Operator Learning

Scientific Machine Learning

Alena Kopaničáková

ENSEEIHT / Toulouse INP

Spring 2026

Motivation: The Many-Query Problem

From a Single Solve to a Family of Solves

Classical PDE solvers and PINNs compute u for *one* input at a time

Many applications need u for *thousands* of different inputs:

1. **Design optimisation:** Evaluate drag for each candidate geometry
2. **Uncertainty quantification:** Propagate uncertain coefficients $a(\boldsymbol{x}; \omega)$ through the PDE
3. **Real-time control:** Need $u(a)$ at actuator sampling rate
4. **Inverse problems:** Forward map evaluated repeatedly in MCMC / optimisation
5. **Digital twins:** Continuously update u as new data arrives

The many-query bottleneck

Running a PDE solver 10^4 - 10^6 times is often intractable
Idea: *Amortise* all solves into a single offline learning phase

The Operator Learning Perspective

The PDE defines a **map between function spaces**:

$$\mathcal{G}^\dagger : \mathcal{X} \longrightarrow \mathcal{Y}, \qquad \mathcal{G}^\dagger(a) = u_a$$

- $a \in \mathcal{X}$: input function (forcing, coefficient, IC, geometry)
- $u_a \in \mathcal{Y}$: corresponding PDE solution
- Both $\mathcal{X}$ and $\mathcal{Y}$ are *infinite-dimensional* function spaces

Example: Darcy flow on $\Omega = [0,1]^2$

$$-\nabla \cdot \big(a(\boldsymbol{x})\nabla u(\boldsymbol{x})\big) = f(\boldsymbol{x}), \quad u\big|_{\partial\Omega} = 0$$

$\mathcal{G}^\dagger : a \mapsto u$ maps permeability $\rightarrow$ pressure
Each evaluation costs $\mathcal{O}(N_h^{1.5})$; UQ may require 10^4–10^6 solves

Operator Learning: The Core Idea

Replace $\mathcal{G}^\dagger$ with a **learned surrogate** $\widehat{\mathcal{G}} \approx \mathcal{G}^\dagger$:

Key properties of $\widehat{\mathcal{G}}$

- Trained **offline** on N input–output pairs $\{(a_n, u_n)\}$
- Evaluated **online** in $\mathcal{O}(1)$ or $\mathcal{O}(N_h \log N_h)$
- Generalises to new inputs $a \sim \mu$ not seen during training

Operator Learning Problem

Observe N i.i.d. pairs $\{(a_n, u_n)\}_{n=1}^N$, $a_n \sim \mu$, $u_n = \mathcal{G}^\dagger(a_n) + \varepsilon_n$

Find $\widehat{\mathcal{G}} \in \widehat{\mathcal{G}}$ minimising the **population risk**:

$$\mathcal{R}(\widehat{\mathcal{G}}) = \mathbb{E}_{a\sim\mu}\left[\frac{\left\|\widehat{\mathcal{G}}(a) - \mathcal{G}^\dagger(a)\right\|_{\mathcal{Y}}^2}{\|\mathcal{G}^\dagger(a)\|_{\mathcal{Y}}^2}\right]$$

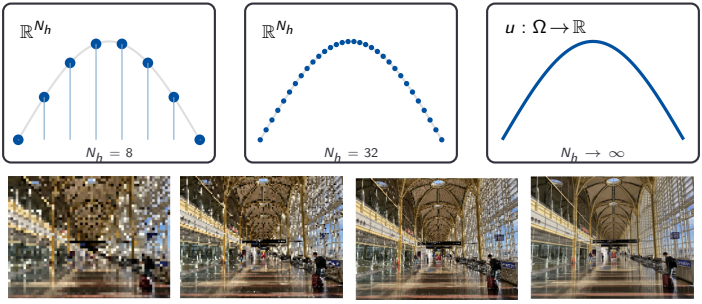
In practice, estimated by the **empirical risk**:

$$\widehat{\mathcal{R}}(\widehat{\mathcal{G}}) = \frac{1}{N}\sum_{n=1}^{N}\frac{\left\|\widehat{\mathcal{G}}(a_n) - u_n\right\|_{\mathcal{Y}}^2}{\|u_n\|_{\mathcal{Y}}^2}$$

Why relative norm? Prevents the loss from being dominated by outputs with large $\|u\|_{\mathcal{Y}}$

Vectors vs. Functions: A Conceptual Shift

A discretised field can be viewed as a vector *or* as a function sampled at different resolutions:



(a) Fixed resolution

Design the algorithm at a specific N_h
CNN: $\mathbb{R}^{N_h} \to \mathbb{R}^{N_h}$
Changing resolution $\Rightarrow$ retrain from scratch

(b) Continuum first

Define the algorithm in function space ($N_h = \infty$), then discretise
Same model works at *any* resolution
$\Rightarrow$ **Discretisation invariance**

Sampling Input Functions

How to Generate Training Data

Need N samples $a_n \sim \mu$ from a distribution on *function space*

Most common: Gaussian Random Fields (GRF)

GRF definition

$a \sim \mathrm{GRF}(m, k)$:

- Mean $\mathbb{E}[a(\boldsymbol{x})] = m(\boldsymbol{x})$ (usually 0)
- Covariance $\mathrm{Cov}[a(\boldsymbol{x}), a(\boldsymbol{x}')] = k(\boldsymbol{x}, \boldsymbol{x}')$

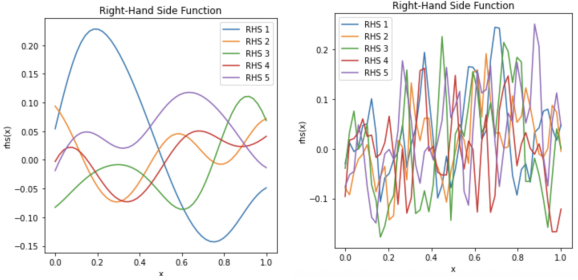
Common kernels

RBF: $k(r) = \sigma^2 e^{-r^2/(2\ell^2)} \to C^\infty$ samples

Matérn-ν: $\to C^{\lceil \nu \rceil - 1}$ samples

$\nu = 1/2$: continuous, not differentiable

$\nu \to \infty$: recovers RBF



Role of ℓ

Left: Large ℓ - smooth, low-dim

Right: Small ℓ - rough, high-dim

Overview of Sampling Strategies

Strategy	**Regularity of a**	**Eff. dimension**	**Typical use**
Pointwise i.i.d.	None (white noise)	N_h	Stress tests; rarely used
Piecewise constant	L^∞, discontinuous	K (num. blocks)	Layered media, two-phase
GRF (RBF kernel)	C^∞	Low ($\lesssim 20$)	Most benchmarks (Darcy, NS)

Distribution matters!

If μ produces only smooth fields but deployment needs rough inputs, the surrogate *will fail* on out-of-distribution data.

Always match μ to the **deployment distribution**.

Classical Approach: POD and ROM

The Solution Manifold

Key observation

PDE solutions do *not* fill the output space $\mathcal{Y}$ - the PDE imposes strong constraints (e.g., smoothing)
The solutions $\{u_a : a \in \mathcal{X}\}$ trace out a "thin" subset

Solution manifold

$$\mathcal{M} = \{\mathcal{G}^\dagger(a) : a \in \mathcal{X}\} \subset \mathcal{Y}$$

Even though $\mathcal{Y}$ is ∞-dimensional, $\mathcal{M}$ often has *low intrinsic dimension*

Why low-dimensional?

1. **Smoothing by PDE:** elliptic/parabolic operators suppress high-frequency content
2. **Smooth dependence on input:** nearby $a \Rightarrow$ nearby u

Kolmogorov r-width

$$d_r(\mathcal{M}) = \inf_{\substack{V_r \subset \mathcal{Y} \\ \dim V_r = r}} \sup_{u \in \mathcal{M}} \inf_{v \in V_r} \|u - v\|_{\mathcal{Y}}$$

Best worst-case error of any r-dim linear subspace

Decay rates

Elliptic: $d_r = \mathcal{O}(e^{-cr})$
Transport: $d_r = \mathcal{O}(r^{-1/2})$

POD: The Optimal Linear Subspace

Snapshot matrix from N solutions on N_h-point mesh:

$$\boldsymbol{U} = [u_1 \mid \cdots \mid u_N] \in \mathbb{R}^{N_h \times N}$$

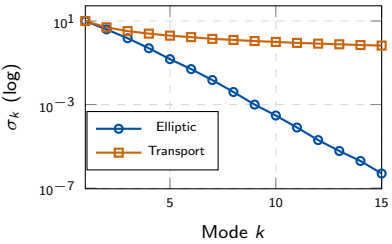
SVD: $\boldsymbol{U} = \boldsymbol{\Phi}\boldsymbol{\Sigma}\boldsymbol{V}^\top$

Columns $\phi_1, \ldots, \phi_r$ of $\boldsymbol{\Phi}$ are **POD modes**

Truncation error:

$$\frac{1}{N}\sum_{n=1}^{N} \|u_n - \Pi_{V_r} u_n\|_2^2 = \sum_{k=r+1}^{N} \sigma_k^2$$

Choose r so that 99%–99.9% of energy is retained



POD-Based Reduced Order Model

ROM approximation: $u_a \approx \sum_{k=1}^{r} c_k(a)\,\phi_k = \mathbf{\Phi}_r \boldsymbol{c}(a), \quad \boldsymbol{c}(a) \in \mathbb{R}^r$

Intrusive (Galerkin projection)

Substitute into PDE weak form, project onto V_r:

$$\underbrace{(\mathbf{\Phi}_r^\top \boldsymbol{A}(a)\mathbf{\Phi}_r)}_{\mathbb{R}^{r\times r}} \boldsymbol{c}(a) = \mathbf{\Phi}_r^\top \boldsymbol{f}$$

- Cost: $\mathcal{O}(r^3 + N_h r^2)$ vs. $\mathcal{O}(N_h^{1.5})$
- Requires access to PDE

Non-intrusive (data-driven)

- Learn $a \mapsto \boldsymbol{c}(a)$ from training data using regression (linear, GP, or neural network)
- No PDE access needed
- This is the approach adopted by **PCA-Net**

POD error decomposition

$$\left\| u_a - \widehat{\mathcal{G}}(a) \right\| \leq \underbrace{d_r(\mathcal{M})}_{\text{approx (truncation)}} + \underbrace{\text{regression error}}_{\text{learning } a\mapsto \boldsymbol{c}(a)} + \underbrace{\text{optimisation error}}_{\text{zero for Galerkin}}$$

Limitations of POD-ROM

1. **No mesh transfer:**
   - POD modes $\phi_k \in \mathbb{R}^{N_h}$ are discrete vectors
   - Refining the mesh ($N_h \to 4N_h$) $\Rightarrow$ rebuild everything from scratch

2. **Offline cost = solver cost:**
   - Generating N snapshots costs $\mathcal{O}(N \cdot N_h^{1.5})$

3. **Slow for transport:**
   - $d_r = \mathcal{O}(r^{-1/2}) \Rightarrow$ need large r

4. **No off-mesh evaluation:**
   - Output is a vector of nodal values, not a continuous function

Darcy example (64^2 mesh)

	Cost
Full solve	$\mathcal{O}(N_h^{1.5})$
ROM ($r = 12$)	$\mathcal{O}(N_h r)$
Speedup	$\sim 5\times$

How to switch to resolution: 128^2?
$\Rightarrow$ Recompute all 500 snapshots, re-run SVD, retrain regression

Motivation for neural operators

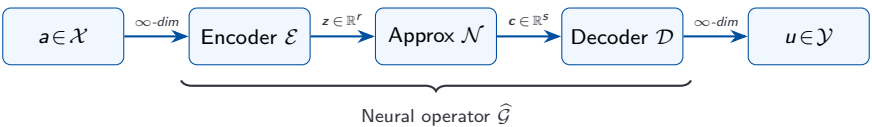
Replace discrete POD modes with *continuous* representations $\Rightarrow$ discretisation invariance

The Encode–Approximate–Decode Framework

The Universal Template

All operator learning architectures follow the same three-step template:

$$\widehat{\mathcal{G}}(a) = \underbrace{\mathcal{D}}_{\text{decode}} \circ \underbrace{\mathcal{N}}_{\text{approximate}} \circ \underbrace{\mathcal{E}(a)}_{\text{encode}}$$



1. **Encode** $\mathcal{E} : \mathcal{X} \to \mathbb{R}^r$: compress the input to a latent vector
2. **Approximate** $\mathcal{N} : \mathbb{R}^r \to \mathbb{R}^s$: nonlinear map in latent space
3. **Decode** $\mathcal{D} : \mathbb{R}^s \to \mathcal{Y}$: reconstruct a function in $\mathcal{Y}$

How Architectures Instantiate the Template

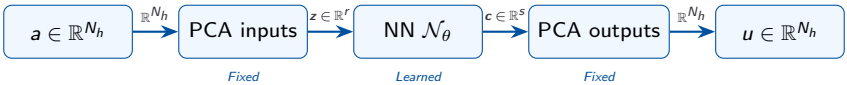
The architectures differ in *which components are fixed vs. learned*:

	Encoder $\mathcal{E}$	**Approx** $\mathcal{N}$	**Decoder** $\mathcal{D}$
PCA-Net	Fixed (PCA proj.)	**Learned** (deep NN)	Fixed (POD basis)
DeepONet	**Learned** (branch)	Trivial (dot product)	**Learned** (trunk)
FNO	**Learned** (lift $\mathcal{P}$)	**Learned** (T layers)	**Learned** (proj $\mathcal{Q}$)

PCA-Net: From POD to Learned Regression

PCA-Net Architecture

Idea: Two separate PCA decompositions (inputs & outputs) + a neural network in between



Three steps

1. **Encode:** $z = \Psi_r^\top (a - \bar{a}) \in \mathbb{R}^r$
2. **Approximate:** $c = \mathcal{N}_\theta(z) \in \mathbb{R}^s$
3. **Decode:** $\widehat{\mathcal{G}}(a) = \bar{u} + \Phi_s c$

Training loss

Only the NN $\mathcal{N}_\theta$ is trained:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \frac{\|\bar{u} + \Phi_s \mathcal{N}_\theta(z_n) - u_n\|_2^2}{\|u_n\|_2^2}$$

Standard finite-dimensional regression!

PCA-Net: Algorithm

Offline phase

1. Compute **input PCA**: SVD of centred input matrix $\boldsymbol{A} = [a_1 - \bar{a} \mid \cdots \mid a_N - \bar{a}]$
   Keep top-r left singular vectors $\boldsymbol{\Psi}_r$
2. Compute **output PCA**: SVD of centred output matrix
   $\boldsymbol{U} = [u_1 - \bar{u} \mid \cdots \mid u_N - \bar{u}]$
   Keep top-s left singular vectors $\boldsymbol{\Phi}_s$
3. Encode all pairs: $\boldsymbol{z}_n = \boldsymbol{\Psi}_r^\top (a_n - \bar{a})$, $\boldsymbol{c}_n = \boldsymbol{\Phi}_s^\top (u_n - \bar{u})$
4. Train $\mathcal{N}_\theta : \mathbb{R}^r \to \mathbb{R}^s$

Online phase (new input a_{new})

1. Encode: $\boldsymbol{z} = \boldsymbol{\Psi}_r^\top (a_{\text{new}} - \bar{a})$
2. Map: $\boldsymbol{c} = \mathcal{N}_\theta(\boldsymbol{z})$
3. Decode: $u_{\text{pred}} = \bar{u} + \boldsymbol{\Phi}_s \boldsymbol{c}$

Why two separate PCAs?

- Functions a and u can live in different spaces with different structure, e.g. rough a vs. smooth u
- No reason the same basis compresses both well

PCA-Net: Error Decomposition and Limitations

Error decomposition (Lanthaler 2023)

$$\mathcal{R}(\widehat{\mathcal{G}}) \leq \underbrace{C_1 d_r(\mathcal{M})^2}_{\text{encoding error}} + \underbrace{C_2 d_s(\mathcal{M})^2}_{\text{decoding error}} + \underbrace{\mathcal{R}_{\mathrm{NN}}(\mathcal{N}_\theta)}_{\text{NN regression error}}$$

Encoding/decoding errors are purely geometric (SVD truncation)
NN regression brings approximation-generalisation-optimisation

Limitations:

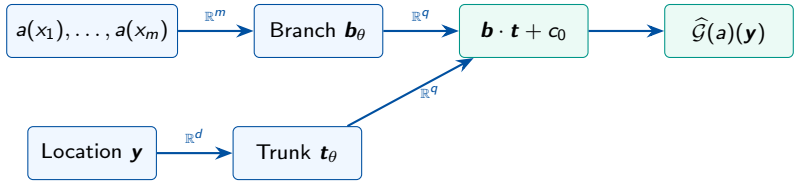
- **Linear encoder/decoder:** PCA assumes linear subspace For transport PDEs with slow Kolmogorov decay $\Rightarrow$ need very large r
- **Fixed discretisation:** PCA modes Φ_s computed on training mesh; *not* discretisation invariant
- **Offline SVD cost:** $\mathcal{O}(\min(N_h, N) \cdot N_h \cdot N)$, mitigated by randomised SVD

DeepONet: Universal Approximation for Operators

DeepONet Architecture

Lu et al. (2021): replace single hidden layers with **deep networks**:

$$\widehat{\mathcal{G}}_\theta(a)(\boldsymbol{y}) = \boldsymbol{b}_\theta\big(a(x_1), \ldots, a(x_m)\big) \cdot \boldsymbol{t}_\theta(\boldsymbol{y}) + c_0$$



Through the EAD lens

- **Encoder** = branch net (learned, nonlinear)
- **Approx** = dot product (trivial, no params)
- **Decoder** = trunk net (learned, continuous)

vs. PCA-Net

PCA-Net: fixed encode/decode, learned middle
DeepONet: learned encode/decode, trivial middle
All the work shifts to the *encoder* and *decoder*

DeepONet: Training and Properties

Training loss:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{N\,N_h} \sum_{n=1}^{N} \sum_{j=1}^{N_h} \frac{\left| \widehat{\mathcal{G}}_\theta(a_n)(\boldsymbol{y}_j) - u_n(\boldsymbol{y}_j) \right|^2}{\|u_n\|_2^2}$$

Optimised by Adam or L-BFGS

Advantages

- **Continuous output:** Trunk evaluates at any $\boldsymbol{y}$
- **Universal:** Chen & Chen guarantee
- **Low-rank kernel:** $\boldsymbol{b}(a)\cdot\boldsymbol{t}(\boldsymbol{y})$ is rank-q

Limitations

- **Fixed sensors:** Branch requires same m sensor points
- **Not disc. invariant** (original formulation)
- **Bilinear bottleneck:** Output is rank-q in trunk basis

Example: Darcy flow

Branch: 4 layers, width 128, ReLU; $m = 100$ sensors. Trunk: 4 layers, width 128, tanh; $q = 64$.
$N = 1000$; relative L^2 error $\approx 1.5\%$. Online: $\sim$0.3ms vs. 150ms (FEM) = 500$\times$ speedup.

The Chen & Chen Theorem (1995)

Theorem (Universal Approximation for Operators)

Let σ be continuous non-polynomial, $K \subset \mathcal{X}$ compact, $\mathcal{G}^\dagger : K \to C(\Omega_y)$ continuous. For any $\varepsilon > 0$, $\exists\, m, p$ s.t.:

$$\widehat{\mathcal{G}}(a)(\boldsymbol{y}) = \sum_{k=1}^{p} \underbrace{b_k\left(\sigma\left(\sum_{i=1}^{m} w_{ki}\, a(x_i) + \beta_k\right)\right)}_{\text{branch net: encodes } a} \cdot \underbrace{t_k(\boldsymbol{y})}_{\text{trunk: encodes } \boldsymbol{y}}$$

satisfies $\sup_{a,\boldsymbol{y}} \left|\widehat{\mathcal{G}}(a)(\boldsymbol{y}) - \mathcal{G}^\dagger(a)(\boldsymbol{y})\right| < \varepsilon$

Key insight: The output is a **dot product** between branch and trunk $\Rightarrow$ **rank-p decomposition** of the operator kernel:

$$\mathcal{G}(\boldsymbol{y}, \boldsymbol{x}) \approx \sum_{k=1}^{p} t_k(\boldsymbol{y})\, \phi_k(\boldsymbol{x})$$

DeepONet Variants

The bilinear bottleneck $\widehat{\mathcal{G}}(a)(\boldsymbol{y}) = \sum_k b_k(a)\, t_k(\boldsymbol{y})$ motivates several extensions:

	Encoder	**Approx.**	**Decoder**
PCA-Net	Fixed (PCA)	**Learned** (NN)	Fixed (POD)
DeepONet	**Learned** (branch)	Trivial (dot prod.)	**Learned** (trunk)
POD-DeepONet	**Learned** (branch)	Trivial (dot prod.)	Fixed (POD)
NOMAD	**Learned** (branch)	**Learned** (nonlinear decoder)	
Full autoencoder	**Learned**	**Learned**	**Learned**

- **POD-DeepONet:** - fixed POD decoder frees branch for nonlinear encoding
- **Shift-DeepONet:** - adds a learned shift - helps for transport operators
- **NOMAD:** - nonlinear decoder $\mathcal{D}_\theta(\boldsymbol{b}(a), \boldsymbol{y})$ - breaks bilinear bottleneck

Fourier Neural Operator (FNO)

Preliminaries

Work on $\mathbb{T}^d = [0, 2\pi]^d$ (periodic). For $f \in L^2(\mathbb{T}^d)$:

Fourier series

$$\widehat{f}_k \;=\; \frac{1}{(2\pi)^d} \int_{\mathbb{T}^d} f(x)\, e^{-ik\cdot x}\, dx, \qquad f(x) \;=\; \sum_{k\in\mathbb{Z}^d} \widehat{f}_k\, e^{ik\cdot x}$$

- $\{(2\pi)^{-d/2}\, e^{ik\cdot x}\}_{k\in\mathbb{Z}^d}$ is a complete orthonormal basis of $L^2(\mathbb{T}^d)$

Plancherel / Parseval

$$\|f\|_{L^2}^2 \;=\; (2\pi)^d \sum_{k\in\mathbb{Z}^d} |\widehat{f}_k|^2$$

The Fourier transform $\mathcal{F} : L^2(\mathbb{T}^d) \to \ell^2(\mathbb{Z}^d)$ is an isometry — mode truncation errors are L^2 errors

Decay of Fourier coefficients

For $f \in H^s(\mathbb{T}^d)$:

$$|\widehat{f}_k| \;=\; \mathcal{O}(|k|^{-s}) \;\text{ as } |k| \to \infty$$

Smoother function $\Rightarrow$ faster decay

Preliminaries: The Fourier transform on $\mathbb{R}^d$

For $f \in L^1(\mathbb{R}^d) \cap L^2(\mathbb{R}^d)$:

Fourier transform and its inverse

$$\widehat{f}(\xi) \;=\; (\mathcal{F}f)(\xi) \;=\; \int_{\mathbb{R}^d} f(x)\, e^{-2\pi i\, \xi\cdot x}\, dx, \qquad f(x) \;=\; (\mathcal{F}^{-1}\widehat{f})(x) \;=\; \int_{\mathbb{R}^d} \widehat{f}(\xi)\, e^{2\pi i\, \xi\cdot x}\, d\xi$$

Properties (used throughout FNO)

Physical space	Fourier space
$\alpha f + \beta g$	$\alpha\widehat{f} + \beta\widehat{g}$
$\partial_{x_j} f$	$2\pi i\, \xi_j\, \widehat{f}(\xi)$
$D^\alpha f$	$(2\pi i\, \xi)^\alpha \widehat{f}(\xi)$
$f(x-a)$	$e^{-2\pi i\, \xi\cdot a}\, \widehat{f}(\xi)$
$(f * g)(x)$	$\widehat{f}(\xi)\, \widehat{g}(\xi)$
$f(x)\cdot g(x)$	$(\widehat{f} * \widehat{g})(\xi)$

The key property for PDEs

$\mathcal{F}$ diagonalises differential operators: e.g. $-\Delta u = f$ becomes $4\pi^2|\xi|^2\, \hat{u}(\xi) = \widehat{f}(\xi)$, so $\hat{u}(\xi) = \widehat{f}(\xi)/(4\pi^2|\xi|^2)$ — explicit solution by pointwise division

Preliminaries: Integral kernels

Many operators on function spaces are *integral operators*.

Definition

An *integral operator* $\mathcal{K} : L^2(D_y) \to L^2(D_x)$ has the form

$$(\mathcal{K}v)(x) \;=\; \int_{D_y} \kappa(x,y)\, v(y)\, dy,$$

where $\kappa : D_x \times D_y \to \mathbb{R}$ is the *kernel*: the influence of $v(y)$ on the output at x.

Discrete view: On a uniform grid with n points: $(Kv)_i = (|D|/n) \sum_j \kappa(x_i, y_j)\, v_j$, $K \in \mathbb{R}^{n \times n}$

Familiar examples

- Identity: $\kappa(x,y) = \delta(x-y)$
- Global mean: $\kappa(x,y) = |D|^{-1}$
- Projection onto $\{\phi_k\}$: $\kappa(x,y) = \sum_k \phi_k(x)\phi_k(y)$

The example we care about

Green's function $\kappa(x,y) = G(x,y)$ — kernel of the *solution operator* $\mathcal{L}^{-1}$ of a linear PDE (next slide)

The Green's function: kernel of the solution operator

Let $\mathcal{L}$ be a linear differential operator on $D \subset \mathbb{R}^d$
The *Green's function* $G(x, y)$ is the response at x to a unit point source at y:

$$\mathcal{L}_x\, G(x, y) \;=\; \delta(x - y), \quad x \in D \quad (+\text{boundary conditions})$$

By linearity, the solution to $\mathcal{L}u = f$ is

$$\boxed{u(x) \;=\; (\mathcal{L}^{-1} f)(x) \;=\; \int_D G(x, y)\, f(y)\, dy}$$

The solution operator *is* an integral operator with kernel G

Three canonical examples

- *1D Poisson* $-u'' = f$ on $[0, 1]$: $G(x, y) =$ triangle function (peaked at y)
- *3D Poisson* $-\Delta u = f$: $G(x, y) = \frac{1}{4\pi|x-y|}$ (Newtonian potential)
- *Heat* $\partial_t u - \Delta u = 0$: $G_t(x, y) = (4\pi t)^{-d/2}\, e^{-|x-y|^2/(4t)}$

FNO learns G (or rather: its Fourier multiplier) directly from data

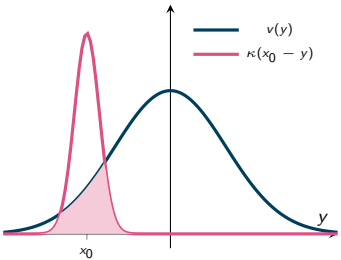
Convolution: translation-invariant integral kernels

When $\kappa(x, y) = \kappa(x - y)$ the integral operator becomes a *convolution*:

$$(\mathcal{K}v)(x) \;=\; (\kappa * v)(x) \;:=\; \int_{\mathbb{R}^d} \kappa(x - y)\, v(y)\, dy$$

Storage gain

- General kernel: n^2 values (one per pair (x_i, y_j))
- Convolution kernel: n values (one per offset $x - y$)
- Weight sharing: the same filter is applied at every position
- Physically: *translation-invariant local physics*



Worked example: 1D convolution (box-averaging kernel)

Signal f sampled at $y = 0, 1, 2, 3, 4$ and a 3-point averaging kernel κ:

y	0	1	2	3	4
$f(y)$	1	2	3	2	1

z	-1	0	1
$\kappa(z)$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

$\kappa(z) = 0$ for $|z| > 1$

Discrete convolution: $(f * \kappa)(x) = \sum_y f(y)\,\kappa(x - y)$

One output position: $x = 2$

y	0	1	2	3	4
$x - y$	2	1	0	-1	-2
$\kappa(x-y)$	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0
$f(y)$	1	2	3	2	1
$f(y)\,\kappa(x-y)$	0	$\frac{2}{3}$	1	$\frac{2}{3}$	0

Sum: $(f * \kappa)(2) = \frac{2}{3} + 1 + \frac{2}{3} = \frac{7}{3} \approx 2.33$

All output positions (sliding window)

x	window weights	$(f * \kappa)(x)$
0	$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2$	1
1	$\frac{1}{3}(1+2+3)$	2
2	$\frac{1}{3}(2+3+2)$	$\frac{7}{3}$
3	$\frac{1}{3}(3+2+1)$	2
4	$\frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 1$	1

Boundary effect - zero-padding

Convolution with a positive kernel always smooths $\Longrightarrow$ *Output* $[1, 2, \frac{7}{3}, 2, 1]$ *is smoother than the input* $[1, 2, 3, 2, 1]$

Worked example: convolution as a matrix-vector product

The full computation $(f * \kappa)$ can be written as Cf, where C is the *matrix* built from κ:

$$Cf = \frac{1}{3}\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 7/3 \\ 2 \\ 1 \end{pmatrix}$$

Structural properties

- Each row is the same pattern of weights, *shifted* by one position
- Constant along every diagonal $\Rightarrow$ Toeplitz structure
- $\mathcal{O}(n)$ distinct entries (one per diagonal), not $\mathcal{O}(n^2)$
- *Weight sharing* of CNNs is exactly this Toeplitz structure

The convolution theorem

Theorem (convolution $\leftrightarrow$ multiplication)

For $f, \kappa \in L^2$:

$$\boxed{\mathcal{F}(f * \kappa) \;=\; \widehat{f} \cdot \widehat{\kappa} \quad \Longleftrightarrow \quad f * \kappa \;=\; \mathcal{F}^{-1}\big(\widehat{f} \cdot \widehat{\kappa}\big)}$$

Convolution in physical space $\equiv$ pointwise multiplication in Fourier space

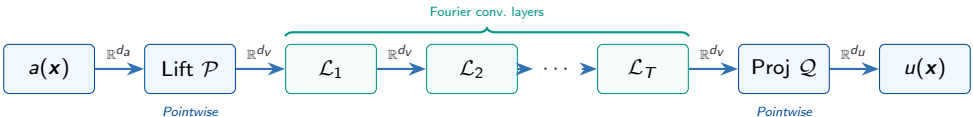
Three immediate consequences — all used in FNO:

1. **Computational efficiency:** $f * \kappa$ on N points: FFT ($\mathcal{O}(N \log N)$), multiply pointwise ($\mathcal{O}(N)$), inverse FFT $\Rightarrow \mathcal{O}(N \log N)$ total, vs. $\mathcal{O}(N^2)$ for direct convolution
2. **Mode truncation = ideal low-pass filter:** Keeping only $|k| \le k_{\max}$ corresponds to convolution with the Dirichlet kernel: $P_{k_{\max}} f = \mathcal{F}^{-1}(1_{|k| \le k_{\max}} \widehat{f})$
3. **Parametrisation by Fourier multiplier:** Any convolution $f \mapsto f * \kappa$ is fully characterised by $\widehat{\kappa}(k)$. FNO replaces the analytic $\widehat{\kappa}$ with a *learned* complex matrix $R_\theta(k) \in \mathbb{C}^{d_v \times d_v}$ at each retained mode — a nonlinear, multi-channel Fourier multiplier

Full FNO Architecture

Three stages: **Lift** $\rightarrow$ **FNO layers** $\rightarrow$ **Project**:

$$\widehat{\mathcal{G}}_\theta(a) = \underbrace{\mathcal{Q}}_{\text{project}} \circ \underbrace{(\mathcal{L}_T \circ \cdots \circ \mathcal{L}_1)}_{T \text{ FNO layers}} \circ \underbrace{\mathcal{P}(a)}_{\text{lift}}$$



- Dimensions ($\mathbb{R}^{d_a}$, $\mathbb{R}^{d_v}$, $\mathbb{R}^{d_u}$) = **channel dimension** at each x
- Spatial grid is preserved throughout (N_h points); only channels change
- Typical: $d_a = 1$, $d_v = 32$–128, $d_u = 1$, $T = 4$

The Lifting Layer $\mathcal{P}$ (Encoder)

Pointwise affine embedding: $\mathbb{R}^{d_a} \to \mathbb{R}^{d_v}$ at each spatial point:

$$\mathcal{P}(a)(\boldsymbol{x}) = W_P\, a(\boldsymbol{x}) + b_P, \qquad W_P \in \mathbb{R}^{d_v \times d_a},\ b_P \in \mathbb{R}^{d_v}$$

"Colours" a scalar input into d_v channels of features

Coordinate augmentation variant

Concatenate spatial coordinate $\boldsymbol{x}$ to input before lifting:

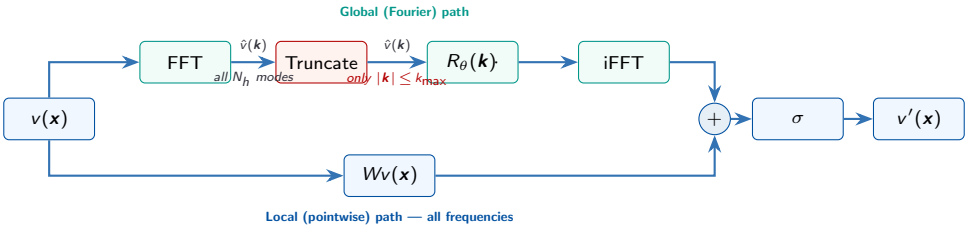
$$\mathcal{P}(a)(\boldsymbol{x}) = W_P \begin{pmatrix} a(\boldsymbol{x}) \\ \boldsymbol{x} \end{pmatrix} + b_P, \qquad W_P \in \mathbb{R}^{d_v \times (d_a + d)}$$

Gives explicit access to position - helps for *non*-translation-invariant problems

The FNO Layer (Fourier Convolution Block)

Each layer: **global Fourier path** + **local linear path** → sum → activation:

$$v'(\boldsymbol{x}) = \sigma\left(\underbrace{W v(\boldsymbol{x})}_{\text{local}} + \underbrace{\mathcal{K}_\theta[v](\boldsymbol{x})}_{\text{global Fourier}}\right), \qquad \mathcal{K}_\theta[v](\boldsymbol{x}) = \mathcal{F}^{-1}\big(R_\theta(\boldsymbol{k}) \cdot \mathcal{F}[v](\boldsymbol{k})\big)$$



- **Truncate**: Discard all modes $|\boldsymbol{k}| > k_{\max}$ (e.g. keep 12 out of $N_h/2$ modes)
- $R_\theta(\boldsymbol{k}) \in \mathbb{C}^{d_v \times d_v}$: Learnable weight matrix applied *only* to the $k_{\max}$ retained modes
- Local path $Wv(\boldsymbol{x})$ compensates: Pointwise, captures the high-frequency details that the Fourier path discards

The Projection Layer $\mathcal{Q}$ (Decoder)

After T layers: $v_T : \Omega \to \mathbb{R}^{d_v}$. Map back to output dimension d_u:

$$\mathcal{Q}(v_T)(\boldsymbol{x}) = W_2\, \sigma(W_1\, v_T(\boldsymbol{x}) + b_1) + b_2, \quad W_1 \in \mathbb{R}^{d_v \times d_v},\ W_2 \in \mathbb{R}^{d_u \times d_v}$$

Two-layer pointwise MLP: nonlinear channel mixing $\to$ linear projection

FNO Through the Encode-Approximate-Decode Lens

The Encode-Approximate-Decode template applies at **two levels**:

Global level (full pipeline)

- $\mathcal{E} = \mathcal{P}$ (lift): channels $d_a \to d_v$
- $\mathcal{N} = \mathcal{L}_T \circ \cdots \circ \mathcal{L}_1$
- $\mathcal{D} = \mathcal{Q}$ (project): channels $d_v \to d_u$

- No spatial compression - full grid preserved

Per-layer level (each $\mathcal{L}_\ell$)

- Encode: FFT $\to$ truncate to $k_{\max}$ modes
- Approximate: multiply by $R_\theta(\boldsymbol{k})$
- Decode: iFFT back to physical space

- Compression in *frequency domain* at every layer

	Encoder	**Approx.**	**Decoder**	**Latent space**
PCA-Net	Fixed (PCA)	Learned (NN)	Fixed (POD)	$\mathbb{R}^r$ (vector)
DeepONet	Learned (branch)	Trivial (dot)	Learned (trunk)	$\mathbb{R}^q$ (vector)
FNO	Learned (lift)	Learned (T layers)	Learned (proj.)	Functions on Ω

FNO: Training, and Discretisation Invariance

Training loss: $\widehat{\mathcal{R}}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \frac{\left\|\widehat{\mathcal{G}}_\theta(a_n) - u_n\right\|_2^2}{\|u_n\|_2^2}$

Backprop through FFT layers via differentiable FFT

Why discretisation invariant?

Weights $R_\theta(\boldsymbol{k})$ are indexed by *frequency* $\boldsymbol{k}$, not grid point
FFT on any grid with $N_h > 2k_{\max}$ yields the same low modes $\Rightarrow$ same weights, any resolution
In contrast, DeepONet's branch net is tied to m fixed sensors