pi^2 \sin(\pi x)$. The PDE to solve is $-u''(x) = \pi^2 \sin(\pi x)$ on $[0, 1]$, $u(0) = u(1) = 0$.

The network $u_\theta(x)$ sees only the **equation and boundary conditions** — not u^*. The residual is $r_\theta(x) = u''_\theta(x) + \pi^2 \sin(\pi x)$ ($\pi^2 \sin(\pi x)$ is the known input f, not the unknown solution). The loss is

$$L(\theta) = \frac{1}{N_r} \sum_i r_\theta(x_i)^2 \; + \; u_\theta(0)^2 \; + \; u_\theta(1)^2.$$

After training, compare u_θ against the known $u^* = \sin(\pi x)$ to measure the error. u^* enters only here, as a yardstick.

A spectral method solves the same problem to machine precision almost instantly. The PINN is slower and less accurate on this 1-D problem.

Soft and hard constraints

The boundary condition can enter the network in two ways.

Soft add it as a penalty term in the loss — the standard PINN. Flexible for any geometry, but it trades off against the residual and is a source of training trouble.

Hard build it into the architecture so it holds exactly. For $u(0) = u(1) = 0$, write $u_\theta(x) = x(1-x)\,\mathrm{MLP}(x)$, which vanishes at both ends by construction.

Hard constraints remove one loss term and the weight that goes with it. The cost is a construction tailored to each domain. Soft constraints need no such construction, but leave the boundary only approximately satisfied.

Collocation and adaptive sampling

The residual is enforced at sampled interior points — the collocation points.

- Uniform random sampling spends as many points where the solution is smooth as where it is sharp.
- Residual-based adaptive refinement adds points where the residual is currently large — near shocks and boundary layers.

The sharp regions are exactly where the network is least accurate. Concentrating points there is what makes a PINN usable on problems with fronts.

The optimizer and the loss landscape

The loss is non-convex in the weights θ, so there is no guarantee gradient descent reaches the global minimum.

- In practice: Adam first (a robust adaptive-gradient method), then L-BFGS (a curvature-using method) to drive the residual down sharply at the end.
- The loss weights λ_b, λ_0 on the boundary and initial terms matter: set wrong, one term dominates and the others are barely trained.

An approximation theorem can promise a small accurate network exists. Yet the optimizer may not find it. The optimizer, not the network's approximation power, is usually what limits a PINN.

PINN convergence: consistency (Shin, Darbon, Karniadakis 2020)

Shin, Darbon, Karniadakis, *Commun. Comput. Phys.* 28 (2020). PDE class: linear second-order elliptic and parabolic with **sign condition** $\tilde{c}(x) = \sum_i D_i b_i(x) + d(x) \leq 0$ (needed for the maximum principle). Here $D_i = \partial/\partial x_i$ are first-order differential operators, b_i their coefficient functions, $d(x)$ the zeroth-order term.

Let h_{m_r} minimize the PINN loss within a network class chosen with m_r collocation points, sampled i.i.d. from a fixed distribution on the domain U. As $m_r \to \infty$ (probabilities under the joint law of the samples):

- **Theorem 3.2 — PINN loss rate.** With high probability, $\mathrm{Loss}^{\mathrm{PINN}}(h_{m_r}) = O\left(m_r^{-\alpha/d}\right)$, where $\alpha \in (0, 1)$ is the Hölder exponent of the PDE data and d is the spatial dimension.
- **Theorem 3.3 — consistency.** Almost surely, $h_{m_r} \to u^*$ uniformly on $\overline{U}$ (convergence in C^0). No rate is proved for the C^0 solution error.
- **Theorem 3.4.** If the network class enforces the boundary condition exactly (hard constraint), the convergence strengthens to $H^1(U)$ (functions whose weak gradient is in L^2).

PINN error bound (Mishra, Molinaro 2020)

Mishra, Molinaro, arXiv:2006.16144; *IMA J. Numer. Anal.* 43 (2023).

Theorem 2.6. Let u_θ^* be a PINN trained on N collocation points, $\|\cdot\|_X$ a norm on the solution (e.g. $L^2(D)$ for elliptic, $L_t^2 L_x^2$ for parabolic), $p \geq 1$ the residual exponent in the loss. Then

$$\|u - u_\theta^*\|_X \;\leq\; C_{\mathrm{pde}}\, E_T \;+\; C_{\mathrm{pde}}\, C_{\mathrm{quad}}^{1/p}\, N^{-\kappa/p},$$

Small training error $\Rightarrow$ small solution error only when C_{pde} is moderate. An unstable PDE can have a tiny residual yet a large solution error.

PINN error bound: the terms

In the bound of Theorem 2.6:

- $y_1, \ldots, y_N$ are the collocation points, $w_i \geq 0$ their quadrature weights.
- $E_T = \left(\sum_i w_i |r_{\theta^*}(y_i)|^p\right)^{1/p}$ is the computable training residual.
- κ is the quadrature convergence rate (the paper's α, renamed to avoid collision with the Hölder α).
- For Monte Carlo points $\kappa = \frac{1}{2}$, giving $O(N^{-1/(2p)})$.
- C_{pde} is the PDE *stability constant*: how much u changes per unit residual.
- C_{quad} is the quadrature constant.

The Deep Ritz method (E & Yu, 2018)

Many PDEs are the condition for a function to *minimize an energy*. The Poisson problem $-\Delta u = f$ with zero boundary values has the same solution as the minimizer of the **Dirichlet energy**

$$I[u] = \int \left(\tfrac{1}{2} \left| \nabla u \right|^2 - f\, u \right) dx.$$

Deep Ritz replaces u by a network u_θ. It estimates the integral by Monte-Carlo sampling, then minimizes $I[u_\theta]$ over θ. The name is from the **Ritz method** of the calculus of variations: approximating an energy's minimizer within a chosen family, here a deep network.

- Advantage: the energy uses only the first derivative ∇u, not Δu — numerically gentler.
- Cost: a variational (energy) formulation must exist, and the boundary condition is no longer automatic.
- The **Deep Galerkin Method (DGM)** (Sirignano & Spiliopoulos, 2018) is the strong-form sibling. It fits the operator and boundary/initial data at randomly sampled points, not a mesh. Shown in up to 200 dimensions.

Weak and adversarial formulations (Zang, Bao, Ye, Zhou, 2020)

Input: PDE in weak form; domain; boundary/initial conditions.

Output: trained primal network u_θ satisfying the weak PDE.

Objective: saddle-point — u_θ minimizes, φ_η maximizes the weak residual; train both networks jointly (GAN-style).

The **weak form** multiplies the PDE residual by a **test function** φ and integrates by parts onto φ. A **weak adversarial network (WAN)**: u solves the PDE if and only if this weak residual is zero against *every* φ.

This is rewritten as a min–max (saddle-point) problem:

- the solution is a **primal** network u_θ;
- the worst-case test function is an **adversarial** network φ_η;
- u_θ is trained to minimize, and φ_η to maximize, the weak residual — the structure of a generative adversarial network.

It is mesh-free and aimed at high-dimensional PDEs on irregular domains. The route is distinct: the weak form (one fewer derivative, like Deep Ritz) combined with a *learned* test function in place of a fixed energy.

Backward SDE: the problem

A **forward SDE** gives $X_0 = x_0$ and integrates forward in time. The path is $\mathcal{F}_t$-adapted by construction: it depends only on W_s, $s \le t$.

A **backward SDE (BSDE)** replaces the initial condition with a *terminal* condition and asks for the rest. Find two $\mathcal{F}_t$-adapted (depending only on past W_s) processes (Y_t, Z_t) satisfying

$$dY_t = -f(t, X_t, Y_t, Z_t)\, dt + Z_t \cdot dW_t, \qquad Y_T = g(X_T).$$

- $Y_t \in \mathbb{R}$: the value process.
- $Z_t \in \mathbb{R}^d$: the martingale integrand — a second unknown, not auxiliary. The right Z_t steers $\int Z_t\, dW_t$. Then Y_T hits the target $g(X_T)$ using no future information.

Simplest case ($f = 0$): $Y_t = \mathbb{E}[g(X_T) \mid \mathcal{F}_t]$ (a martingale); Z_t is given by the martingale representation theorem.

Pardoux–Peng (1990): for f Lipschitz in (y, z) and $g(X_T) \in L^2(\Omega)$, the BSDE has a unique square-integrable adapted solution (Y, Z).

From PDE to BSDE: Itô's formula

Target: a semilinear parabolic PDE for $u : \mathbb{R}^d \times [0,T] \to \mathbb{R}$,

$$\frac{\partial u}{\partial t} + \mathcal{L}u + f\Big(t, x, u, \sigma^\top \nabla u\Big) = 0, \quad u(T,x) = g(x),$$

where $\mathcal{L}u = \mu \cdot \nabla u + \frac{1}{2}\operatorname{tr}(\sigma\sigma^\top \nabla^2 u)$ is the generator of the SDE $dX_t = \mu\, dt + \sigma\, dW_t$.

Apply Itô's formula to $u(t, X_t)$ along a path of the SDE:

$$d\big[u(t,X_t)\big] = \Big(\tfrac{\partial u}{\partial t} + \mathcal{L}u\Big)dt + (\sigma^\top \nabla u) \cdot dW_t.$$

The PDE says $\frac{\partial u}{\partial t} + \mathcal{L}u = -f(t, X_t, u, \sigma^\top \nabla u)$, so:

$$d\big[u(t,X_t)\big] = -f\Big(t, X_t,\, u(t,X_t),\, \sigma^\top \nabla u(t,X_t)\Big)\, dt + \sigma^\top \nabla u(t,X_t) \cdot dW_t.$$

Set $Y_t := u(t, X_t)$ and $Z_t := \sigma^\top \nabla u(t, X_t)$. This is exactly a **backward SDE (BSDE)**:

$$dY_t = -f(t, X_t, Y_t, Z_t)\, dt + Z_t \cdot dW_t, \qquad Y_T = g(X_T).$$

To recover $u(0, x_0) = Y_0$: simulate sample paths of X_t, and solve this BSDE along each. This is Monte-Carlo, not an N^d grid over $\mathbb{R}^d$.

Deep BSDE: the PDE–BSDE equivalence (Han, Jentzen, E, 2018)

Here a network does what no grid can: solve nonlinear parabolic PDEs in hundreds of dimensions. The mechanism is a classical equivalence.

A **semilinear parabolic PDE** has the form $\partial_t u + \mathcal{L}u + f(t, x, u, \sigma^\top \nabla u) = 0$. The generator $\mathcal{L}u = \mu \cdot \nabla u + \frac{1}{2}\mathrm{tr}[\sigma\sigma^\top \nabla^2 u]$ is the same as on the previous frame. $\mathcal{L}$ is linear. The nonlinearity is f, which may depend nonlinearly on u and $\sigma^\top \nabla u$.

- The **nonlinear Feynman–Kac formula** (derived on the previous frame): the BSDE solution is the PDE solution evaluated *along* the random path X_t — $Y_t = u(t, X_t)$ and $Z_t = \sigma^\top \nabla u(t, X_t)$. So (Y_t, Z_t) is a stochastic process. The number we want is its start value $u(0, x_0) = Y_0$.

Deep BSDE: the method

Discretize time $0 = t_0 < \cdots < t_{N_t} = T$ into N_t steps. Simulate forward paths by the **Euler–Maruyama scheme**: $X_{t_{n+1}} = X_{t_n} + \mu(t_n, X_{t_n})\Delta t + \sigma(t_n, X_{t_n})\,\Delta W_n$, $\Delta W_n \sim \mathcal{N}(0, \Delta t\, I_d)$. Then:

- at each step approximate the integrand $Z_{t_n} = \sigma^\top \nabla u(t_n, X_{t_n})$ by a **small network**, one per step. Stacked, these form a single deep network across time.
- the unknown start value $Y_0 = u(0, x_0)$ is a trainable scalar;
- run the BSDE forward along each path with the network-supplied Z; the loss is the *expected* squared terminal mismatch $\mathbb{E}\,|Y_{t_{N_t}} - g(X_{t_{N_t}})|^2$.

Input: the PDE (drift, diffusion, nonlinearity, terminal condition) and a starting point x_0.

Output: the value $u(0, x_0)$, and $\sigma^\top \nabla u$ along sampled paths.

Objective: minimize the expected terminal mismatch. The BSDE solution is unique (Pardoux–Peng), so its only minimizer is the true (Y_0, Z) — hence the trained scalar returns $u(0, x_0)$.

Deep BSDE: 100 dimensions

The method was demonstrated in $d = 100$ dimensions on three equations:

- nonlinear Black–Scholes (option pricing on d assets);
- Hamilton–Jacobi–Bellman (HJB: optimal control, d state variables);
- Allen–Cahn ($\partial_t u = \Delta u + u - u^3$, phase-field model for interface motion between two stable phases).

It avoids the grid's N^d blow-up by sampling paths, not nodes: the per-path **Monte-Carlo** error $\sim 1/\sqrt{M}$ is independent of d.

The harder question: does the network representing Z also stay small in d? Deep BSDE has only an a posteriori error bound, not an end-to-end proof.

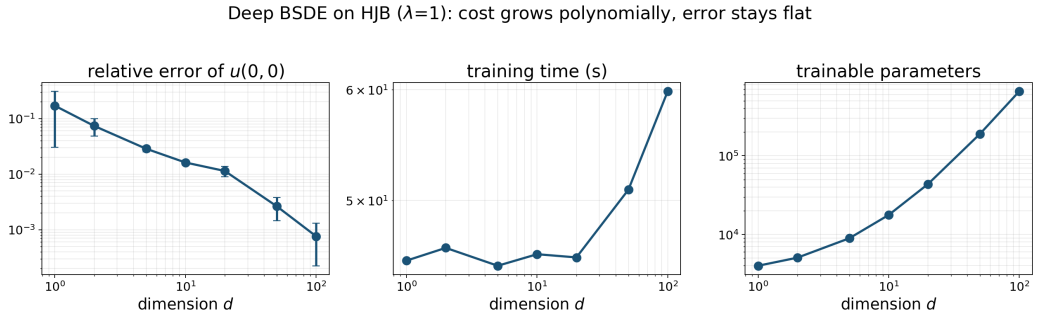
Deep BSDE on the cluster (exp41): HJB in $d = 10$ to 100

Equation (linear-quadratic HJB, Han–Jentzen–E 2018, §4.2):
$\partial_t u + \Delta u - \lambda \|\nabla u\|^2 = 0$ on $\mathbb{R}^d \times [0,1]$, terminal condition
$u(1,x) = g(x) = \ln\!\left(\frac{1+\|x\|^2}{2}\right)$, λ=1. Quantity of interest: $u(0,\mathbf{0})$ at the origin.
Cole–Hopf exact solution $u(t,x) = -\frac{1}{\lambda}\ln \mathbb{E}\big[\exp(-\lambda\, g(x+\sqrt{2}\,W_{T-t}))\big]$, evaluated by 10^7-sample Monte Carlo at every d. **Our run (XMU HPC, exp41).** N_t=20 time steps, 2,500 Adam iterations, 5 seeds, 8 CPU cores.

d	rel. error (ours)	wall time
10	1.6%	46 s
50	0.26%	51 s
100	0.076%	60 s

Error stays below 2% across all dimensions tested. Wall time grows only from 46 s to 60 s. No grid is built. Training cost scales with the number of network parameters, not with the N^d grid a classical solver would need.

The dimension sweep as a picture



Log–log against d: relative error (left), training time (middle), parameter count (right). A grid for $d = 100$ would need 10^{100} nodes. The network trains to 0.076% in one minute on 8 CPU cores. The paper's result: 0.17% in 330 s on a 2017 laptop.

Two families for high-dimensional PDEs

High-dimensional PDE solvers split into two families (E, Han, Jentzen, review, 2021):

Deep learning, high-d proven empirically represent the solution by a network and train it. Demonstrated in hundreds of dimensions: **Deep BSDE** (semilinear parabolic) and **Deep Galerkin Method** (DGM, up to 200-d free boundary problems, Sirignano–Spiliopoulos 2018). *Not on this list*: Deep Ritz, which degrades already by $d \approx 10$ on eigenvalue problems.

Nonlinear Monte Carlo stochastic recursions with no network — in particular multilevel Picard iteration (next frame: the only family with end-to-end curse-free proofs).

Both sample paths rather than gridding space, so both can avoid the N^d cost. They differ in what carries the approximation: a trained network, or a fixed Monte-Carlo recursion.

Multilevel Picard: a proven curse-free method

Mechanism.

- Feynman–Kac rewrites the PDE as a fixed-point equation $u = \Phi(u)$.
- The iteration $u_{k+1} = \Phi(u_k)$ is expanded as a *multilevel* sum of corrections; each level estimates only the change from the level below.
- Nested Monte-Carlo evaluates it: many cheap samples on coarse levels, few expensive ones on fine. No network — the recursion carries the approximation.

Theorem (Hutzenthaler, Jentzen, Kruse, Nguyen, von Wurstemberger, Proc. R. Soc. A, 2020). For semilinear heat equations on a bounded time horizon T, with *gradient-independent* Lipschitz nonlinearity $f(t, x, u)$ and Lipschitz terminal g, multilevel Picard approximates $u(0, x)$ to mean-square error $\left(\mathbb{E}|U^{\mathrm{MLP}} - u(0,x)|^2\right)^{1/2} \leq \varepsilon$. The cost grows only *polynomially* in d and $1/\varepsilon$ — a complete proof these PDEs escape the curse. Gradient-dependent extension: Hutzenthaler–Jentzen–Kruse, Found. Comp. Math. 2022.

Open: which PDEs this extends to, and whether training reaches the small network the *approximation* theorems guarantee exists.

The dimension ladder

Three successive algorithmic ideas pushed the empirical record from $d = 100$ to $d = 10^5$. Each fixed a different computational bottleneck. "Theorem?" = end-to-end error bound proven (not just convergence in principle).

d	method	source	theorem?
100	Deep BSDE	Han–Jentzen–E, PNAS 2018	a posteriori only
$10{,}000$	deep splitting	Beck et al., SISC 2021	no
$100{,}000$	SDGD-PINN	Hu et al., NN 2024	no
$1{,}000$	multilevel Picard	Hutzenthaler et al., PRSA 2020	**yes**
$10{,}000$	multilevel Picard	Neufeld–Wu, SPDE-AC 2025	**yes**

- Rows 1–3: neural-network methods (semilinear parabolic / HJB). Empirical records; the last column shows no end-to-end bound exists yet.
- Rows 4–5: **multilevel Picard** — a Monte Carlo algorithm, not a neural network. Its proofs cover d in the thousands with explicit rates.

$d = 100 \to 10{,}000$: deep splitting (Beck et al., SISC 2021)

Input: semilinear PDE + terminal condition in d dimensions.

Output: approximate $u(t, x)$ at any queried (t, x).

Objective: split time into steps; each step is an independent regression (no backpropagation through time).

Bottleneck of Deep BSDE at large d: it backpropagates through all N_t chained subnetworks at once. The whole path's activations and per-step parameters are held in memory simultaneously — the joint computational graph blows up at $d = 10^4$.

Fix (Beck et al., SISC 2021): train one **fresh small network** $V_n(x) \approx u(t_n, x)$ per step. Each is trained independently, by a single least-squares regression on simulated paths:

$$\text{target:} \quad V_{n+1}(X_{t_{n+1}}) + \Delta t\, f(V_{n+1}, \sigma^\top \nabla V_{n+1}),$$

marching backward from the terminal data $V_N = g$. No backpropagation through time; each step is a self-contained regression.

Result: relative error 1.0×10^{-3} at $d = 10{,}000$ on $\partial_t u + \Delta u - \|\nabla u\|^2 = 0$, in $1{,}688\,\text{s}$ on a 2019 gaming GPU (against a Cole–Hopf Monte Carlo reference).

$d = 10{,}000 \to 100{,}000$: stochastic dimension gradient descent (SDGD-PINN)

Input: high-dimensional PDE, random coordinate subset I per step.

Output: u_θ satisfying the PDE.

Objective: use an unbiased Laplacian estimator (below) at cost $O(|I|) \ll O(d)$ instead of all d second-derivative terms.

Bottleneck of standard PINN at large d: the residual needs $\Delta u_\theta = \sum_{i=1}^{d} \partial^2 u_\theta / \partial x_i^2$ — d second-derivative evaluations per step ($O(d)$). This is prohibitive at $d = 10^5$.

Fix (Hu et al., Neural Networks 2024): at each step sample a *uniformly* random subset $I \subset \{1, \dots, d\}$ of coordinates. Use

$$\widehat{\Delta} u_\theta = \frac{d}{|I|} \sum_{i \in I} \frac{\partial^2 u_\theta}{\partial x_i^2}$$

as an unbiased estimator of Δu_θ.

Result: HJB and Schrödinger-type equations in $100{,}000$ effective dimensions in 12 h on a single GPU. No end-to-end error theorem exists for these numbers.

Deep splitting on the cluster (exp46, A100 GPU)

Beck–Becker–Cheridito–Jentzen–Neufeld (SISC 2021).

Equation. HJB $\partial_t u + \Delta u - \|\nabla u\|^2 = 0$ on $\mathbb{R}^d \times [0,1]$, terminal condition $u(1,x) = \varphi(x) = \|x\|^{1/2}$. **QoI**: $u(0,\mathbf{0})$ at the origin; reference by 10^7-sample Monte Carlo.

d	rel. error (ours)	rel. error (SISC 2021)
100	0.20%	0.14%
1,000	0.30%	0.022%
10,000	did not converge	0.10%, 1,688 s

At $d = 100$ and $d = 1{,}000$ the method works; both errors are well under 1%. The difference at $d = 1{,}000$ (0.30% vs 0.022%) reflects hyperparameter sensitivity (the paper uses a longer tuning schedule). At $d = 10{,}000$ we did not reproduce the paper's result. The original run used a 2019 gaming GPU with specific batch-normalisation tuning.

SDGD-PINN on the cluster (exp47, A100 GPU)

Hu–Shukla–Karniadakis–Kawaguchi (Neural Networks 2024).
Equation (Hu et al. §5.1). Sine-Gordon-type semilinear elliptic $\Delta u + \sin(u) = f(x)$ on the unit ball $\{x \in \mathbb{R}^d : \|x\| \le 1\}$, f chosen so a specified inseparable manufactured solution u^* holds. **QoI**: relative L^2 error against u^* on a held-out set of $20{,}000$ points. SDGD samples $d_{\text{sub}} = 4$ Laplacian terms per step (instead of all d).

d	vanilla PINN (rel. L^2)	SDGD (rel. L^2)
100	3.6%	0.63%
1,000	infeasible	0.059%
10,000	infeasible	0.24%

Both columns are our runs. Vanilla PINN becomes infeasible above $d = 100$: computing all d Laplacian terms costs $O(d)$ per step. SDGD samples only $d_{\text{sub}} = 4$ terms, staying feasible at any d.

Multilevel Picard on the cluster (exp45, NumPy CPU)

Pure NumPy re-implementation of the numerical multilevel-Picard scheme of Becker et al. (Commun. Comput. Phys. 28, 2020); no neural network.
Equation class. Semilinear heat $\partial_t u + \Delta u + f(u) = 0$ on $\mathbb{R}^d \times [0, 1)$, terminal $u(1, x) = g(x)$. Allen–Cahn: $f(u) = u - u^3$. Semilinear heat (paper's row): f a sigmoid-type nonlinearity. **QoI**: $u(0, \xi)$ at a fixed reference point ξ (the value below).

equation	d	our value	CiCP 2020
Allen–Cahn	10	0.29589	0.29555
Allen–Cahn	1,000	0.00339	0.00339
Semilinear heat	10,000	0.28427	0.28433

Table 1 values reproduced to 3–4 significant figures. Each case runs in seconds on a single CPU core. This is the only solver on the dimension ladder with an end-to-end polynomial-cost proof — and it is not a neural network.

A theorem: networks overcome the curse (Grohs, Hornung, Jentzen, von Wurstemberger, 2018)

Statement. Consider linear Kolmogorov PDEs $\partial_t u + \frac{1}{2}\mathrm{tr}(\sigma\sigma^\top \nabla^2 u) + \mu \cdot \nabla u = 0$. Assume *affine* drift $\mu(x)$ and *affine* diffusion $\sigma(x)$ (Black–Scholes is one case), Lipschitz payoff g at $t = T$, on a bounded box $[a, b]^d$. For every $\varepsilon > 0$ there is a neural network u_θ with

$$\|u(0,\cdot) - u_\theta\|_{L^2([a,b]^d)} \le \varepsilon, \qquad \#\text{parameters of } u_\theta \;\le\; C \cdot d^{c_1}\, \varepsilon^{-c_2},$$

for absolute constants C, c_1, c_2 (independent of d).

A grid resolving each of the d axes to accuracy ε needs $\sim\varepsilon^{-1}$ nodes per axis, i.e. ε^{-d} in all — the curse. The theorem proves a network's representational cost is instead $\mathrm{poly}(d, 1/\varepsilon)$.

Caveat. Existence only: a small accurate network exists. The theorem does not say gradient descent finds it. It does not bound training time. (arXiv 2018; Memoirs of the American Mathematical Society 284(1410), 2023.)

A posteriori error bound from the training loss (Han, Long, 2020)

In $d = 100$ there is usually no exact solution to compare against. A theorem closes this gap for Deep BSDE.

Han–Long (Probability, Uncertainty and Quantitative Risk 5:5, 2020). For coupled forward–backward SDEs satisfying the paper's structure hypotheses (Lipschitz μ, σ, f, g *plus* a monotonicity condition on (μ, σ, f) guaranteeing a unique solution), the Deep BSDE approximation (Y^θ, Z^θ) with N_t Euler–Maruyama steps satisfies

$$\underbrace{\left\|Y^* - Y^\theta\right\|^2 + \left\|Z^* - Z^\theta\right\|^2}_{\text{distance to the true solution}} \lesssim \underbrace{\mathbb{E}\left|Y_T^\theta - g(X_T)\right|^2}_{\text{training loss}} + \underbrace{\frac{1}{N_t}}_{\text{time-disc. error}} .$$

The right-hand training objective is computable from simulated paths — an *a posteriori* bound: small training loss ⇒ provably close to the true solution.

Caveats. Lipschitz + $g \in L^2$ alone is not enough — the monotonicity hypothesis is required; the constant is non-explicit; observed non-convergence cases in stochastic control are explained by Negyesi–Huang–Oosterlee, 2024 (arXiv:2403.18552).

The positive ladder, by strength of statement

1. **Approximation.** Networks of size $\mathrm{poly}(d, 1/\varepsilon)$ *exist* for affine-coefficient Kolmogorov PDEs in $L^2([a,b]^d)$ (Grohs–Hornung–Jentzen–von Wurstemberger, Memoirs of the AMS 284(1410), 2023) and for semilinear heat equations (Hutzenthaler–Jentzen–Kruse–Nguyen, 2020).
2. **Generalization.** Empirical risk minimization needs only $\mathrm{poly}(d, 1/\varepsilon)$ samples for affine-coefficient Kolmogorov equations (Berner–Grohs–Jentzen, SIAM J. Math. Data Sci. 2020).
3. **Conditional variational rates.** Deep Ritz has *dimension-independent* generalization rates in H^1 for Poisson and static Schrödinger problems — provided u^* lies in the **spectral Barron space** $\mathcal{B}^s(\Omega) = \{f : \int (1+|\omega|)^s |\widehat{f}(\omega)|\, d\omega < \infty\}$ (Lu–Lu–Wang, COLT 2021; Barron 1993). The curse is relocated into a regularity assumption, not abolished.
4. **End-to-end algorithmic.** Only multilevel Picard — which is not a neural network.

Missing from every neural rung: a guarantee that the *optimizer* finds the promised network.

The negative results

- **The theory-to-practice gap is a theorem.** Take the deep-ReLU approximation classes $\mathcal{N}_{p,p}^{L}$ of fixed depth $L \geq 3$ (p controls the function-space norm: $p = 2$ gives L^2 approximation). No algorithm using point samples (stochastic gradient descent (SGD) included) matches the approximation rate the existence theorems guarantee (Grohs–Voigtlaender, Found. Comput. Math. 24:1085–1143, 2024).
- **Training time can carry the curse.** Take shallow ReLU networks trained by gradient flow on a target u^* of Sobolev smoothness r with $2r < d$. The population risk after time t decays no faster than $t^{-4r/(d-2r)}$. The exponent shrinks toward 0 as d grows, so training time explodes with d (Na–Yang, Information and Inference, 2025).
- **Replication study (McGreivy–Hakim, 2024).** Of ML-for-fluid-PDE papers claiming to beat a standard solver, 79% (60 of 76) compared against a weak baseline — a classical solver run below its real best (Nature Machine Intelligence 6:1256–1269). Caveat: that study concerns low-dimensional fluid problems with strong classical baselines. In high dimension no classical baseline exists — the regime's appeal, and why Monte-Carlo cross-checks matter.

The electronic Schrödinger equation

For N electrons and fixed nuclei (Born–Oppenheimer), the Hamiltonian in atomic units is

$$H = -\tfrac{1}{2}\sum_{i=1}^{N}\Delta_{\mathbf{r}_i} - \sum_{i,I}\frac{Z_I}{|\mathbf{r}_i - \mathbf{R}_I|} + \sum_{i<j}\frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} + \text{const.}$$

Ground state: $H\psi = E_0\psi$ on $\mathbb{R}^{3N}$ (102 dimensions for bicyclobutane, $N = 34$). One constraint: ψ must flip sign whenever two electrons are exchanged (Pauli exclusion principle).

Why classical methods fail.

- No grid (10^{100} nodes at $d = 100$, $N = 10$).
- The exact method **Full Configuration Interaction (Full CI)** writes ψ as a sum over *every* way to place the N electrons in a set of orbitals. Each placement is one antisymmetric basis function. $O(N!)$ terms, exponential cost.
- The e–e repulsion $1/|\mathbf{r}_i - \mathbf{r}_j|$ couples all electrons: ψ is not a product of one-electron functions.

Variational Monte Carlo: escaping the exponential

The variational principle: $E_0 \le \langle\psi|H|\psi\rangle / \langle\psi|\psi\rangle$ for any antisymmetric ψ.

The VMC idea. Parametrize a network $\psi_\theta : \mathbb{R}^{3N} \to \mathbb{R}$. It maps all electron coordinates to *one scalar* (the wavefunction value). *Sample* from $|\psi_\theta|^2$:

$$\langle H\rangle_\theta = \mathbb{E}_{\mathbf{R}\sim|\psi_\theta|^2}[E_L(\mathbf{R})]\,, \quad E_L(\mathbf{R}) = \frac{H\psi_\theta(\mathbf{R})}{\psi_\theta(\mathbf{R})}.$$

Monte Carlo error is $O(1/\sqrt{M})$, independent of d. **No labeled dataset**: the Hamiltonian H is the loss. Samples come from the network's own density.

Training loop.

1. Draw $\mathbf{R}_1, \ldots, \mathbf{R}_M \sim |\psi_\theta|^2$ by Metropolis–Hastings MCMC.
2. Compute E_L: kinetic $-\nabla^2\psi/\psi$ via autodiff, potential explicit.
3. Gradient: $\nabla_\theta\langle H\rangle \approx \frac{2}{M}\sum_m (E_L(\mathbf{R}_m) - \langle H\rangle)\nabla_\theta \log\psi_\theta(\mathbf{R}_m)$.
4. Adam step on all network parameters (weights $W^{(\ell)}$, biases $b^{(\ell)}$, orbital envelope weights $V_i^k, w_{iI}^k, \pi_{iI}^k$; defined in the architecture frame below).

The ansatz: from independent electrons to correlated ones

The standard starting point — a single Slater determinant (Hartree–Fock):

$$\psi_{\mathrm{HF}} = \det\big[\phi_i(\mathbf{r}_j)\big]_{i,j=1}^{N}, \quad \phi_i(\mathbf{r}_j) \text{ depends only on } \mathbf{r}_j.$$

Antisymmetry is automatic. But each electron moves in the *average* field of all others. The individual motions are independent. This misses **correlation**: electrons dynamically avoiding each other. The gap $E_{\mathrm{exact}} - E_{\mathrm{HF}}$ is the **correlation energy**. It is typically $\sim 1\%$ of the total energy. But it governs bond-breaking, dispersion forces, and chemical accuracy.

FermiNet's fix: replace one-electron orbitals with *generalized orbitals* that depend on **all** electrons:

$$\psi_\theta = \sum_{k=1}^{K} \det\big[\phi_i^k(\mathbf{r}_j;\, \{\mathbf{r}_{\neq j}\})\big]_{i,j=1}^{N}.$$

Still antisymmetric (swapping electrons $\leftrightarrow$ swapping determinant columns $\Rightarrow$ sign flip). Now each orbital "knows" where all other electrons are. Correlation is captured inside each ϕ_i^k.

FermiNet: the equivariant backbone

Build a feature vector $\mathbf{h}_i \in \mathbb{R}^{256}$ for each electron i that encodes its whole environment — where it sits relative to the nuclei *and* to every other electron. The orbitals are read off the $\mathbf{h}_i$ (next frame).

Inputs. Electron i: the displacements $\mathbf{r}_i - \mathbf{R}_I$ to the nuclei and their lengths, plus spin, give $\mathbf{h}_i^{(0)}$. Each pair (i, j) gives $\mathbf{g}_{ij} = \big(\mathbf{r}_i - \mathbf{r}_j,\ |\mathbf{r}_i - \mathbf{r}_j|\big)$.

Each layer lets an electron see the others: mix its own feature with the *average* feature of the same-spin and of the opposite-spin electrons (and likewise average the pair terms $\mathbf{g}_{ij}$):

$$\mathbf{h}_i^{(\ell+1)} = \mathbf{h}_i^{(\ell)} + \tanh\!\big(W^{(\ell)}[\mathbf{h}_i^{(\ell)};\, \overline{\mathbf{h}}_i^{\uparrow};\, \overline{\mathbf{h}}_i^{\downarrow};\, \overline{\mathbf{g}}_i^{\uparrow};\, \overline{\mathbf{g}}_i^{\downarrow}] + b^{(\ell)}\big), \quad \overline{\mathbf{h}}_i^{\uparrow} = \operatorname*{mean}_{j:\, s_j=\uparrow} \mathbf{h}_j^{(\ell)}.$$

Four such layers (width 256, tanh, residual).

Why an average? It ignores how the electrons are numbered, so the map is **permutation-equivariant**: relabel the electrons and the $\mathbf{h}_i$ relabel the same way. Each electron keeps its *own* $\mathbf{h}_i$ — the N outputs that become the N columns of the determinant.

FermiNet: orbitals from the embedding

Each **generalized orbital** (determinant k, orbital i, evaluated at electron j) is an envelope times a linear readout of the embedding:

$$\phi_i^k(\mathbf{r}_j) = \underbrace{\left(\sum_I w_{iI}^k \, e^{-\pi_{iI}^k \|\mathbf{r}_j - \mathbf{R}_I\|}\right)}_{\text{envelope}} \times \left(V_i^k \, \mathbf{h}_j^{(L)} + c_i^k\right).$$

- The **envelope** (a sum of exponentials decaying from each nucleus $\mathbf{R}_I$) forces $\psi \to 0$ far from the molecule.
- The **linear readout** $V_i^k \mathbf{h}_j^{(L)} + c_i^k$ uses the all-electron-aware $\mathbf{h}_j^{(L)}$, so each orbital carries correlation — unlike a Hartree–Fock orbital, which depends on $\mathbf{r}_j$ alone.

The wavefunction sums K determinants, $\psi_\theta = \sum_{k=1}^{K} \det[\phi_i^k(\mathbf{r}_j)]_{i,j=1}^{N}$ (ansatz frame); more determinants = more expressive.

FermiNet and beyond: accuracy and the timeline

Results (Pfau, Spencer, Matthews, Foulkes, PRR 2:033429, 2020).
Bicyclobutane (C_4H_6, 34 electrons, 102 dims): $\sim 97\%$ of the correlation energy — near chemical accuracy. At stretched N_2: more accurate than CCSD(T) (coupled-cluster singles, doubles, and perturbative triples). CCSD(T) is quantum chemistry's gold standard method, and it fails there.

Key papers since:

- **PauliNet** (Hermann–Schätzle–Noé, Nature Chem. 12:891, 2020): same idea; Hartree–Fock baseline and electron–nuclear cusps built in analytically instead of learned.
- **Psiformer** (von Glehn–Spencer–Pfau, ICLR 2023): self-attention replaces the hand-designed pairwise streams; accuracy gain grows with system size.
- **LapNet** (Li et al., Nature MI 2024): Laplacian in one forward pass, $\sim 10\times$ faster, 116 electrons demonstrated.
- **Orbformer** (arXiv:2506.19960, 2025): pretrained on $22{,}000$ structures, fine-tuned per molecule. The foundation-model pattern applied to the Schrödinger equation.

Learning the solution operator

Solution operators

The **solution operator** of a PDE is the map from its input data to its solution.

- For steady Darcy flow, $G:\ a(x) \mapsto u(x)$ sends the permeability field to the pressure field.
- For a time-evolution PDE, $G:\ u(\cdot, 0) \mapsto u(\cdot, T)$ sends the initial state to the state a time T later.

A classical solver recomputes from scratch for every input. Many tasks solve the *same* PDE for thousands of inputs — design optimization, uncertainty quantification, weather. Learn G once, then evaluate it cheaply to amortize the cost.

G maps between **function spaces** (infinite-dimensional sets of functions), not between fixed-length vectors. An operator network takes a whole function in and returns a whole function out, independent of the grid each is sampled on.

Universal approximation: from functions to operators

Functions (Cybenko 1989; Hornik 1989; Leshno–Lin–Pinkus 1993). Let σ be a continuous non-polynomial activation, $K \subset \mathbb{R}^n$ compact. For every continuous $f : K \to \mathbb{R}$ and every $\varepsilon > 0$ there exist m, c_i, w_i, b_i with

$$\sup_{x \in K} \Big| f(x) - \textstyle\sum_{i=1}^{m} c_i \, \sigma(w_i \cdot x + b_i) \Big| < \varepsilon.$$

Operators (Chen & Chen 1995, IEEE Trans. Neural Networks 6(4)). Let $K_1 \subset \mathbb{R}^d$, $K_2 \subset \mathbb{R}^{d'}$ be compact, $V \subset C(K_1)$ compact, $G^* : V \to C(K_2)$ a continuous (possibly nonlinear) operator, σ continuous non-polynomial. For every $\varepsilon > 0$ there exist integers m, p, sensor points $x_1, \ldots, x_m \in K_1$, and weights such that

$$\sup_{u \in V,\, y \in K_2} \Big| \, G^*(u)(y) - \textstyle\sum_{k=1}^{p} \mathrm{branch}_k(u) \cdot \mathrm{trunk}_k(y) \Big| < \varepsilon.$$

$\mathrm{branch}_k(u) = \sum_i c_i^k \sigma(\sum_j \xi_{ij}^k u(x_j) + \theta_i^k)$ reads u at sensors.
$\mathrm{trunk}_k(y) = \sigma(w_k \cdot y + \zeta_k)$ reads the query point. Both branch and trunk can be implemented as deep networks (see the next frame).

Training an operator: the data question

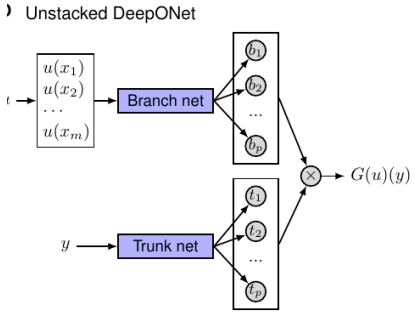
Supervised operator learning needs (input, solution) pairs, and the solutions come from a classical solver run many times in advance.

- The expensive classical solve is paid once, at training time.
- Inference — solving a new instance — is then a single cheap forward pass.

Operator learning fits **many-query** problems: the same PDE for thousands of inputs (optimization, uncertainty quantification, weather). It does not fit a one-off solve. There, running the classical solver once would be cheaper than generating a training set.

DeepONet: branch network + trunk network

Two subnets combined by inner product (Lu–Jin–Pang–Zhang–Karniadakis, *Nat. Mach. Intell.* 3:218–229, 2021, Fig. 1D):



Input: m sensor readings of u (branch) and query point y (trunk).

Output: $G(u)(y) = \mathbf{b}^\top \mathbf{t} \in \mathbb{R}$ — the operator value at y.

Training: supervised on (input function, solution) pairs from a classical solver; generalizes to input functions unseen during training.

DeepONet error and the operator curse (Lanthaler, Mishra, Karniadakis, 2022)

Setting: separable Hilbert spaces $\mathcal{X}, \mathcal{Y}$ of functions, a probability measure $\mathbb{P}$ on inputs, error measured by $\mathcal{E}(G) = \left(\mathbb{E}_{u\sim\mathbb{P}}\|G(u) - G^*(u)\|_{\mathcal{Y}}^2\right)^{1/2}$. The DeepONet error splits into three pieces (encoder, approximator, reconstructor). Each is bounded separately. Two regimes:

- **General Lipschitz operator** $G^* : \mathcal{X} \to \mathcal{Y}$ with $\mathbb{P}$ a centred Gaussian: achieving $\mathcal{E}(G) \leq \varepsilon$ forces a DeepONet of size $\exp(c/\varepsilon^\beta)$ ($c > 0$ a constant, $\beta > 0$ the exponent from the operator's complexity). The curse reappears at the operator level.
- **Solution operators of specific PDEs** — linear elliptic with Lipschitz coefficient, the heat semigroup, Burgers, Darcy flow, incompressible Navier–Stokes. Here the operator inherits the PDE's smoothing/regularity: $\mathcal{E}(G) \leq \varepsilon$ with size only *algebraic* in $1/\varepsilon$.

Neural operators: from kernel integrals to Fourier modes

A **neural operator** maps a whole input function $u(x)$ to a whole output function. Each layer takes a vector-valued function $z(x) : D \to \mathbb{R}^{d_v}$ — the *feature field*, with more channels than the original scalar field. Each layer produces another such function.

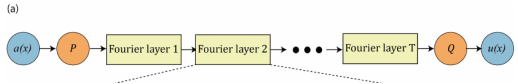
Kernel integral operator. Given a learned kernel $\kappa : D \times D \to \mathbb{R}^{d_v \times d_v}$, one layer's main step maps

$$z(x) \;\mapsto\; (Kz)(x) \;=\; \int_D \kappa(x, y)\, z(y)\, dy.$$

If $\kappa(x, y) = \kappa(x - y)$ (translation invariant), this is a **convolution** — the same filter is slid across the domain.

Convolution theorem. Convolution in space equals pointwise multiplication in Fourier space: $\mathcal{F}(\kappa * z)(\xi) = \widehat{\kappa}(\xi)\, \widehat{z}(\xi)$. So the **Fourier neural operator (FNO)** does not store the kernel κ. Instead it stores and learns the kernel's Fourier coefficients $\widehat{\kappa}(\xi)$ directly — one complex weight per kept Fourier mode (Li, Kovachki, Azizzadenesheli, Liu, Bhattacharya, Stuart, Anandkumar, ICLR 2021).

The FNO architecture: lift, Fourier layers, project (Li et al., ICLR 2021)



Three stages, with d_v hidden channels (one feature vector per grid point):

- P **(lift)** — one linear map applied *identically at every grid point*, input channels $\to d_v$. *Darcy example*: the 1 scalar $a(x)$ per point $\mapsto$ a 64-vector.
- T **Fourier layers** $v_\ell \mapsto v_{\ell+1}$ — mix information across the whole domain via frequency modes (next frame).
- Q **(project)** — again pointwise, $d_v \to$ output channels (Darcy: the 64-vector $\mapsto$ the pressure value $u(x)$).

Resolution invariance. P, Q act point-by-point. The Fourier layers keep a fixed $k_{\max}$ low modes. Neither depends on how finely the domain is sampled. So the same weights run at *any* grid: train on 64^2, evaluate on 256^2 with no retraining. (A CNN is locked to its training resolution.)

Inside one Fourier layer (Li et al., ICLR 2021, Fig. 2b)

One layer maps the feature field $v_\ell(x)$ to $v_{\ell+1}(x)$ by summing a **global spectral path** and a **local linear path**, then a pointwise nonlinearity σ (e.g. GeLU):

$$v_{\ell+1}(x) = \sigma\Big(\underbrace{W\, v_\ell(x)}_{\text{local}} + \underbrace{\big[\mathcal{F}^{-1}\big(R \cdot \mathcal{F} v_\ell\big)\big](x)}_{\text{global / spectral}}\Big).$$

$\mathcal{F}, \mathcal{F}^{-1}$ FFT to Fourier coefficients and back; only the lowest $k_{\max}$ modes are kept (higher modes truncated to zero).

R the **spectral weights**: for each kept mode k, a learned complex $d_v \times d_v$ matrix R_k multiplies that mode's coefficient vector, $\widehat{v}(k) \mapsto R_k\, \widehat{v}(k)$. These $k_{\max} d_v^2$ complex numbers are most of the parameters.

W a pointwise 1×1 linear map $d_v \to d_v$ in physical space — a **local bypass** carrying the high frequencies the truncation drops.

Universal approximation for neural operators (Kovachki et al., 2023)

A different, more general theorem than Chen & Chen (1995). That one covered the branch–trunk (DeepONet) form. Here Kovachki, Li, Liu, Azizzadenesheli, Bhattacharya, Stuart, Anandkumar (*JMLR* 24, 2023) frame the FNO inside a wider class. A **neural operator** is a composition of linear integral operators and pointwise nonlinearities between Banach function spaces. They prove:

Theorem (universal approximation). For every continuous operator $G^* : \mathcal{A} \to \mathcal{U}$ between Banach function spaces on bounded domains, every compact subset $K \subset \mathcal{A}$, and every $\varepsilon > 0$, there exists a neural operator G with

$$\sup_{a \in K} \|G(a) - G^*(a)\|_{\mathcal{U}} \; < \; \varepsilon.$$

Four parameterizations of the integral kernel sit inside this framework: graph kernels, multipole, low-rank, and Fourier (the FNO). The theorem says the framework is expressive enough. It does not bound the network size. It does not say that gradient descent finds the small accurate operator the theorem guarantees.

FNO on the cluster (exp42): task, data, setup

Task. 1-D viscous Burgers on $[0, 1]$, periodic, viscosity $\nu = 0.1$, terminal time $T = 1$. Operator $G : u_0 \mapsto u(\cdot, 1)$ (initial state to state at $t = 1$).

Data source (`exp42-operator-learning`, course cluster). $1{,}400$ pairs $\big(u_0, u(\cdot, 1)\big)$ generated by an integrating-factor RK4 Fourier spectral solver on $M = 4096$ points, $\Delta t = 10^{-4}$. Initial conditions sampled from a **Gaussian random field (GRF)** $u_0 \sim \mathcal{N}(0,\, 625(-\Delta + 25I)^{-2})$ — a random function whose Fourier modes are independent Gaussians with the given covariance. Split: $1{,}200$ train / 200 test.

FNO setup (official `neuraloperator` library, Caltech/NVIDIA). $k_{\max} = 16$ Fourier modes per layer, channel width $d_v = 64$, $L = 4$ layers, $198{,}337$ trainable parameters. 300 epochs (one epoch = one pass over the n_{train} inputs), single random seed. Trained at grid resolution $s = 256$.

Metric. Average over the 200 test inputs of the *relative* L^2 error $\big\|G_\theta(u_0) - u(\cdot, 1)\big\|_{L^2} / \|u(\cdot, 1)\|_{L^2}$. *Zero-shot* = evaluated at a grid resolution not used during training, with no retraining.

FNO measured: resolution invariance and the data curve

n_{train}	train time	s=256	s=512	s=1024	s=2048	s=4096
100	29 s	0.0406	0.0404	0.0403	0.0403	0.0402
400	100 s	0.0090	0.0090	0.0090	0.0090	0.0090
1000	253 s	0.0126	0.0123	0.0122	0.0121	0.0121

- Each row constant in s: train at 256, test at 4096 with no retraining and no degradation — resolution invariance, observed directly.
- Data curve: $\sim 4\%$ at $n_{\text{train}} = 100$, $\sim 1\%$ from $n_{\text{train}} = 400$ on (the 1000-input row is slightly worse at this fixed 300-epoch budget, single seed, untuned).
- The `neuraloperator` library (Caltech/NVIDIA) was used; numbers from exp42 on the course cluster.

Amortization measured: 4.21 s per solve vs 3.5 ms per forward pass

- The classical spectral solver (integrating-factor RK4, $M = 4096$, $dt = 10^{-4}$) costs **4.21 s** per new instance — it generated the 1,400-solve training set.
- The trained FNO costs **3.5 ms** per new instance at $s = 4096$ — a factor of $\sim 1{,}200$.
- The one-off cost: 1,400 training solves plus ~ 100 s of training. The saving exists only in the *many-query* regime — amortization, measured.

Python-vs-Python on the same CPU: the order of magnitude is the result, not the digits.

DeepONet: implementation and named successors

Reference implementation. `DeepXDE` (Lu et al., *SIAM Rev.* 63, 2021; `github.com/lululxvi/deepxde`): PyTorch / TF / JAX backend, DeepONet and PINN built in.

Named successors.

- **POD-DeepONet** (Lu et al., 2022): replace the learned trunk with modes from a **proper orthogonal decomposition (POD)** of the training data; often the most accurate variant in the 16-benchmark study below.
- **Physics-informed DeepONet** (Wang, Wang, Perdikaris, 2021): add the PDE residual to the loss; trains with little or no labelled data.
- **Feature-expansion DeepONet** (Lu et al., 2022): add harmonics $(\cos \omega\xi, \sin \omega\xi, \dots)$ to the trunk for oscillatory targets.
- **dFNO+ / gFNO+ / dgFNO+** (Lu et al., 2022) — Fourier Neural Operator (FNO) extensions, fitting FNO to different-dimensional I/O (d) and complex-geometry domains (g).

DeepONet on the cluster (exp48): same Burgers operator

Same Burgers operator data as exp42 (FNO frame). The rel. L^2 FNO column = exp42 results, for comparison. Vanilla DeepONet via DeepXDE's `DeepONetCartesianProd`: branch / trunk FNN, each 4×128 tanh, $132{,}225$ params; $50{,}000$ Adam ($lr = 10^{-3}$) then $5{,}000$ L-BFGS; single seed; same 16 CPU cores.

n_{train}	train time	rel. L^2 DeepONet	rel. L^2 FNO
100	1,462 s	0.833	0.0406
400	1,467 s	0.160	0.0090
1,000	1,495 s	0.193	0.0126

Vanilla DeepONet is 15–20× worse than FNO at every n_{train} on this smooth, periodic, regular-grid task. The $n = 1{,}000$ row is slightly worse than $n = 400$ at this fixed iteration budget. Inference $\sim 33\,\mu$s per instance — a branch+trunk dot product, faster than FNO's 3.5 ms at $s = 4096$.

Lu et al. 2022's published DeepONet/FNO parity uses POD-DeepONet with input/output normalization, not vanilla DeepONet.

DeepONet vs FNO: the 16-benchmark study (Lu et al., 2022)

Lu et al., *CMAME* 393:114778, 2022. Relative L^2 lower is better, same training budget. PDEs: Darcy ($-\nabla\cdot(a\nabla u) = f$, a random), 1-D advection (square-wave IC), 2-D N–S vorticity, airfoil.

benchmark	DeepONet	POD-DeepONet	FNO
	rel. L^2 error —lower is better		
Darcy (rectangle)	1.36%	1.26%	$\mathbf{1.19}\%$
Darcy (pentagram, hole)	1.19%	$\mathbf{0.82}\%$	n/a
Advection II (square wave)	0.27%	$\mathbf{0.08}\%$	54.4%
Navier–Stokes (vorticity-velocity formulation)	1.78%	$\mathbf{1.71}\%$	1.81%
Flapping airfoil	$\mathbf{3.65}\%$	n/a	16.2%

FNO n/a (pentagram): FFT needs a Cartesian grid; irregular domain unsupported. POD n/a (airfoil): POD needs fixed sensor locations; not tested on moving boundary. Advection 54.4%: spectral truncation → Gibbs ringing near discontinuity.

Noise sensitivity (0.1% noise, Figs. 6, 7C): Advection: FNO $\to 270\%$, DeepONet $\to 0.36\%$. Instability waves: FNO $\sim$10,000$\times$ worse, DeepONet $\sim 2\times$.

Open benchmark datasets for operator learning

Standardized datasets make methods directly comparable:

PDEBench (Takamoto et al., NeurIPS 2022) Uniform training/test splits and baselines across a broad PDE family (advection, diffusion–reaction, Navier–Stokes, shallow water, …).

PDEArena (Gupta et al., 2022) Over 20 surrogate architectures — FNO, U-Net, Transformers, graph nets — evaluated in a single unified framework.

The Well (Polymathic AI, NeurIPS 2024) 15 TB across 16 datasets from fluid dynamics, plasma physics, and beyond. Designed for pretraining and transfer experiments.

Parametric PDE: the notation $\mathcal{N}_a[u] = 0$

The solution operator $G : a \mapsto u_a$ was introduced in the operators section. Here $a \in \mathcal{A}$ is the parameter that varies from instance to instance (permeability field, initial condition, source term, …).

The compact notation $\mathcal{N}_a[u] = 0$ names the PDE whose solution is u_a. The subscript a means: the differential operator has a as a coefficient or data. Different $a \Rightarrow$ different PDE $\Rightarrow$ different solution.

Two examples.

Darcy flow $\mathcal{N}_a[u] = -\nabla \cdot (a(x)\, \nabla u) - f$. The subscript a is the permeability field; changing a changes the conductivity and therefore the pressure u.

Burgers equation $\mathcal{N}_a[u] = u_t + u\, u_x - \nu u_{xx}$ with $u(0, \cdot) = a$. The subscript a is the initial condition; each initial profile gives a different evolution.

PINO (physics-informed neural operator; Li et al., 2021) trains one FNO $G_\theta \approx G$ on data $\{(a_i, u_i)\}$ with an additional PDE-residual penalty. Trained once, it solves for any a_* in one forward pass.

PINO: training loss and inference (Li et al., arXiv:2111.03794)

Network. An FNO $G_\theta : \mathcal{A} \to \mathcal{U}$.

Training data. N pairs (a_i, u_i) from a classical solver; $N = 0$ is also allowed (physics residual only, no solver data).

Loss ($\lambda > 0$: physics weight balancing data fidelity vs. PDE residual).

$$L(\theta) \;=\; \underbrace{\tfrac{1}{N}\sum_i \|G_\theta(a_i) - u_i\|_{L^2}^2}_{\text{data term}} \;+\; \lambda\; \underbrace{\tfrac{1}{N}\sum_i \|\mathcal{N}_{a_i}[G_\theta(a_i)]\|_{L^2}^2}_{\text{PDE residual}}.$$

- The residual $\mathcal{N}_{a_i}[G_\theta(a_i)]$ is computed by automatic differentiation through G_θ.
- The residual can be evaluated on a grid *finer* than the training-data grid: coarse labeled solutions, fine physics.
- With $N = 0$ PINO has only the PDE residual term — and can still converge.

Inference. For a new a_*, one forward pass $G_\theta(a_*)$ returns the solution — same cost as a vanilla FNO query.

FINO: a finite-difference-inspired neural operator (Cheng, Dong, Schönlieb, Aviles-Rivero, 2025)

Position. Hyperbolic PDEs have finite-speed propagation. The solution at (x, t) depends only on a local region. FINO argues that locality should be a hard architectural constraint — not an emergent property of global mixing (FNO, attention). (arXiv:2509.26186)

Local Operator Block (LOB). One layer = three steps:

1. **Learnable stencil**: a $(2r+1)^2$ convolution

$$S(\mathbf{u})[h, w] = \sum_{p,q=-r}^{r} \sum_{c} w_{c,p,q}\, \mathbf{u}_{c,h+p,w+q} + b$$

   Weights replace fixed stencils (5-point Laplacian, etc.); cf. PDE-Net (Long et al. 2018).
2. **Gating mask** $M(\mathbf{u}) = \sigma(W_g * S(\mathbf{u})) \odot S(\mathbf{u})$ — per-location, per-channel importance weight in $[0, 1]$ suppresses irrelevant derivative responses. (σ = sigmoid; written M to avoid collision with the operator G_θ used in PINO.)
3. **Linear fuse** $\partial_t U_t = W_c * M(\mathbf{u})$.

FINO: explicit time stepping, stability, and results

Explicit time step with learnable Δt: $U_{t+\Delta t} = U_t + \Delta t\, \partial_t U_t$, then $\mathrm{ReLU}(W_p * \cdot)$ (W_p: 1×1 pointwise channel-mixing convolution) to form one FINO Block. Three blocks are stacked, wrapped in a U-Net.

Notation. $\Phi_{\Delta t}$: the true time-Δt solution map (one exact PDE step); Ψ_θ: the learned FINO block approximating it.
Composition error bound (Prop. 1). If $\sup_u \|\Psi_\theta(u) - \Phi_{\Delta t}(u)\| \le \varepsilon'$ and $\Phi_{\Delta t}$ is Lipschitz with constant C, then

$$\left\|(\Psi_\theta)^K(u_0) - (\Phi_{\Delta t})^K(u_0)\right\| \;\le\; \tfrac{C^K-1}{C-1}\,\varepsilon' \quad (K\varepsilon' \text{ if } C{=}1).$$

Stability under rollout from per-step approximation quality.
Thm 2 (universal approximation): for every compact $\mathcal{U}$ and $\varepsilon > 0$, a depth-K FD-NET matches $(\Phi_{\Delta t})^K$ uniformly within ε on $\mathcal{U}$.

Results. Up to 44% lower error and $\sim2\times$ faster than FNO / CNO (Convolutional Neural Operator, Raoníc et al., 2023) / transformer baselines. Measured on six PDEBench tasks + a climate task.

In-context operator learning: setup (Yang, Liu, Meng, Osher, 2023)

Goal: train *one* network. At inference it takes a few demos of an unknown PDE/ODE operator and applies it to a new input — no weight update. (arXiv:2304.07993; PNAS 121:11, 2024.)

Vocabulary. *Condition* = the operator's input function (e.g. for the ODE $u' = \alpha u + \beta c + \gamma$: the source $c(t)$ and the initial value $u(0)$). *QoI* (quantity of interest) = the output function $u(t)$. *Demo* = one (condition, QoI) pair. *Prompt* = J demos + one *question condition*. *Query* = keys where the question QoI is asked.

Token representation (key, value, index). Every function is encoded as a sequence of columns; column i holds:

key $\in \mathbb{R}^{d_k}$ term indicator, time coord., space coord., ...

value $\in \mathbb{R}$ the function value at that key

index $\in \mathbb{R}^{J+1}$ $+\mathbf{e}_j$ for demo j's condition; $-\mathbf{e}_j$ for demo j's QoI; $\mathbf{e}_{J+1}$ for the question.

Prompt matrix = concatenate all demos' and question's columns along the column axis. Column order is irrelevant: the index column tells the model who is who.

ICON architecture, training, inference

Architecture (DETR-style encoder–decoder; DETR = DEtection TRansformer, Carion et al. 2020; Yang et al. 2023, Fig. 2).

- Prompt matrix → shared linear → **transformer encoder** (self-attention over all prompt columns) → embedding $\mathbf{E}$ of the unknown operator + question.
- Queries → shared linear → **transformer decoder** (cross-attention into $\mathbf{E}$; *no self-attention* over queries) → predicted value of the question QoI at each query key.

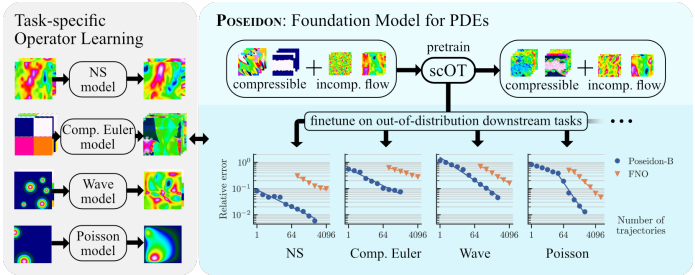
Training (Algorithm 2). Each step:

- pick a random problem type + random parameters (defines an operator);
- sample J (condition, QoI) demos + one question;
- build prompt, queries, ground-truth values; MSE loss; backprop.

J varies across batches (transformer handles variable sequence length).

Inference. New operator → collect J demos, a question condition, and a list of query keys → one forward pass returns predictions. The operator is *never* named. It exists only as the embedding $\mathbf{E}$.

Poseidon: a pretrained PDE operator model (Herde et al., NeurIPS 2024)



arXiv:2405.19101; `github.com/camlab-ethz/poseidon`.

What. A neural operator pretrained on a large corpus of fluid-dynamics simulations (compressible and incompressible), then fine-tuned on new PDE types.
Result. Tested on 15 unseen PDE tasks: Navier-Stokes (NS), Compressible Euler, Wave equation, Poisson, …Poseidon-B (base, 158M parameters; sizes T/B/L defined in the next frame) matches an FNO trained from scratch. It uses *fewer* training trajectories (right panel in figure).

scOT (scalable convolutional operator transformer): input, output, architecture

Input. Initial state $a(x) \in C(D; \mathbb{R}^n)$ on $D = [0,1]^2$ (n channels — density, velocity, …), sampled on a regular grid. *Plus* a lead time $t \in [0, T]$.

Output. Predicted state $S_\theta^*(t, a) \approx u(t, \cdot)$ on the same grid. Sizes: Poseidon-T / -B / -L $= 21\text{M} / 158\text{M} / 629\text{M}$ parameters.

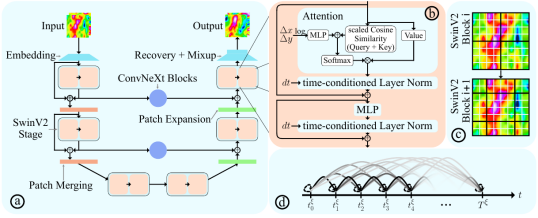


Figure 2: (a) scOT, the model underlying POSEIDON; (b) SwinV2 Transformer block; (c) Shifting Window over patch-based tokens with window (patch) boundaries with black (white); (d) all2all Training for time-dependent PDEs.

Source: Herde et al., arXiv:2405.19101, Fig. 2. (a) scOT; (b) SwinV2 (Shifted-Window Transformer v2) block; (c) shifted window; (d) all2all: train on pairs $(u(t_i), u(t_j))$, $t_i \leq t_j$.

scOT components (1/2): patch embedding, SwinV2 block, time-LN

Patch embed $\hat{E}$ Partitions D into $w \times w$ patches ($H \times W$ = input grid resolution); linearly maps per-patch weighted averages to tokens $\mathbf{v} \in \mathbb{R}^C$ on a $(H/w) \times (W/w)$ grid. Role: dense field $\rightarrow$ token sequence.

SwinV2 block Two residual updates per layer (Eq. 3):

$$\mathbf{v}'_\ell = \mathbf{v}_{\ell-1} + LN_{\alpha,\beta}(W\text{-MSA}(\mathbf{v}_{\ell-1}))$$
$$\mathbf{v}_\ell = \mathbf{v}'_\ell + LN_{\alpha,\beta}(\mathrm{MLP}(\mathbf{v}'_\ell))$$

W-MSA: windowed self-attention inside $w \times w$ tiles (cost $O(N)$; tiles shift each layer so global context accumulates over depth).

Time-conditioned LayerNorm Injects lead time t as scale and shift (Eq. 4):

$$LN_{\alpha(t),\beta(t)}(\mathbf{v}) = \alpha(t) \odot \frac{\mathbf{v} - \mu_{\mathbf{v}}}{\sigma_{\mathbf{v}}} + \beta(t), \quad \alpha(t) = \alpha t + \bar{\alpha}$$

One set of weights handles all $t \in [0, T]$.

scOT components (2/2): hierarchy, skips, recover

Patch merging *Takes* a 2×2 block of neighbouring tokens, each $\in \mathbb{R}^C$. *Does* concatenate into $\mathbb{R}^{4C}$, then linearly project to $\mathbb{R}^{2C}$. *Outputs* 1 token at half spatial resolution and double channels. *Role*: the U-Net down-step — coarser scale gives the next SwinV2 stage a larger effective receptive field for the same window size.

Patch expansion *Takes* 1 token $\in \mathbb{R}^{2C}$. *Does* linearly project to $\mathbb{R}^{4C}$, then reshape into a 2×2 block of tokens $\in \mathbb{R}^C$. *Outputs* 4 tokens at double spatial resolution and half channels. *Role*: the U-Net up-step — restores fine spatial resolution for the output.

ConvNeXt blocks 7×7 depthwise conv + 1×1 channel-mixing MLP (a modern conv block; Liu et al., 2022). Used in *two* places: as the **bottleneck** at the coarsest scale and as the **skip** that connects encoder and decoder features at the same scale.

Recover $\hat{R}$ inverse of $\hat{E}$: linear projection per token, patches un-folded into a continuous output field $u(t, x)$ on the original grid.

Poseidon training: loss, all2all trick, dataset, finetune

Standard loss on M trajectories sampled at $0 = t_0 < \cdots < t_K = T$:

$$\mathcal{L}(\theta) = \frac{1}{M(K+1)} \sum_{i,k} \left\| S_\theta^*(t_k, a_i) - S(t_k, a_i) \right\|_{L^p(D)}^p, \qquad K{+}1 \text{ pairs per trajectory.}$$

all2all loss (Eq. 6). Solution operators satisfy the **semigroup property** $u(t^*) = S(t^* - t, u(t))$ for $0 \le t \le t^*$, so every $\bar{k} \le k$ gives a valid pair $(u_i(t_{\bar{k}}), u_i(t_k))$:

$$\widehat{\mathcal{L}}(\theta) = \frac{1}{M\widehat{K}} \sum_{i,\, \bar{k} \le k} \left\| S_\theta^*(t_k - t_{\bar{k}}, u_i(t_{\bar{k}})) - S(t_k - t_{\bar{k}}, u_i(t_{\bar{k}})) \right\|^p, \quad \widehat{K} = \tfrac{(K+1)(K+2)}{2}.$$

Quadratic in K: $K{+}1 = 11$ snapshots $\Rightarrow \widehat{K} = 66$ pairs per trajectory.

Pretraining data. 6 operators (4 compressible Euler + 2 incompressible NS), $77{,}840$ trajectories, 11 snapshots each $\Rightarrow \sim$5.11M pairs.

Finetune. Partition $\theta = [\hat{\theta}, \tilde{\theta}, \tilde{\theta}_{\mathcal{N}}]$. $\hat{\theta}$ (body) initialised from pretraining. $\tilde{\theta}_{\mathcal{N}}$ (time-LN) transferred with higher learning rate (LR). $\tilde{\theta}$ (embed/recover, when the downstream channel count differs) trained from scratch.

PDEformer-2: input is the PDE itself (Ye et al., Dong group, 2025)

Design point vs Poseidon. Poseidon fixes the PDE family (fluid), varies the initial condition. PDEformer-2 takes the *symbolic PDE itself* as input, so one model handles many PDE forms. (arXiv:2507.15409)

Input: a PDE as a directed acyclic graph (DAG) of nodes and operations. Plus coefficient fields, initial condition, and domain shape Ω via its signed distance function SDF_Ω. **Output**: mesh-free $\hat{u}(t, x, y)$ at any query coordinate.

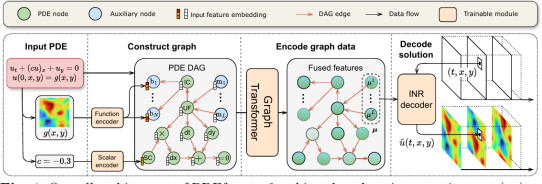


Fig. 1: Overall architecture of PDEformer-2, taking the advection equation $u_t + (cu)_x + u_y = 0$, $u(0, \boldsymbol{r}) = g(\boldsymbol{r})$ on $\Omega = [0, 1]^2$ with periodic boundary conditions as an example. This equation involves only a single unknown field variable $u_1 = u$, and we omit the subscript $j = 1$ for u_j and $\boldsymbol{\mu}_j$ in the figure.

Source: Ye et al., arXiv:2507.15409, Figure 1 (advection example $u_t + (cu)_x + u_y = 0$, $u(0, r) = g(r)$ on $\Omega = [0, 1]^2$ periodic).

PDEformer-2 components: DAG, encoders, Graphormer, Poly-INR

DAG node types `UF` (unknown field), `SC` (scalar coef), `CF` (coefficient field), `VC` (varying coef), `IC` (initial condition), `SDF` (domain shape). **Operation nodes**: `dt`, `dx`, `dy`, `+`, $\times$, $-$, $(\cdot)^2$, $=0$, $\sin$, $\cos$, $\exp_{10}$, $\log_{10}$, `AT` (arbitrary transform).

Scalar encoder 2-layer MLP, 256 hidden units, output dim $d_e = 768$ (the graph embedding dim).

Function encoder CNN on 128×128 field: three stride-4 4×4 convs ($32\to128\to768$ channels), flatten, split into $N{=}4$ branch features.

Graph Transformer (Graphormer; Ying et al. 2021) 12 layers, $d_e = 768$, FFN 1536, 32 heads; $h_i^{(0)} = x_{\mathrm{type}(i)} + \xi_i + z^-_{\deg^-(i)} + z^+_{\deg^+(i)}$ (ξ_i: positional; $z^\pm$: degree; attn. bias $B_{ij} = b^+_{\phi(i,j)} + b^-_{\phi(j,i)}$, ϕ = shortest-path length, $d_{ij}{=}-\infty$ if disconnected).

Poly-INR decoder an implicit neural representation (INR: maps a query coordinate to its field value); $L = 11$ layers, width 768 (base) / 256 (fast), activation $\sigma = \sqrt{2}\sin$. A hypernetwork turns branch features into per-layer scale/shift modulations.

PDEformer-2: pretraining data + use modes

Pretraining dataset ~ 40 **TB.** Eight *generic* PDE families; each yields many specific PDEs by randomly setting some coefficients to 0 or 1. Data from Dedalus V3 + FEniCSx; accuracy is nRMSE (normalized root-mean-square error).

family	samples	data	nRMSE (base)
Diffusion–Convection–Reaction (DCR)	745k	4.85 TB	6.9%
Wave	605k	4.34 TB	6.5%
Multi-variable DCR	660k	12.24 TB	12.6%
Divergence-constrained DCR	400k	7.49 TB	17.0%
MV-Wave / DC-Wave / G-SWE / Elasticity	875k	11.13 TB	3.8–13.0%
Total	3.29**M**	40 **TB**	8.7% avg

Use modes. *Zero-shot*: PDE form within the 8 families, one forward pass returns $\hat{u}$ on the full space-time domain. *Few-shot finetune*: unseen PDE form, < 100 task-specific samples beat specialised neural operators trained from scratch. *Inverse problems*: gradient descent on unknown coefficient scalars or fields with observed u as targets.

Attention patterns in neural PDE solvers

Where the tokens come from decides what attention does. (N = grid tokens; $|V|$ = DAG nodes, $|V| \ll N$.)

token	attention scope	cost	example
grid patch	full self-attention (Q, K, V from patches)	$O(N^2)$	OFormer, GNOT, Transolver
grid patch	$w \times w$ window, shifted per layer	$O(N)$	scOT in Poseidon
DAG node	shortest-path bias $\phi(i,j)$; masked if disconnected	$O(\|V\|^2)$	Graphormer/PDEformer-2
query coord	cross-attention: query vs encoded input	$O(\|q\|\|\text{in}\|)$	HAMLET, ICON

Time conditioning. Same attention weights handle every lead time t. Inject t through scale/shift in the layer norm ($LN_{\alpha(t),\beta(t)}$ in Poseidon's scOT), not into $Q/K/V$ directly.

The opposing camp. FNO replaces global mixing with truncated Fourier modes (no $QK^\top$ at all). FINO (arXiv:2509.26186) argues that for PDEs with finite propagation speed even windowed attention mixes too much. Its alternative: strict locality via learnable finite-difference stencils.

Learned weather models: three architectures

Numerical weather prediction (NWP) integrates the atmosphere's PDEs on supercomputers. The operational standard is the ECMWF **IFS** (Integrated Forecasting System). Three models trained on **ERA5 reanalysis** (decades of hourly global state) now match or beat it — each a different backbone:

FourCastNet an **adaptive Fourier neural operator (AFNO)**: a vision transformer whose token mixing is an FNO-style Fourier filter (global mixing at linear cost), at $0.25°$, 6 h step. *New*: Fourier token mixing makes a ViT tractable at weather resolution. First $0.25°$ data-driven model to match IFS at short range.

Pangu-Weather a **3D Earth-Specific Transformer** (Swin-style 3D attention over the pressure-level cube, with an Earth-specific positional bias) plus hierarchical time-stepping. *New*: first to beat operational IFS across all variables and lead times.

GraphCast an **encoder–processor–decoder graph network** on a multi-mesh icosahedral grid (16 message-passing layers). *New*: one mesh hop spans short and long range; trained end-to-end on multi-step error.

Learned weather models: speed, skill, and the catch

model	params	training	headline vs IFS / HRES
FourCastNet	ViT–AFNO	ERA5 '79–'15	matches IFS short-range; week in $< 2\,\mathrm{s}$
Pangu	$\sim 256\mathrm{M}$	ERA5 '79–'21	5-day Z_{500} RMSE 297 vs IFS 334; $1.4\,\mathrm{s}$
GraphCast	$37\mathrm{M}$	ERA5 '79–'17	beats HRES on 90.3% of targets; $< 60\,\mathrm{s}$

The catch. All three are *operator surrogates*: they learn a single-step map on fixed-resolution reanalysis and roll it out autoregressively. Trained on mean-squared error, they **blur fine structure and underpredict extremes at long lead times**. They also carry **no first-principles guarantees**: no enforced conservation laws. Skill is validated against the reanalysis they were fit to, not derived from the governing equations.

Spectral bias and training failure

Spectral bias

The approximation theorems say a good network *exists* and is *small*. Training is a separate matter.

An **empirical observation** (Rahaman et al., 2019; later derived in the neural tangent kernel (NTK) regime — see next frame). Networks trained by gradient descent fit a target frequency by frequency:

- **low-frequency** components first;
- **high-frequency** components last, or not at all.

Take a PDE whose solution has sharp features — a shock, a boundary layer, a high-frequency oscillation. Training plateaus with the coarse shape right and the fine structure wrong. The PINN loss reaches a small value. The error stays concentrated at the shock or boundary layer.

The neural tangent kernel account (Wang, Yu, Perdikaris, 2022)

The **neural tangent kernel (NTK)** (Jacot, Gabriel, Hongler, 2018) makes spectral bias quantitative. In the wide / linearized regime, gradient-descent training behaves like kernel regression with a fixed kernel

$$\Theta(x, x') = \left\langle \nabla_\theta u_\theta(x),\ \nabla_\theta u_\theta(x') \right\rangle.$$

Decompose the error along the eigenvectors of Θ. Each component decays at a rate proportional to its **eigenvalue**. The NTK eigenvalues decay rapidly with frequency. So high-frequency error components decay polynomially slowly — spectral bias, derived rather than observed.

A second pathology is specific to PINNs. The residual loss and the boundary loss have NTK eigenvalues of very different magnitude. Under equal weighting one term dominates the gradient and the other is barely trained (the gradient pathologies of Wang & Perdikaris, 2021).

Remedies

- **Loss weighting / learning-rate annealing** (Wang & Perdikaris, 2021): rescale the loss terms during training using their gradient statistics, so no term dominates.
- **Fourier-feature input embeddings** (Tancik et al., 2020; Wang, Wang, Perdikaris, 2021): map the input $x \mapsto [\cos(2\pi B x), \sin(2\pi B x)]$ through a bank of frequencies B, so the network sees high frequencies directly and the NTK spectrum flattens.
- Further directions: domain decomposition, causal/curriculum training that respects the time direction, and architectures that build the boundary condition in as a hard constraint.

Reference implementation. `jaxpi` (Wang, Sankaran, Wang, Perdikaris, arXiv:2306.06520, 2023) packages all three remedies — loss re-weighting, Fourier-feature embeddings, and causal training — in a single JAX library with worked examples.

Demonstration: three PINN experiments on the cluster

PINNs trained in PyTorch (2.12, CPU) on the course cluster; per-mode error traces and summary numbers recorded for every run.

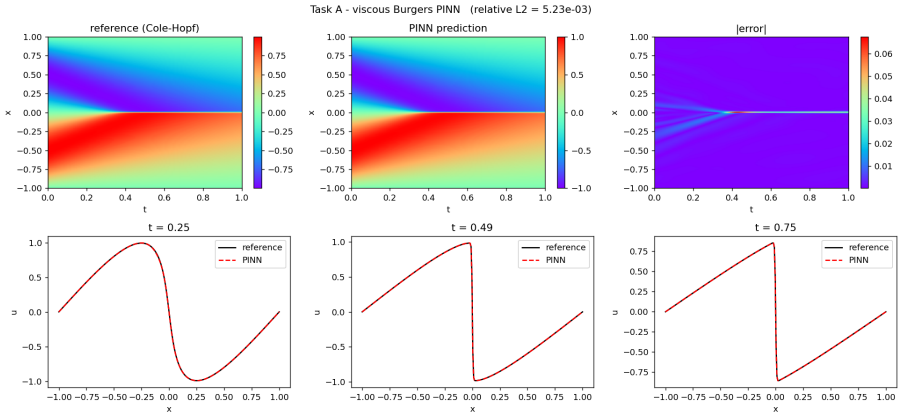
A — Burgers forward Burgers $u_t + u\,u_x = \nu\,u_{xx}$ on $[-1,1]\times[0,1]$, $\nu = 0.01/\pi$, $u(0,x) = -\sin(\pi x)$, $u(t,\pm 1) = 0$; a 4×64 tanh MLP (12,737 params), 10,000 collocation points, 15,000 Adam + 500 L-BFGS steps, ≈12 min on CPU.

B — spectral bias per mode target $u^*(x) = \sin(2\pi x) + \frac{1}{2}\sin(2\pi K x)$; every 400 steps project $u_\theta - u^*$ onto $\sin(2\pi m x)$ and record $|c_m|$. Sweep: {regression, PINN residual} × {plain, Fourier features} × $K \in \{5, 10, 15, 30\}$ × 3 seeds = 48 jobs, 40,000 steps each.

C — inverse recovery an inverse problem — $u_t + \lambda_1 u\,u_x = \lambda_2 u_{xx}$ with the coefficients λ_1, λ_2 as trainable parameters; 5,000 noisy samples of the reference solution.

Convergence threshold throughout: $|c_m| \le 10^{-2}$.

Task A result: Burgers forward solve



Relative L^2 error vs the Cole–Hopf reference: 5.2×10^{-3}. Maximum pointwise error 0.068, confined to the thin shock layer at $x = 0$ (top-right panel). The Cole–Hopf reference is more accurate and far cheaper on this 1-D problem.

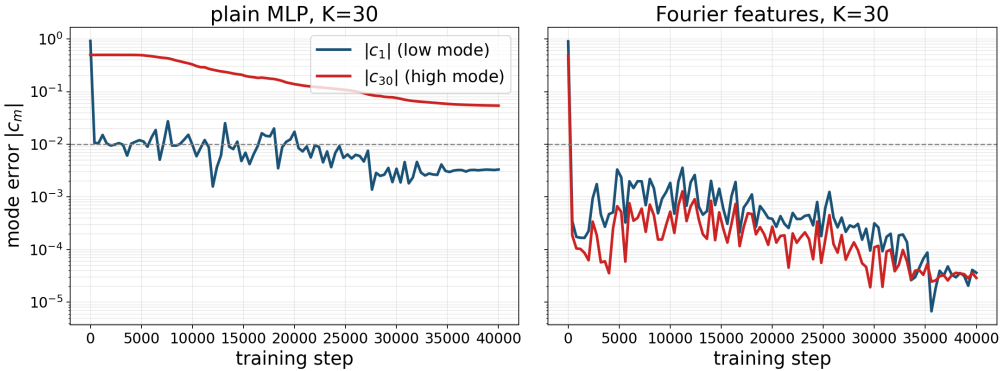
Task B result: steps for each mode to reach 10^{-2} (regression)

Fit u^* by mean-squared error — the pure frequency effect, no operator. Steps for $|c_m| \le 10^{-2}$, mean over 3 seeds:

K	plain, low c_1	plain, high c_K	Fourier, low	Fourier, high
5	400	1,600	400	400
10	1,200	3,600	400	400
15	1,200	7,600	400	400
30	2,000	never (final 5.4×10^{-2})	400	400

The plain network's low mode converges in a few hundred steps. Its high mode lags by a factor growing with K: $1{,}600 \to 3{,}600 \to 7{,}600 \to$ never within 40,000 steps. Fourier features converge both modes in $\sim$400 steps at every K.

Task B: the per-mode error curves ($K = 30$, regression)



Left (plain MLP): $|c_1|$ drops below 10^{-2} within 2,000 steps; $|c_{30}|$ stays near 10^{-1} for all 40,000 steps. Right (Fourier features): both modes fall below 10^{-2} in $\sim$400 steps. Dashed line: the 10^{-2} threshold.

Task B, PINN mode: the K^4 conditioning on top of the bias

Train on the residual of $-u'' = f$ instead of on u^*. By Parseval, the L^2 residual is

$$\left\| u_\theta'' + f \right\|_2^2 \;=\; \sum_k (2\pi k)^4 \, |e_k|^2,$$

where $e_k = \langle u_\theta - u^*, \sin(2\pi k x) \rangle$ is the Fourier error coefficient of mode k.

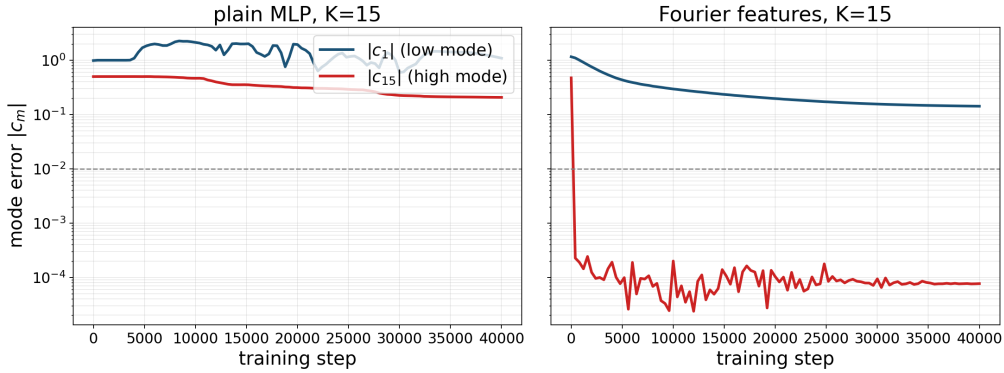
so the mode-k error enters with weight $(2\pi k)^4$. The loss has condition number $\sim K^4$ on top of the network's frequency bias.

Measured (steps to 10^{-2}, mean over 3 seeds):

- Plain PINN: fails across the family — at $K = 5, 15, 30$ neither mode reaches the threshold in 40,000 steps; at $K = 10$ the high mode crosses at 23,200.
- Fourier features: the high mode converges in ~ 400 steps at every K; at $K \geq 15$ the *low* mode is now the unconverged one (final 0.14 at $K = 15$, 1.06 at $K = 30$).

The imbalance is not removed. It moves to the other end of the spectrum. The full remedy pairs the embedding with loss re-weighting based on the neural tangent kernel.

Task B: the per-mode error curves ($K = 15$, PINN mode)



Left (plain): both modes stuck near 1 — the PINN does not train. Right (Fourier features): $|c_{15}|$ falls below 10^{-4} almost immediately; $|c_1|$ declines slowly and is still ~ 0.14 at 40,000 steps.

Task C result: inverse recovery of (λ_1, λ_2)

$u_t + \lambda_1 u\, u_x = \lambda_2 u_{xx}$; data = 5,000 samples of the reference solution + Gaussian noise; 40,000 Adam + 3,000 L-BFGS steps per noise level.

noise	λ_1 (true 1)	error	λ_2 (true 3.183×10^{-3})	error
0%	0.99717	0.28%	3.468×10^{-3}	8.95%
1%	0.98850	1.15%	4.796×10^{-3}	50.7%
5%	0.99854	0.15%	3.476×10^{-3}	9.21%

- The convection coefficient λ_1 is recovered to $\leq 1.2\%$ at every noise level.
- The viscosity λ_2 is the coefficient of the smallest, highest-curvature term. It errs 9–51%, not monotone in the noise (single seed). Away from the thin shock layer the data barely distinguish $\nu = 3.2 \times 10^{-3}$ from 4.8×10^{-3}.

Scope of the PINN experiments

- The spectral bias (slow high-frequency convergence) and the Fourier-feature remedy are established results in the PINN literature; these experiments measure them on the Burgers equation.
- The Cole–Hopf reference is more accurate and far cheaper than the PINN on these 1-D problems.
- Task C uses synthetic data generated by the reference solver; the governing equation is known in advance.

Failure modes beyond frequency (Krishnapriyan et al., 2021)

On convection, reaction, and reaction–diffusion problems, a PINN that solves the easy regime *fails* as a single parameter — for instance the convection speed — grows.

The cause is not limited network capacity. The soft PDE-residual penalty lets the optimizer trade off residual in one region against another. As stiffness grows, spurious local minima proliferate. The global minimum becomes unreachable. Two fixes:

- curriculum training — start from an easy version of the PDE and harden it during training;
- pose the problem as sequence-to-sequence in time, rather than fitting all of space–time at once.

This difficulty is an optimization and loss-landscape problem, distinct from the neural-tangent-kernel spectral bias.

Respecting causality in time (Wang, Sankaran, Perdikaris, 2022)

Train a time-dependent PINN on all times at once and it can fit late times before the early times are correct. This violates the physical arrow of time. It fails on chaotic or turbulent dynamics.

The fix weights the loss so that the residual at a given time is trained only after the earlier times already have small residual — a weight that decays with the accumulated earlier loss.

This has three effects:

- it restores temporal causality;
- it improves convergence;
- it gives a quantitative signal of how far training has progressed in time.

Spectral bias as a feature: hybrid neural–classical solvers

Spectral bias says a network learns *low* frequencies fast, high ones slowly. Classical relaxation (Jacobi, Gauss–Seidel) is the *opposite*: it removes high-frequency error fast and stalls on the smooth, low-frequency part. Combine them and each covers the other's weakness.

HINTS (Zhang, Kahana, ..., Karniadakis, *Nature Mach. Intell.* 2024). Run a classical relaxation loop. Every few sweeps swap in one **DeepONet** correction that removes the smooth error. On a 2D Helmholtz case where plain multigrid *diverged*, HINTS reached machine precision in ~ 3 V-cycles.

The network accelerates the solver; it does not replace it. Related lines:

- learned **preconditioners** for CG/GMRES — a GNN predicts a sparse approximate factorization; the Krylov method keeps its guarantees;
- learned **multigrid** (Greenfeld et al. 2019; Hsieh et al. 2019): the true solution stays a fixed point *for any* weights, so the network changes only the speed, never the answer;
- neural-operator **warm starts** — an FNO initial guess, then a classical corrector.

Networks versus finite elements (He, Li, Xu, Zheng, 2020)

A network with the **ReLU** activation (rectified linear unit, $\max(0, x)$) computes a continuous piecewise-linear function — exactly the class that linear finite elements use.

- Every linear-finite-element function in dimension d can be represented by a ReLU network.
- At least two hidden layers are necessary for $d \geq 2$.

So a ReLU network contains the finite-element space and strictly more. The difference is how the partition is chosen.

- A network adapts its own piecewise-linear partition through training.
- The finite-element method fixes the mesh in advance.

When to use a neural PDE method

Low-dimensional smooth forward problems classical FDM / FEM / spectral methods win on speed and accuracy. Networks do not replace them.

High-dimensional PDEs ($d \gg 3$) classical solvers fail; neural methods (Deep BSDE, deep splitting, SDGD = stochastic dimension gradient descent) and multilevel Picard are the available options, with the Grohs et al. theorem behind the proven rung.

Inverse problems, data assimilation, unknown coefficients sparse noisy data; PINNs are natural — physics and data live in one objective.

Many-query settings the same PDE for thousands of inputs; neural operators (FNO, DeepONet) amortize the cost — one training, cheap inference.

Complex geometry networks are mesh-free, avoiding the meshing that strains classical solvers.

Open: reliable training in the presence of spectral bias; error bounds for the network the optimizer actually finds.

What these networks do not do

Neural PDE methods produce numerical solutions and data-driven conjectures. They do not:

- prove existence, uniqueness, or regularity of solutions. The Navier–Stokes global-regularity question is a Millennium Prize problem. A network that computes approximate Navier–Stokes flows leaves it untouched;
- return a symbolic solution or a theorem — the output is a trained function or a fitted equation, to be verified separately;
- carry an error certificate by default — the approximation theorems say a good network exists, not that the one you trained is good.

Discovering the equation from data

PDE-FIND: the method (Rudy, Brunton, Proctor, Kutz, 2017)

Invert the problem: given measurements of $u(x,t)$, find the governing PDE.
PDE-FIND recovers a hidden PDE from a grid of measurements:

1. build a large **library** of candidate terms — u, u_x, u_{xx}, $u\,u_x$, u^2, …— each evaluated from the data (spatial derivatives by finite differences or local polynomial fits);
2. write the time derivative as a linear combination, $u_t \approx \Theta\,\xi$, the columns of Θ the candidate terms, ξ the unknown coefficients;
3. solve by **sparse regression** — STRidge (sequential threshold ridge regression): repeatedly fit and zero out small coefficients — so only a few terms survive.

The surviving terms with their coefficients *are* the discovered PDE. A Pareto trade-off between fit accuracy and the number of active terms selects the model.

PDE-FIND: input, output, results

Input: spatiotemporal measurements and a candidate-term library.

Output: the few active terms and their coefficients — a PDE.

Objective: fit u_t with the fewest terms.

PDE-FIND rediscovers Burgers' equation, the Korteweg–de Vries (KdV) equation, the Schrödinger equation, and Navier–Stokes from data.

It extends **SINDy** (sparse identification of nonlinear dynamics; Brunton, Proctor, Kutz, 2016) from ordinary differential equations to PDEs. Like **symbolic regression** — automatic search for a closed-form expression fitting numerical data — it returns a *form* fitted to data. The form is a hypothesis; correctness must be checked separately (analytic derivation, or a prover).

Library. `PySINDy` (de Silva et al., *J. Open Source Softw.* 2020; Kaptanoglu et al., 2022) implements SINDy and PDE-FIND in Python; maintained by the Brunton–Kutz group.

Inverse PINNs: coefficients from data

Input: known PDE form with one or more unknown coefficients; sparse, possibly noisy pointwise measurements of the solution u.

Output: fitted coefficients $(\lambda_1, \lambda_2, \dots)$ and a network u_θ approximating the full solution.

Objective: jointly minimize the PDE residual and the mismatch with the observed data, treating unknown coefficients as trainable parameters.

The second route is the inverse mode of the 2019 PINN: the unknown coefficients become **trainable parameters**, fitted jointly with u_θ from sparse, noisy observations of u.

- Less general than PDE-FIND — you must guess the *form* of the equation and only fit its coefficients.
- Robust to noise and effective with little data.

Example. Burgers' equation $u_t + \lambda_1 u\, u_x = \lambda_2 u_{xx}$ with unknown (λ_1, λ_2); supply 5,000 noisy pointwise measurements of u; the network and the two coefficients train jointly.

Noise-robust discovery, and the network's role (Weak-SINDy / WSINDy)

Input: noisy time-series measurements $u(t_i, x_j)$ of a physical system.

Output: a sparse set of library terms $\{\theta_k\}$ with coefficients ξ_k such that $\dot{u} = \sum_k \xi_k \, \theta_k(u, \nabla u)$.

Objective: identify the governing ODE/PDE without amplifying measurement noise via numerical differentiation.

PDE-FIND (see previous frame) differentiates the measured data to form candidate library terms, which amplifies measurement noise. Weak-form variants (WSINDy, Messenger–Bortz 2021; Reinbold–Grigoriev 2020) integrate each term against smooth test functions φ: $\int u_t \varphi \, dx = \int \theta_k(u) \varphi \, dx$. Integrating by parts moves derivatives from noisy u onto the known φ. This is far more robust to noise, and exact on piecewise data without differentiating.

Across the discovery methods, the network appears in one of two roles:

- as the regressor — the inverse PINN fitting unknown coefficients;
- as an interpretable function — a Kolmogorov–Arnold network whose edges can be read.

Kolmogorov–Arnold networks (Liu et al., 2024)

PDE-FIND expresses the governing equation as a *sum of library terms* and selects the active ones. KAN offers a complementary route: train a network whose *edges are explicit one-variable functions*, then read the edge shapes as a symbolic formula. Discovery happens by inspecting the trained model directly.

Kolmogorov–Arnold representation theorem (Kolmogorov 1957). For every continuous $f : [0,1]^n \to \mathbb{R}$ there exist $2n+1$ continuous functions $\Phi_q : \mathbb{R} \to \mathbb{R}$ and $n(2n+1)$ continuous monotone functions $\psi_{q,p} : [0,1] \to \mathbb{R}$ such that

$$f(x_1, \ldots, x_n) \;=\; \sum_{q=0}^{2n} \Phi_q\Big(\sum_{p=1}^{n} \psi_{q,p}(x_p)\Big).$$

The inner functions $\psi_{q,p}$ can be chosen *independent of* f (Lorentz 1962; Sprecher 1965).

KAN architecture: a learnable function on every edge

An **MLP** puts a fixed activation σ on each node and a learnable scalar weight on each edge — one layer is $x \mapsto \sigma(Wx + b)$. A **KAN** reverses both. The nodes only *sum*. Each edge carries its own **learnable one-variable function** ϕ — no weight matrix, no node nonlinearity.

One layer $(n_{\text{in}} \to n_{\text{out}})$ is a matrix of edge functions $\{\phi_{j,i}\}$, with output

$$(\Phi(x))_j = \sum_{i=1}^{n_{\text{in}}} \phi_{j,i}(x_i), \qquad j = 1, \ldots, n_{\text{out}}.$$

Each edge is a fixed residual plus a spline, $\phi(x) = w_b\, b(x) + w_s \sum_m c_m B_m(x)$, where $b(x) = \text{silu}(x)$ and the B_m are cubic $(k = 3)$ **B-splines** on a grid of knots. The trainable parameters are the coefficients c_m and the two scales w_b, w_s.

A **deep KAN** stacks L layers, $\Phi_{L-1} \circ \cdots \circ \Phi_0$. The Kolmogorov–Arnold theorem is the shallow case (shape $[n, 2n{+}1, 1]$). KANs lift it to arbitrary depth and width. The spline grid can be **refined** mid-training (more knots) to raise accuracy with no restart.

KAN for equation discovery: prune, then symbolify

Because every edge is an explicit one-variable function, a trained KAN can be turned into a closed-form equation:

1. **Sparsify** — train with an L_1 penalty (plus an entropy term) on the edge functions, driving unneeded edges toward zero.
2. **Prune** — remove edges and nodes below a magnitude threshold, leaving a small graph.
3. **Symbolify** — fit each surviving edge spline to a library of forms ($\sin$, x^2, e^x, $\log$, …), picking the best by R^2 and allowing an affine fit $c\,f(ax+b)+d$.
4. **Refit** — freeze the symbolic forms, re-optimize only the constants (a, b, c, d), and read off the formula.

Evidence (Liu et al., ICLR 2025).

- KANs with $\mathcal{O}(10^2)$ parameters match MLPs with $\mathcal{O}(10^3)$ parameters on PDE-solving and physics data-fitting benchmarks.
- Trained on a known physical law (special relativity; a quantum system), a KAN recovers the symbolic expression by inspecting its edge splines.

LLM-SR: equation discovery via LLM program proposals (Shojaee et al., ICLR 2025)

Paradigm shift. PDE-FIND (see previous frame) fixes a *library* and runs sparse regression. LLM-SR replaces the library by an LLM. The LLM proposes whole **equation skeletons as Python programs** $f(\mathbf{x}, \texttt{params})$. Then it numerically fits the placeholders. (arXiv:2404.18400)

Loop (three steps per iteration):

1. LLM (Mixtral-8x7B or GPT-3.5) gets prompt = problem spec + a few high-scoring past hypotheses; emits b new Python skeletons.
2. Each skeleton runs `numpy+BFGS` (Broyden–Fletcher–Goldfarb–Shanno) or `torch+Adam` to fit `params`; score $s = -\mathrm{MSE}(\hat{y}, y)$ ($\hat{y}$: predicted, y: measured).
3. FunSearch-style island buffer (Romera-Paredes et al., Nature 2024): m populations, Boltzmann sampling, cluster-by-score diversity.

Worked example (nonlinear damped oscillator, "Oscillation 1"): $\dot{v} = F\sin(\omega x) - \alpha v^2 - \beta x^3 - \gamma x v - x\cos(x)$. Mixtral proposes a skeleton with driving / damping / restoring blocks. BFGS fits the constants. Best score reached in $\sim$2,500 iterations (vs $>$2M for PySR, Python Symbolic Regression, Cranmer 2023).

LLM-SR vs PySR: out-of-domain error on four benchmarks

Each row is an ODE fitted to training data. Both methods are then evaluated on a *held-out input range* not seen during fitting. **OOD NMSE** $= \|\hat{y} - y\|^2 / \|y\|^2$. **Lower is better**. A value > 1 means the formula fails outside the training range.

Task (OOD NMSE ↓)	PySR ($\geq$2M iter)	LLM-SR Mixtral (2.5k iter)
Oscillation 1 — custom nonlinear ODE	0.310	$\mathbf{2\times10^{-4}}$
Oscillation 2 — custom nonlinear ODE	**0.010**	0.029
E. coli bacterial growth ODE	10.14	**0.0037**
Steel stress-strain (experimental data)	0.130	**0.095**

Bold = smaller error (better) on that row. LLM-SR wins 3 of 4 tasks with $2{,}500$ iterations vs PySR's $\geq$2M ($800\times$ fewer). On *E. coli*, PySR's extrapolation error (10.14) is $2{,}700\times$ higher than LLM-SR's (0.0037): the evolutionary method found a formula that fits in-distribution but fails outside it. Exception: Oscillation 2, where PySR is $3\times$ more accurate.

Software and tools

DeepXDE versus a hand-coded PINN, on identical ground

Burgers' equation $u_t + u\,u_x = \nu u_{xx}$, $\nu = 0.01/\pi$, $u(0,x) = -\sin(\pi x)$ on $[-1,1]$; a 4×64 $\tanh$ MLP (12,737 parameters), 10,000 collocation points, 15,000 Adam steps followed by L-BFGS (PyTorch 2.x, CPU). DeepXDE 1.15.0 (Lu Lu et al., *SIAM Rev.* 2021) versus the same architecture and optimizer budget, coded from scratch:

implementation	rel. L^2	wall time
hand-rolled (15k Adam + 500 L-BFGS steps)	5.2×10^{-3}	715 s
DeepXDE (15k Adam + L-BFGS to convergence)	3.1×10^{-4}	486 s

The library is $\sim17\times$ more accurate and $1.5\times$ faster on the same problem. The difference is almost entirely the L-BFGS phase: DeepXDE runs it to its convergence criterion, the hand-rolled version stopped at 500 steps. The optimizer schedule, not the network, was the hand-rolled version's bottleneck.

The high-dimensional toolboxes

- **DeepBSDE** (github.com/frankhan91/DeepBSDE; maintained by Jiequn Han) — the official code, TensorFlow 2, MIT license, still updated; runs the $d = 100$ HJB out of the box from a JSON config.
- **HighDimPDE.jl** (github.com/SciML/HighDimPDE.jl; Julia, MIT, v2.3.0 April 2026) — the only *maintained multi-algorithm* toolkit: Deep BSDE, deep splitting, multilevel Picard, optimal stopping — the neural line and the proven Monte-Carlo line side by side.
- **SDGD_PINN** (github.com/zheyuanhu01/SDGD_PINN; the first author's official PyTorch code) — the 10^5-dimensional scripts.
- **ferminet** (github.com/google-deepmind/ferminet; official, JAX, actively developed) — includes Psiformer, excited states, periodic systems. Molecules also: **deepqmc** (PauliNet lineage); lattice spins: **NetKet** (github.com/netket/netket).

References

References: solving one PDE

- I. E. Lagaris, A. Likas, D. I. Fotiadis, *Artificial neural networks for solving ordinary and partial differential equations*, IEEE Trans. Neural Networks 9 (1998) 987–1000.
- M. Raissi, P. Perdikaris, G. E. Karniadakis, *Physics-informed neural networks*, J. Comput. Phys. 378 (2019) 686–707.
- W. E, B. Yu, *The Deep Ritz Method*, Commun. Math. Stat. 6 (2018) 1–12.
- J. Sirignano, K. Spiliopoulos, *DGM: a deep learning algorithm for solving partial differential equations*, J. Comput. Phys. 375 (2018) 1339–1364.
- J. Han, A. Jentzen, W. E, *Solving high-dimensional partial differential equations using deep learning*, PNAS 115 (2018) 8505–8510.
- P. Grohs, F. Hornung, A. Jentzen, P. von Wurstemberger, *A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of Black–Scholes PDEs*, Mem. AMS (2023); arXiv:1809.02362.

References: operators, training, discovery

- T. Chen, H. Chen, *Universal approximation to nonlinear operators by neural networks…*, IEEE Trans. Neural Networks 6 (1995) 911–917.
- L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis, *Learning nonlinear operators via DeepONet*, Nat. Mach. Intell. 3 (2021) 218–229.
- Z. Li, N. Kovachki, K. Azizzadenesheli, et al., *Fourier Neural Operator for Parametric PDEs*, ICLR (2021).
- N. Kovachki, Z. Li, et al., *Neural Operator: Learning Maps Between Function Spaces*, JMLR 24 (2023).
- S. Wang, X. Yu, P. Perdikaris, *When and why PINNs fail to train: a neural tangent kernel perspective*, J. Comput. Phys. 449 (2022); and Wang & Perdikaris, SIAM J. Sci. Comput. 43 (2021).
- S. H. Rudy, S. L. Brunton, J. L. Proctor, J. N. Kutz, *Data-driven discovery of partial differential equations*, Sci. Adv. 3 (2017) e1602614.
- G. E. Karniadakis, et al., *Physics-informed machine learning*, Nat. Rev. Phys. 3 (2021) 422–440.

References: theory and recent work

- M. Hutzenthaler, A. Jentzen, T. Kruse, T. A. Nguyen, P. von Wurstemberger, *Overcoming the curse of dimensionality…semilinear parabolic PDEs* (multilevel Picard), Proc. R. Soc. A 476 (2020); W. E, J. Han, A. Jentzen, *…from nonlinear Monte Carlo to machine learning*, Nonlinearity 35 (2021).
- Y. Shin, J. Darbon, G. E. Karniadakis, *On the convergence of PINNs*, Commun. Comput. Phys. 28 (2020); S. Mishra, R. Molinaro, *Estimates on the generalization error of PINNs*, IMA J. Numer. Anal. 43 (2023).
- S. Lanthaler, S. Mishra, G. E. Karniadakis, *Error estimates for DeepONets*, Trans. Math. Appl. 6 (2022).
- A. S. Krishnapriyan, et al., *Characterizing possible failure modes in PINNs*, NeurIPS (2021); S. Wang, S. Sankaran, P. Perdikaris, *Respecting causality…*, arXiv:2203.07404 (2022).
- J. He, L. Li, J. Xu, C. Zheng, *ReLU deep neural networks and linear finite elements*, J. Comput. Math. 38 (2020).
- Y. Zang, G. Bao, X. Ye, H. Zhou, *Weak adversarial networks…*, J. Comput. Phys. 411 (2020); Z. Li, et al., *Physics-Informed Neural Operator*, arXiv:2111.03794 (2021).
- L. Yang, S. Liu, T. Meng, S. J. Osher, *In-context operator learning…*, PNAS 120 (2023); M. Herde, et al., *Poseidon*, NeurIPS (2024); Z. Liu, et al., *KAN: Kolmogorov–Arnold Networks*, ICLR (2025).