Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Lecture notes on Physics Informed Neural Networks, Neural Operators, and their applications

From the PhD course held at the Free University of Bozen-Bolzano, valid for the Ph.D. in Computer Science

Alessandro Bombini
Istituto Nazionale di Fisica Nucleare, Sezione di Firenze
2026

Abstract

This is the set of lecture notes for the PhD course *Physics Informed Neural Network*, held at the University of Bozen in the academic year 2025/2026.

The goal of the course is to introduce the concept of Physics Informed Deep Neural Networks (PINN) and Neural Operators (NOs), discuss their implementation from scratch in PyTorch and using advanced ad-hoc developed open-source libraries such as NVIDia PhysicsNeMo to address real-world problems in various fields (engineering, physics, petroleum reservoir). We discuss recent topics such as Mixture-of-Models, Fourier Neural Operators, Physics-Informed Kolmogorov-Arnold Networks (PIKANs) and Physics-Informed Computer Vision.

Acknowledgements

I would like to thank Prof. Oswald Lanz for inviting me to held a 20 hours PhD course on this subject at the University of Bozen (academic year 2025/2026), and for motivating me in start drafting this lecture notes.

I would also like to thank Prof. Stefano Berretti for granting me the opportunity to held this course for two years (a.y. 2023/2024, 2024/2025), in a smaller form factor.

Furthermore, I would like to thank Dr. Lucio Anderlini, for pointed me towards this topic, and for for the help in our joint work on the subject.

Finally, I would like to thank Dr. Alessandro Rosa for having read the draft, and for all his invaluable comments, suggestions, critiques to the draft, which really helped me improving the quality of this lecture notes. All errors, imprecisions, confusing points, etc., are all entirely my fault.

Contents

List of Figures

List of Tables

PART I

Introductory Lectures

CHAPTER 1

Lecture 1: General Introduction to the course

chap:1

1.1 A brief introduction: why should I care?

sec:1:1

The goal of the course is to introduce the concept of Physics Informed Deep Neural Networks (PINNs) and Neural Operators (NOs), discuss their implementation from scratch in PyTorch and using advanced ad-hoc developed open-source libraries such as nvidia-PhysicsNeMo to address real-world problems in various fields (engineering, physics, oil).

The goal is to have each lecture composed by two parts:

1. Theory (frontal lectures)
2. Practice (hands-on using JuPyTer notebooks)

The code is available on GitHub[1]. The latest version of the slides for the frontal lecture, and of the code for the hands-on lecture, as well as this lecture notes as pdf file, are also freely available [Bom26].

For the interested reader, [Zha26] contains an updated (as of 3rd April 2026) living review of paper on the subject.

1.2 A brief introduction: what is a Partial Differential Equation?

sec:1:2

Let us start by introducing a bit of notation. Let us be $f : [a, b] \subset \mathbb{R} \to \mathbb{R}$ a function from a compact interval on the real numbers to the real numbers, and let it be continuous, i.e. $f \in C([a, b])$. The *different quotient* $R_h[f](x)$ of f in x is

$$R_h[f](x) \coloneqq \frac{f(x+h) - f(x)}{h} \,. \tag{1.1}$$

We say that f is differentiable in x if the limit

$$\lim_{h\to 0} R_h[f](x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \,, \tag{1.2}$$

exists. If f is differentiable $\forall x \in [a, b]$, we say it is differentiable. The function obtained by this limit is the *derivative* of f, and it is denoted with the Newton

[1] https://github.com/androbomb/PINN_Course_2026

notation $f'(x)$, or with the Leibniz notation, $\frac{\mathrm{d}f}{\mathrm{d}x}$. If a function $f \in C([a,b])$ is differentiable, and its derivate is continuous (a,b), we write it as $f \in C^1((a,b))$[2]. We can define derivates of higher order by repeating the process,

We denote with curly braces the *open* intervals, and with *squared braces* the *closed* intervals.

$$f^{(n)}(x) = \frac{\mathrm{d}^n f(x)}{\mathrm{d}x} \coloneqq \lim_{h\to 0} \frac{f^{(n-1)}(x+h) - f^{(n)}(x)}{h}\,, \tag{1.3}$$

and if a function is continuous with up to n continuous derivates in the interval $[a,b]$, we write $f \in C^n([a,b])$.

The interpretation of $f'(x)$ is that it defines the angular coefficient of the tangent line to f in x.

If, for example, $f \in C^2([a,b])$, we can approximate the behaviour of f in a small interval h nearby any point $x \in [a,b]$ via its Taylor expansion

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + O(h^2), \tag{1.4}$$

where we have introduced the *big-O* notation, where $O(h^2)$ describes the order of error (in powers of h) we commit by approximating f with its second-order Taylor expansion. We can generalise it to higher powers of n; nevertheless, even if a function if $C^\infty([a,b])$, i.e. for every $n \in \mathbb{N}$, $f^{(n)}$ exists and it is continuous, it does not always admits a complete Taylor expansion. Such functions are $f \in C^\omega([a,b])$, and for those we have

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} f^{(n)}(x)\,, \tag{1.5}$$

{eq:1:taylorexp}

which is an *exact* relation, i.e. we can *represent* $f \in C^\omega([a,b])$ with its Taylor expansion. If $0 \in [a,b]$, the Taylor expansion in 0 of $f \in C^\omega([a,b])$ is called the MacLaurin series. Fortunately, the most famous functions (like $\sin, \cos, \exp, \log$) are C^ω in certain subsets of $\mathbb{R}$.

We can move to higher dimensional settings, e.g. $\mathbf{f} : \Omega \subset \mathbb{R}^m \to \mathbb{R}^n$. Let select an orthonormal basis for $\mathbb{R}^n$ as $\{\hat{\mathrm{e}}_i\}_{i=1,\dots n}$, i.e. $\hat{\mathrm{e}}_i \cdot \hat{\mathrm{e}}_j = \delta_{ij}$. We can then write without loss of generality $\mathbf{f}(x) = f_i(x)\,\hat{\mathrm{e}}_i$, where f_i are the function components $f_i : \Omega \subset \mathbb{R}^m \to \mathbb{R}$. If $\Omega \subset \mathbb{R}^m$ is open, we can define the *directional derivative* of $\mathbf{f}$ along the vector $\mathbf{v} \in \mathbb{R}^m$ if the limit

$$\lim_{t\to 0} \frac{\mathbf{f}(\mathbf{x}+t\mathbf{v}) - f(\mathbf{x})}{t} \tag{1.6}$$

exists. In that case, we write it as

$$\nabla_{\mathbf{v}} \mathbf{f}(\mathbf{x}) \coloneqq \lim_{t\to 0} \frac{\mathbf{f}(\mathbf{x}+t\mathbf{v}) - f(\mathbf{x})}{t}\,, \tag{1.7}$$

If $\mathbf{v}$ is one of the basis versor $\{\hat{\mathbf{u}}_j\}_{j=1,\dots,m}$ of $\mathbb{R}^m$, we have the *partial derivative*

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_j} \coloneqq \lim_{t\to 0} \frac{\mathbf{f}(\mathbf{x}+t\hat{\mathbf{u}}_j) - f(\mathbf{x})}{t}\,. \tag{1.8}$$

[2] Actually, to be rigorous, we need also that exists finite the right limit in a and the left limit in b of the different quotient.

The vector of the derivatives along all directions is called *gradient* of $\mathbf{f}$

$$\nabla \mathbf{f}(\mathbf{x}) = \left(\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_1}, \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_m} \right). \tag{1.9}$$

If $\nabla \mathbf{f}$ exists and it is continuous on Ω, we can easily see that

$$\nabla_{\mathbf{v}} \mathbf{f}(\mathbf{x}) = \mathbf{v} \cdot \nabla \mathbf{f}(\mathbf{x}). \tag{1.10}$$

After this fast and dirty recap of real analysis, we can introduce the concept of differential equation. While an equation is a relation between two functions, and comprises the standard operations (sum, multiplications, and composition), a *differential equation* is a relation between functions *and their derivates*. If it is a relation between univariate functions (i.e., functions of a single variable) and their derivatives, it is called an *ordinary differential equation* (ODE). If it is a relation between multivariate functions - sometimes called *fields* - it is a *partial differential equation* (PDE). The order of the highest derivate appearing in the relation is the order of the differential equation.

When dealing with differential problems, we need to furnish a set of additional constraints, called boundary conditions (BC), which describes the behaviour of the solution $u : \overline{\Omega} \subset \mathbb{R}^m \to \mathbb{R}^n$ of the differential problem at the boundary of the space, i.e. $u|_{\partial\Omega}$. If the function is defined also to have a time variable, i.e. $u : [0, T] \times \Omega \to \mathbb{R}^n$, we must furnish an initial condition (IC), describing the behaviour of the solution when $t = 0$, i.e. $u(t = 0, x)$.

With $\overline{\Omega}$ we denote the closure of Ω, i.e. $\overline{\Omega} \coloneqq \Omega \cup \partial\Omega$. We will later on simplify this notation, at the sole scope of enraging our mathematician friends, and to not put an OVERLINE latex command.

In compact form, a PDE can be written as

$$\mathcal{F}[u(x), x; \gamma] = J(x), \qquad x \in \Omega, \tag{1.11}$$
$$\mathcal{B}[u(x), x; \gamma] = g(x), \qquad x \in \partial\Omega, \tag{1.12}$$

where $\mathcal{F}$ is the differential operator describing the evolution of the system, J is the source term, $\mathcal{B}[u(x), x; \gamma]$ is the operator describing the boundary condition, and g is the boundary value.

Usually, we will work with second-order linear PDE systems, which are the most common in nature (yet, higher order PDE systems exists and may be quite relevant, e.g. Allen-Cahan PDE [AC75] or the Korteweg–de Vries PDE [KV95]). Furthermore, (contanst coefficients) second order linear PDE operator can be *classified*: the general form is

$$\mathcal{F}[u] = L^{ij} \partial_i \partial_j u, \tag{1.13}$$

where L_{ij} is the $m \times m$ constant-coefficient symmetric matrix

It is symmetric by the Schwarz theorem, by which $\partial_i \partial_j u = \partial_j \partial_i u$.

We can classify the second-order linear operators by the eigenvalues of L; we say that L is

1. *elliptic*, if all the eigenvalues have the same sign (and thus there are no zero eigenvalues);
2. *parabolic*, if one eigenvalue is zero and all the others have the same sign ;
3. *hyperbolic*, if one eigenvalue has a sign, and all the others have opposite sign;
4. *ultra-hyperbolic*, in all other cases (i.e., we have a multiplicity of eigenvalues with different signs).

A notable *elliptic* operator is the *Laplacian*, $\Delta \coloneqq \nabla^2$; a notable *parabolic* operator is the *diffusion operator* $\partial_t - D^2\Delta$; a notable *hyperbolic* operator is the *wave* operator $\Box \coloneqq \partial_t^2 - c^2\Delta$.

Generally, the boundary conditions are

- **Dirichlet**: $u(x) = g(x), x \in \partial\Omega$, i.e. the value of the function is *fixed* at the boundary.
- **Neumann**: $\nabla_{\hat{n}} u = h(x), x \in \partial\Omega$, i.e. the value of the part of the gradient of the function normal to the boundary is fixed ($\hat{n}$ is the normal to $\partial\Omega$). This fixes the *flux* through the boundary.
- **Robin**: $u(x) + \alpha\nabla_{\hat{n}} u = r(x), x \in \partial\Omega$, i.e. a linear combination of the Dirichlet and Neumann conditions.

1.2.1 A first example: the Heat equation in 1+1 dimensions

1.2.1.1 A heuristic derivation of 1+1D heat equation as thermal diffusion

subsubsec:1:2:1:1:DiffEqDer

There are many way of deriving the diffusion equation; one example is starting from the continuity equation

$$\partial_t u + \nabla \cdot \mathbf{j} = 0\,,$$

and assume that the flow $\mathbf{j}$ follows the Fick's Law, i.e., that it is somehow proportional to the spatial rate change of u,

$$\mathbf{j}(t,x) = -D^2 \nabla u(t,x)\,.$$

Another approach is to describe a random walk. In 1+1 (time+space) dimension (without loss of generality, WLOG), suppose we want to model a random walk of a particle that moves at each time step τ of h in a random direction, i.e. $x \to x \pm h$. Thus, the probability of finding the particle at time $t+\tau$ in x is

$$p(t+\tau, x) = \frac{1}{2}[p(t, x+h) + p(t, x-h)].$$

Now, by Taylor expanding both side we see that

$$\begin{aligned} p(t+\tau,x) &\approx p(t,x) + \tau\partial_t p(t,x) + O(\tau^2)\,,\\ p(t,x\pm h) &\approx p(t,x) \pm h\partial_x p(t,x) + \frac{h^2}{2}\partial_x^2 p(t,x) + O(h^3)\,, \end{aligned}$$

so that we get

$$\begin{aligned} p(t,x) + \tau\partial_t p(t,x) = \frac{1}{2}\Bigg[&\left(p(t,x) + h\partial_x p(t,x) + \frac{h^2}{2}\partial_x^2 p(t,x)\right)\\ &+ \left(p(t,x) - h\partial_x p(t,x) + \frac{h^2}{2}\partial_x^2 p(t,x)\right)\Bigg]\\ \Rightarrow \partial_t p(t,x) = \frac{h^2}{2\tau}\,&\partial_x^2 p(t,x)\,. \end{aligned}$$

Finally, a last way to obtain the heat equation, is to think about heat balancing, still in 1+1D. Suppose that we want to describe the diffusion in time of heat; we can model it as saying that the temperature u at time $t+\tau$ in x is

$$u(t+\tau, x) = u(t+x) + Q(t, x),$$

where Q is the heat flux. If we see the We can suppose that such heat flux is proportional to the difference in temperature between the cell of size h in x, and the two nearby ones, the one at $x+h$ and the one at $x-h$, i.e.

$$Q(t, x) \propto [u(t, x+h) - u(t, x)] + [u(t, x-h) - u(t, x)]\,,$$

Notice that we can generalise that in higher dimensions and in the continuous limit, by integrating the (signed) difference $u(t, \mathbf{y}) - u(t, \mathbf{x})$ over a ball $B_d \subseteq \mathbb{R}^d$ of radius h centred around x.

so that the flux is *positive* if $u(t, x \pm h) > u(t, x)$, i.e. the cell at $x \pm h$ is at higher temperature (heat flows from higher temperature to lower ones).

Now, again, by Taylor expanding, and calling α the proportionality constant, we get

$$\Rightarrow \partial_t u(t, x) = \frac{\alpha h^2}{2\tau}\, \partial_x^2 u(t, x)\,. \tag{1.14}$$

1.2.1.2 Heated bar thermalisation via diffusion

Let us suppose we want to describe the thermalisation of a metallic bar of length $\ell = 1$, We thus describe our system with the coordinates $(t, x) \in [0, T] \times [-1, +1]$. Let now suppose that we have heated the centre of the bar (with a cosinusoidal shape, i.e. $u(t=0, x) = \cos \frac{\pi x}{2}$, and suppose we put the boundary of the bar on a thermal bath, keeping it at $u(t, x = \pm 1) = 0$. The system is described by the Cauchy problem

$$\partial_t u(t, x) = D^2 \partial_x^2 u(t, x)\,, \quad (t, x) \in [0, T] \times [-1, +1] \tag{1.15}$$

$$u(t=0, x) = \cos\left(\frac{\pi x}{2}\right), \tag{1.16}$$

$$u(t, x) = 0, \quad x \in \partial[-1, 1] = \{-1, +1\}. \tag{1.17}$$

This system can be easily solved by *the separation of variables*: let us suppose that we can rewrite the solution as $u(t, x) = T(t)X(x)$, where T, X are two univariate functions we need to find. In this case, the PDE becomes

$$\dot{T}(t)X(x) = D^2 T(t) X''(x) \Rightarrow \frac{\dot{T}(t)}{T(t)} = D^2 \frac{X''(x)}{X(x)}\,, \tag{1.18}$$

which is possible only if

$$\frac{\dot{T}(t)}{T(t)} = -\lambda^2 = D^2 \frac{X''(x)}{X(x)}\,, \quad \lambda \in \mathbb{R}\,. \tag{1.19}$$

This gives two simple ODEs, whose general solutions are

$$T(t) = T_0 \exp\left(-\lambda^2 (t - t_0)\right), \tag{1.20}$$

$$X(x) = X_0 \cos\left(\frac{\lambda}{D}(x - x_0)\right) + \tilde{X}_0 \sin\left(\frac{\lambda}{D}(x - x_0)\right), \tag{1.21}$$

where T_0, X_0, x_0 are constants to be determined.

Now, imposing the boundary condition, we see that we can fix $t_0 = 0 = x_0$, $T_0 = X_0 = 1, \tilde{X}_0 = 0$. Finally, to satisfy the initial condition, we need that $2\lambda = \pi D$. Thus the solution is

$$u(t,x) = \mathrm{e}^{-\frac{\pi D}{2}t} \cos\left(\frac{\pi x}{2}\right). \tag{1.22}$$

Let us now interpret the role of the *Dirichlet boundary conditions*: we have imposed that the function has to have constant value at the boundary (being in contact with the thermal bath); thus, at the boundary, we have a non-zero *outward flux*

$$\begin{aligned}\Phi|_{\partial[0,1]} &= \sum_{i=L,R} \hat{n}_i \cdot \nabla u = \sum_{\pm 1} \mp \partial_x u \\ &= \sum_{\pm 1} \pm \frac{\pi}{2} T(t) \sin\left(\frac{\pi x}{2}\right)\Big|_{x=\pm 1} \\ &= \pi \mathrm{e}^{-\frac{\pi D}{2}t},\end{aligned} \tag{1.23}$$

which signals the outward flux, i.e. the loss of heat through the boundary. Indeed, the thermal bath we have inserted our bar in is *thermalising* our object.

1.2.2 The effect of Robin conditions: the loose end violin string

subsec:1:2:1: Loose_String

We want to describe the effect of a pluck on a violin string, which is fixed at the boundary [GA98]; the system is

$$\begin{aligned}&\Box u(t,x) = 0\,, \qquad (t,x) \in [0,T] \times [0,1]\,, \\ &\begin{cases} u(t,x=0) = 0\,, \\ u(t,x=1) = 0\,, \end{cases} \\ &\begin{cases} u(t=0,x) = f(x)\,, \quad \text{(pluck)} \\ \partial_t u(t=0,x) = 0\,, \end{cases}\end{aligned} \tag{1.24}$$

where $\Box := \partial_t^2 - c^2 \partial_x^2$.

Let us now suppose that we are cheap, and we have a cheap violin of poor quality. Its peg[3] does not perfectly holds the string, which can thus move *orthogonally*. This effects is modelled by a possible flux of u from a side; the problem this requires a *Robin* boundary condition

With $\alpha = 0$ we recover the pure Dirichlet Cauchy problem.

$$\begin{aligned}&\Box u(t,x) = 0\,, \qquad (t,x) \in [0,T] \times [0,1]\,, \\ &\begin{cases} u(t,x=0) = 0\,, \\ u(t,x=1) + \alpha \partial_x u(t,x=1) = 0\,, \end{cases} \\ &\begin{cases} u(t=0,x) = f(x)\,, \quad \text{(pluck)} \\ \partial_t u(t=0,x) = 0\,. \end{cases}\end{aligned} \tag{1.25}$$

Both these systems can be solved analytically, with small differences. Using again the separation of variables, we see that we can write

I.e., we set

$$u(t,x) = \sum_n X_n(x) T_n(t)$$

$$\Rightarrow \begin{cases} X'' + \lambda_n X = 0, \\ \ddot{T} + c^2 \lambda_n T = 0 \end{cases} \tag{1.26}$$

Indeed, we have:

$$\begin{aligned}\partial_t^2 u &= -c^2 u\,, \\ \partial_x^2 u &= -u\,.\end{aligned} \tag{1.28}$$

[3] In Italian, *il bischero*.

$$u(t,x) = \sum_{n=0}^{\infty} c_n \sin\left(\sqrt{\lambda_n}\, x\right) \cos\left(\sqrt{\lambda_n}\, ct\right) , \tag{1.30}$$

where λ_n is a sequence of real numbers labelled by a natural number n. We can now show that we can express the coefficients c_n in terms of the pluck $f(x)$ as

$$c_n = \frac{\int_0^1 f(x) \sin\left(\sqrt{\lambda_n}\, x\right) \mathrm{d}x}{\int_0^1 \sin^2\left(\sqrt{\lambda_n}\, x\right) \mathrm{d}x}. \tag{1.31}$$

Indeed, from the first initial condition,

$$\begin{aligned} f(x) = u(t=0,x) &= \sum_{n=0}^{\infty} c_n \sin\left(\sqrt{\lambda_n}\, x\right) \\ \Rightarrow c_n &= \frac{\int_0^1 f(x) \sin\left(\sqrt{\lambda_n}\, x\right) \mathrm{d}x}{\int_0^1 \sin^2\left(\sqrt{\lambda_n}\, x\right) \mathrm{d}x} , \quad \forall n = 1, \ldots, \infty. \end{aligned} \tag{1.32}$$

where we have used the *orthogonality* property of the sinusoidal basis, i.e.

$$\int_0^1 \sin\left(\sqrt{\lambda_n}\, x\right) \sin\left(\sqrt{\lambda_m}\, x\right) \mathrm{d}x = \delta_{mn} \int_0^1 \sin^2\left(\sqrt{\lambda_n}\, x\right) \mathrm{d}x. \tag{1.33}$$

This relation holds for these λ_n because of the *orthogonality of the eigenfucntions of the wave operator* respecting the Dirichlet/Robin BC.

Indeed, let us pick 2 random eigensolutions, X_a, X_b; now, let's multiply the equation satisfied by one for the other, and subtract them:

I.e., $X''_{a,b} + \lambda_{a,b} X_{a,b} = 0$.

$$\begin{cases} X_b(X''_a + \lambda_a X_a) = 0 \\ X_a(X''_b + \lambda_b X_b) = 0 \end{cases} \implies (X''_a X_b - X_a X''_b) + (\lambda_a - \lambda_b) X_a X_b = 0. \tag{1.34}$$

{eq:1:2:2: Robin1}

The first term can be written as a total derivative

$$(X''_a X_b - X_a X''_b) = \partial_x (X'_a X_b - X_a X'_b) \tag{1.35}$$

so that, if we integrate Equation (1.34) in the whole domain the relation above, we get

$$[X'_a X_b - X_a X'_b]_0^1 + (\lambda_a - \lambda_b) \int_0^1 X_a(x) X_b(x) \mathrm{d}x = 0. \tag{1.36}$$

But, by the Dirichlet BC, $X_{a,b}(0) = 0$, while, by the Robin BC, $\alpha X_a(1) = -X'_a(1)$, and so, the left term cancels out! This means that, if $\lambda_a \neq \lambda_b$ the integral must be zero, and this

$$\int_0^1 X_a(x) X_b(x) \mathrm{d}x = \delta_{a,b} \int_0^1 X_a(x)^2 \mathrm{d}x. \tag{1.37}$$

While the above results is general, we can explicitly write λ_n. For the *good quality violin*, i.e. pure Dirichlet boundary conditions, we need to impose that

$$\text{(pure Dirichlet)}: \quad u(t, x=1) = 0 \Rightarrow \sin\left(\sqrt{\lambda_n}\right) = 0 \Rightarrow \lambda_n = n^2 \pi^2 \, \forall n \in \mathbb{N} . \tag{1.38}$$

	C	C#	D	Eb	E	F	F#	G	G#	A	Bb	B
0	16.35	17.32	18.35	19.45	20.60	21.83	23.12	24.50	25.96	27.50	29.14	30.87
1	32.70	34.65	36.71	38.89	41.20	43.65	46.25	49.00	51.91	55.00	58.27	61.74
2	65.41	69.30	73.42	77.78	82.41	87.31	92.50	98.00	103.8	110.0	116.5	123.5
3	130.8	138.6	146.8	155.6	164.8	174.6	185.0	196.0	207.7	220.0	233.1	246.9
4	261.6	277.2	293.7	311.1	329.6	349.2	370.0	392.0	415.3	440.0	466.2	493.9
5	523.3	554.4	587.3	622.3	659.3	698.5	740.0	784.0	830.6	880.0	932.3	987.8
6	1047	1109	1175	1245	1319	1397	1480	1568	1661	1760	1865	1976
7	2093	2217	2349	2489	2637	2794	2960	3136	3322	3520	3729	3951
8	4186	4435	4699	4978	5274	5588	5920	6272	6645	7040	7459	7902

Figure 1.1: The tables of standard musical note frequencies. We move from column to column by changing ℓ of a certain amount. Instead, we move from row to row by increasing n. From [MSR18].

fig:lect1: frequencies-of-notes

Reintroducing the *length* of the string ℓ (via the shift $x \mapsto x/\ell$), we have

$$(\text{pure Dirichlet}) : \lambda_n = \frac{n^2\pi^2}{\ell^2} \,. \tag{1.39}$$

This gives the standard behaviour of *tempered* notes while playing string instruments, like violins or guitars: moving the finger on the fretboard increases the frequency (since we reduce ℓ), and a well given pluck creates all the harmonics of a certain *note*. See Figure 1.1.

If we instead have the cheap violin (or we are describing us playing, and we are *bad* musical players), we need to solve

$$\begin{aligned} (\text{Robin}) : \qquad & u(t, x=1) + \alpha \partial_x u(t, x=1) = 0 \\ & \Rightarrow \sin\left(\sqrt{\lambda_n}\right) + \alpha\sqrt{\lambda_n}\cos\left(\sqrt{\lambda_n}\right) = 0 \\ & \Rightarrow \tan\sqrt{\lambda_n} = \alpha\sqrt{\lambda_n} \,. \end{aligned} \tag{1.40}$$

$\forall n \in \mathbb{N}$, such that $\sqrt{\lambda_n} \neq \frac{\pi}{2} + k\pi, \forall z \in \mathbb{Z}, \forall n \in \mathbb{N}$, which is a *transcendental* equation. It cannot be solved in closed form analytically, but we can obtain a *numerical* approximation of it, via numerical methods (see Listing 1.1).

lst:1:2: transcendental

Listing 1.1: Python function to solve the transcendental equation.

```python
import numpy as np
from scipy.optimize import brentq

def solve_transcendental(a, N):
    """
    Solve   tan(sqrt(\lambda)) = a sqrt(\lambda) for the first N
        positive solutions.
    Returns array of x solutions.
    """
    roots = []
    n = 0
    while len(roots) < N:
        # Interval between asymptotes of tan(y) ; y = \sqrt{\lambda}
        left  = n * np.pi
        right = n * np.pi + np.pi/2 - 1e-6  # avoid asymptote
        # Define function f(y) = a*tan(y) - y
        f = lambda y:  np.tan(y) - a * y
```

```
        # Check if sign change exists in interval
        try:
            if np.sign(f(left + 1e-6)) != np.sign(f(right)):
                y_root = brentq(f, left + 1e-6, right)
                roots.append((y_root)**2)  # convert back to x
        except ValueError:
            pass
        n += 1
    return np.array(roots)
```

If we numerically solve it for various α, we see that the effect of the loose end (i.e., $\alpha > 0$) is more relevant for *lower* notes, as reported in Figure 1.2.

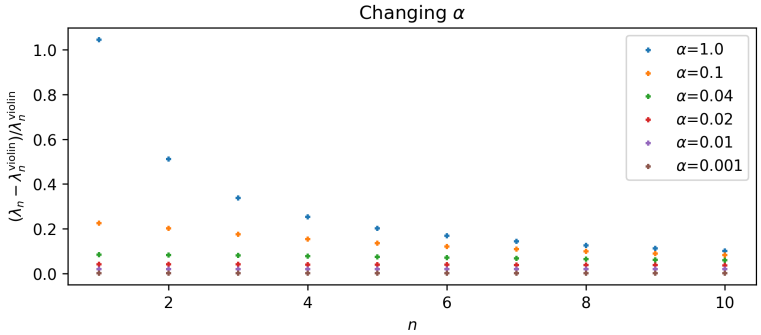


Figure 1.2: Numerical λ_n for various α.

fig:1:2: violin_robin_1

1.3 Recap of Functional Analysis

sec:1:3: func_analys

To properly understand the approximation power of neural networks and the weak formulation of PDEs, we need a few notions from functional analysis.

For a brief and formal introduction to the subject, see [Sal22], chapter 7, or [Qua20], chapter 2, or the lecture notes [Rod21].

Definition 1.3.1 (Vector space)**.** A **vector space** V over a field $\mathbb{K}$ (e.g., $\mathbb{R}$ or $\mathbb{C}$) is a set $V = \{v_i\}_{i=1,2,\dots}$ with two operation defined: an addition $+ : V \times V \to V$ and a scalar multiplication $\cdot : \mathbb{K} \times V \to V$, satisfying certain axioms: commutativity, associativity, null element, and inverse of addition, identity of multiplication, and distributivity.

Definition 1.3.2 (Finite-dimensional vector space)**.** A vector space V is **finite-dimensional** if every linearly independent set is a finite set. In other words, if $a = (a_1, \dots a_N) \in V$ for all sets $E \subseteq V$ such that

$$\sum_{i=1}^{N} a_i v_i = 0 \implies a_1 = a_2 = \cdots = a_N = 0 \quad \forall v_1, \dots, v_N \in E,$$

then E has a finite cardinality. V is **infinite-dimensional** if it is not finite-dimensional.

Definition 1.3.3 (Norm)**.** A **norm** on a vector space V is a function $\|\cdot\| : V \to [0, \infty)$ satisfying the following three properties:

1.3.3.1. (Definiteness) $\|v\| = 0$ if and only if $v = 0$,

1.3.3.2. (Homogeneity) $\|\lambda v\| = |\lambda|\|v\|$ for all $v \in V$ and $\lambda \in \mathbb{K}$,

1.3.3.3. (Triangle inequality) $\|v_1 + v_2\| \leq \|v_1\| + \|v_2\|$ for all $v_1, v_2 \in V$.

A **seminorm** is a function $\|\cdot\| : V \to [0, \infty)$ which satisfies (2) and (3) but not necessarily (1), and a vector space equipped with a norm is called a **normed space**.

Proposition 1.3.4 (Metric induced by the norm)**.** *Let $(V, \|\cdot\|)$ be a normed vector space. Then*

$$d(v, w) = \|v - w\|$$

defines a metric on V, which we call the metric induced by the norm.

Definition 1.3.5. Let $\{x_n\} \subset V$ be a sequence in a metric space (V, d). Then:

$\{x_n\}$ is **fundamental** if $d(x_n, x_m) \to 0$ as $n, m \to \infty$.

$\{x_n\}$ **converges** to $x \in V$ if $d(x_n, x) \to 0$ as $n \to \infty$.

Proposition 1.3.6. *If $\{x_n\}$ converges in V, then it is fundamental.*

Observation 1.3.7. *The converse is not always true. For example:*

$$x_n = \left(1 + \frac{1}{n}\right)^n \to e \notin \mathbb{Q}, \quad x_n \in \mathbb{Q}, \tag{1.41}$$

i.e., $x_n \in \mathbb{Q}$ for all finite n, but the limit $\lim_{n\to\infty} x_n = e \notin \mathbb{Q}$.

If the converse holds for all fundamental sequences in V, then V is called **complete**.

The Euclidean norm on $\mathbb{R}^n$ or $\mathbb{C}^n$, given by

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2\right)^{1/2},$$

is indeed a norm (the standard notion of "distance" that we're used to). But we can also define

$$\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|,$$

which measures the largest magnitude among the components, and more generally (for $1 \leq p < \infty$),

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p},$$

which is the $p-$norm.

Definition 1.3.8 (Banach space)**.** A normed space is a **Banach** space if it is complete with respect to the metric induced by the norm.

Why Banach space are relevant? because of the following

Theorem 1.3.9. *For any metric space X, the space of bounded, continuous function on X, $C(X;\mathbb{R})$, is complete, and thus a Banach space with respect to the uniform norm (or sup norm)*

$$\|F\|_\infty \coloneqq \sup_{x\in X} |F(x)| .$$

Usually, we works with more stringent spaces, where we have a concept of *projectability*, or *decomposition*, via a bilinear form (the scalar product).

Definition 1.3.10 (bilinear forms)**.** Let X be a Banach space. A **form** is an application

$$a : X \times X \to \mathbb{R}$$

that assigns a real number to each pair of elements in X. A form is called:

1.3.10.1. **Bilinear** if it is linear in both arguments, i.e.,

$$a(\lambda u + \mu w, v) = \lambda a(u, v) + \mu a(w, v), \quad a(u, \lambda w + \mu v) = \lambda a(u, v) + \mu a(u, w)$$

for all $\lambda, \mu \in \mathbb{R}$ and $u, v, w \in X$;

1.3.10.2. **Continuous** if there exists a constant $M > 0$ such that

$$|a(u, v)| \le M\|u\|_X\|v\|_X \quad \forall u, v \in X;$$

As a consequence, $a(0, u) = a(u, 0) = 0\ \forall u \in X$.

1.3.10.3. **Symmetric** if

$$a(u, v) = a(v, u) \quad \forall u, v \in X.$$

1.3.10.4. **Positive** if

$$a(v, v) > 0, \forall v \in V \setminus \{0\}.$$

1.3.10.5. **Coercive** if exists $\alpha \in \mathbb{R}^+$ (i.e. $\alpha > 0$) such that

$$a(v, v) \ge \alpha\|v\|_V^2, \quad \forall v \in V.$$

Definition 1.3.11. Let X be a vector space over $\mathbb{R}$. X is called a **pre-Hilbertian space** if there exists a bilinear form, called inner product $\langle\cdot,\cdot\rangle : X \times X \to \mathbb{R}$, satisfying:

1.3.11.1. **Positive definiteness:** $\langle x, x\rangle \ge 0$, and $\langle x, x\rangle = 0 \Leftrightarrow x = 0$,

1.3.11.2. **Symmetry:** $\langle x, y\rangle = \langle y, x\rangle$ for all $x, y \in X$,

1.3.11.3. **Bilinearity:** $\langle z, ax + by\rangle = a\langle z, x\rangle + b\langle z, y\rangle$ for all $x, y, z \in X$ and $a, b \in \mathbb{R}$.

Definition 1.3.12 (Hilbert space)**.** If X is complete with respect to the metric induced by the norm $\|x\| = \sqrt{\langle x, x\rangle}$, then X is called a **Hilbert space**.

Why are Banach and (pre-)Hilbert spaces relevant for us? because we can study functions - and thus, solutions of our differential problems - in these spaces. Let $\Omega \subset \mathbb{R}$ be open, and let $p \geq 1$. Then $L^p(\Omega)$ is the set of functions f such that $|f|^p$ is Lebesgue integrable, equipped with the norm

$$\|f\|_{L^p(\Omega)} \equiv \left(\int_\Omega |f|^p \right)^{1/p}.$$

This space is a Banach space (up to equivalence classes). When $p = 2$, (i.e., $L^2(\Omega)$) and we define the scalar product $\langle \cdot, \cdot \rangle : L^2(\Omega) \times L^2(\Omega) \to \mathbb{R}$ by

$$\langle f, g \rangle = \int_\Omega fg,$$

then $L^2(\Omega)$ becomes a Hilbert space.

Furthermore, Hilbert spaces are the ideal setting to solve problems in infinitely many dimensions. They unify through the inner product and the induced norm, both an analytical and a geometric structure. We can use *geometrical* structures of the space to separate the Hilbert space in two *orthogonal* subsets (orthogonal w.r.t. the bilinear form), i.e. $H = V \oplus V^\perp$. This is possible thanks to the following

Theorem 1.3.13 (Projection Theorem)**.** *Let V be a closed subspace of a Hilbert space H. Then, for every $x \in H$, there exists a unique element $P_V x \in V$ such that*

$$\|P_V x - x\| = \inf_{v \in V} \|v - x\|. \tag{6.17}$$

Moreover, the following properties hold:

1.3.13.1. *$P_V x = x$ if and only if $x \in V$,*

1.3.13.2. *Let $Q_V x = x - P_V x$. Then $Q_V x \in V^\perp := H \setminus V$ and*

$$\|x\|^2 = \|P_V x\|^2 + \|Q_V x\|^2.$$

This lays down the basis for a more stringent, useful result, for certain Hilbert spaces (which, usually, are the one we have to work with):

Definition 1.3.14 (Separable Hilbert Space)**.** A Hilbert space H is said to be **separable** if there exists a countable dense subset of H.

Those are relevant because they admits the concept of *basis*:

Definition 1.3.15. An orthonormal basis in a separable Hilbert space H is a countable set $\{\mathbf{v}_i\}_{i \in \mathbb{N}} \subset H$ which is dense in H and

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle = \delta_{ij},$$

where $\langle \cdot, \cdot \rangle$ is the scalar product on H.

With this concept at hand, we can represent any element of a separable Hilbert space (e.g., all functions in $L^2(\Omega)$) with a series over $\mathbb{N}$, the *generalised Fourier series*.

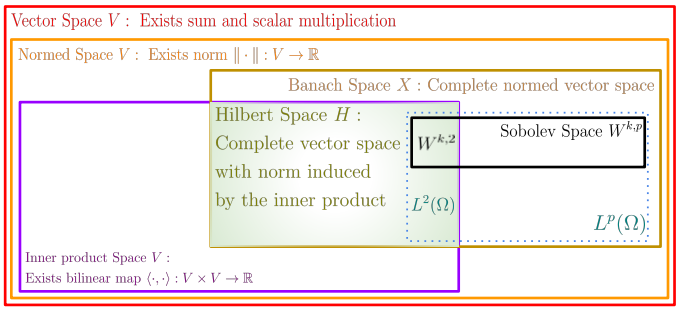


fig:1:3:spaces2

Figure 1.3: Pictorial representation of the spaces we have introduced.

def:1:3: genfourierseries

Definition 1.3.16 (Generalised Fourier Series)**.** If H separable Hilbert space, any $\mathbf{x} \in H$ can be written as

$$\mathbf{x} = \sum_{i=1}^{\infty} \langle \mathbf{v}_i, \mathbf{x} \rangle \mathbf{v}_i.$$

This is already our **second** way to *represent* a function we have seen so far; after the Taylor expansion, we have the (generalised) Fourier series. We will see a more relevant representation in a moment.

Before going on, we want to describe a more general function space, introducing also the derivatives of the elements of L^p-spaces.

Definition 1.3.17 (Weak derivative)**.** Let $\Omega \subset \mathbb{R}^d$ open bounded, and let $f \in L^1(\Omega)$. We say that f admits a *weak derivative* $\tilde{\partial}_{x_j} f$ if there exists a function $g \in L^1(\Omega)$ such that

$$\int_\Omega f \partial_{x_j} \varphi = -\int_\Omega g\varphi\,, \quad \forall \varphi \in C^1(\Omega),\, \varphi|_{\partial\Omega} = 0\,.$$

We define

$$\tilde{\partial}_{x_j} f := g.$$

Definition 1.3.18 (Sobolev space)**.** Let $\Omega \subset \mathbb{R}^d$ and let $f \in L^p(\Omega)$. If $k \in \mathbb{N}$, we say that $f \in W^{k,p}(\Omega)$ if $\exists D^i f \in L^p(\Omega)$, $\forall i = 1, \dots, k$, such that

$$\|f\|_{k,p} = \sum_{i=0}^{k} \|D^i f\|_p,$$

where $D^i f$ denotes the multi-index (weak) derivative of order i, and $\|\cdot\|_p$ is the $L^p(\Omega)$ norm. The space $W^{k,p}(\Omega)$ equipped with this norm is called Sobolev space, which is a Banach space.

We have seen that $L^2(\Omega)$ is an Hilbert space. Similarly, in particular, $W^{k,2}(\Omega)$ is a Hilbert space.

In Figure 1.3, we see the relation between the spaces we have defined.

1.3.1 From functions to functionals

We know that *functions* maps (vector) spaces to (vector) spaces. Yet, we are discussing (vector) spaces of *functions*. So, how do we generalise these concept, i.e. a map from function spaces to function spaces?

Definition 1.3.19 (operators & functionals)**.** Let H_1, H_2 be Hilbert spaces. An **operator** $L \in \mathcal{L}(H_1, H_2)$ is a map

$$L : H_1 \to H_2,$$

which is *bounded*, i.e. $\|Lx\|_{H_2} \leq c\|x\|_{H_1}, \forall x \in H_1$. If $L(h_1 + ah_2) = L(h_1) + aL(h_2)$, then L is called a *linear operator*. If $H_2 = \mathbb{R}$, i.e., $L \in \mathcal{L}(H_1, \mathbb{R})$, then L is called a (bounded) *functional*.

Proposition 1.3.20. *The space $\mathcal{L}(H_1, H_2)$, endowed with the supremum norm, i.e.*

$$\|L\|_{\mathcal{L}(H_1,H_2)} = \sup_{x \in H_1} \|Lx\|_{H_2}$$

is a Banach space.

Definition 1.3.21 (Dual space)**.** Let H be a Hilbert space. The collection of all bounded linear functionals on H is called the **dual space** of H, and is denoted by H^* (instead of $\mathcal{L}(H, \mathbb{R})$).

Remark 1.3.22. $(L^2(\Omega))^* = L^2(\Omega)$.

To recap, we have that a **function** maps a space X to a space Y, sending points in X to points in Y. An **operator** maps a function space H_1 to another function space H_2, sending functions to functions. A **functional** maps a function space to a field (e.g., $\mathbb{R}$), associating scalar values to functions.

Like matrices represents the realisation of the action of a linear transformation on a vector space on a certain vector fields, linear operators are their generalisation on function spaces. Examples of linear operators on (certain) function spaces are derivatives, and thus differential operators.

Example 1.3.23. Let $u, v \in C^\infty(\Omega)$, and $a \in \mathbb{R}$. Then

$$\Delta(u + av) = \Delta u + a\Delta v.$$

Example 1.3.24. Let $u, v \in C^\infty(\mathbb{R} \times (\Omega)$, $a \in \mathbb{R}$, and define $L = \partial_t - D^2\Delta$. Then

$$L(u + av) = L(u) + aL(v).$$

Definition 1.3.25. Let $L \in \mathcal{L}(H_1, H_2)$ be a linear operator between Hilbert spaces. The set of points in H_1 mapped to the zero element of H_2 is called the **kernel** (or *nucleus*) of L,

$$\ker(L) = \{h \in H_1 : L(h) = 0_{H_2}\}.$$

Observation 1.3.26. *Let $u \in H(X)$ be a solution of the PDE defined by an operator L, i.e., $L(u) = f$. Then, for all $v \in \ker(L)$, the function $u + v$ is also a solution.*

A relevant property of scalar products and Hilbert spaces is that exists a relevant representation theorem for bounded functionals (e.g., some bounded differential operators):

theorem:1:3:1: Riesz

Theorem 1.3.27 (Riesz Representation Theorem)**.** *Let H be a Hilbert space. For each linear and bounded functional f on H, there exists a unique element $h_f \in H$ such that*

$$f(y) = \langle y, h_f \rangle_H \quad \forall y \in H, \quad \textit{and} \quad \|f\|_{H^*} = \|h_f\|_H.$$

Conversely, each element $h \in H$ defines a linear and bounded functional f_h on H such that

$$f_h(y) = \langle y, h\rangle_H \quad \forall y \in H, \quad \textit{and} \quad \|f_h\|_{H^*} = \|h\|_H.$$

In the case of $L^2(\Omega)$, another formulation of the Riesz representation theorem is that any bounded linear functional $L \in \mathcal{L}(L^2(\Omega), \mathbb{R})$ can be represented as an integral

$$L[y] = \int_X y(x) h_f(x) \mathrm{d}x\,, \tag{1.42}$$

for some $h_f \in L^2(\Omega)$, which can be rewritten, in Lebesgue integral formulation, as a *measure* on the X space:

$$L[y] = \int_X y(x) \mathrm{d}\mu(x)\,. \tag{1.43}$$

{eq:1:3: Riesz4Cyb}

Why all this functional analysis stuff? is it really useful? Well.... YES.

Remark 1.3.28. The main idea behind the use of Hilbert (or, more generally, Banach) spaces is that the PDEs are difficult to solve using derivates in the classical sense (the one introduce before). While dealing with the *weak formulation* of the PDE (which we will formally encounter later) is easier, which is, roughly speaking, the study of the functional associated to the PDE as the Energy is the functional associated to te Euler-Lagrange equations in Physics; and the study of these functionals requires spaces that admits a "representation", in the sense of Theorem 1.3.27, to obtain existence and uniqueness of solutions; or, at least, spaces that are complete wit respect to norms that descend from the functionals that we are considering (which are in general L^p or $W^{k,p}$). On the contrary, $C^m(\Omega)$ spaces are **not** complete w.r.t. those norms, so that we cannot apply functional results, which will also be used in numerical methods.

Let $f \subset C^0(\Omega)$, with $\Omega \subset \mathbb{R}^d$ open bounded, and let $\Delta \equiv \nabla \cdot \nabla$ be the Laplace operator. Suppose we want to solve the PDE

$$\Delta u = f.$$

This is called the **strong formulation** of the PDE. Now, let $v \in H$ be an arbitrary test function in a Hilbert space H to be determined. Then

$$\langle v, \Delta u\rangle = \langle v, f\rangle,$$

where $\langle\cdot,\cdot\rangle$ is the inner product in $L^2(\Omega)$, i.e.,

$$\int_\Omega v(x)\Delta u(x)\,\mathrm{d}x = \int_\Omega v(x) f(x)\,\mathrm{d}x.$$

Applying the Gauss-Green-Ostrogradskiĭ-Stokes Theorem yields

$$\int_{\partial\Omega} v(x)\nabla u(x)\cdot \mathrm{d}\Sigma - \int_\Omega \nabla v(x)\cdot\nabla u(x)\,\mathrm{d}x = \int_\Omega v(x) f(x)\,\mathrm{d}x. \tag{1.44}$$

This is called the **weak formulation** of the PDE, if we impose that this relation must hold for all the *test functions* v. Indeed, we can see that, while the first term represent the boundary terms, we have rewritten our PDE in terms of two bilinear form $a, b : H \times H \to \mathbb{R}$ on the Hilbert space H

$$b(v, u) = a(v, f), \tag{1.45}$$

where $b : u, v \mapsto \int \nabla v \cdot \nabla u \mathrm{d}x$.

1.3.2 The Cybenko's Theorem

subsec:1:3:2: UniversalApproxThm

We have now almost all ingredients to prove one very elegant theorem: the Cybenko's Universal Approximation Theorem [Cyb89]. The last two ingredients we need are the Hahn-Banach separation theorem and the discriminatory measure definition

Theorem 1.3.29 (Hahn-Banach Theorem)**.** *Let V be a normed vector space, and let $M \subset V$ be a subspace. Suppose $u \in M^*$ is a linear map such that*

$$|u(t)| \leq C\|t\| \quad \forall t \in M,$$

i.e., u is a bounded linear functional. Then there exists a continuous extension U, with $U \in V^*$, such that

$$U|_M = u \quad \text{and} \quad |U(t)| \leq C\|t\| \quad \forall t \in V,$$

where the constant C is the same as above.

This mean that we can continuously extend a bounded linear map from a subspace of a functional space to the whole space, keeping its bounded property.

Nevertheless, we need this result in a *slightly* different setting, called the *Hahn-Banac Separation Theorem*, which is actually a corollary of the former theorem

thm:1:3:HBST

Theorem 1.3.30 (Hahn-Banach Separation Theorem)**.** *Let V be a normed vector space and let $R \subset V$ be a closed subspace. Then there exists a bounded linear functional $L \in V^*$, $L \neq 0$, such that*

$$L(r) = 0 \qquad \forall r \in R.$$

Equivalently, the annihilator $\{L \in V^* : L(r) = 0 \ \forall r \in R\}$ contains a non-zero functional.

Proof. Let V be a normed vector space and $R \subset V$ a proper closed subspace.

1. Choose a point outside R.

Since R is proper, pick $f_0 \in V \setminus R$. Then the distance

$$d = \inf_{r \in R} \|f_0 - r\|$$

2. Define a functional on a one-dimensional subspace.

is strictly positive. Choose $r_0 \in R$ such that

$$\|f_0 - r_0\| = d.$$

Define the one-dimensional subspace

$$M = \text{span}\{f_0 - r_0\},$$

and define a linear functional $u : M \to \mathbb{R}$ by

$$u(\lambda(f_0 - r_0)) = \lambda d.$$

3. Extend the functional using Hahn–Banach.

This functional is bounded with $\|u\| = 1$. By the Hahn–Banach extension theorem, u extends to a bounded linear functional $U : V \to \mathbb{R}$ with the same norm, i.e.

$$U|_M = u, \qquad \|U\| = 1.$$

For any $r \in R$,

4. Show that the extension annihilates R.

$$U(r) = U\big((r - r_0) + r_0\big) = U(r - r_0) + U(r_0).$$

But $r - r_0 \in R$ and $r_0 \in R$, and by minimality of r_0 the extension satisfies

$$U(r) = 0 \qquad \forall r \in R.$$

Moreover,

$$U(f_0) = U\big((f_0 - r_0) + r_0\big) = U(f_0 - r_0) = d \neq 0.$$

Thus we have constructed a nonzero bounded linear functional U such that $U(r) = 0$ for all $r \in R$, proving the separation form of the Hahn–Banach theorem. ■

Finally, we introduce the concept of *discriminatory* functions for a certain measure in a measurable space

Definition 1.3.31 (Discriminatory function)**.** Let be X a Banach space of functions over the real interval $I_n = [0,1]^n \subset \mathbb{R}^n$. A function $\sigma \in X$ is said to be *discriminatory* if the following implication holds: Let $\mu \in \mathcal{M}(I_n)$ be a measure. If

$$\int_{I_n} \sigma(y^T x + \theta)\, \mathrm{d}\mu(x) = 0$$

for all $y \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$, then $\mu = 0$. In other words, the vanishing of all such integrals implies that the measure μ must be identically zero.

The universal approximation property of neural networks is a key result that justifies their use as function approximators. We now sketch its proof, which relies on the functional analysis tools we just introduced.

Universal Approximation Theorem

Let be X a Banach space of functions over the real interval $I_n = [0,1]^n \subset \mathbb{R}^n$. Let function $\sigma : \mathbb{R} \to \mathbb{R}$ be any continuous discriminatory function. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j) \tag{1.46}$$

{eq:1:3:ANNCyb}

are dense in $C(I_n)$ (w.r.t. the uniform norm). That is, for any $f \in C(I_n)$ and $\varepsilon > 0$, there exists a sum $G(x)$ of the above form such that

$$|G(x) - f(x)| < \varepsilon \quad \text{for all} \quad x \in I_n.$$

Proof. Let $S \subset C(I_n)$ be the set of functions of the form $G(x)$ as in Equation (1.46). It is easy to see that S is a linear subspace of $C(I_n)$. We will show that the closure of S is all of $C(I_n)$.

Assume that the closure of S is not all of $C(I_n)$. Then the closure of S, called, let us say, R, is a closed proper subspace of $C(I_n)$. By the Hahn–Banach theorem, there is a bounded linear functional on $C(I_n)$, call it L, with the

1. We work by contraddiction.

2. Use Hahn-Banach Separation Theorem Theorem 1.3.30

property that $L \neq 0$ but $L(R) = L(S) = 0$.

By the Riesz Representation Theorem, this bounded linear functional L is of the form

3. Use Equation (1.43).

$$L(h) = \int_{I_n} h(x)\,\mathrm{d}\mu(x)$$

for some $\mu \in \mathcal{M}(I_n)$, for all $h \in C(I_n)$. In particular, since S is contained in its closure R, and $\sigma(w^T x + b) \in S$ for all w, b, it is also in R, and thus we must have

$$\int_{I_n} \sigma(w^T x + b)\,\mathrm{d}\mu(x) = 0$$

for all $w \in \mathbb{R}^n$, $b \in \mathbb{R}$.

4. Impose discriminatory definition.

However, we assumed that σ is discriminatory, so this condition implies $\mu = 0$, contradicting our assumption. Hence, the subspace S must be dense in $C(I_n)$. ■

Remark 1.3.32 (Representations of functions). We have seen already three different way to represent *any* function (under certain conditions). We have encountered *Taylor expansion* (Equation (1.5)), so that, for any $f \in C^\omega(\mathbb{R})$

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}\,(x - x_0)^n = \sum_{n=0}^{\infty} a[f](x_0)\,(x - x_0)^n\,,$$

i.e., we can represent any function with a polynomial; furthermore, Taylor theorem furnishes two additional crucial ingredients:

- a way to define the polynomial coefficients $a[f](x_0)$, in terms of the function f (and its derivatives) and of the point x_0 around which we are "expanding" it;
- a way to compute the *residual* for a truncated expansion:

  $$f(x) \approx \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\,(x - x_0)^n + R_N(f; x_0)\,, \quad R_N(f; x_0) \sim (x - x_0)^{N+1}\,,$$

  which allows us to estimate the numerical representation error efficiently and precisely.

The Taylor representation naturally extends in higher dimensions,

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}\,(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + R_2\,,$$

where $\mathbf{H}$ is the Hessian matrix, with $H_{ij} = \partial_i \partial_j f$.

Unfortunately, the Taylor recipe is numerically expensive, and very limited to a certain region (within the "radius of convergence" of the series expansion). Another relevant expansion we have seen is the *(generalised)-Fourier expansion* (Definition 1.3.16) of any element of an Hilbert space,

$$\mathbf{x} = \sum_{i=1}^{\infty} \langle \mathbf{v}_i, \mathbf{x} \rangle \mathbf{v}_i,$$

which, using the standard Fourier basis for an Hilbert space like $L^2([-\pi,+\pi];\mathbb{R})$, for $f \in L^2([-\pi,+\pi]];\mathbb{R})$ can be conveniently written as the more familiar formula:

$$f(x) = \sum_{n=-\infty}^{+\infty} c_n \mathrm{e}^{+\mathrm{i}nx}\,, \qquad c_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)\,\mathrm{e}^{-\mathrm{i}nx}\mathrm{d}x\,,$$

where the basis vector are $\{v_n\} = \{\mathrm{e}^{\mathrm{i}nx}, n \in \mathbb{Z}\}$ and the product is

$$\langle g, f\rangle = \frac{1}{2\pi}\int_{-\pi}^{+\pi} \overline{g(x)} f(x)\mathrm{d}x.$$

Also here we have the same two properties as above, with the *finite approximation* being [Eps05]

$$f(x) \approx \sum_{n=-N}^{+N} c_n \mathrm{e}^{+\mathrm{i}nx} + R_N\,.$$

The crucial point for Fourier representation, is that we have a relation between *modes* (i.e., the n) and *scale* of the *features* in the function: lower $|n|$, long scale; higher $|n|$, short scale.

Nevertheless, we have a dependence on the number of basis elements used to the error rate, as expressed in the form of R_N. This is similar, yet even more convoluted, for the *Universal Approximation Theorem* (Equation (1.46)); there, the N we are referring to, are the *Number of Neurons*, and the limit $N \to \infty$ is the *infinite width limit* of the neural network. What Universal Approximation Theorem has, as an empirical property, is that we can *stack* multiple finite representation formulas (the *layers* of the Multi-Layer Perceptron) and approximate the unknown function *better* that widening the single-layer representation, in a more computationally efficient and effective manner.

1.4 Recap of Montecarlo Integration methods

sec:1:4:MCI

For an extended introduction to the subject we refer to [Lin20; LPB15; SA24], and references therein.

Now, we know that it is easy to sample from a uniform distribution in a (pseudo)random manner; e.g., we can perform modulus operations on CPU time (in milliseconds since 01/01/1970) when running the script. But what if we want to generate more general *probability density functions* pdf(x)? Well, we can use the *Monte Carlo Sampling Method* (one of its incarnations, at least). Indeed, we can draw $y \sim \mathcal{U}[0,1]$, the uniform distribution on the interval $[0,1] \subset \mathbb{R}$. Its pdf is simply the constant function of value 1 in the interval. The intuition here is that we can see the $[0,1]$ as the image of the cumulative distribution function (cdf) of *any* pdf $f(x)$. Since the cdf is a right-continuous, monotone increasing, bounded function $F : \mathbb{R} \to [0,1]$ for any pdf f, we can see that any $x \sim f$ is mapped to $y = F(x)$, or, equivalently, that, drawing $y \in [0,1]$ can be mapped on $x = F^{-1}(y)$. But how such x are distributed if $y \sim \mathcal{U}[0,1]$. Well...

Equivalently, we can define the cdf as the linear, bounded functional over the Banach space $X := L^1(\mathbb{R})$, $\Phi \in X^* = \mathcal{L}(X;\mathbb{R})$:

$$\Phi : X \to \mathbb{R},\ \text{s.t.}$$
$$\Phi : f \mapsto \int_{-\infty}^{x} f(s)\mathrm{d}s. \tag{1.48}$$

as $f(x)$! Indeed, by a simple change of coordinates

$$\begin{aligned} 1 &= \int_0^1 \mathrm{d}y = \int_{F^{-1}(0)}^{F^{-1}(1)} \frac{\mathrm{d}F(x)}{\mathrm{d}x}\,\mathrm{d}x \quad (\text{where } y = F(x)) \\ &= \int_{-\infty}^{+\infty} f(x)\,\mathrm{d}x \qquad\qquad\qquad\qquad (F'(x) = f(x)). \end{aligned} \tag{1.49}$$

So the algorithm is:

0. Chose the $f(x)$ to sample, (numerically) compute $F(x)$, and then its inverse, $F^{-1}(y)$.
1. Draw $y \sim \mathcal{U}[0,1]$
2. compute $x = F^{-1}(y)$, so that $x \sim f(x)$.

lst:1:3: MCSampling

Listing 1.2: Python function to draw a normal distribution from the uniform distribution via Monte Carlo Sampling.

```
1 import numpy as np
2
3 def inv_cumulative_func(y):
4     return np.sqrt(2) * scs.erfinv(2*y-1)
5
6 def draw_x(N: int = 1024):
7     y = np.random.rand(N)      # random uniform extraction in [0,1]
8     x = inv_cumulative_func(y) # retrieve x, distributed as Normal
9     return x
```

In Figure 1.4 we see an example of a normal distribution $\mathcal{N}(0,1)$ obtained using the Monte Carlo Sampling method.

We can use a similar strategy to compute *integrals*. We know that integrals can be thought of *the area under the curve*; in higher dimensions, for a closed, bounded domain Ω, we can think as the integral over Ω as the (hyper)volume contained within it. So we can use an accept/reject algorithm based on sampling the hypercube containing it. More formally, suppose we have $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}^n$, and a (closed, bounded) domain $D \subseteq \Omega$. Suppose we want to compute

$$I \coloneqq \int_D f(x)\mathrm{d}\mu(x)\,.$$

We can numerically evaluate I via the so-called *Monte Carlo integration*. Suppose we have a random sampler of the integration domain D, with a $\mathrm{pdf} : D \to \mathbb{R}$, respecting

$$\int_D \mathrm{pdf}(x)\mathrm{d}\mu(x) = 1.$$

We can then extract $\{X_i\}_{i=1,\dots,N} \in D$, and compute

$$I_N \coloneqq \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{\mathrm{pdf}(X_i)}\,. \tag{1.50}$$

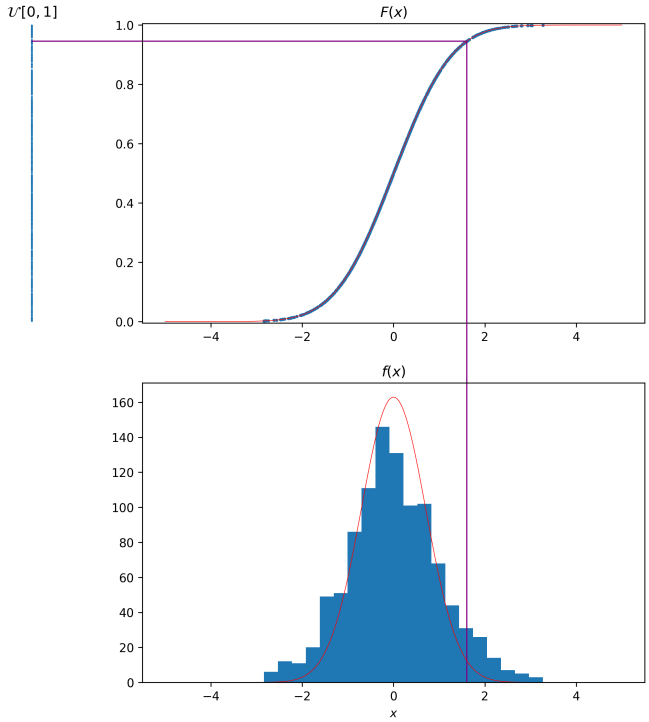


Figure 1.4: Example of the Monte Carlo Sampling of Listing 1.2 with $N = 1024$ draws.

fig:1:3: MCSampling

We can see that I_N approximate I, as it can be seen as its expectation value:

$$\begin{aligned}
\mathbb{E}[I_N] &= \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{\mathrm{pdf}(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}\left[\frac{f(X_i)}{\mathrm{pdf}(X_i)}\right] && \text{(linearity of } \mathbb{E}) \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_D \frac{f(x)}{\mathrm{pdf}(x)}\mathrm{pdf}(x)\mathrm{d}\mu(x) && \text{(definition of } \mathbb{E}) \\
&= \int_D f(x)\mathrm{d}\mu(x) && \text{(i.i.d.)} \\
&= I.
\end{aligned} \tag{1.51}$$

So, we have seen that it is an *unbiased* estimator of the integral. So we can *numerically* integrate any function defined on a bounded region D, in any dimensions. Obviously, the rate of convergence depends on N. Indeed, let

$$Y \coloneqq \frac{f(X)}{\mathrm{pdf}(X)}, \quad X \sim \mathrm{pdf}(x).$$

Then $I_N = \frac{1}{N}\sum_{i=1}^N Y_i$ with i.i.d. copies Y_i of Y. We have $\mathbb{E}[Y] = I$ and

$$\begin{aligned}\mathrm{Var}(I_N) &= \mathrm{Var}\left(\frac{1}{N}\sum_{i=1}^N Y_i\right)\\ &= \frac{1}{N^2}\sum_{i=1}^N \mathrm{Var}(Y_i)\\ &= \frac{1}{N^2}\cdot N\cdot \mathrm{Var}(Y)\\ &= \frac{\mathrm{Var}(Y)}{N}.\end{aligned} \tag{1.52}$$

Moreover,

$$\mathrm{Var}(Y) = \mathbb{E}\left[\left(\frac{f(X)}{\mathrm{pdf}(X)}\right)^2\right] - I^2 = \int_\Omega \frac{f(x)^2}{\mathrm{pdf}(x)}\,\mathrm{d}\mu(x) - I^2.$$

Therefore the mean squared error is

$$\mathrm{Var}(I_N) = \mathbb{E}\big[(I_N - I)^2\big] = \frac{1}{N}\left(\int_\Omega \frac{f(x)^2}{\mathrm{pdf}(x)}\,\mathrm{d}\mu(x) - I^2\right),$$

which gives a $O(N^{-1/2})$ convergence rate in root-mean-square.

1.4.1 Relevance for Machine Learning

Now one may ask: *why should I care?* Well, we have seen that functions space may have the concept of *distance*, like (pre-)Hilbert spaces, in particular $L^p(\Omega)$ spaces.

Let (X, Y) be a pair of random variables with values in $\mathscr{X} \subseteq \mathbb{R}^m$ and $\mathscr{Y} \subseteq \mathbb{R}^n$, respectively, defined on some probability space, and let P_X denote the distribution of X on $\mathscr{X}$. We assume that there exists a (possibly unknown) target function

$$f : \mathscr{X} \to \mathscr{Y}$$

such that f belongs to the Banach space $L^p(\mathscr{X}, P_X)$, equipped with the norm

$$\|F\|_{L^p(P_X)} \coloneqq \left(\int_{\mathscr{X}} \|F(x)\|^p \,\mathrm{d}P_X(x)\right)^{1/p},$$

for $1 \le p < \infty$. We consider a parametric family of functions

$$f_\theta : \mathscr{X} \to \mathscr{Y}, \quad \theta \in \Theta,$$

for instance a deep neural network (DNN). The (population) L^p distance between f_θ and f is given by

$$\|f_\theta - f\|^p_{L^p(P_X)} = \int_{\mathscr{X}} \|f_\theta(x) - f(x)\|^p \, dP_X(x).$$

In practice, the distribution P_X and the function f are unknown. Instead, we observe a dataset

$$\{(X_i, Y_i)\}_{i=1,\dots,N},$$

where (X_i, Y_i) are i.i.d. copies of (X, Y). The integral above can be approximated by Monte Carlo integration using the samples $X_1, \ldots, X_N$:

$$\int_{\mathscr{X}} \|f_\theta(x) - f(x)\|^p \, dP_X(x) \approx \frac{1}{N} \sum_{i=1}^{N} \|f_\theta(X_i) - f(X_i)\|^p.$$

Since $f(X_i)$ is not observed, we replace it by the corresponding label Y_i. This yields the empirical loss

$$R_N(f_\theta) \coloneqq \frac{1}{N} \sum_{i=1}^{N} \|f_\theta(X_i) - Y_i\|^p,$$

which is precisely the *empirical risk* associated with the loss $\ell(y', y) = \|y' - y\|^p$.

In the special case $p = 2$, we obtain the mean squared error (MSE)

$$R_N(f_\theta) = \frac{1}{N} \sum_{i=1}^{N} \|f_\theta(X_i) - Y_i\|^2,$$

whose square root is the empirical L^2-norm (root mean squared error, RMSE) of the difference between f_θ and the observed outputs.

Thus, we can see the entire *learning process* as the iterative algorithm to minimise the distance between the approximator DNN and the real function, evaluating such distance using the Monte Carlo approach, which yields the minimisation of the *empirical risk* [Sca24].

CHAPTER 2

Lecture 2: An introduction to numerical resolution of differential equations

chap:2

For a broader introduction to the subject, see [Ins25; Ise09; LPB15; Qua20; Sul21], and references therein.

In this lecture, we will briefly introduce a few possible way to *numerically* solve a generic Partial Differential Cauchy problem, in the general form

$$\begin{aligned} \mathcal{F}[u(x), x, \gamma] &= J(x)\,, \qquad x \in \Omega \subseteq \mathbb{R}^d\,, \\ \mathcal{B}[u(x), x, \gamma] &= g(x)\,, \qquad x \in \partial\Omega. \end{aligned} \tag{2.1}$$

Overall, the main goal of numerical methods is to find a way to recast the *differential* problem into an *algebraic* problem, which can be easily solved with standard numerical methods. In essence, the different methods we will presents, mainly differ in *how* they recast the differential problem into the algebraic problem:

1. Finite Difference Methods (FDMs) recast it by approximating the function on the nodes of a regular grid, and replacing derivatives with incremental ratios of the function on neighbouring nodes;

2. Finite Element Methods (FEMs) recast it by approximating the unknown function on the interior regions of an irregular mesh via a (multidimensional) linear approximation, and imposing stitching conditions on edges and nodes of the mesh, in order to solve the *weak formulation* of the PDE;

3. Finite Volume Methods (FVMs) recast it by approximating the unknown function on the interior regions of an irregular mesh via a (multidimensional) linear approximation, and imposing *flux* conditions on edges, in order to satisfy continuity.

We will close with a completely different approach, which is a *meshless* approach, called the **Kansa Method** [Kan90], which instead uses a *collocation* method to use an approximator to satisfy the differential equations, recasting the differential problem into an *optimisation* problem.

2.1 Finite Difference Methods (FDM)

sec:2:1:FDM

The core of FDM methods is to approximate the derivatives of unknown fields (multivariate functions) in terms of the value of the functions in a neighbourhood of the evaluation point. We have already encountered the Taylor expansion of a function

$$\begin{aligned} u(x+\Delta x) &= u(x) + \sum_{n=1}^{\infty} \frac{(\Delta x)^n}{n!} \left.\frac{\partial^n u}{\partial x^n}\right|_x \\ &= u(x) + \Delta x \partial_x u(x) + \mathcal{O}(\Delta x). \end{aligned}$$

This means that, up to an error $\mathcal{O}(\Delta x)^2$, we can express the derivative as

$$\partial_x u(x) = \frac{u(x+\Delta x) - u(x)}{\Delta x} + \mathcal{O}(\Delta x)\,, \qquad \text{(Forward Difference)} \tag{2.2}$$

which is the so-called *Forward Difference Formula*. But we can also compute

$$\begin{aligned} u(x-\Delta x) &= u(x) + \sum_{n=1}^{\infty} \frac{(-1)^n \Delta x^n}{n!} \left.\frac{\partial^n u}{\partial x^n}\right|_x \\ &= u(x) - \Delta x \partial_x u(x) + \mathcal{O}(\Delta x). \end{aligned}$$

so that

$$\partial_x u(x) = \frac{u(x) - u(x-\Delta x)}{\Delta x} + \mathcal{O}(\Delta x)\,, \qquad \text{(Backward Difference)} \tag{2.3}$$

which is the so-called *Backward Difference Formula*. Now, subtracting the two, we get a more precise approximation

$$\begin{aligned} &- \begin{cases} u(x+\Delta x) = u(x) + \Delta x \partial_x u(x) + \mathcal{O}(\Delta x) \\ u(x-\Delta x) = u(x) - \Delta x \partial_x u(x) + \mathcal{O}(\Delta x) \end{cases} \\ &\Rightarrow \partial_x u(x) = \frac{u(x+\Delta x) - u(x-\Delta x)}{2\Delta x} + \mathcal{O}((\Delta x)^2)\,, \qquad \text{(Central Difference)} \end{aligned} \tag{2.4}$$

Notice that the *Central Difference* has an higher precision on $\mathcal{O}((\Delta x)^n)$; to notice this explicitly, we simply expand to a higher order

$$\begin{aligned} &- \begin{cases} u(x+\Delta x) = u(x) + \Delta x \partial_x u(x) + \frac{\Delta x^2}{2} \partial_x^2 u(x) + \mathcal{O}((\Delta x)^2) \\ u(x-\Delta x) = u(x) - \Delta x \partial_x u(x) + \frac{\Delta x^2}{2} \partial_x^2 u(x) + \mathcal{O}((\Delta x)^2) \end{cases} \\ &\Rightarrow \partial_x u(x) = \frac{u(x+\Delta x) - u(x-\Delta x)}{\Delta x} + \mathcal{O}((\Delta x)^2)\,, \qquad \text{(Central Difference)} \end{aligned} \tag{2.5}$$

Higher derivative order can be found iterating the Taylor expansion, and linearly combine them, e.g. [For88]

$$\partial_x^2 u(x) = \frac{u(x+2\Delta x) + u(x) - 2u(x+\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x), \qquad \text{(Forward Difference)} \tag{2.6}$$

$$\partial_x^2 u(x) = \frac{u(x-2\Delta x) + u(x) - 2u(x+\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x), \qquad \text{(Backward Difference)} \tag{2.7}$$

Order of derivative	Order of accuracy	Approximations at $x = 0$; x-coordinates at nodes: -4	-3	-2	-1	0	1	2	3	4
0	∞					1				
1	2				$\frac{-1}{2}$	0	$\frac{1}{2}$			
	4			$\frac{1}{12}$	$\frac{-2}{3}$	0	$\frac{2}{3}$	$\frac{-1}{12}$		
	6		$\frac{-1}{60}$	$\frac{3}{20}$	$\frac{-3}{4}$	0	$\frac{3}{4}$	$\frac{-3}{20}$	$\frac{1}{60}$	
	8	$\frac{1}{280}$	$\frac{-4}{105}$	$\frac{1}{5}$	$\frac{-4}{5}$	0	$\frac{4}{5}$	$\frac{-1}{5}$	$\frac{4}{105}$	$\frac{-1}{280}$
2	2				1	-2	1			
	4			$\frac{-1}{12}$	$\frac{4}{3}$	$\frac{-5}{2}$	$\frac{4}{3}$	$\frac{-1}{12}$		
	6		$\frac{1}{90}$	$\frac{-3}{20}$	$\frac{3}{2}$	$\frac{-49}{18}$	$\frac{3}{2}$	$\frac{-3}{20}$	$\frac{1}{90}$	
	8	$\frac{-1}{560}$	$\frac{8}{315}$	$\frac{-1}{5}$	$\frac{8}{5}$	$\frac{-205}{72}$	$\frac{8}{5}$	$\frac{-1}{5}$	$\frac{8}{315}$	$\frac{-1}{560}$
3	2			$\frac{-1}{2}$	1	0	-1	$\frac{1}{2}$		
	4		$\frac{1}{8}$	-1	$\frac{13}{8}$	0	$\frac{-13}{8}$	1	$\frac{-1}{8}$	
	6	$\frac{-7}{240}$	$\frac{3}{10}$	$\frac{-169}{120}$	$\frac{61}{30}$	0	$\frac{-61}{30}$	$\frac{169}{120}$	$\frac{-3}{10}$	$\frac{7}{240}$
4	2			1	-4	6	-4	1		
	4		$\frac{-1}{6}$	2	$\frac{-13}{2}$	$\frac{28}{3}$	$\frac{-13}{2}$	2	$\frac{-1}{6}$	
	6	$\frac{7}{240}$	$\frac{-2}{5}$	$\frac{169}{60}$	$\frac{-122}{15}$	$\frac{91}{8}$	$\frac{-122}{15}$	$\frac{169}{60}$	$\frac{-2}{5}$	$\frac{7}{240}$

fig:2:1: table_fdm_1

Figure 2.1: Higher order and Higher precision derivatives. From [For88].

$$\partial_x^2 u(x) = \frac{u(x+\Delta x) - 2u(x) - u(x-\Delta x)}{\Delta x^2} + \mathcal{O}((\Delta x)^2), \qquad \text{(Central Difference)} \tag{2.8}$$

Is it possible to compute higher order derivatives, but also higher precision at fixed order, by enlarging the neighbouring points, from $\{-1, 0, +1\} \cdot \Delta x$ to $\{-q, \dots, 0, \dots, +q\} \cdot \Delta x$.

Now, let us pause for a moment. If we define $x_n \coloneqq n\Delta x, u_n \coloneqq u(x_n)$, we can see that we are expressing the derivative $\partial_x u_i \coloneqq \partial_x u(x)|_{x=x_i}$ as a *convolution*

$$\partial_x u_i = k \star u \coloneqq \sum_{j=-q}^{q} k_j u_{i+j} \tag{2.9}$$

where k is the *discretised* differential operator, expressed as a *linear kernel*. This is completely analogous to what a Convolutional Neural Network (CNN) is (a part the fact that it has *learnable* parameters). Similarly to CNNs, enlarging the receptive fields *increases* accuracy in the representation of data, but also *increases* the computational cost, with the possible additional effect of oversmoothing local effects.

2.1.1 A first application of finite differences: Sobel Edge detection algorithm

In computer vision task, *edge detection* has always be a relevant task. An algorithmic way to find the borders is the *Sobel filter*.

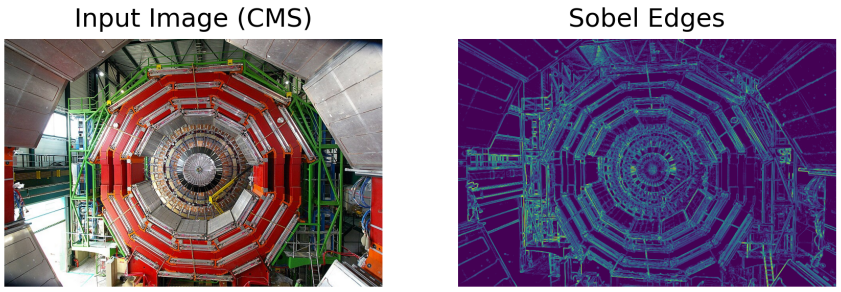


fig:2:1:
sobel_edge_
detection_img

Figure 2.2: Example of the usage of Sobel Edge detection algorithm

alg:2:1:Sobel

Algorithm 1 Sobel Edge Detection Algorithm

procedure SOBEL(image I, threshold θ)
  $G_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * [-1, 0, +1]$
  $G_y = \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix} * [1, 2, 1]$
  for $i \in [0, H]$ **do** ▷ Iterate over x pixels
    for $j \in [0, W]$ **do** ▷ Iterate over y pixels
      $X_{ij} = G_x \star I_{ij}$ ▷ Apply $x-$derivative, $y-$smoothing
      $Y_{ij} = G_y \star I_{ij}$ ▷ Apply $x-$smoothing, $y-$derivative
      $O_{ij} = \sqrt{X_{ij}^2 + Y_{ij}^2}$ ▷ Compute average magnitude
      if $O_{ij} < \theta$ **then**
        $O_{ij} = 0$ ▷ Threshold low values
  return O

The $[-1, 0, +1]$ piece is the first order derivative in its finite difference approximation. The $[1, 2, 1]$ instead is a discrete gaussian smoothing kernel.

lst:2:1:Sobel

Listing 2.1: Sobel Edge Detection Algorithm in Python

```python
import numpy as np

def sobel_edges(gray: np.ndarray, theta: float, mode: str = "reflect",
     return_gradients: bool = False):
    """
    Compute Sobel edge magnitude for a 2D grayscale image using pure
        NumPy.

    Parameters
    ----------
    gray : np.ndarray
        2D array (H, W), grayscale image.
    mode : str
        Padding mode for np.pad (e.g. "reflect", "edge", "constant").
    return_gradients : bool
        If True, also return (Gx, Gy).

    Returns
```

```
    -------
    mag : np.ndarray
        Edge magnitude image, same shape as input.
    (optional) Gx, Gy : np.ndarray
        Horizontal and vertical gradients.
    """
    if gray.ndim != 2:
        raise ValueError("Input must be a 2D grayscale array")
    if gray.max() > 0:
        gray /= 255.
        theta/= 255.
    # Work in float for accuracy
    img = gray.astype(np.float32, copy=False)
    # Pad by 1 pixel on all sides
    p = np.pad(img, 1, mode=mode)
    # Horizontal gradient Gx
    Gx = (
        (p[1:-1, 2:] - p[1:-1, :-2]) +
        2 * (p[2:, 2:] - p[2:, :-2]) +
        (p[:-2, 2:] - p[:-2, :-2])
    )
    # Vertical gradient Gy
    Gy = (
        (p[2:, 1:-1] - p[:-2, 1:-1]) +
        2 * (p[2:, 2:] - p[:-2, 2:]) +
        (p[2:, :-2] - p[:-2, :-2])
    )
    # Edge magnitude
    mag = np.hypot(Gx, Gy)
    # threshold
    mag = mag * (mag > theta)
    if return_gradients:
        return mag, Gx, Gy
    return mag
```

2.1.2 Using FDM to solve Poisson Equation: the Jacobi method

Suppose we need to solve a Poisson equation in three dimension

$$\begin{aligned} \Delta V(x,y,z) &= f(x,y,z)\,, \quad (x,y,z) \in \Omega \coloneqq [0,1]^3\,, \\ V(x,y,z) &= g(x,y,z)\,, \quad (x,y,z) \in \partial\Omega_D\,, \\ \hat{n}\cdot\nabla V(x,y,z) &= h(x,y,z)\,, \quad (x,y,z) \in \partial\Omega_N\,, \end{aligned} \tag{2.10}$$

with mixed Dirichlet/Neumann boundary conditions. As we have seen, the Finite Difference Method relies on the discretisation of the space, as $V(x_i, y_j, z_k) = V_{ijk}$, where

$$x_i = i\Delta x, y_i = j\Delta y, z_k = k\Delta z, \quad (i,j,k) = (1,1,1), \ldots, (N_x, N_y, N_z) \in \mathbb{N}^3\,, \tag{2.11}$$

where $N_x\Delta x = N_y\Delta y = N_z\Delta z = 1$.

Now, this implies that we can approximate the Laplace operator as

The generalisation to less general interval is trivial, and left as an exercise to the reader.

Again, the generalisation to different spatial dimension is trivial, as is trivial the generalisation to non-constant intervals $\Delta x \to \Delta x_i$.

$$\begin{aligned}\Delta V(\vec{x}) \to \Delta V_{ijk} \simeq & \frac{V_{(i+1),j,k} - 2V_{i,j,k} + V_{(i-1),j,k}}{(\Delta x)^2} \\ & + \frac{V_{i(j+1)k} - 2V_{ijk} + V_{i(j-1)k}}{(\Delta y)^2} \\ & + \frac{V_{ij(k+1)} - 2V_{ijk} + V_{ij(k-1)}}{(\Delta z)^2}\,.\end{aligned} \tag{2.12}$$

{eq:2:1: lapl_fdm}

The three lines in Equation (2.12) are, respectively, the three central difference of the operators $\partial_x^2, \partial_y^2, \partial_z^2$.

This finite difference approximation of the Laplace operator allows us to recast the PDE as an algebraic relation

$$\begin{aligned}f_{ijk} = & \frac{V_{(i+1),j,k} - 2V_{i,j,k} + V_{(i-1),j,k}}{(\Delta x)^2} \\ & + \frac{V_{i,(j+1),k} - 2V_{i,j,k} + V_{i,(j-1),k}}{(\Delta y)^2} \\ & + \frac{V_{i,j,(k+1)} - 2V_{i,j,k} + V_{i,j,(k-1)}}{(\Delta z)^2}\,,\end{aligned} \tag{2.13}$$

where $f(x_i, y_j, z_k) = f_{i,j,k}$.

Now, we can solve this algebraic relation via a trick, that is known as the *Jacobi Method*, due to its relation to the Jacobi Method to solve linear systems [Ise09]. The intuition behind the Jacobi method (mutated from the approach in solving linear systems) is to update iteratively the values of the field by bringing on the LHS the V_{ijk} terms, while the other terms are kept or put in the RHS, so that the field at the step $n+1$ is determined in terms of the field at steps n, as

$$\begin{aligned}V_{i,j,k}^{n+1} = \frac{\Delta^2}{2}\Big(& \frac{V_{i+1,j,k}^n + V_{i-1,j,k}^n}{(\Delta x)^2} \\ & + \frac{V_{i,j+1,k}^n + V_{i,j-1,k}^n}{(\Delta y)^2} \\ & + \frac{V_{i,j,k+1}^n + V_{i,j,k-1}^n}{(\Delta z)^2} - f_{ijk}\Big) \coloneqq U_{ijk}^n,\end{aligned} \tag{2.14}$$

{eq:2:1: jacobi_poisson}

where we have defined the RHS as the update operator U_{ijk}^n, and where we have defined

$$\begin{aligned}\frac{1}{\Delta^2} &\coloneqq \left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}\right) \\ &= \frac{(\Delta x)^2(\Delta y)^2 + (\Delta x)^2(\Delta z)^2 + (\Delta y)^2(\Delta z)^2}{(\Delta x)^2(\Delta y)^2(\Delta z)^2}\,.\end{aligned} \tag{2.15}$$

We thus have the following algorithm to solve the Poisson Cauchy problem:

alg:2:1:Jacobi

Algorithm 2 Jacobi Method for solving Poisson Problem

```
1: procedure JACOBI(grid (N_z, N_y, N_z), discretisation (Δx, Δy, Δz), threshold θ)
2:     init V_ijk                                          ▷ Initialise V_ijk
3:     ϵ = ∞                                               ▷ Initialise update Error
4:     while ϵ > θ do
5:         compute update U_ijk
6:         ϵ = |V_ijk − U_ijk|                             ▷ Update error
7:         V_ijk ← U_ijk                                   ▷ Update field
8:     return V
```

2.1.2.1 Jacobi Method as a Relaxation of a Diffusion Equation

Notice that, starting from the Jacobi method equation Equation (2.14), we can add zero, written up as

$$\begin{aligned} 0 &= V_{ijk}^n - V_{ijk}^n \\ &= V_{ijk}^n - \frac{\Delta^2}{2}\left(\frac{2V_{ijk}^n}{\Delta x^2} + \frac{2V_{ijk}^n}{\Delta y^2} + \frac{2V_{ijk}^n}{\Delta z^2}\right), \end{aligned} \tag{2.16}$$

and thus we get

$$\begin{aligned} V_{ijk}^{n+1} - V_{ijk}^n = \frac{\Delta^2}{2}\Big(& \frac{V_{i+1,j,k}^n - 2V_{ijk}^n + V_{i-1,j,k}^n}{\Delta x^2} \\ & + \frac{V_{i,j+1,k}^n - 2V_{ijk}^n + V_{i,j-1,k}^n}{\Delta y^2} \\ & + \frac{V_{i,j,k+1}^n - 2V_{ijk}^n + V_{i,j,k-1}^n}{\Delta z^2} - f_{ijk}\Big), \end{aligned} \tag{2.17}$$

Which is the Forward Euler approximation of the Diffusion equation with a constant source term

$$\frac{\partial V(t,\vec{x})}{\partial t} = \Delta V(t,\vec{x}) - f(\vec{x}), \tag{2.18}$$

with

$$\Delta t = \frac{\Delta^2}{2}$$

where the initialisation of the field is, here, interpreted as the initial condition.

This means that the Jacobi method converges when the diffusive process ends.

2.1.3 An application in Computer Vision: image inpainting with Laplace Equation

Image inpainting focuses on reconstructing missing regions or structures in an image through interpolation. The problem is inherently ill-posed: once part of an image is lost, it cannot be perfectly and unequivocally recovered unless the full original image is already known. There exist many inpainting techniques, often based on interpolation algorithms, but partial differential equation (PDE)-based approaches are among the most successful ones [Ber+00;

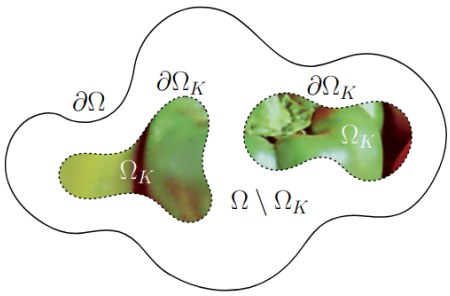


Figure 2.3: Generic inpainting model as given in Equation (2.19) with known data image f in Ω_K. The task consists in recovering a reasonable reconstruction of the image f in $\Omega \setminus \Omega_K$ by solving the PDE. From [Hoe+19].

fig:2:1:generic_inpainting_problem

GL14; Hoe+19]. A very popular strategy to tackle the inpainting problem is the 2D Laplace PDE with mixed Dirichlet/Neumann boundary condition:

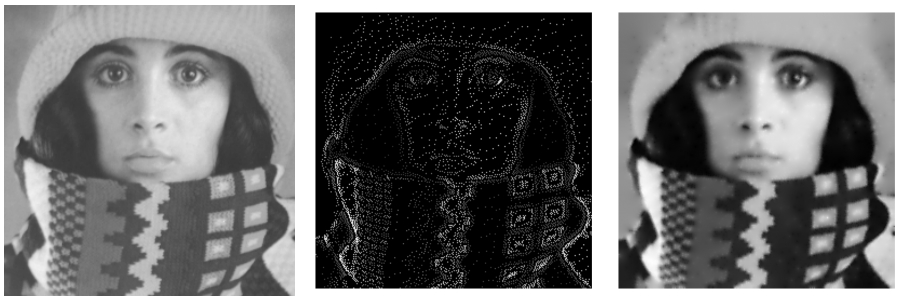
$$\begin{aligned} -\Delta u(x,y) &= 0\,, && (x,y) \in \Omega \setminus \Omega_K\,, \\ u(x,y) &= f(x,y)\,, && (x,y) \in \partial\Omega_K\,, \\ \hat{n} \cdot \nabla u &= 0\,, && (x,y) \in \partial\Omega \setminus \partial\Omega_K\,, \end{aligned} \tag{2.19}$$

{eq:2:1:inpainting_laplace}

Here, f represents known image data in a region $\Omega_K \subset \Omega$ (respectively on the boundary $\partial\Omega_K$) of the whole image domain Ω. A graphical sketch of the problem is given in Figure 2.3.

Another approach, equivalent to the former up to certain smoothness requirement, is to use the indicator function $c(x,y)$ of the optimal set Ω_K (i.e. $c(x,y) = 1$ if $(x,y) \in \Omega_K$, and $c(x,y) = 0$ otherwise) and define the pure Laplace-Neumann problem on Ω

$$\begin{aligned} c(x,y)[u(x,y) - f(x,y)] - [1 - c(x,y)]\nabla^2 u(x,y) &= 0, && (x,y) \in \Omega \\ \nabla_n u &= 0\,, && (x,y) \in \partial\Omega \setminus \Omega_K. \end{aligned} \tag{2.20}$$



(a) Original Image (b) Optimal masking. It has $\sim 90\%$ of missing data. (c) Inpainted solution using solving the Laplace problem Equation (2.19).

fig:2:1:inpainting_trui

A relevant result of this approach is that it is possible to have a high compression of the data. In Section 2.1.3, an optimal compression, containing

less than 10% of pixels of the original image, is inpainted using the 2D Laplace pde. A python sketch of the algorithm is reported in Listing 2.2.

lst:2:1:Laplace_Inpainting

Listing 2.2: Laplace inpainting

```python
import numpy as np

def inpaint_with_laplace(real: np.array, bnd: np.array, N_iter: int):
    # mask: True = unknown pixel
    unknown = (bnd == 0)
    # initialize solution
    V = bnd.copy()
    # Jacobi iteration
    for _ in tqdm.tqdm(range(N_iter)):
        V_new = V.copy()
        # Laplace update only on unknown pixels
        laplace =  (
            V[2:, 1:-1] + V[:-2, 1:-1] +
            V[1:-1, 2:] + V[1:-1, :-2]
        ) / 4.
        # update unknown pixels
        V_new[1:-1, 1:-1][unknown[1:-1, 1:-1]] = laplace[unknown[1:-1,
             1:-1]]
        # enforce boundary conditions
        # neumann
        V_new[ 0, :] = V_new[ 1, :]
        V_new[-1, :] = V_new[-2, :]
        V_new[ :, 0] = V_new[ :, 1]
        V_new[ :,-1] = V_new[ :,-2]
        # data
        V_new[~unknown] = bnd[~unknown]
        # enforce V <= 1
        V_new[V_new>=1.]=1.
        # update V
        V = V_new
    return V
```

2.1.4 Solving Burgers equation with FDM

A simple example may be the tackling of the Burgers equation; a very nice set-up is the following Cauchy problem

$$\begin{aligned} \partial_t u(t,x) + u(t,x)\partial_x u(t,x) - \nu^2 \partial_x^2 u(t,x) = 0\,, &\qquad (t,x) \in [0,1] \times [-1,+1], \\ u(t=0,x) = -\sin(\pi x)\,, &\qquad x \in [-1,+1], \\ u(t,x=\pm 1) = 0\,, &\qquad t \in [0,1], \end{aligned} \tag{2.21}$$

{eq:2:1:Burgers_FDM}

Where $\nu = 0.01/\pi$. A very minimal, pure python (plus numpy) implementation is

lst:2:1:Burgers_FDM

Listing 2.3: FDM for Burgers problem Equation (2.21)

```python
import numpy as np

use_cfl = False
```

```
# Parameters
nu = 0.01 / np.pi          # viscosity
Nx = 201                   # number of spatial points
Nt = 5000                  # number of time steps
xmin, xmax = -1.0, 1.0
tmax = 1.0
# Grid
x = np.linspace(xmin, xmax, Nx)
dx = x[1] - x[0]
dt = tmax / Nt
# Optionally: choose dt using a crude Courant-Friedrichs-Lewy (CFL) condition-like restriction instead
if use_cfl:
    cfl_adv = 0.4
    cfl_dif = 0.4
    dt = min(cfl_adv * dx / np.pi, cfl_dif * dx**2 / nu)   # since |u| <= 1 from initial data
    Nt = int(tmax / dt)
# Initial condition: u(0,x) = -sin(pi x)
u = np.zeros([Nt, Nx])
u[0, :] = - np.sin(np.pi * x)
# Enforce boundary conditions at t = 0
u[:,  0] = 0.0
u[:, -1] = 0.0
# Time stepping: forward Euler in time, central differences in space
for n in tqdm.tqdm( range(1, Nt) ):
    un = u[n-1,:].copy()
    # First derivative u_x (central difference)
    ux = (un[2:] - un[:-2]) / (2.0 * dx)
    # Second derivative u_xx (central difference)
    uxx = (un[2:] - 2.0 * un[1:-1] + un[:-2]) / dx**2
    # Interior update: u_t = -u u_x + nu u_xx
    u[n, 1:-1] = un[1:-1] + dt * (-un[1:-1] * ux + nu * uxx)
    # Dirichlet boundary conditions u(t, \\pm 1) = 0
    u[:, 0 ] = 0.0
    u[:, -1] = 0.0
```

The code reported in Listing 2.3 produces a solution which can be plotted in a 2D plot, as shown in Figure 2.5. We see why the Burgers equation is an interesting example: it develops a *shock* in a finite time.

2.1.5 A heuristic connection to CNN and Graphs: Stencil representation & Heat equation on Graphs

We have seen in Section 1.2.1.1 that we can derive the diffusion equation by balancing heat flux in spatially discretised cells, obtaining a relation between the rate change in time with the Laplacian in space (i.e., the rate of the rate of change in space). We have seen here that we can stencil-approximate the Laplacian as

$$(\Delta u)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2},$$

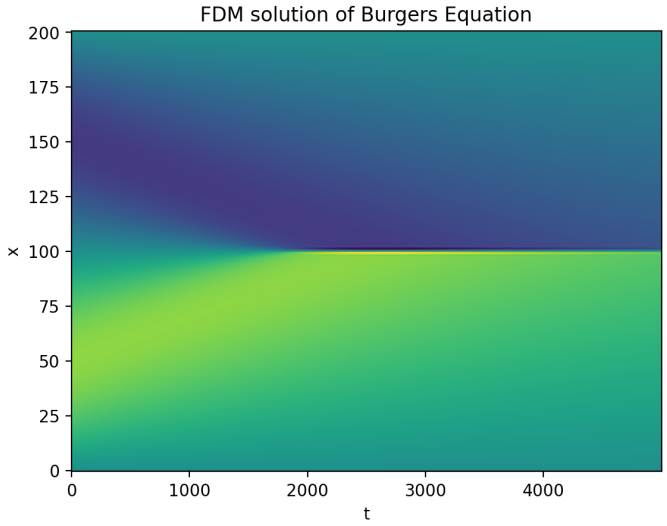


fig:2:1: FDM_burgers

Figure 2.5: FDM results of the simple code Listing 2.3.

using cental difference finite differences. We can go on, and generalise it to higher differences

$$(\Delta u)_{ij} = \frac{u_{i+1;j} - 2u_{ij} + u_{i-1;j}}{h_x^2} + \frac{u_{i;j+1} - 2u_{ij} + u_{i;j+1}}{h_y^2} \,,$$

and so on. We can represent *graphically* this *stencil* operators [Ise09], e.g.

$$(\Delta u)_i \;\coloneqq\; \left[\; \mathbf{1} \;\; \mathbf{-2} \;\; \mathbf{1} \;\right] \star u_i \,, \tag{2.22}$$

in 1D, or

$$(\Delta u)_{ij} \;\coloneqq\; \begin{bmatrix} & \mathbf{1} & \\ \mathbf{1} & \mathbf{-4} & \mathbf{1} \\ & \mathbf{1} & \end{bmatrix} \star u_{ij} \,, \tag{2.23}$$

in 2D, and

$$(\Delta u)_{ijk} \;\coloneqq\; \left[\begin{array}{ccc} & \mathbf{1} & \\ & \mathbf{1} & \\ \mathbf{1} & \mathbf{-6} & \mathbf{1} \\ \mathbf{1} & & \\ & \mathbf{1} & \end{array}\right] \star u_{ijk} \,. \tag{2.24}$$

in 3D, and so on. From here, it is almost immediate to see the connection with convolutional neural networks, in the sense that we have a *kernel* (the stencil operator) convolved with the input (the field), in 1D, 2D and 3D.

But we can draw an additional connection, to the field of Graph Theory, as well as Graph Machine Learning and Graph Neural Networks [Bac+20; Sca24]. Indeed, let us suppose we have a graph $\mathcal{G} = \{V, E\}$, where $V = \{v_i\}_{i=1,\dots N_{\text{nodes}}}$ are the vertices and $E = \{e_{ij}\}_{(i,j)=(1,1),\dots(N_{\text{nodes}},N_{\text{nodes}})}$ the edges.

Suppose we want to describe the evolution of temperature on a graph. It may be, for example, that we have a set of sensors in a building (the nodes), and few of those are "connected" by, e.g., corridors (the edges). We may define the "temperature" on the graph, i.e. the temperature gotten by the sensor i at time t. Formally, we have an application $u : [0, T] \times \mathcal{G} \to \mathbb{R}$, that we denote as $u_i(t)$. Now, as we have done in Section 1.2.1.1, we suppose that the rate of evolution of the temperature depends on the heat flow, which is proportional to the difference in temperature between a node i and all its neighbouring nodes $j \in \mathcal{N}_i$, i.e.

$$\partial_t u_i(t) \propto \sum_{j \in \mathcal{N}_i} [u_j(t) - u_i(t)] \,.$$

With $\mathcal{M}_{N\times N}(\mathbb{R})$ we denote the space of the $N \times N$ matrices with entry values in $\mathbb{R}$.

Now, we recall the concept of *Adjacency Matrix* $A_{ij} \in \mathcal{M}_{N\times N}(\mathbb{R})$, which is the matrix defined as

$$A_{ij} = \begin{cases} 1 \,, & (i,j) \text{ are connected} \,, \\ 0 \,, & \text{otherwise,} \end{cases}$$

which is, by construction, symmetric, at least for undirected graphs, as in this case. Now, the above equation is, calling α the proportionality constant,

$$\begin{aligned} \partial_t u_i(t) &= \alpha \sum_{j=1}^{N} A_{ij}[u_j(t) - u_i(t)] \\ &= \alpha \sum_{j=1}^{N} A_{ij} u_j(t) - \alpha \left(\sum_{j=1}^{N} A_{ij} \right) u_i(t) \,, \end{aligned}$$

now, the last term is simply computing the *connectivity* of the $i-th$ node, i.e. the number of other nodes connected (sharing an edge) with the $i-th$ node, which is called *degree*, d_i,

$$d_i \coloneqq \sum_{j=1}^{N} A_{ij} \,,$$

and the diagonal matrix defined as

$$\mathrm{D} \coloneqq \mathrm{diag}(d_1, \dots d_N) \,, \quad D_{ij} = \delta_{ij}\, d_i \,,$$

is the *Degree Matrix*. Now, we see that we can rewrite the *diffusion equation in the graph* in a matricial form; called $\mathbf{u}(t) = (u_1(t), \dots, u_N(t))$, we have

$$\partial_t \mathbf{u}(t) = -\alpha[\mathrm{D} - \mathrm{A}] \cdot \mathbf{u}(t). \tag{2.25}$$

{eq:2:1:5: GraphDiff}

Thus, the matrix appearing there

$$\mathrm{L} \coloneqq \mathrm{D} - \mathrm{A} \,, \qquad L_{ij} = D_{ij} - A_{ij} \,, \tag{2.26}$$

{eq:2:1:5: GraphLapl}

is, by analogy, called *Graph Laplacian* matrix of the graph.

2.1.5.1 The spectral theory on the graph and Laplacian Eigenmaps as an unsupervised dimensional reduction method

Disclaimer: for a more in-depth introduction to the subject, see e.g. [Chu96].

We have thus seen that the diffusion on graphs behave like the diffusion PDE Equation (2.25), where the Laplacian is implemented as the above Graph Laplacian L Equation (2.26). But, by definition, for undirected graphs, the Graph Laplacian is real and symmetric, $L = L^T$. Furthermore, for any function $g : \mathcal{G} \to \mathbb{R}$, i.e. $g : v \mapsto g(v)$ (like the heat above at fixed time), we can see that

$$\begin{aligned}
\frac{\langle \mathrm{g}, \mathrm{Lg} \rangle}{\langle \mathrm{g}, \mathrm{g} \rangle} &= \frac{\sum_{i,j} L_{ij} g_i g_j}{\sum_{i=1}^N g_i^2} \\
&= \frac{\sum_{i,j} [D_{ij} - A_{ij}] g_i g_j}{\sum_{i=1}^N g_i^2} \\
&= \frac{1}{2} \frac{\sum_{i,j} A_{ij}(g_i^2 + g_j^2) - 2\sum_{i,j} A_{ij} g_i g_j}{\sum_{i=1}^N g_i^2} \\
&= \frac{\sum_{i,j} A_{ij}(g_i - g_j)(g_i - g_j)}{\sum_{i=1}^N g_i^2} \\
&= \frac{\sum_{i\sim j} (g_i - g_j)^2}{\sum_{i=1}^N g_i^2} \geq 0 \,,
\end{aligned}$$

where $i \sim j$ we denote the sum over (i, j) which shares an edge, and where we have noted that, since D is diagonal, A traceless symmetric, and $d_i = \sum_j A_{ij}$

$$\begin{aligned}
\sum_{i,j} D_{ij} g_i g_j = \sum_{i,j} D_{ij} g_i^2 &= \sum_i g_i^2 \sum_j A_{ij} \\
&= \frac{1}{2} \sum_{i,j} A_{ij}(g_i^2 + g_j^2) \,.
\end{aligned}$$

Thus, we have proved that the Graph Laplacian is a (real symmetric) semi-positive definite matrix; thus, (i) by the Spectral Theorem, it is a diagonalisable matrix, i.e. $\exists\{\lambda_i, \mathrm{v}_i\}_{i=1,\dots,N}$ eigenvalues - (orthonormal) eigenvectors pairs, such that,

$$\mathrm{L} = U \Lambda U^T \,, \quad \Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_N) \,, \; U = (\mathrm{v}_1 | \dots | \mathrm{v}_N) \,,$$

and that, since $\langle g, Lg \rangle \geq 0$, $\forall g$, we can specialise to any v_i and see that

$$0 \leq \langle \mathrm{v}_i, L\mathrm{v}_i \rangle = \lambda_i \langle \mathrm{v}_i, \mathrm{v}_i \rangle = \lambda_i \,, \qquad \forall i = 1, \dots, N \,,$$

i.e. all eigenvalues of the graph Lagrangian are greater or equal to zero.

Notice that it is easy to see that, if a graph is connected, there is a eigenvector with zero eigenvalues, which is $\mathrm{v}_1 = (1, \dots, 1)/N$. So the eigenvalues of the graph Laplacian can be ordered as

$$0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_N \,.$$

This fact is quite relevant for the Laplacian Eigenmaps.

The reader may have noticed the minus sign in the definition of the Graph Laplacian w.r.t. the Laplacian operator analogy in the diffusion equation. The sign is inserted to have the graph Laplacian semi-positive definite, and not semi-negative definite.

In general, the number of zero eigenvalues are the number of connected subgraph present in the original graph.

Laplacian Eigenmaps Laplacian Eigenmaps is an unsupervised learning method to non-linearly dimensionally reduce a dataset $X \in \mathbb{R}^{N\times F}$, with N data having F features [BN03]. In the abstract, they state

« [...] *We consider the problem of constructing a representation for data lying on a lowdimensional manifold embedded in a high-dimensional space. Drawing on the correspondence between the graph Laplacian, the Laplace Beltrami operator on the manifold, and the connections to the heat equation, we propose a geometrically motivated algorithm for representing the high dimensional data. The algorithm provides a computationally efficient approach to nonlinear dimensionality reduction that has locality-preserving properties and a natural connection to clustering.* »

Laplacian Eigenmaps utilises the fact that the eigenvalues of the graph Laplacian are ordered (and positive), and represents the scale of the rate of changes of the node features over the graph. Indeed, working in analogy with the Laplace (Beltrami) operator, $-\Delta$, and its eigenfunctions, the Fourier modes $\mathrm{e}^{\mathrm{i}k\cdot x}$, the authors uses the low-eingevalues eigenvectors of W_{ij}, i.e. the weighted-graph Laplacian, also called *Heat Kernel*, defined as

$$W_{ij} = A_{ij}\, \exp\left[-\beta \|\mathrm{x}_i - \mathrm{x}_j\|\right] ,$$

(where β is an hyper-parameter) to extract *global* properties of the graph, built on top of the dataset X. The algorithm is

1. Build the Graph: Starting from the dataset X, an $\epsilon-$neighbourood is built, i.e., for each data-point x_i (i.e., the node), we say that $\mathrm{x_j}$ is connected if $\|\mathrm{x}_i - \mathrm{x}_j\| \leq \epsilon$, where the norm is computed in the feature space.

Another approach could be use kNN algorithm to selected k neighbours for each data point.

2. The Heat Kernel is computed; Thus, from there, we defined the (weighted) Laplacian as
$$\mathrm{L} = \mathrm{D} - \mathrm{W} , \qquad D_{ij} = \sum_j W_{ij} .$$

Sometimes, Laplacian Eigenmaps are perfomed on the standard graph Laplacian L, so using $W = A$.

3. The following eigen-problem is solved:
$$\mathrm{Lv} = \lambda \mathrm{Dv} .$$

3. The reduced dimensional components are thus selected as the first k (non-trivial) eigenvectors (v_1 is the trivial one),
$$\mathbf{x}_i \mapsto (\mathrm{v}_2(i), \dots, \mathrm{v}_{k+1}(i)) .$$

The analogy $L \longleftrightarrow -\Delta$ is

$$\begin{aligned} L &\longleftrightarrow -\Delta \\ (\lambda_i, \mathrm{v}_i) &\longleftrightarrow (k, \mathrm{e}^{\mathrm{i}k\cdot x}) \\ g = \sum_i \langle \mathrm{v}_i, g\rangle\, \mathrm{v}_i &\longleftrightarrow f(x) = \int f_k \mathrm{e}^{\mathrm{i}k\cdot x}\mathrm{d}k , \quad f_k \coloneqq \langle f, \mathrm{e}^{\mathrm{i}k\cdot x}\rangle \end{aligned}$$

so that, if k (λ_k) is low, is like $\mathrm{e}^{\mathrm{i}k\cdot x}$ (v_k) oscillates slowly, thus extracting long-range behaviour of the function; on the contrary, the higher k represents

rapidly oscillating functions, which captures small-scale features of the function. The idea behind Laplacian Eigenmaps is thus to keep the lowest ones, because the highest ones may be contain noise effects in the data, effectively reducing the feature spaces in a non-linear way.

Notice that the spectral problem solved here is a *modified* one; or, better said (if L is the standard Laplacian), we are solving the spectral problem for the *random-walk* graph Laplacian,

$$\mathrm{L}_{\text{random-walk}} \coloneqq \mathrm{I} - \mathrm{D}^{-1}\mathrm{A}.$$

Spectral Graph Convolutions In [Bru13; DBV16] the authors wanted to extend the concept of *convolution* from images to graphs. The idea was that, since in the graph context there is no simple meaningful way to define a *translation* operator in the vertex domain, it is better to define the convolution operator on the graph in the Fourier space, i.e. the space of the eigenvectors of the graph Laplacian, such that

$$x \star_{\mathcal{G}} y = U\left[(U^T x) \odot (U^T y)\right],$$

where $\odot$ is the element-wise Hadamard product. There, we can see that x signal is filtered by (a possible learnable) g_θ via

$$y = g_\theta(L)x = g_\theta(U\Lambda U^T)x = U g_\theta(\Lambda)U^T x.$$

So the idea of the graph spectral convolution is

(i) Diagonalise the Graph Laplacian, and construct the Fourier space;

(ii) Transform the input into the Fourier space ($y \mapsto U^T x$;

(iii) Apply a learnable transformation g_θ in Fourier space

(iv) Transform back to direct space, using $U^{-1} = U^T$, i.e., by multiplying by U.

In Chapter 5, we will encounter a family of Neural Operators building on this idea, the so-called *Fourier Neural Operators*.

2.2 Finite Element Methods (FEM)

sec:2:2:FEM

In this section, we briefly introduce the *Finite Element Method* (FEM). We have no intention of furnishing an exhaustive introduction on the subject: this section is merely intendend as a first contact with the numerical resolution of PDEs in the weak sense. For a broader introduction to the subject, see [Ise09; Qua20; Smi22], and references therein.

It is convenient to begin the section by outlining the five main ideas behind the finite element method:

1. Formulate the Cauchy problem as a *variational* problem, e.g. by seeking a function that minimizes a nonlinear functional;
2. If possible, reduce differentiability requirements in the above functional, using integration by parts;

We will use the term *Cauchy Problem* with a more general sense, as an encompassing term referring to the PDE + adequate number of Boundary and Initial Conditions.

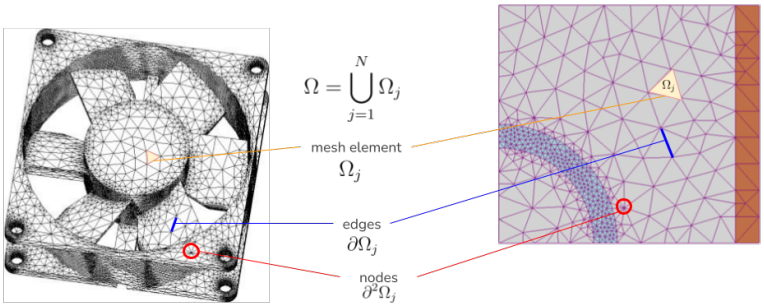


Figure 2.6: Visual representation of the *meshing* of the domain Ω for the FEM analysis (2D surface in 3D space on the left, and a plain 2D region in the right). Adapted from [BCT02].

fig:2:2:FEM1

3. If integration by parts have been used in step 2., take appropriate care with boundary conditions, and distinguish between essential and natural conditions;

4. Replace the underlying problem by an approximate one, restricted to a finite-dimensional space;

5. Choose a finite-dimensional space which is spanned by functions with a small support (finite element functions).

Ok, before delving into it further, let us recap the problem. We have to find $u : \Omega \to \mathbb{R}$, which satisfies the Cauchy problem

$$\begin{aligned} \mathcal{F}[u(x)] &= J(x)\,, \quad x \in \Omega \quad \text{(PDE)} \\ \mathcal{B}[u(x)] &= g(x)\,, \quad x \in \partial\Omega \quad \text{(BC)} \end{aligned} \tag{2.27}$$

where, for the moment, we are focusing on *pure boundary* problems (i.e., time-free). We assume that the aforementioned problem can be written in its *weak formulation* via a bilinear form $\mathfrak{g} : X(\Omega) \times X(\Omega) \to \mathbb{R}$, where $X(\Omega)$ is the Banach space where u lives, $u \in X(\Omega)$.

We then define a *mesh* of the $\Omega \subset \mathbb{R}^d$, i.e. we represent Ω as an union of subdomains, the *elements*,

$$\Omega = \bigcup_{j=1}^{N} \Omega_j\,,$$

where $\overset{\circ}{\Omega}_i \cap \overset{\circ}{\Omega}_j = \emptyset$.

These elements have specific *overlapping* regions, which are the *edges*, $\partial\Omega_j$. A common approach in 2D is to use *triangulation*, i.e. represents 2D domains as an union over triangles. In any case, we may have that our elements are manifold with *corners*, i.e. we have a non-negligible $\partial^2\Omega_j$, which are *corners*.

Let us now build a basis family $\{\phi_j : \Omega_j \to \mathbb{R}\}$, with appropriate behaviour under differentiation (up to a certain order, e.g. $\phi_j \in C^1(\Omega_j)$. We can represent the solution u as

$$u(x) = \sum_{j=0}^{N} \alpha_j \phi_j(x)\,. \tag{2.28}$$

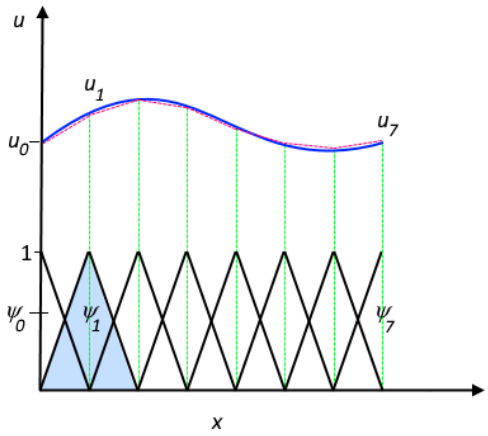
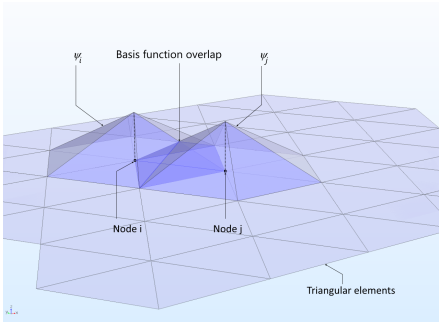
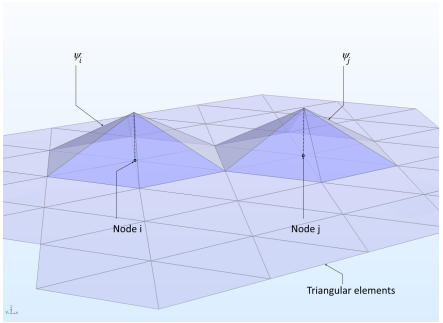


Figure 2.7: The function u (solid blue line) is approximated with u_h (dashed red line), which is a linear combination of linear basis functions $\{\phi_j\}$, represented by the solid black lines. From [Mul].

fig:2:2:FEM_repr



(a) Overlap between basis functions. From [Mul].



(b) No overlap between basis functions. From [Mul].

fig:2:2:FEM2

Now, let us rewrite the weak formulation of the problem,

$$\begin{aligned} &\mathfrak{g}(v, \mathcal{F}[u]) = \mathfrak{g}(v, J)\,, \qquad \forall v \in X(\Omega), \\ \text{i.e.} \quad &\int_\Omega v(x)\mathcal{F}[u(x)]\mathrm{d}^d x = \int_\Omega v(x)J(x)\mathrm{d}^d x\,. \end{aligned} \tag{2.29}$$

Sometimes, we will use also $\mathscr{G}(u(x), v(x)) \coloneqq \langle u(x), v(x)\rangle_{\mathbb{R}^d}$, so that we can write

$$\int_\Omega \mathscr{G}(\nabla v(x), \nabla u(x))\mathrm{d}^d x = \int_\Omega \nabla v(x) \cdot \nabla u(x)\mathrm{d}^d x\,.$$

Now, let us focus on a specific case: Laplace operator on $\mathbb{R}^d$, i.e. we have the Poisson problem

$$\nabla^2 u(x) = J(x), \quad x \in \Omega \subset \mathbb{R}^d,$$

so that

$$\int_\Omega v(x)\nabla^2 u(x)\mathrm{d}^d x = \int_\Omega v(x)J(x)\mathrm{d}^d x\,.$$

Now, integrating by parts via the Gauss-Green-Ostrogradskij-Stokes Theorem, we get

$$\underbrace{\int_{\partial\Omega} v(x)\nabla u\cdot\hat{n}_\Sigma \mathrm{d}^{d-1}\Sigma}_{\text{Boundary Term}} - \int_\Omega \nabla v(x)\cdot\nabla u(x)\mathrm{d}^d x = \int_\Omega v(x)J(x)\mathrm{d}^d x\,, \tag{2.30}$$

obtaining the *weak formulation* of the PDE.

Why is it useful? well, now, we can represent our functions on the $\{\phi_j\}$ basis

Notice that we are using the Riesz theorem under the hood, to see $J \in X(\Omega)$.

$$u(x) = \sum_{j=0}^N \alpha_j\phi_j(x)\,, \qquad J(x) = \sum_{j=0}^N J_j\phi_j(x)\,. \tag{2.31}$$

Using this, we recast our *differential* problem into an *algebraic* one, specialising the basis in a way that

$$\int_\Omega \phi_j(x)\phi_k(x)\mathrm{d}^d x \neq 0\,, \qquad \text{if} \quad \Omega_j\cap\Omega_k\neq\emptyset \tag{2.32}$$

$$\int_\Omega \mathscr{G}\left(\nabla\phi_j(x),\nabla\phi_k(x)\right)\mathrm{d}^d x \neq 0\,, \qquad \text{if } \partial\Omega_j\cap\partial\Omega_k\neq\emptyset \tag{2.33}$$

so that we get

$$-\sum_{j=0}^N \alpha_j\int_\Omega \mathscr{G}\left(\nabla\phi_j(x),\nabla\phi_k(x)\right)\mathrm{d}^d x = \sum_{j=0}^N J_j\int_\Omega \phi_j(x),\phi_k(x)\mathrm{d}^d x\,, \tag{2.34}$$

where we have picked a specific test function to simply the computations. Now, since the integrals can be explicitly computed once we have selected the basis, we can define the matrices L_{jk} and M_{jk} as

$$\mathrm{L}_{jk} \coloneqq \int_\Omega \mathscr{G}\left(\nabla\phi_j(x),\nabla\phi_k(x)\right)\mathrm{d}^d x\,, \qquad \mathrm{M}_{jk} \coloneqq \int_\Omega \phi_j(x)\phi_k(x)\mathrm{d}^d x\,,$$

and thus we get the algebraic problem

$$-\mathrm{L}\alpha = \mathrm{M}J\,, \qquad \alpha = (\alpha_1,\dots\alpha_N), J = (J_1,\dots,J_N), \tag{2.35}$$

which can formally solved as

$$\alpha = -\mathrm{L}^{-1}\mathrm{M}J\,.$$

2.2.1 Ritz vs Galerkin method

subsec:4:2:1: RitzVsGalerkin

Let us now standardize a little bit the terminology, as well as introducing relevant formulation of the FEM approaches.

Definition 2.2.1 (Weak formulation)**.** Let $\Omega\subset\mathbb{R}^d$ be a bounded domain and set V to be a suitable Sobolev space (typically $V = H_0^1(\Omega)$). Given a bilinear form $a(\cdot,\cdot): V\times V\to\mathbb{R}$ and a bounded linear functional $f\in V'$, the *weak* (or *variational*) formulation of the boundary value problem is: find $u\in V$ such that

$$a(u,v) = f(v) \qquad \forall v\in V\,.$$

Notice that, up to now, we have said nothing on the bilinear forms. Let us relax the generalisation a bit, and let us keep in mind the Poisson problem of before. Using far away memories from physics courses, we may recall that, in some conservative systems, we may defined the solution trajectory of a point-particle of mass m as the one which *minimise* the energy *functional*; indeed, if a particle moves in a *potential* $U = U(q)$ with a trajectory $q : t \in [0, T] \mapsto q(t) \in \mathbb{R}^d$, the energy functional $E : q \in X \mapsto \mathbb{R}$ is

$$E[q] = \frac{1}{2} m\dot{q}^2 + U(q)\,,$$

which can be also seen as a function $Et \in [0, T] \mapsto \mathbb{R}$ once q is selected; but energy is conserved, thus $\partial_t E = 0$, i.e.

$$\begin{aligned} 0 = \partial_t E &= \partial_t \left(\frac{1}{2} m\dot{q}^2 + U(q) \right) \\ &= \dot{q}\,(m\ddot{q} + \nabla_q U) \Longrightarrow m\ddot{q} = -\nabla_q U, \quad \text{since } \dot{q} \neq 0, \end{aligned} \tag{2.36}$$

which is the standard Newton law for conservative forces, since the force is $\mathrm{F} = m\ddot{q}$. Thus, we can think to solve the differential problem by *minimising* the Energy functional of the system (describing the *status* of the system), instead of solving the differential equation (describing the *dynamics* of the system).

The derivation above is heuristic: the correct variational route is via Hamilton's principle (or the Euler-Lagrange equations) rather than energy conservation alone, and a rigorous derivation requires specifying the admissible function space and boundary conditions.

These two approaches are at the core of two similar, yet different, schemes: the *Ritz Method* and the *Galerkin Method*.

Definition 2.2.2 (Ritz method (energy formulation))**.** Assume the PDE admits an energy functional $E : V \to \mathbb{R}$ whose Euler–Lagrange equation is equivalent to the weak form above. The continuous problem can then be written as the variational minimisation

$$u = \arg\min_{v \in V} E[v].$$

The *Ritz* discretisation is obtained by restricting E to a finite dimensional trial space $V_h \subset V$ and solving

$$u_h = \arg\min_{v \in V_h} E[v].$$

Notice that, when E is quadratic, one obtains a symmetric coercive bilinear form $a(\cdot, \cdot)$ and the discrete equations follow from

$$a(u_h, v) = f(v), \qquad \forall v \in V_h\,.$$

Instead, if we don't want (or can't) use the energy functional minimisation trick, we are back to what we have seen before (maybe via the integration by parts of the Laplace operator): the Galerkin problem.

Definition 2.2.3 (Galerkin method)**.** Given the weak form $a(u, v) = f(v)$, choose finite-dimensional trial and test spaces $V_h \subset V$ and $W_h \subset V$. The *Galerkin* (or Petrov–Galerkin) method seeks $u_h \in V_h$ such that

$$a(u_h, w) = f(w) \qquad \forall w \in W_h.$$

When $W_h = V_h$ the method is the (Bubnov) Galerkin method; when $W_h \neq V_h$ it is a Petrov–Galerkin method. Galerkin projection requires a well-posed weak form but does not require the existence of an energy functional.

Remark 2.2.4. The essential conceptual distinction is that *Ritz* originates from an *energy minimisation* (so it requires a variational/Euler-Lagrange structure and typically leads to symmetric positive definite systems), while *Galerkin* is a *projection* of the weak residual onto a test space and therefore applies more broadly (including non-self-adjoint operators). Integration by parts is a common step in deriving weak forms for both approaches and is not the defining difference.

Remark 2.2.5. A relevant case where we encounter a Galerkin problem, but not a Ritz problem, is the general Elliptic PDE

$$-\nabla \cdot [a(x)\nabla u(x)] + b(x)\nabla u(x) + c(x)u(x) = f(x), \tag{2.37}$$

{eq:2:2:RAD_Eq}

where integration by parts cannot be made for *all* the LHS. This PDE contains

- in $a(x)$, the information about *media anisotropy* for the diffusion of u; for modelling the diffusion in anistropic/porous media, we have the Fick Law relating the flux q to the field u, $\mathbf{q} = a(x)\nabla u(x)$, where a is called *diffusion matrix*.

- in $b(x)$, the information about the *transport* or *convection* of the field u; in such case, b is related to the velocity of the object the field u is moving into (e.g., a substance concentration u moving in a stream of liquid with velocity b.

- in $c(x)$, the information about the *reaction* of the media to the diffusion of u; for example, if we are modelling the concentration of a substance with u, c may be the rate of *decomposition* ($c < 0$) or *growth* ($c > 0$).

This is why Equation (2.37) is called *Reaction-Advection-Diffusion* equation.

Example 2.2.6 (Poisson problem, reprise)**.** For $-\Delta u = f$ with $u|_{\partial\Omega} = 0$ the energy functional is

$$E(v) = \tfrac{1}{2}\int_\Omega |\nabla v|^2 \,\mathrm{d}x - \int_\Omega fv \,\mathrm{d}x,$$

whose Euler-Lagrange equation yields the weak form $\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x$ for all $v \in H_0^1(\Omega)$. In this case the Ritz and (Bubnov) Galerkin formulations *coincide*.

Why are these two methods relevant? Well, because of the following:

Theorem 2.2.7 (Lax-Milgram Theorem)**.** *Let $a(\cdot,\cdot) : V \times V \to \mathbb{R}$ be a bounded and coercive bilinear form on the Hilbert space V. Then, for each bounded linear functional $f \in V'$, there is exactly one $u_M \in V$ with*

$$a(u, v) = f(v) \quad \forall v \in V.$$

This assures the *existence and uniqueness of the solution* for our problem.

Ritz vs Galerkin - Quick Comparison

Aspect	Ritz	Galerkin
Principle	Minimise an energy functional; solution is variational minimiser.	Enforce weak form residual orthogonality to test space.
Applicability	Requires PDE to admit a variational formulation; typically self-adjoint.	Applies to weak forms even without an energy functional.
Boundary handling	Natural BCs arise from variational derivative; essential BCs on trial space.	Essential BCs on trial space; natural BCs appear in weak form.
Advantages	Energetic interpretation; often symmetric positive definite systems.	Broad applicability; flexible test/trial choices (Petrov–Galerkin).
Drawbacks	Limited to variational problems.	May yield non-symmetric systems; stability depends on pairing.

2.3 Finite Volume Methods (FVM)

sec:2:3:FVM

As for the Finite Element Method, also for this section we have no intention of furnishing an exhaustive introduction on the Finite Volume Methods: this section is merely intended as a first contact with the numerical resolution of PDEs with the conservation approach. For a broader introduction to the subject, see [EGH00; Qua20], and references therein.

Finite Volume Methods are usually employed when we need to tackle differential problems describing *conservation laws*, which is quite canonical in various fields, such as (but not limited to) fluid mechanics, heat and mass transfer or petroleum engineering. The finite volume method is locally conservative because it is based on a "balance" approach: a local balance is written on each discretization cell which is often called "control volume"; by the divergence formula, an integral formulation of the fluxes over the boundary of the control volume is then obtained. The fluxes on the boundary are discretized with respect to the discrete unknowns.

The core essence of FVM, is that volume integrals in a partial differential equation that contain a divergence term are converted to surface integrals, using the divergence theorem. These terms are then evaluated as fluxes at the surfaces of each finite volume. Because the flux entering a given volume is identical to that leaving the adjacent volume, these methods are conservative (i.e., automatically satisfies conservation equation!) Another advantage of the finite volume method is that it is easily formulated to allow for unstructured meshes."Finite volume" refers to the small volume surrounding each node point on a mesh.

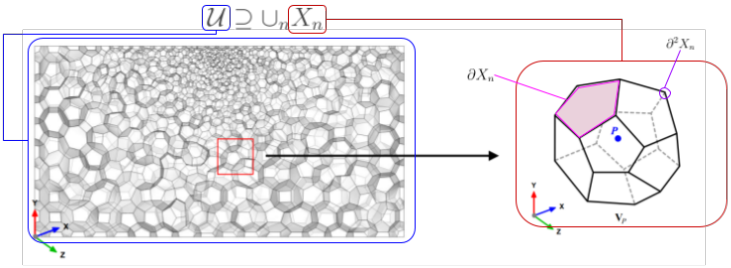


fig:2:3:FVM1

Figure 2.9: Example of finite volume tessellation.

Suppose now we have a general conservation equation

$$\partial_t q(t,\mathbf{x}) + \nabla \cdot \mathbf{F}(q(t,\mathbf{x}),t,\mathbf{x}) = f(t,\mathbf{x})\,, \tag{2.38}$$

defined on $\mathcal{U} \subset \mathbb{R}^d$ (d=3 WLOG), which can covered by the union of tassels $\bigcup_{n=1}^{N} X_n$.

Thus, for the $n-$th cell, we can take the volume integral over the total volume of the cell

$$\int_{X_n} \partial_t q(t,\mathbf{x})\mathrm{d}\omega_n + \int_{X_n} \nabla \cdot \mathbf{F}(q(t,\mathbf{x}),t,\mathbf{x})\mathrm{d}\omega_n = \int_{X_n} f(t,\mathbf{x})\mathrm{d}\omega_n\,.$$

Now, integrating the first and third term to get the *volume averaged* quantities $\bar{q}(t,\mathbf{x}_n), \bar{f}(t,\mathbf{x}_n)$, applying the divergence theorem on the second term, we get

$$\partial_t \bar{q}(t,\mathbf{x}_n) + \int_{\partial X_n} \hat{\mathbf{n}}_n \cdot \mathbf{F}(q(t,\mathbf{x}),t,\mathbf{x})\mathrm{d}\Sigma_n = \bar{f}(t,\mathbf{x}_n)\,.$$

This expression can now be recast into an (implicit) algebraic equation, discretising the time derivative via, for example, the forward Euler scheme:

$$\frac{\bar{q}_{i+1,n} - \bar{q}_{i,n}}{\Delta T} + F_{i,n} = f_{i,n}\,, \tag{2.39}$$

where we have discretised the time via $t_i = t_0 + i\Delta T$, and where we have defined

$$F_{i,n} \coloneqq \int_{\partial X_n} \hat{\mathbf{n}}_n \cdot \mathbf{F}(\bar{q}_{i,n},t_i,\mathbf{x})\mathrm{d}\Sigma_n(\mathbf{x})\,, \qquad \bar{f}_{i,n} \coloneqq f(t_i,\mathbf{x}_n)\,.$$

In a more explicit form, we can separate the fluxes into their contribution for each faces of the tessellation

$$\frac{\bar{q}_{i+1,n} - \bar{q}_{i,n}}{\Delta T} + \sum_{k-\text{faces}} F_{i;n,k} = f_{i,n}\,,$$

where we have defined the flux from the $n-$th to the $k-$th sub-volume:

$$F_{i;n,k} \coloneqq \int_{\partial X_n \cap \partial X_k} \hat{\mathbf{n}}_n \cdot \mathbf{F}(\bar{q}_{i,n},t_i,\mathbf{x})\mathrm{d}\Sigma_n(\mathbf{x})\,, \qquad \bar{f}_{i,n} \coloneqq f(t_i,\mathbf{x}_n)\,.$$

2.4 Meshless methods; Kansa's approach

sec:2:4:Kansa

Again, the same warning as before. So, for a more detailed introduction to the subject, see [Fas14; Mar18; Sch07], as well as the original paper by Kansa [Kan90].

Before delving into the Kansa method, let us (very briefly) recall the *spline* approximation of functions, and the *radial basis* approach. Sometimes, we have to face the problem of building an *unknown* function f from a set of data (which is usually *small*). Suppose that these available data consist of two sets: the *collocation points* (or *data sites*) $X = \{x_1, \dots, x_N\}$, and the *data values* $f_j = f(x_j), \forall j = 1, \dots, N$. Our goal is to find a *reconstructing* function, s; we can either *interpolate* the data, i.e. $s(x_j) = f_j$, or *approximate* the data, $s(x_j) \approx f_j$. Usually, the latter is more relevant, especially in statistical applications, since the data may contain *noise*.

For the 1D case (i.e., the *univariate* setting), we can approximate with a spline; indeed, it is a well-known fact that a large data set is better dealt with splines than by polynomials. Furthermore, one aspect to notice in contrast to polynomials, the accuracy of the interpolation process using splines is not based on the polynomial degree but on the spacing of the data sites.

We can represent any spline via the so-called *B-splines* (Basis Spline) [Emb16]; B-splines are defined *recursively* from the constant B-spline $B_{i,0}$

$$B_{i,0}(x) = \begin{cases} 1, & t_i \le x \le t_{i+1}\,, \\ 0, & \text{otherwise,} \end{cases}$$

via the recurrence relation

$$B_{i,k}(x) \coloneqq \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)\,.$$

So, to interpolate a dataset $\{(x_j, f_k\}_{j=1,\dots,N}$ via a spline $S_k(x)$ (using order k B-spline), such that $S_k(x_j) = f_j,\ \forall j = 1, \dots, N$, we have

$$S_k(x) = \sum_{j=-k}^{N} c_{j,k} B_{j,k}(x)\,. \tag{2.40}$$

Notice that, by construction, $S_k \in C^1[x_1, x_N]$.

Notice that, to find the $\{c_{j,k}\}$ coefficients, we need to solve the algebraic problem

$$f_\ell = S_k(x_\ell) = \sum_{j=-k}^{N} c_{j,k} B_{j,k}(x_\ell)\,, \qquad \forall \ell = 1, \dots, N.$$

Another relevant approximation is the cubic spline $s : [a, b] \subset \mathbb{R} \to \mathbb{R}$ as

$$s(x) = \sum_{j=1}^{N} a_j (x - x_j)_+^3 + \sum_{k=0}^{3} b_k x^k\,, \quad x \in [a, b].$$

Nevertheless, the take away home message is that [Mar18]

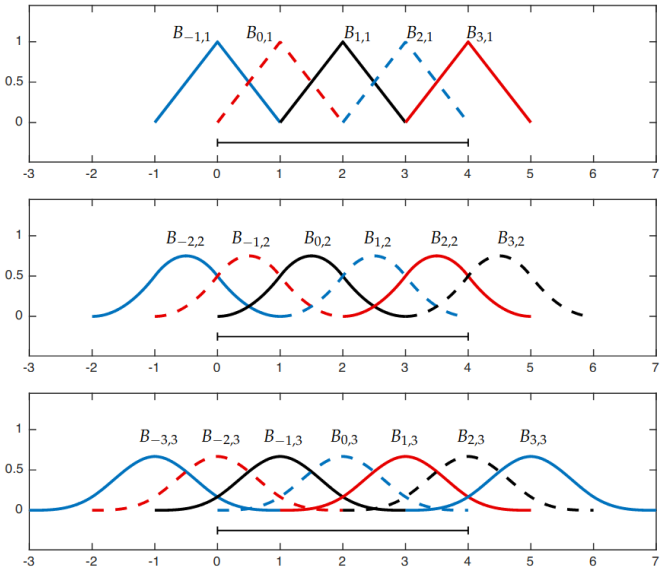


fig:2:4:Bsplines

Figure 2.10: B-splines of degrees 1,2,3. (top, centre, bottom). From [Emb16].

Proposition 2.4.1. *Every natural spline s has the representation*

$$s(x) = \sum_{j=1}^{N} a_j \phi(|x - x_j|) + p(x), \quad x \in \mathbb{R} \tag{2.41}$$

{eq:2:4:RBF1}

where $\phi(r) = r^3$, $r \geq 0$, and $p \in \mathcal{P}_1(\mathbb{R})$, the space of polynomials of degree 1. The coefficients $\{a_j\}$ have to satisfy the relations

$$\sum_{j=1}^{N} a_j = \sum_{j=1}^{N} a_j x_j = 0\,. \tag{2.42}$$

{eq:2:4:RBFCond}

On the contrary, for every set $X = \{x_1, \ldots, x_N\} \subset \mathbb{R}$ of pairwise distinct points and for every $f \in \mathbb{R}^N$, there exists a function s of the form Equation (2.41) satisfying Equation (2.42), that interpolates the data, i.e. $s(x_j) = f(x_j)$, $1 \leq j \leq N$.

The generalisation to functions of the form $\mathbb{R}^d \to \mathbb{R}$ is straightforward, and justifies where the name *radial basis function* (RBF) become more evident [Buh00]:

$$s(x) = \sum_{j=1}^{N} a_j \phi(\|x - x_j\|_2) + p(x), \quad x \in \mathbb{R}^d\,, \tag{2.43}$$

where $\phi : [0, \infty) \to \mathbb{R}$ is a fixed univariate function and $p \in \mathcal{P}^d_{m-1}$ is a low degree m polynomial.

The resulting interpolant is, up to a low-degree polynomial, a linear combination of shifts of a radial function.

Some relevant *radial basis* are

$$\phi(r) = \exp(-\epsilon r^2), \qquad \text{(gaussian)} \tag{2.44}$$

$$\phi(r) = \frac{1}{1+\epsilon r^2}, \qquad \text{(inverse quadratic)} \tag{2.45}$$

$$\phi(r) = \frac{1}{\sqrt{1+\epsilon r^2}}. \qquad \text{(inverse multiquadratic)} \tag{2.46}$$

2.4.1 The Kansa approach: RBF for numerical PDE resolution

We have now a way of interpolating, i.e. approximate, in a *meshless* way, any function, via radial basis. Furthermore, those functions are univariate (being radial) and analytical. Let us now suppose we need to solve a Cauchy problem on a domain $x \in \Omega \subset \mathbb{R}^d$

$$Lu(x) = f(x), \qquad x \in \Omega\,, \tag{2.47}$$

$$u(x) = b(x), \qquad x \in \partial\Omega_D\,, \tag{2.48}$$

$$\nabla_n u(x) = 0, \qquad x \in \partial\Omega_N\,, \tag{2.49}$$

$$\tag{2.50}$$

where L is a linear operator acting on the function u, and where $\nabla_n u(x) \coloneqq \hat{n} \cdot \nabla u(x)$ is the outward normal derivative at the boundary ($\hat{n}$ is the outward versor of $\partial\Omega$ in $x \in \partial\Omega$). Now, let us approximate the unknown function via a radial basis function on N_c *centres* $\{c_i\}_{i=1,\dots,N_c}$:

$$u(x) = \sum_{i=1}^{N_c} a_i \phi(r_i(x))\,, \qquad r_i(x) \coloneqq \|x - c_i\|. \tag{2.51}$$

We can now plug this in the PDE and compute *analytically* the relation

$$Lu(x) = L\sum_{i=1}^{N_c} a_i \phi(r_j(x)) = \sum_{i=1}^{N_c} a_i L\phi(r_j(x)) \coloneqq \sum_{i=1}^{N_c} a_i \ell(r_j(x)),$$

where we have defined the *explicitly, analytically* computable $L\phi(r_j(x))$ as a convenient function ℓ. Selecting now N_I *collocation points* $\{x_j\}_{j=1,\dots,N_I}$, we can, once again, reduce the differential problem to an algebraic one

$$L_x\Phi \cdot \mathbf{a} = \mathbf{f}\,, \qquad \text{i.e.} \qquad \sum_{i=1}^{N_c} (L_x\Phi)_{ji} a_i = f_j\,, \quad \forall j = 1,\dots,N_I\,, \tag{2.52}$$

where

$$(L_x\Phi)_{ji} = L_x\Phi_{ji} = L_x\phi(r_i(x_j))\,, \qquad \Phi_{ji} \coloneqq \phi(r_i(x_j))\,,$$

and where we have defined

$$\mathbf{a} = (a_1,\dots,a_{N_c})\,, \qquad \mathbf{f} = (f_1,\dots,f_{N_I})\,.$$

We have additional constraints to satisfy, i.e. the boundary conditions; by selecting the *boundary collocation points* $\{b_k\}_{k=1,\dots,N_d} \in \partial\Omega_D$, $\{v_p\}_{p=1,\dots,N_n} \in \partial\Omega_N$, we have two additional sets of constraints

$$\Phi\,\mathbf{a} = \mathbf{b}\,, \qquad \text{i.e.} \qquad \sum_{i=1}^{N_c} \Phi_{ki} a_i = b_k\,, \quad \forall k = 1,\dots,N_d\,, \tag{2.53}$$

$$(\nabla_n\Phi)\,\mathbf{a} = \mathbf{0}\,, \qquad \text{i.e.} \qquad \sum_{i=1}^{N_c} \nabla_n\Phi_{pi} a_i = 0\,, \quad \forall p = 1,\dots,N_n\,. \tag{2.54}$$

$$\tag{2.55}$$

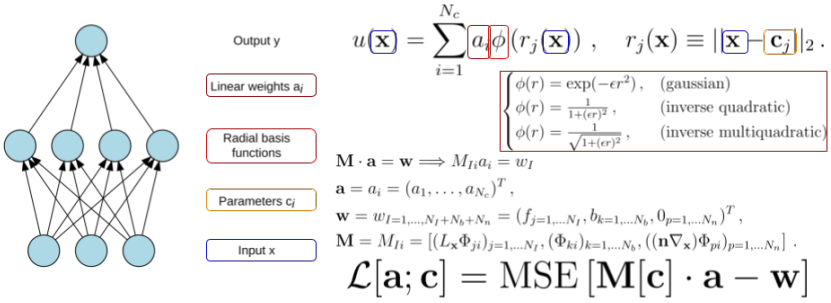


fig:2:4:Kansa1

Figure 2.11: Visual representation of the Kansa method.

Putting all together, we get a simple algebraic problem:

$$\mathbf{M}\cdot\mathbf{a} = \mathbf{w} \quad \Longrightarrow \sum_{i=1}^{N_c} M_{Ii}\, a_i = w_I\,, \quad \forall I = 1, \ldots, N_I + N_d + N_c \tag{2.56}$$

$$\mathbf{a} = a_i = (a_1, \ldots, a_{N_c})^T, \tag{2.57}$$

$$\mathbf{w} = w_{I=1,\ldots,N_I+N_b+N_n} = (f_{j=1,\ldots,N_I},\ b_{k=1,\ldots,N_d},\ 0_{p=1,\ldots,N_n})^T, \tag{2.58}$$

$$\mathbf{M} = M_{Ii} = [(L_x\Phi_{ji})_{j=1,\ldots,N_I},\ (\Phi_{ki})_{k=1,\ldots,N_d},\ (\nabla_n\Phi_{pi})_{p=1,\ldots,N_n}]\,. \tag{2.59}$$

So... End of story? Spoiler: NO. Finding M^{-1} is *hard*. Its well posedness is not a priori guaranteed. Nevertheless, we don't need to have an exact solvable system; we are interested in numerical viable solution. Something that minimises our error, or empirical risk, i.e. we can recast the problem to an *optimisation* problem

$$\mathbf{a}^* = \underset{\mathbf{a}\in\mathbb{R}^{N_c}}{\operatorname{argmin}}\, \mathcal{L}[a]\,, \qquad \mathcal{L}[a] \coloneqq \mathrm{MSE}[\mathbf{M}\cdot\mathbf{a} - \mathbf{w}]. \tag{2.60}$$

{eq:2:4:KansaLoss1}

We can also make the *centers* learnable parameters in $\mathbb{R}^d$,

$$\mathbf{a}^*, \mathbf{c}^* = \underset{\mathbf{a},\mathbf{c}}{\operatorname{argmin}}\, \mathcal{L}[a;c]\,, \qquad \mathcal{L}[a;c] \coloneqq \mathrm{MSE}[\mathbf{M}[\mathbf{c}]\cdot\mathbf{a} - \mathbf{w}]. \tag{2.61}$$

Having set up our problem as an optimisatin problem, we can employ our favourite optimisation scheme, e.g. *Stochastic Gradient Descent*, Newtonian methods, etc., and stop when a certain error threshold is surpassed.

Let us stop here to think for a moment: what Kansa method, in the essence, is

tbox:2:4:
kansa_method

Recipe 2.4.1: Kansa Method basic recipe

1. Find an *universal approximation* for any unknown function, here, the RBF.
2. Transform the Cauchy problem into an *optimisation problem*, defining a certain error function(al) over an alternative representation of the differential problem (i.e., the algebraised representation).
3. Find the solution solving the optimisation problem via our favourite optimisation algorithm, e.g., SGD.

PART II

Physics Informed Neural Networks and Neural Operators

CHAPTER 3

Lecture 3: Introduction to Physics Informed Neural Networks

chap:3

Let us restart with our beloved Cauchy problem

{eq:3:0:cauchy1}

$$\mathcal{F}[u(x), x, \gamma] = J(x)\,, \qquad x \in \Omega\,, \tag{3.1}$$

$$\mathcal{B}[u(x), x] = g(x)\,, \qquad x \in \partial\Omega\,, \tag{3.2}$$

where we have explicit a few ingredients:

- **The Geometry**: the domain Ω, its boundary $\partial\Omega$, and the behaviour of the field at the boundary;
- **The physics**: the source J, the dynamics $\mathcal{F}$, which may depend on a set of parameters γ.

Let us now introduce a bit of terminilogy here:

Definition 3.0.1 (Forward, inverse, parametric problems)**.** We say we are solving a **forward problem** when the geometry and the physics are fixed, i.e., we have explicit $(\Omega, \mathcal{F}, \gamma)$, and we need to find the solution $u(x)$. We say we are solving an **inverse problem** when we have, as an additional set of informations, a set of *observations* $\{(x_i, u_i^* \coloneqq u(x_i)\}_{i=1,\dots,N}$, but we have no explicit knowledge on (some of) the physical parameters γ. We say we are solving a *parametric problem* when we want to find a set of solution $\{\gamma, u_\gamma\}$ for the class of differential problems defined by $\mathcal{F}[\cdot, \cdot, \gamma]$.

3.1 Forward Problems: Vanilla PINNs

We now suppose the reader knows what a Deep Neural Network (DNN) is, in particular, without loss of generality, what a Multi-Layer Perceptron (MLP) is. Nevertheless, to set up and standardise the notation, we say that a Deep Neural Network is a representative of the space of the functions satisfying the hypothesis of one of the *Universal Approximation Theorems*.

Now, suppose we need to approximate an unknown function $f : x \in \Omega \subseteq \mathbb{R}^d \to \mathbb{R}^n$ with one of those universally approximating functions $\mathrm{DNN}_\theta : x \mapsto \mathrm{DNN}_\theta(x)$, where $\theta \in \Theta$ are parameters, such that

$$\mathrm{DNN}_\theta(x) \approx f(x).$$

Usually this is done by solving an optimisation problem; we define a cost function (or loss function)

$$\mathcal{L}[\theta] = \mathcal{L}[\mathrm{DNN}_\theta, f]\,,$$

such that we define our optimal approximator as the one described by the optimal parameter set θ^*

$$\theta^* = \operatorname*{argmin}_{\theta\in\Theta} \mathcal{L}[\theta]\,.$$

To achieve that, is commonly employed (a variation of) gradient descent, i.e. an iterative process where we continuosly update the parameter by a local exploration of the *loss landscape*

$$\theta \leftarrow \theta - \eta\nabla\mathcal{L}[\theta]\,.$$

We have seen that the loss function can be seen as a Monte Carlo approximation of the distance between the real function f, and the approximator DNN_θ.

This is usually facilitated by the fact that the loss $\mathcal{L}[\theta]$ is a smooth function, with an explicit form for its derivatives. Also, DNN_θ is itself a smooth function, arranged somehow in *layers*, i.e. repeated composition of elemental functions, and, using repeatedly the formula for the derivative of a composite function, its derivative in each point can be *analytically explicitly computed*.

For example, the standard MLP is a composition of *layers*, where each layer $\lambda_i : \mathbb{R}^{m_i} \to \mathbb{R}^{m_{i+1}}$ is the composition of a learnable *affine transformation* $x \mapsto Wx + b$ and and *activation function* $\phi : \mathbb{R} \to \mathbb{R}$, whose derivative is explicitly, analytically known (in the sense of its analytical expression, ϕ'):

$$\lambda_i(\theta_i; x) = \phi(W_i x + b_i), \qquad \theta_i \coloneqq (W_i, b), \quad W_i \in \mathcal{M}(\mathbb{R}^{m_i} \times \mathbb{R}^{m_{i+1}}),\ b_i \in \mathbb{R}^{m_i}\,,$$

so that [GBC16; Sca24] a MLP is

$$\begin{aligned}\mathrm{DNN}_\theta(x) &= \lambda_N(\theta_N) \circ \lambda_{N-1}(\theta_{N-1}) \circ \cdots \circ \lambda_1(\theta_1, x) \\ &= \lambda_N(\theta_N, \lambda_{N_1}(\theta_{N-1}, \cdots \lambda_1(\theta_1, x) \cdots)),\end{aligned} \tag{3.3}$$

where $\theta = (\theta_1, \ldots, \theta_N)$. Since we can compute

$$\nabla_{\theta_i}\lambda_i(\theta_i; x) \coloneqq \begin{cases} \nabla_{W_i}\lambda_i(\theta_i; x) &= \phi'(W_i x + b_i)\cdot x_i^T \\ \nabla_{b_i}\lambda_i(\theta_i; x) &= \phi'(W_i x + b_i), \end{cases}$$

where we have used the rule of the derivative of a composed function, we can iterate over the MLP layers and easily compute the generic derivative $\nabla_{W_i,b_i}\mathrm{DNN}_\theta$, in a *closed, analytical* form.

But this means that we can compute also $\nabla_x \mathrm{DNN}_\theta(x)$ in a *closed, analytical* form, i.e., the derivative with respect to the input x. This is quite known and standard in the literature; for example **contractive autoencoders** are denoising autoencoders with a penalisation term to the loss to reduce the variation under small variations of the input, which is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function $h(x)$, i.e. $\|\nabla_x h(x)\|_2$ [GBC16].

So this means that (almost) *any* differential operator $\mathcal{F}$ acting on a DNN can be represented in a *closed, analytical* form! so that, we can define a risk

functional for our Cauchy problem in Equation (3.1) where we are approximating the solution $u(x) \sim \text{DNN}_\theta(x)$, simply by

$$\begin{aligned}\mathcal{L}[\theta] &= \|\mathcal{F}[\text{DNN}_\theta(x), x, \gamma] - J(x)\|_2 + \lambda\|\mathcal{B}[\text{DNN}_\theta(x), x] - g(x)\|_2 \\ &= \int_\Omega |\mathcal{F}[\text{DNN}_\theta(x), x, \gamma] - J(x)|^2 \text{d}\omega(x) \\ &\quad + \lambda \int_{\partial\Omega} |\mathcal{B}[\text{DNN}_\theta(x), x] - g(x)|^2 \text{d}\Sigma(x) \\ &\coloneqq \mathcal{L}_{PDE}[\theta] + \lambda\mathcal{L}_{BC}[\theta]\end{aligned} \tag{3.4}$$

{eq:3:1: vanillapinnloss}

which, if we sample N random points $\{x_i\} \in \Omega$, and M random points in $z_j \in \partial\Omega$, can be computed via Monte Carlo approximation

$$\mathcal{L}[\theta] \approx \frac{1}{N}\sum_{i=1}^{N} |\mathcal{F}[\text{DNN}_\theta(x_i), x_i, \gamma] - J(x_i)|^2 + \frac{\lambda}{M}\sum_{i=1}^{M} |\mathcal{B}[\text{DNN}_\theta(x), x] - g(x)|^2 . \tag{3.5}$$

This loss definition transforms our differential problem into an optimisation problem, i.e. the solution $u(x)$ of our differential problem Equation (3.1) is approximated

$$\begin{aligned}&u(x) \sim \text{DNN}_{\theta^*}(x)\,, \\ &\theta^* = \underset{\theta\in\Theta}{\text{argmin}} \Big[\text{MSE}(\mathcal{F}[\text{DNN}_\theta(x_i), x_i, \gamma] - J(x_i), 0) \\ &\qquad\qquad + \lambda\,\text{MSE}(\mathcal{B}[\text{DNN}_\theta(x), x] - g(x), 0)\Big].\end{aligned}$$

The optimisation problem can now solved using standard algorithms, like stochastic gradient descent; furthermore, (more or less) all the standard libraries used for training neural networks can be used, with very little fuzz, to solve this optimisation problem.

This framework for solving differential problem using deep neural networks is called Physics Informed Machine Learning; a DNN trained to solve a differential problem, is called (Vanilla) **Physics Informed Neural Network** (PINN).

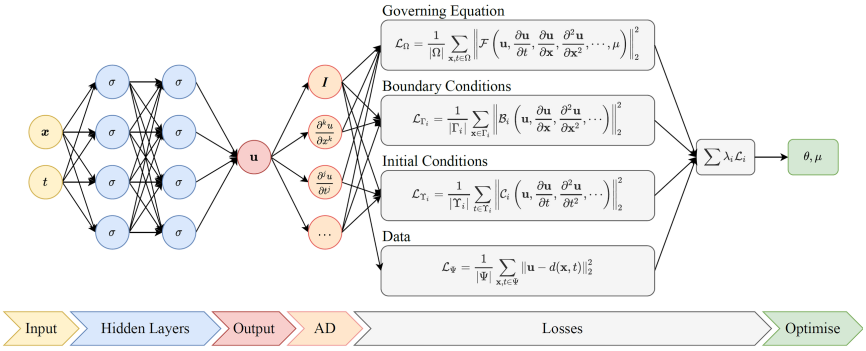
We can summarise the basic recipe of a PINN to solve the differential problem Equation (3.1):

Recipe 3.1.1: Vanilla PINN basic recipe

tbox:3:1: vanilla_pinn

1. Define a *geometry sampler*, i.e. a method to draw $x_i \in \Omega$.
2. Define a Deep Neural Network to use as a function approximator (using your favourite framework, like PyTorch, TensorFlow, Jax, etc.), $\hat{u}_\theta(x) \leftarrow \text{DNN}_\theta(x)$.
3. Define the action of the differential operators on the DNN (e.g., using PyTorch autograd library), $\mathcal{F}[\hat{u}_\theta, x, \gamma], \mathcal{B}[\hat{u}_\theta(x), x]$.
4. Choose an optimistion algorithm to solve the optimisation problem over the cost function defined in Equation (3.4).

A visual representation of a PINN is reported in Figure 3.1.



Note: θ: weights/biases of network; μ: unknown PDE parameters; λ_i: scaling factor for loss term i

Figure 3.1: Visual representation of a PINN. From here.

fig:3:1: PINN_visual_representation

3.1.1 An historical note

Before going on, let us make a brief historical note. One may wonder why *such a simple* approach emerged only as recently as 2017s.

Indeed, the terminology *Physics Informed Deep Neural Network* first appearance traces back to 2017's series of papers by Maziar Raissi, Paris Perdikaris, George Em Karniadakis [RPK17a], [RPK17b], [RPK19], but the concept traces way back: In 1990, in [LK90], Lee and Kang present a paper titled *Neural algorithm for solving differential equations*, where they minimise FDM implicit relations by recasrting the problem to an optimisation problem.

In 1994, [DP94], Dissanayake and Phan-Thien presented a paper with an abstract saying:

«*A numerical method, based on neural-network-based functions, for solving partial differential equations is reported in the paper. Using a 'universal approximator' based on a neural network and point collocation, the numerical problem of solving the partial differential equation is transformed to an unconstrained minimization problem.*»

In 1998, [LLF98], Lagaris, Likas, Fotiadis, wrote in the abstract:

«*We present a method to solve initial and boundary value problems using artificial neural networks. [...] This part involves a feedforward neural network containing adjustable parameters (the weights).*»

Similarly, in 2001 Aarts & Van Der Veer wrote in the abstract of paper [AV01]:

«*A method is presented to solve partial differential equations (pde's) and its boundary and/or initial conditions by using neural networks. It uses the fact that multiple input, single output, single hidden layer feedforward networks with a linear output layer with no bias are capable of arbitrarily well approximating arbitrary functions and its derivatives.*»

In 2011 appeared even a survey on such methods [KY11]!

As usual, the success obtained by PINNs was that they were carefully branded to repack, reformulate, modernise, optimise, streamline, and simplify already known methods, but uplifting them to be truly powerful.

This is somewhat similar to what happened with transformers and the attention mechanism (for a nice historical review on Attention Mechanism, see [Soy22]). The 2017 Transformer paper [Vas+17] triggered a revival because it reframed attention from an auxiliary component into the core computational engine of a model. Indeed, inceptions of similar ideas were as old as the early '90s, with the work on *fast weights*, where a secondary mechanism dynamically modulates connections [Sch92]. Earlier attention mechanisms (e.g.,[BCB14]) were attached to recurrent or convolutional networks; the Transformer removed recurrence and convolution entirely, showing that self-attention alone could model long-range dependencies more efficiently and with superior performance. The paper's clarity, empirical success, and architectural simplicity made attention not just a useful mechanism but the foundational primitive of contemporary deep learning.

So, the early attempts to solve differential equations with neural networks, such as the works of [AV01; LLF98], introduced the essential idea that neural networks could approximate PDE solutions by embedding boundary conditions and minimizing physics-related residuals. However, these methods remained isolated techniques, limited by the computational tools and modelling paradigms of their time. The 2017 introduction of Physics-Informed Neural Networks (PINNs) by Raissi, Perdikaris, and Karniadakis played a role analogous to the paper "Attention Is All You Need" in the attention literature: it reframed a scattered set of earlier ideas into a unified, scalable, and conceptually clean framework. PINNs demonstrated how automatic differentiation, modern optimization, and deep architectures could be systematically combined to enforce physical laws as soft constraints, transforming a niche methodology into a general paradigm for scientific machine learning. This synthesis, together with compelling demonstrations on forward, inverse, and data-assimilation problems, catalysed the widespread adoption and rapid growth of PINNs in a way the earlier works could not.

3.1.2 A brief comment on vanilla PINNs

Let us stop here for a moment, again. Let us look again, closely, to the basic recipe of the Vanilla PINN Section 3.1. The process we have described is essentially organised in three macro-blocks:

1. A Representation Model (e.g., the DNN)
2. The Governing Equations (the Cauchy problem)
3. An Optimisation process (e.g., the SGD)

At this moment, there is **no** data incorporation. Thus, strictly speaking, is **not** machine learning. It is more similar to a *meshless, numerical solver method*.

We have already encountered a *meshless, numerical solver method*, the Kansa Method Section 2.4.1. We have seen that we can use a radial basis function network to solve a PDE problem, by minimising the loss using the Stochastic Gradient Descent (or any of its variant).

In Figure 3.3 we report a visual comparison of Kansa method Section 2.4.1 and vanilla PINN method Section 3.1. Let us observe them side-by-side, from the perspective ot the three main ingredients above:

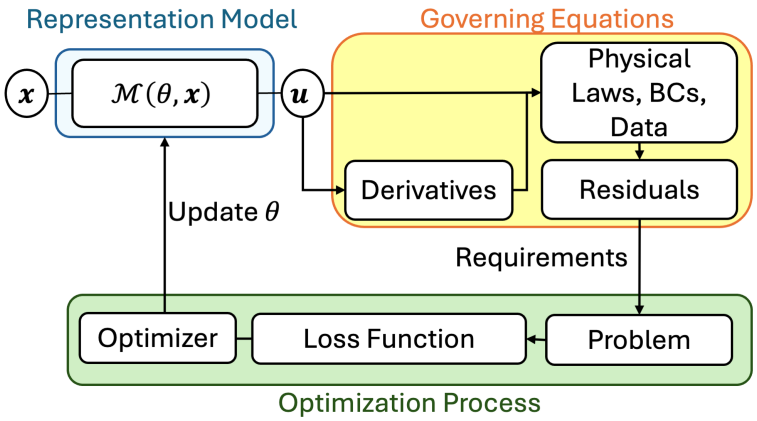


Figure 3.2: Visual representation of the PIML training components. A representation model provides an approximation $\hat{u}$ of the ODE/PDE solution u. The mismatch (residuals) between the approximated solution and the known components of the true solution, such as physical laws and boundary conditions, is calculated. These governing equations define a set of requirements that generate an optimization problem. This problem is reformulated as a loss function and then sent to an optimizer that updates the model parameters. From [Tos+25].

fig:3:1: PIML_overview

1. **Representation Model**:

Kansa: It uses the Radial Basis Function (RBF) as a representation of the solution, $\hat{u}_\theta(x) \sim \text{RBF}_\theta(x)$;

RBFs are local or semi-global kernels with explicit smoothness.

PINN: It uses a Multi-Layer Perceptron (MLP) as a representation of the solution, $\hat{u}_\theta(x) \sim \text{DNN}_\theta(x)$;

MLPs are global, compositional, adaptive basis functions. This helps explain why PINNs scale differently.

2. **Governing Equations**:

Kansa: Classically, it leads to a dense linear system obtained by enforcing the PDE at collocation points. In modern reinterpretations, one may also minimise the residual via gradient-based optimisation;

PINN: It computes residuals on a Monte Carlo sampling of the domain;

3. **Optimisation Process**:

Kansa: At the vanilla level, it computes a linearised problem; more realistically, it computes SGD on the linear loss Equation (2.60).

PINN: It uses SGD on the *multi-objective* loss Equation (3.4).

To summarise, the structural similarity between the Kansa method and PINNs becomes evident when viewed through the lens of representation, physics enforcement, and optimisation. Both approaches construct a global trial function (RBF expansions in the Kansa method, and neural networks in PINNs) and enforce the governing equations at a set of collocation points. In the classical Kansa formulation, this yields a dense linear system derived from analytic

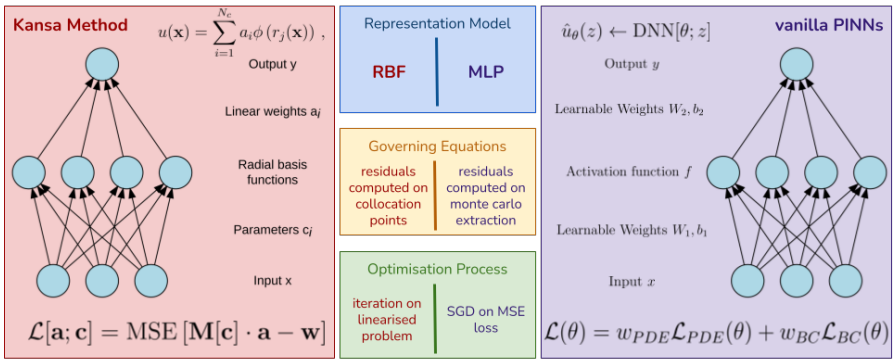


Figure 3.3: Visual comparison of the Kansa Method and the PINN method. The colour code of the centred box follows the one of Figure 3.2.

fig:3:1: KansaVsPINN

derivatives of the RBFs, whereas PINNs rely on automatic differentiation and nonlinear optimisation to minimise a multi-objective residual loss. From this perspective, a vanilla PINN without data is not a machine-learning model but a modern meshless collocation method, differing from Kansa primarily in the choice of basis functions, the use of stochastic sampling, and the optimisation machinery (up to a certain point, as we have seen). This connection places vanilla PINNs within a long lineage of meshfree numerical solvers, rather than outside it.

3.1.3 A *key* comment on vanilla PINNs: strenghts... and limits

Before going on, let's pause on Vanilla PINN for a moment.

It seems that we have found the *Holy Grail* of computational physics. We simply have to define a DNN, and compute the PDE (and BC/IC), and use it as Loss. *That's it.*

Obviously, if life has a rule, it is:

« *There is no such thing as a free lunch* [Hei66].»

For example, in [MA23], authors have shown that

- (vanilla) PINNs are known to suffer from failures in some classes of problems.
- Literature provides many solutions to deal with these failures in specific contexts.
- The effectiveness of these strategies in different contexts *is not always guaranteed*.

The main takeaway lessons are (in the author's opinion):

1. (vanilla) PINNs are an *additional* tool to tackle the issue of solving differential problems.
2. (vanilla) PINNs have difficulties in performing the task (as the normal solver do, but in a different, novel way).

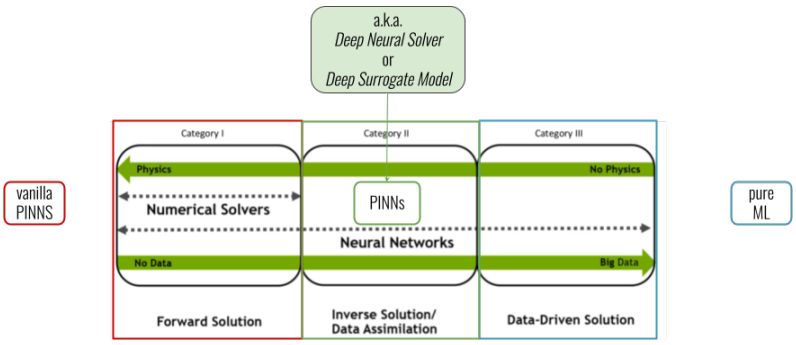


Figure 3.4: Visual representation of the Physics Informed Machine Learning landscape. Adapted from the NVIDIA Physics-Informed NeMo SYM documentation, which categorizes scientific machine learning approaches by their reliance on physics and data [Nvi26].

fig:4:1: PINN_landscape

  - This means that we need to address such difficulties in a *different way* w.r.t. standard numerical solver;
  - But we can tackle (most of) them using insights from *Deep learning approaches*.

3. PINNS works better when they are additionally *data-informed*.

In Figure 3.4, we present a schematic overview of the landscape of machine learning applications to physics-based systems, particularly differential problems, whether well-posed or not. Vanilla PINNs occupy the same corner as traditional numerical solvers: they rely solely on the governing equations, without incorporating observational data. In this regime, PINNs function as meshless solvers, leveraging neural networks as expressive trial spaces.

The term *expressive power* refers to the ability of neural networks to approximate complex, high-dimensional functions with intricate structure. This capacity stems from their compositional architecture and nonlinearity, which allow them to represent solution manifolds that would be intractable with classical basis functions.

What distinguishes PINNs, however, is their capacity to transcend this regime. Neural networks offer remarkable expressive power and an inherent ability to model complex, high-dimensional data. This makes them particularly well-suited for tasks involving *data assimilation* or *inverse problems*, where observational data must be integrated into the solution process. In these contexts, PINNs (no longer vanilla) can outperform classical methods by combining physical constraints with data-driven inference. Their ability to extract latent structure from data while respecting physical laws positions them as a versatile and powerful tool within the broader PIML landscape.

3.2 Inverse & Parametric Problems

3.2.1 PINNs for inverse problems

Let us restart from Equation (3.1),

$$\begin{aligned} \mathcal{F}[u(x), x, \gamma] &= J(x)\,, \qquad x \in \Omega\,, \\ \mathcal{B}[u(x), x] &= g(x)\,, \qquad x \in \partial\Omega\,, \end{aligned}$$

and suppose for now that the parameter(s) γ are *unknown*, but we have access to a set of *observations*, i.e., a dataset $\{x_i, u_i^*\}_{i=1,\dots,N_{\text{data}}}$. This setting defines

a prototypical **inverse problem**, where the goal is to infer hidden parameters, coefficients, or source terms from partial observations of the solution field. More formally,

Definition 3.2.1 (Inverse Problem)**.** Given a physical system described by a differential operator $\mathcal{F}[u(x), x, \gamma] = J(x)$, an inverse problem seeks to infer unknown parameters γ, source terms, or boundary conditions from partial observations of the solution $u(x)$, typically under constraints imposed by the governing equations. Unlike forward problems, which compute $u(x)$ from known γ, inverse problems are often ill-posed and require regularization or prior knowledge to ensure stability and uniqueness.

Inverse problems are notoriously challenging because they often violate the classical notion of *well-posedness*, introduced by Jacques Hadamard in the early 20th century.

Definition 3.2.2 (Well-posedness)**.** A problem is said to be *well-posed* if it satisfies three conditions [Had02]:

3.2.2.1. **Existence**: a solution u to the problem exists;

3.2.2.2. **Uniqueness**: the solution is unique;

3.2.2.3. **Stability**: the solution depends continuously on the data (small perturbations in the input produce small perturbations in the output).

Inverse problems frequently violate one or more of these criteria. For instance, multiple parameter configurations may explain the same observations (non-uniqueness), or small measurement noise may lead to large deviations in the inferred parameters (instability). This intrinsic ill-posedness motivates the use of regularization strategies, prior knowledge, or physics-based constraints, such as those embedded in PINNs, to obtain meaningful and stable solutions.

Physics-Informed Neural Networks (PINNs) offer a natural framework for tackling such problems. By embedding the governing equations into the loss function and simultaneously fitting the observed data, PINNs enable the recovery of unknown physical quantities while respecting the underlying physics. This dual enforcement, data fidelity and physical consistency, makes PINNs particularly attractive for inverse problems, especially in regimes where traditional solvers struggle due to ill-posedness, sparsity, or noise.

Inverse problems addressed with PINNs span a wide range of applications:

- **Petroleum engineering**: monitoring multiphase flow in reservoirs [Cal+20; Cal+23];
- **Quantum mechanics**: solving eigenvalue problems with unknown potentials [JMP22]; reconstruction of unknown potentials and eigenvalues [JMP22].
- **Groundwater hydrology**: estimation of hydraulic conductivity fields in unsaturated flow models [Dep+22].
- **Structural mechanics**: identification of stiffness parameters in beam and rail systems [Kap+23].

- **Nano-optics and metamaterials**: recovery of effective permittivity and material parameters from scattering measurements [Che+20].
- **Fluid dynamics**: parameter estimation in Burgers, Euler, and Navier–Stokes equations [Jag+22; RPK17b].
- **Epidemiology**: inference of transmission rates and latent variables in compartmental models [Pat+20].
- **Battery modelling**: estimation of reaction kinetics and diffusion coefficients in electrochemical PDEs [Wan+24].

A living repository of PINN-based inverse problem examples is maintained at [Zha22], showcasing implementations across domains such as harmonic oscillators, reaction-diffusion systems, and parameter estimation in PDEs.

These examples illustrate the flexibility of PINNs in inverse modeling: the neural network acts as a surrogate solution model, while the unknown parameters are treated as trainable variables. The optimisation process seeks to minimize both the PDE residuals and the mismatch with observed data, yielding physically consistent reconstructions even in challenging settings.

tbox:3:1: inverse_pinn

Recipe 3.2.1: Inverse Problem PINN basic recipe

1. Define a *geometry sampler*, i.e. a method to draw $x_i \in \Omega$.
2. Define a Deep Neural Network to use as a function approximator **& parameter estimator** (using your favourite framework, like PyTorch, TensorFlow, Jax, etc.), $\hat{u}_\theta(x), \gamma \leftarrow \mathrm{DNN}_\theta(x)$.
3. Define the action of the differential operators on the DNN (e.g., using PyTorch autograd library), $\mathcal{F}[\hat{u}_\theta, x, \gamma], \mathcal{B}[\hat{u}_\theta(x), x]$.
4. Choose an optimistion algorithm to solve the optimisation problem over the cost function defined in Equation (3.4), plus an additiona *data loss* term:

$$\mathcal{L}_{data}[\theta] = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} |\hat{u}_\theta(x_i) - u_i^*|^2 \,. \tag{3.6}$$

A very simple implementation of a PINN for inverse problem is simply to add a learnable parameter to the deep neural network modules (e.g., in PyTorch, using NN.PARAMETER).

A very simple example: PINNs for Burgers equation A simple example may be the tackling of the Burgers equation; a very nice set-up is the following Cauchy problem

$$\partial_t u(t,x) + \lambda u(t,x)\partial_x u(t,x) - \nu^2 \partial_x^2 u(t,x) = 0\,, \qquad (t,x) \in [0,1] \times [-1,+1], \tag{3.7}$$

$$u(t=0,x) = -\sin(\pi x)\,, \qquad x \in [-1,+1], \tag{3.8}$$

$$u(t, x = \pm 1) = 0\,, \qquad t \in [0, 1]. \tag{3.9}$$

which is the generalisation of the one reported in Equation (2.21). We have already seen that the case where $\lambda = 1, \nu = 0.01/\pi$ is a nice set up (see Figure 2.5); furthermore, from there, we can have a *dataset*, obtained from the FDM method, which we can employ to mimick an inverse problem:

Problem: Given the data $\{(t_n, x_i), u^*_{ni} := u(t_n, x_i)\}_{n=1,\dots,N_t,\, i=1,\dots N_x}$ (obtained from the FDM method), define a PINN to solve the inverse problem.

To solve it, we simply need a basic DNN, a learnable parameter, and a method to compute the PDE operator on the DNN via automatic differentiation methods, see Listing 3.1.

lst:3:2:Burgers_PINN_inverse

Listing 3.1: PINN sketch for Burgers inverse problem

```python
import torch
from torch import nn
from .utils import MLP # standard MLP implementation

class BurgersInverse(nn.Module):
    def __init__(self):
        # Network
        self.dnn = MLP(
            n_inputs  = 2, # (t,x)
            n_outputs = 1, # (u)
            hidden_dims=list([32, 32, 16]),
            activation_func=nn.Tanh,
        )
        # learnable parameter
        self.lambda = nn.Parameter(torch.tensro([0.8]), requires_grad=
            True) # requires_grad=True is Default
        self.nu = nn.Parameter(torch.tensro([0.005]), requires_grad=
            True)

    def forward(self, coords: torch.Tensor):
        # in_coordsvars of shape [N_points, 2],
        # with t=coords[0], x=coords[1]
        u_pred = self.dnn(x)
        return u_pred, self.lambda, self.nu

# compute PDE
def compute_burgers_pde(
    coords: torch.Tensor,
    u: torch.Tensor,
    lambda: torch.Tensor,
    nu: torch.Tensor
) -> torch.Tensor:
    """
    coords: (N, 2) tensor with columns [t, x]
    u:      (N, 1) predicted solution u(t,x)
    """
    # First derivatives wrt (t, x)
    grads = torch.autograd.grad(
        u, coords,
```

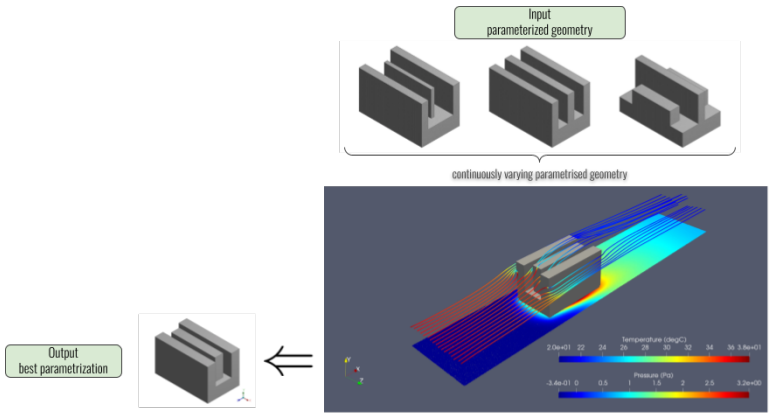


Figure 3.5: Visual example of parametrised geometries, and its connected optimisation task.

fig:3:2: ParametricPINN_ geom

```
        grad_outputs=torch.ones_like(u),
        create_graph=True
    )[0]

    u_t = grads[:, 0:1]   # shape (N,1)
    u_x = grads[:, 1:2]

    # Second derivative wrt x
    grads2 = torch.autograd.grad(
        u_x, coords,
        grad_outputs=torch.ones_like(u_x),
        create_graph=True
    )[0]

    u_xx = grads2[:, 1:2]

    #
    u2_x = torch.autograd.grad(
        u.pow(2), coords,
        grad_outputs=torch.ones_like(u),
        create_graph=True
    )[0][:, 1:2] / 2

    # PDE residual
    return u_t + lambda.unsqueeze(0) * u2_x - (nu.unsqueeze(0).pow(2))
         * u_xx
```

Notice that in Listing 3.1 we have used the *conservative* form of the Burgers equation, where we have simply used that

$$u\partial_x u = \frac{1}{2}\,\partial_x(u^2)\,.$$

3.2.2 PINNs for parametric problems

In many practical scenarios, solving a single instance of a PDE is not sufficient. What we often require is a *family* of solutions, smoothly indexed by one or more parameters γ. This situation arises naturally in design, control, uncertainty quantification, and inverse modelling. Similarly to the inverse setting, and in close analogy with the vanilla PINN formulation, we can extend the basic PINN recipe so that the neural network does not approximate a single function $u(t, x)$, but rather a *solution generator*

$$(t, x, \gamma) \;\mapsto\; u_\gamma(t, x),$$

capable of producing the solution for any admissible value of the parameter γ.

This parametric viewpoint is especially valuable when the dependence on γ cannot be conveniently embedded into the loss function or optimised through gradient descent—for example, when γ is discrete, non-differentiable, or when the downstream objective is itself non-smooth. By learning a surrogate model that spans the entire parameter space, we gain the ability to explore, optimise, or control the system without repeatedly solving the PDE from scratch.

Although in Equation (3.1) the parameter γ appears explicitly only inside the differential operator $\mathcal{F}$, in practice parametric problems arise in several distinct forms. It is therefore useful to classify them according to where the parameter enters the Cauchy problem:

A. **Parametric geometries**: The parameter modifies the *domain* itself (e.g., shape optimisation, moving boundaries, geometry-dependent coefficients).

B. **Parametric PDEs**: The parameter indexes a *family of operators* or physical regimes (e.g., varying diffusivity, Reynolds number, material properties).

C. **Parametric BC/IC**: The parameter controls the *boundary or initial conditions*, effectively generating a family of admissible solutions.

Each of these cases requires slightly different architectural choices and sampling strategies, but the overarching idea remains the same: we embed γ directly into the neural network input, and train a single model that captures the entire solution manifold. Parametric PINNs thus provide a unified framework for learning high-dimensional, continuous families of PDE solutions in a single training phase, enabling fast evaluation and flexible downstream optimisation.

An example of parametric geometry In Figure 3.5 we show a typical *design optimisation problem*, taken from [Nvi26], namely the *Parametrised 3D Heat Sink* example[1]. The key idea is that, once the training is complete, the network can be evaluated for many different values of the geometric parameter γ as a post-processing step, without solving the forward problem again. The parametrisation increases the computational cost only marginally, while covering the entire design space in a single training phase. This gives PINNs a significant advantage over standard solvers for parametric problems.

[1] See https://docs.nvidia.com/physicsnemo/latest/physicsnemo-sym/user_guide/advanced/parametrized_simulations.html.

In this example the goal is to solve a thermal diffusion problem for a family of heat sink geometries. The parameters describe the fin dimensions (thickness, length, and height), so that the design space is generated by varying the height h, length ℓ, and thickness t of the central fin and the two side fins (see again Figure 3.5). A parametric PINN learns the mapping

$$(\mathbf{x}, \gamma) \longmapsto T_\gamma(\mathbf{x}) \,,$$

where γ encodes the geometric configuration.

Once the model is trained, it becomes possible to perform design optimisation directly at inference time. A typical optimisation problem specifies an objective function that is minimised or maximised under physical or engineering constraints. For heat sink design, a common constraint is the maximum allowable temperature at the source chip, which is dictated by the chip's operating limits. Another constraint is the maximum pressure drop that the cooling system can sustain while driving the flow around the heat sink. Parametric PINNs allow the entire optimisation loop to be carried out efficiently, since evaluating new geometries does not require solving the PDE again.

An example of a parametric PDE: stationary wave propagation and the Helmholtz equation A simple yet representative example of a parametric PDE is the Helmholtz equation, which describes stationary wave propagation. In this setting the wave-number k plays the role of a free parameter:

$$\nabla^2 f(\mathbf{x}) + k^2 f(\mathbf{x}) = 0 \,.$$

For each admissible value of k we obtain a different PDE, and the goal of a parametric PINN is to learn the mapping

$$(\mathbf{x}, k) \longmapsto f_k(\mathbf{x}) \,,$$

that is, a generator of solutions for the entire family of Helmholtz problems.

The Helmholtz equation arises naturally from the standard wave equation

$$\Box E(t, \mathbf{x}) \coloneqq \partial_t^2 E(t, \mathbf{x}) - c^2 \nabla^2 E(t, \mathbf{x}) = 0 \,,$$

which models, for example, the propagation of the electromagnetic field in a waveguide. In the stationary regime we look for solutions of the form

$$E(t, \mathbf{x}) = f(\mathbf{x})\, \mathrm{e}^{\mathrm{i}\omega t} \,, \qquad k^2 \coloneqq \frac{\omega^2}{c^2} \,,$$

and by substituting this ansatz into the wave equation we obtain the Helmholtz equation for the spatial profile $f(\mathbf{x})$. Different values of the frequency ω correspond to different values of the parameter k, which makes this a natural example of a parametric PDE.

An example of parametric BC/IC: the vibrating string with a loose end A natural example of a parametric boundary condition arises from the vibrating string model discussed in Section 1.2.2. In that setting the string satisfies the 1+1-dimensional wave equation, and the behaviour at the right endpoint is controlled by a Robin condition of the form

$$u(t, 1) + \alpha\, \partial_x u(t, 1) = 0 \,,$$

where the parameter α describes how tightly the string is held by the peg. The case $\alpha = 0$ corresponds to a perfectly fixed end, while increasing α models a progressively looser constraint. For each value of α the PDE has a different family of eigenvalues $\{\lambda_n(\alpha)\}$, and therefore a different vibration spectrum.

This can be recast as a typical *parametric BC* problem: the differential operator is fixed, but the admissible solutions depend on a parameter that enters through the boundary condition. A parametric PINN can be trained to learn the mapping

We could actually treat also the wave velocity as a parameter.

$$(t, x, \alpha) \longmapsto u_\alpha(t, x)\,,$$

so that, once training is complete, the model can predict the vibration pattern of the string for any value of the looseness parameter α without solving the PDE again.

A similar situation occurs for *parametric initial conditions*. In the same example the pluck profile $f(x)$ determines the coefficients of the modal expansion. If we regard f as a parameter, the task becomes learning the operator

$$f \longmapsto u_f(t, x)\,,$$

that is, the solution corresponding to a given initial displacement. Parametric PINNs provide a unified framework for both types of dependence, since the parameters (either α or a representation of f) can be included directly in the network input.

Connection with Neural Operators. Parametric PINNs naturally point toward a broader and more rigorous framework, namely the theory of Neural Operators. While a parametric PINN learns a map $(x, t, \gamma) \mapsto u_\gamma(t, x)$, a Neural Operator is designed to learn a map between infinite-dimensional function spaces, for example from a coefficient field or forcing term to the corresponding solution of a PDE. We will discuss Neural Operators in Chapter 5. In this sense, Neural Operators provide a mathematically principled generalisation of the parametric PINN idea, since they treat the parameter not as a finite-dimensional vector but as an entire input function. This viewpoint clarifies the role of parametric dependence and offers a systematic way to learn solution operators for families of PDEs. Recent work explicitly analyses the relationship between PINNs and Neural Operators in the context of parametric PDEs, showing how both approaches can be interpreted as operator-learning strategies with different inductive biases and training regimes.

For a detailed discussion of this connection, see for example the analysis in [Zha+25b].

3.3 A brief introduction to the Learning Theory of PINNs

sec:3:3: Learning_Theory

Having introduced the basic formulations of PINNs, we now examine their theoretical foundations, focusing on the sources of error and the dynamics of training.

The next section is based on [Cuo+22; DLM21; DM22; KKB26; Kut22; Tos+25].

Physics-Informed Neural Networks (PINNs) are trained by minimizing a loss function that enforces the differential equation, the boundary conditions, and any additional physical constraints. From a learning-theoretic perspective, this raises a natural question: under which conditions does a PINN converge

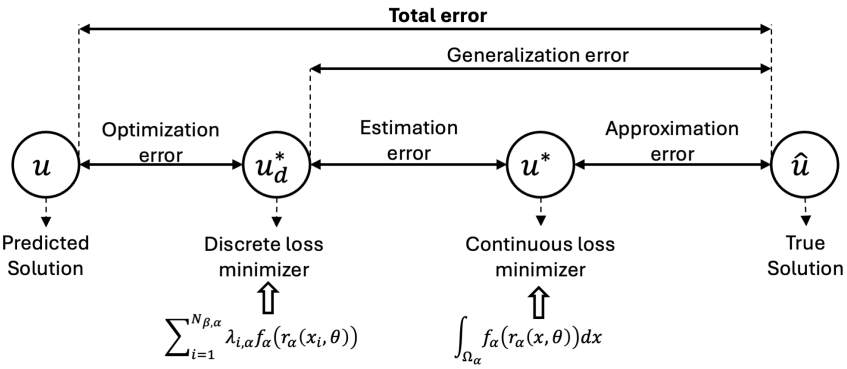


Figure 3.6: Decomposition of total error. The total error in a PIML model can be decomposed into three parts: (1) an optimization error resulting from the optimizer's inability to fully minimize the loss function, (2) an estimation or quadrature error due to discretizing the loss function over a finite number of points, and (3) an approximation error, which reflects the expressivity or capacity of the representation model to capture the true solution. From [Tos+25]

fig:3: 3Errors_PIML

to the true solution of the Cauchy problem? In other words, when does the trained predictor u_θ approximate the exact solution u of the PDE?

This question is subtle, because PINNs do not follow the classical supervised learning paradigm. In standard machine learning the loss is defined on a finite dataset, and the goal is to generalize from the training samples to unseen data. In PINNs the loss is defined by the physics itself, through the residual of the PDE and the boundary or initial conditions. The training points are not given by a dataset, but are chosen by the practitioner, and the loss approximates a continuum functional through numerical quadrature.

For this reason, the learning theory of PINNs must account for three distinct sources of error (for a visual representation, see Figure 3.6):

- the *approximation error*, which measures the expressive power of the neural network class;
- the *estimation error*, which arises from discretizing the continuous loss functional over a finite set of collocation points;
- the *optimization error*, which reflects the fact that the PINN loss is non convex and the optimizer may not reach a global minimizer.

These three contributions combine to determine the total error between the trained model $u_{\hat{\theta}}$ and the true solution u. Understanding their role is essential for explaining both the successes and the limitations of PINNs, and provides a principled framework for improving architectures, sampling strategies, and optimization schemes.

3.3.1 The PINN Learning Problem as Risk Minimization

The training objective of a Physics-Informed Neural Network can be interpreted as the minimization of a suitable risk functional. Given a PDE of the form

$$\mathcal{F}[u(x), x] = J(x) \qquad x \in \Omega,$$

together with boundary or initial conditions, the PINN loss is constructed by evaluating the residual of the PDE and the constraints at a set of collocation points. This leads to a discrete loss of the form

$$\hat{\mathcal{R}}[u_\theta] = \sum_{\alpha} \sum_{i=1}^{N_\alpha} \lambda_{i,\alpha}\, f_\alpha\big(\tau_\alpha(x_i, \theta)\big),$$

where the index α runs over the different components of the loss (PDE residual, boundary conditions, initial conditions, and so on), the points x_i are the collocation nodes, and the functions f_α encode the contribution of each constraint.

From a continuum perspective, the ideal objective is the minimization of the continuous risk

$$\mathcal{R}[u_\theta] = \sum_{\alpha} \int_{\Omega_\alpha} f_\alpha\big(\tau_\alpha(x, \theta)\big)\, dx,$$

which measures how well the neural network satisfies the PDE and the associated constraints over the entire domain. The discrete loss $\hat{\mathcal{R}}$ is therefore a numerical quadrature approximation of the continuous functional $\mathcal{R}$.

This viewpoint highlights an important difference between PINNs and classical supervised learning. In standard machine learning the empirical risk is defined on a fixed dataset, and generalization refers to performance on *unseen data* drawn from the same distribution. In PINNs the training points are chosen by the practitioner, and the empirical loss approximates a continuum integral. The discrepancy between $\mathcal{R}$ and $\hat{\mathcal{R}}$ is therefore more related to a quadrature error rather than a statistical generalization error.

The minimization of $\hat{\mathcal{R}}$ is performed by gradient-based optimization, which produces a trained parameter vector $\hat{\theta}$. The quality of the resulting predictor $u_{\hat{\theta}}$ depends on three factors: the expressivity of the neural network class, the accuracy of the quadrature approximation, and the ability of the optimizer to reach a good minimum of the discrete loss. These three aspects form the basis of the error decomposition discussed in the following subsections.

3.3.2 Approximation Error

The first source of error in a PINN is the *approximation error*, which measures the expressive power of the neural network class used to represent the solution. Given the continuous risk functional $\mathcal{R}$, the approximation error is defined as

$$\mathcal{E}_A = \inf_{\theta \in \Theta} \mathcal{R}[u_\theta], \tag{3.10}$$

that is, the smallest possible value of the loss over all neural networks in the chosen architecture. This quantity reflects how well the network class can represent functions that satisfy the PDE and the associated constraints.

Approximation theory for PINNs relies on classical results for neural networks in Sobolev spaces. A key result [DLM21; DM22], shows that shallow networks with tanh activation can approximate functions and their derivatives with controlled accuracy. More precisely, for a target function $u \in W^{s,\infty}([0,1]^d)$, a two-layer network with N neurons satisfies the bound

$$\|u_{\theta_N} - u\|_{W^{k,\infty}} \leq c_1 \frac{\log(c_2 N)^k}{N^{s-k}}, \qquad k = 0, \ldots, s-1,$$

for suitable constants c_1 and c_2. This estimate shows that neural networks can approximate smooth functions with algebraic convergence rates, and that the approximation improves as the width increases.

In the context of PINNs, the approximation error quantifies the ability of the network to represent a function that satisfies the PDE residual and the boundary or initial conditions. Since the loss involves derivatives of u_θ, the relevant function space is typically a Sobolev space of order one or higher. The approximation error therefore depends on the smoothness of the true solution, the choice of activation function, and the architecture of the network.

In practice, the approximation error is *rarely* the dominant contribution to the total error. Modern neural networks are highly expressive, and even moderately sized architectures can approximate smooth PDE solutions with good accuracy. However, the approximation error becomes relevant when the solution has sharp gradients, discontinuities, or multi-scale features, or when the PDE involves high-order derivatives. In such cases, architectural choices such as depth, width, activation functions, and positional encodings can significantly influence the expressive power of the model. We will see the impact of this fact in the following sections.

3.3.3 Estimation (Quadrature) Error

The second contribution to the total error in a PINN arises from the discrepancy between the continuous risk $\mathcal{R}$ and its discrete approximation $\hat{\mathcal{R}}$. This is the *estimation error*, defined as

$$\mathcal{E}_G = \sup_{\theta \in \Theta} \left| \mathcal{R}[u_\theta] - \hat{\mathcal{R}}[u_\theta] \right|. \tag{3.11}$$

In classical supervised learning this quantity is interpreted as a generalization error, since the empirical risk is computed on a finite dataset and the continuous risk corresponds to the expected loss over the data distribution. In PINNs the situation is different. The training points are not drawn from a distribution, but are chosen by the practitioner, and the discrete loss is a numerical quadrature approximation of a continuum integral. The estimation error is therefore a quadrature error rather than a statistical generalization error.

The magnitude of $\mathcal{E}_G$ depends on the sampling strategy used to select the collocation points. Uniform sampling, Latin hypercube sampling, adaptive sampling, and residual-based refinement all lead to different quadrature accuracies. In high-dimensional problems the estimation error can dominate the total error, since accurate quadrature becomes challenging. However, for many PDEs the structure of the residual and the regularity of the solution mitigate the curse of dimensionality.

A rigorous analysis of the estimation error has been obtained for certain classes of PDEs. In particular, De Ryck and Mishra [DM22] proved that for Kolmogorov-type equations, such as the heat equation or the Black–Scholes equation, the following bound holds:

$$\mathcal{R}[u_\theta] \leq \left(C\,\mathcal{R}[u] + \mathcal{O}(N^{-1/2}) \right)^{1/2},$$

where N is the number of collocation points. This result shows that small training error implies small continuous risk, and that the estimation error decays at a rate comparable to Monte Carlo quadrature. Remarkably, the bound does not depend on the dimension of the problem, which suggests that PINNs can generalize well for certain PDEs even in high-dimensional settings.

In practice, the estimation error is often a critical factor in the performance of PINNs. If the collocation points do not adequately resolve the regions where the PDE residual is large, the discrete loss may underestimate the true error. This explains why adaptive sampling and residual-based refinement can significantly improve the accuracy of PINNs. The estimation error therefore reflects the interplay between numerical quadrature, PDE structure, and the sampling strategy used during training.

3.3.4 Optimization Error

The third contribution to the total error in a PINN is the *optimization error*, which reflects the fact that the discrete loss $\hat{\mathcal{R}}$ is minimized by a gradient-based algorithm that may not reach a global minimizer. The optimization error is defined as

$$\mathcal{E}_O = \hat{\mathcal{R}}[u_{\hat{\theta}}] - \inf_{\theta \in \Theta} \hat{\mathcal{R}}[u_\theta], \tag{3.12}$$

where $\hat{\theta}$ denotes the parameter vector obtained after training. Since the PINN loss is highly non convex, the optimizer may converge to a local minimum, a saddle point, or a flat region of the loss landscape.

The optimization landscape of PINNs is significantly *more complex* than that of standard supervised learning models. The loss involves derivatives of the neural network, and the PDE residual may contain stiff terms, multi-scale features, or high-order operators. These factors create sharp valleys, flat plateaus, and regions where gradients vanish or explode. As a consequence, *the optimization error can dominate the total error*, especially in problems with strong nonlinearities or multi-scale behaviour [Kri+21; WTP21].

Empirical evidence suggests that the choice of optimizer plays a crucial role [Tos+25]. First-order methods such as Adam are effective in the early stages of training, where they rapidly reduce the residual. Second-order methods such as L-BFGS often perform better in the later stages, where they refine the solution and reduce the stiffness of the residual [Jin+21; USP25; WYP22]. Hybrid strategies that combine both methods are widely used in practice.

Despite these empirical successes, the theoretical understanding of optimization in PINNs remains limited. The non convexity of the loss prevents the derivation of global convergence guarantees, and the interaction between the PDE structure and the optimization dynamics is not fully understood. Recent studies indicate that the optimization trajectory may exhibit distinct phases, with an initial period of rapid error reduction followed by a slower

regime where the network adjusts to satisfy the PDE constraints more precisely [Tos+25]. These observations motivate the study of optimization dynamics and information-theoretic perspectives, which will be discussed later on.

3.3.5 Total Error Decomposition

The three sources of error described above combine to determine the total discrepancy between the trained predictor $u_{\hat{\theta}}$ and the true solution u of the PDE. A central result in the learning theory of PINNs is the following inequality, which bounds the continuous risk of the trained model [Kut22]:

$$\mathcal{R}[u_{\hat{\theta}}] \leq \mathcal{E}_O + 2\,\mathcal{E}_G + \mathcal{E}_A. \tag{3.13}$$

This decomposition, introduced in [Kut22], separates the contributions of optimization, estimation, and approximation. Each term reflects a different aspect of the learning process:

- $\mathcal{E}_A$ measures the expressive power of the neural network architecture. It depends on depth, width, activation functions, and the regularity of the true solution [DLM21].
- $\mathcal{E}_G$ quantifies the quadrature error introduced by discretizing the continuous loss. It depends on the sampling strategy, the number of collocation points, and the structure of the PDE [DM22].
- $\mathcal{E}_O$ captures the limitations of gradient-based optimization. It depends on the optimizer, initialization, and the geometry of the loss landscape [Kri+21; WTP21].

This decomposition provides a conceptual framework for understanding both the strengths and the weaknesses of PINNs. For example, if the sampling is insufficient, the estimation error dominates and the network may satisfy the discrete residual while violating the PDE in regions not covered by collocation points. If the optimizer becomes trapped in a poor local minimum, the optimization error dominates. If the architecture is not expressive enough, the approximation error becomes the limiting factor.

In practice, the three errors interact in non-trivial ways. Improving the sampling strategy may reduce the estimation error but increase the stiffness of the optimization landscape. Increasing the network size may reduce the approximation error but make optimization more difficult.

3.3.6 Training Dynamics of PINNs: connection to Optimization Dynamics and Information Flow

The error decomposition provides a static view of the learning process, but PINN training is inherently dynamic. The evolution of the parameters $\theta(t)$ under gradient-based optimization plays a central role in determining which error component dominates at different stages of training. Recent empirical studies show that PINNs often exhibit distinct learning phases [Ana+25; Tos+25], similar to those observed in deep supervised networks [ST17; TZ15; WHM25] (for a nice non-technical introduction, see the *Quanta Magazine* blog post [Wol17], while for a well rounded critic, see [Sax+19]).

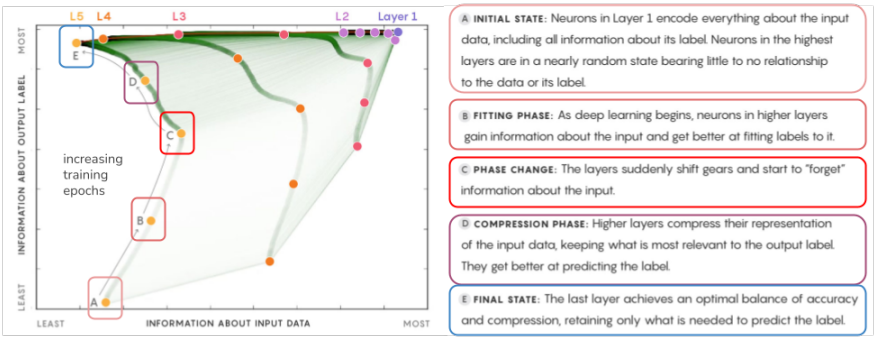


Figure 3.7: Qualitative evolution of the internal representations of a deep neural network during training, visualized in the *information plane*. The horizontal axis reports $I(X;T)$, the mutual information between the input X and the layer representation T, which measures how much information about the input is retained. The vertical axis reports $I(T;Y)$, the mutual information between the representation and the output label Y, quantifying how predictive the representation is. Empirically, layers tend to follow a two-phase dynamics: an initial *fitting phase*, where both $I(X;T)$ and $I(T;Y)$ increase, followed by a *compression phase*, where $I(X;T)$ decreases while $I(T;Y)$ stabilizes. The descriptive points A to E indicate the evolution across training epochs: the model is initialized at A and, during training, moves through B, C, and D, eventually (and hopefully) reaching E. The example shown corresponds to a 5-layer MLP; the layer indices L1-L5 (from last to first) are reported above the trajectories. This picture is inspired by [ST17; TZ15] and adapted from [Wol17].

fig:3:3:DNN_IB

Hints on DNN training dynamics from Information Bottleneck Theory
The Information Bottleneck (IB) theory [Hu+24; Slo02; TPB00] provides an information-theoretic perspective on the training and performance of neural networks.

It presents a framework for forming a condensed representation of layer activations with respect to an input variable $x \in \mathscr{X}$, retaining as much information as possible about an output variable, $y \in \mathscr{Y}$. A key concept in this theory is the mutual information $I(x,y)$ which suggests that optimal model representations preserve all relevant information about the output while discarding irrelevant input information, thus creating an "information bottleneck". The formal definition is

Definition 3.3.1 (Mutual Information)**.** Let X, Y be two random variables with a joint distribution $p(x,y)$. Their *mutual information* is defined as

$$I(X,Y) \coloneqq D_{KL}\left[p(x,y) \| p(x)p(y)\right], \tag{3.14}$$

where D_{KL} is the Kullback-Leibler divergence.

We can show that

$$\begin{aligned} I(X,Y) &= D_{KL}\left[p(x,y)\|p(x)p(y)\right] \\ &= \sum_{x\in\mathscr{X},y\in\mathscr{Y}} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right) \\ &= \sum_{x\in\mathscr{X},y\in\mathscr{Y}} p(x,y)\log\left(\frac{p(x,y)}{p(x)}\right) - \sum_{x\in\mathscr{X},y\in\mathscr{Y}} p(x,y)\log p(y) \\ &= \sum_{x\in\mathscr{X},y\in\mathscr{Y}} p(x)p(y|x)\log p(y|x) - \sum_{x\in\mathscr{X},y\in\mathscr{Y}} p(x,y)\log p(y) \\ &= \sum_{x\in\mathscr{X}} p(x)\left[\sum_{y\in\mathscr{Y}} p(y|x)\log p(y|x)\right] - \sum_{y\in\mathscr{Y}}\left[\sum_{x\in\mathscr{X}} p(x,y)\right]\log p(y) \\ &= -H(Y|x) + H(Y) = H(Y) - H(Y|X), \end{aligned}$$

where $H(Y)$ is the marginal entropy, while $H(Y|X)$ is the conditional Entropy, and where we have used $p(x,y) = p(y|x)p(x)$. Notice that, thus, the Mutual Information is symmetric under $X \longleftrightarrow Y$, so

$$I(X;Y) = H(Y) - H(Y|X) = H(X) - H(X|Y) = I(Y;X)$$

Furthermore, we have [ST17]

Remark 3.3.2. The Mutual Information has the following properties:

1. **Nonnegativity**: $I(X;Y) \geq 0$, by Jensen's Inequality.
2. **Symmetry**: $I(X;Y) = I(Y;X)$.
3. **Relation to Entropies**: We have

$$\begin{aligned} I(X;Y) &= H(Y) - H(Y|X) = H(X) - H(X|Y) \\ &= H(X) + H(Y) - H(X,Y) \\ &= H(X,Y) - H(X|Y) - H(Y|X)\,, \end{aligned}$$

   where $H(X,Y)$ is the joint entropy of X, Y.
4. **Invariance under invertible transformations**:

$$I(X;Y) = I(\psi(X),\phi(Y))\,,$$

   for any invertible functions ψ, ϕ.
5. **Data Processing Inequality** (DPI): For any 3 variables that form a Markov chain $X \to Y \to Z$,

$$I(X;Y) \geq I(X;Z).$$

The last two properties are very important in the context of DNNs. Indeed, in supervised learning we are interested in good representations, $T(X)$, of the input patterns $x \in \mathscr{X}$, to enable good predictions of the target/label (for regression/classification tasks) $y \in \mathscr{Y}$. Moreover, we want to efficiently

learn such representations from an empirical sample of the (unknown) joint distribution $p(X, Y)$, in a way that provides good generalisation. Furthermore, Deep Learning models generate a Markov chain of such representations, a.k.a. the hidden layers, by minimisation of the empirical error over the weights of the network, layer by layer. This optimisation takes place usually via SGD, i.e. using a noisy estimate of the gradient of the empirical error at each weight, through back-propagation.

The insight of [ST17] is to use each whole layer, T, as a single random variable, which can be characterised by the distribution $P(T|X)$ (the "encoding" distribution) and $P(Y|T)$ (the "decoding" distribution). As we are only interested in the information that flows through the network, invertible transformations of the representations, that preserve information, generate equivalent representations even if the individual layers encode entirely different features of the input. For this reason, and for the property 4 of the mutual information, it is possible to quantify the representations by the two mutual informations, $I(T;X)$, the mutual information of T with the input X, and $I(Y;T)$, the mutual information of the output Y with T, and may play the role of "order parameters"; furthermore, by property 4., they are invariant to any invertible re-parametrisation of T.

Another crucial concept, which can be exploit thanks to property 5. of the mutual information, is the *information plane*. Any representation variable, T , defined as a (possibly stochastic) map of the input X, is characterised by its joint distributions with X and Y, $P(T|X)$ and $P(Y|T)$. Given $P(X;Y)$, T is uniquely mapped to a point in the $information plane$ $(I(X;T), I(T;Y))$.

When applied to the deep neural networks with k layers, seen as $k-$step Markov Chain, labelling with $i = 1, \dots, N$ the layer index, the layers are mapped to k monotonic connected points in the plane, called the *information path*. By property 5. of mutual information, it satisfies the DPI chains:

$$I(X;Y) \geq I(T_1;Y) \geq I(T_k;Y) \geq \cdots \geq I(T_k;Y) \geq I(\hat{Y};Y)\,, \tag{3.15}$$

$$H(X) \geq I(X;T_1) \geq I(X;T_2) \geq \cdots \geq I(X;T_k) \geq I(X;\hat{Y})\,. \tag{3.16}$$

This plane is where the acion happens, during training. It was empirically observed that there is a non-trivial, interesting dynamics showing in the information plane during training (treating the epoch as a discretisation of a training time). In Figure 3.7 we report an illustrative example. We can detect five distinct points:

A. **Initial State:** Neurons in the first layer encode essentially almost all information about the input data, including information relevant to the label. Neurons in deeper layers are nearly random and bear little to no relationship to either the input or the output.

B. **Fitting Phase:** As training begins, neurons in higher layers progressively acquire information about the input and become increasingly effective at fitting the labels.

C. **Phase Change:** During training, due to the stochasticity of SGD, the network undergoes a transition in which layers begin to "forget" irrelevant information about the input.

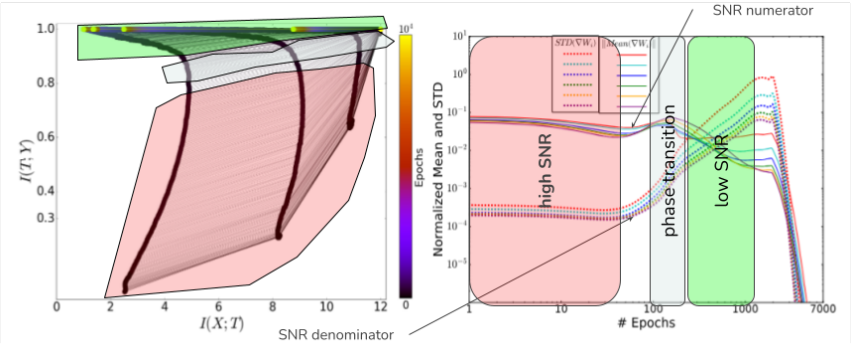


Figure 3.8: On the left, the layers information plane paths during training. On the right, the stochastic gradients means and standard deviations, during training (epochs are on $x-$axis). We have highlighted the three phases of the SNR. Adapted from [ST17].

fig:3:3: DNN_IB_SNR

D. **Compression Phase:** Higher layers compress their internal representation of the input, retaining primarily the information that is most relevant for predicting the output label. Predictive performance improves as irrelevant variability is discarded.

E. **Final State:** The final layer reaches a balance between accuracy and compression, keeping only the information necessary to predict the label.

Another crucial parameter to monitor the training dynamics is the batch-wise *Signal-to-Noise* ratio (SNR), defined as [Ana+25; Shu+24]

$$\text{SNR} := \frac{\|\mu|_2}{\|\sigma\|_2} := \frac{\|\mathbb{E}\left[\nabla_\theta \mathcal{L}\right]\|_2}{\|\text{std}\left[\nabla_\theta \mathcal{L}\right]\|_2} \,. \tag{3.17}$$

As shown in Figure 3.8, the phases of training can be seen as the behaviour of SNR. Furthermore, [Ana+25; Shu+24] have shown that PINNs exhibits *three* distinct training phases:

1. **Fitting Phase**: At the beginning of training, both the loss and its gradients are large across all subdomains. This produces a high batch-wise SNR, since the gradient signal dominates the noise. During this stage, the optimizer follows a clear descent direction and the residuals exhibit an ordered structure. As training progresses and the loss decreases, disagreement between subdomains increases, causing the SNR to drop. This phase is therefore predominantly *deterministic*, transitioning from high to low SNR [Shu+24].

1. **Diffusion Phase**: Once the model has adequately fitted the data, it enters a stochastic exploration regime. Here, the gradients across subdomains fluctuate, the SNR remains low, and the network weights undergo a diffusion-like evolution. This diffusion disrupts the initial ordering of parameters and is associated with improved generalization, as the model

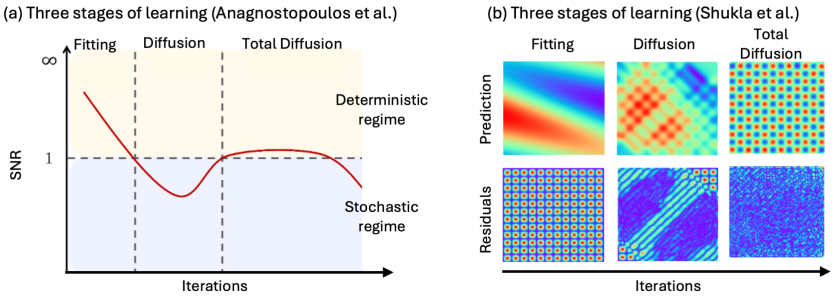


Figure 3.9: Illustration of the three stages of learning in PINNs. (a) The evolution of the batch-wise signal-to-noise ratio (SNR) reveals distinct training regimes: a high-SNR *fitting* phase corresponding to deterministic descent; a low-SNR *diffusion* phase characterized by stochastic exploration; and a final *total diffusion* phase in which the SNR rises sharply and stabilizes, indicating the emergence of a coherent descent direction and accelerated reduction of the generalization error. (b) Prediction fields and residual distributions for the Helmholtz equation at the three stages. Residuals are highly ordered during fitting, become progressively disordered during diffusion, and remain noisy in total diffusion, where the model simplifies its internal representation and closely matches the analytical solution. From [Tos+25].

fig:3:3:PINN_Stages_learning

searches for a direction that consistently reduces the global training error. Residuals become disordered and the SNR oscillates [Ana+25; Shu+24].

1. **Total Diffusion Phase**: [New phase in PINNs!] After the model identifies a coherent descent direction that reduces the generalization error across all subdomains, the SNR rises sharply and stabilizes. In this phase, the network simplifies its internal representation by retaining only the most relevant features while discarding irrelevant ones, effectively reducing model complexity. Residuals remain highly disordered, but the generalization error (e.g., relative L^2) decreases rapidly, indicating convergence. Empirically, the best-performing PINN models are those that enter the total diffusion phase earliest [Ana+25; Shu+24].

The three phases described above are visually summarized in Figure 3.9 (Adapted in [Tos+25] from [Ana+25; Shu+24]). Panel (a) shows how the batch-wise SNR evolves during training, providing a clear diagnostic of the underlying learning regime: the fitting phase corresponds to a high-SNR, largely deterministic descent; the diffusion phase emerges when the SNR drops and oscillates, reflecting stochastic exploration; and the total diffusion phase begins once the model identifies a coherent descent direction, causing the SNR to rise and stabilize. Panel (b) complements this view by displaying predictions and residuals for a Helmholtz problem at each stage. Residuals transition from ordered (fitting) to disordered (diffusion), and remain noisy in total diffusion, where the model achieves its best generalization and most accurate reconstruction of the analytical solution. Together, these visualizations reinforce the interpretation of PINN training as a progression from deterministic fitting to stochastic exploration and finally to a stabilized, highly generalizable regime.

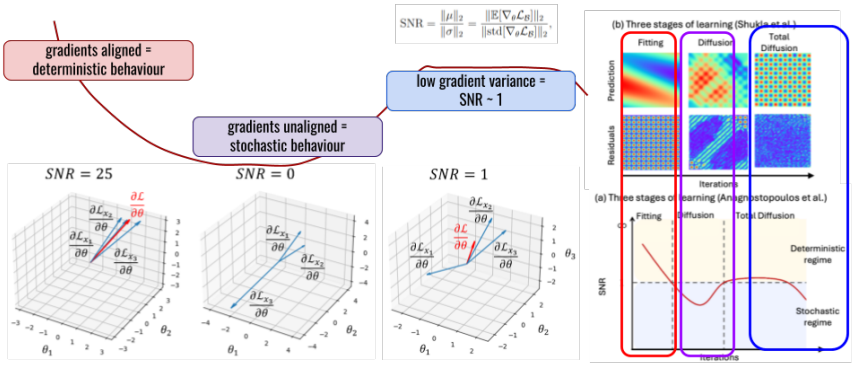


Figure 3.10: Visual representation of the alignment of the mini-batch gradients during the three PINN training phases. Adapted from [Ana+25].

fig:3:3: PINN_IB_align

These three training phases are fundamentally governed by the geometry of the batch-wise gradients (see Figure 3.10). As demonstrated in [Ana+25; Shu+24], the Signal-to-Noise Ratio (SNR) provides a quantitative measure of this geometry: during the *fitting phase*, gradients computed at different collocation points are largely aligned, producing a high SNR and a clear deterministic descent direction; in the subsequent *diffusion phase*, these gradients become increasingly misaligned and partially cancel, yielding a low SNR and a stochastic exploration regime; finally, in the *total diffusion phase*, a coherent global descent direction re-emerges, the SNR rises sharply, and the model rapidly improves its generalization.

Importantly, we have seen that PINNs are trained through a *multi-objective loss* that typically includes PDE residuals, boundary conditions, and possibly data terms. This decomposition can introduce additional sources of gradient conflict, further contributing to misalignment and potentially creating multiple local transitions between deterministic and stochastic regimes.

CHAPTER 4

Lecture 4: Advanced PINNs methods - *Learning strategies, Architectures, Losses, and other approaches*

chap:4

The aim of this section is to introduce and examine various techniques to improve PINN training, grouped into five categories: (i) adaptive weighting of loss terms, (ii) advanced sampling and geometry handling, (iii) architectural innovations, (iv) optimisation strategies, and (v) a synthesis of best practices. This section covers a vast amount of research put into optimising PINN for various tasks. Furthermore, it aim to show the interconnection of PINNs with other machine learning topics, highlighting how hints from one topic may be beneficial in the other.

4.1 Dynamic Hyperparameter Optimisation

sec:4:1:DynHypOp

In Chapter 3 we have seen that PINNs, whether in their vanilla, inverse, or parametric formulation, are inherently *multi-objective* optimisation problems. The loss functional typically combines several heterogeneous components,

$$\mathcal{L}[\theta] = w_{\text{pde}}\,\mathcal{L}_{\text{pde}}[\theta] + w_{\text{bc}}\,\mathcal{L}_{\text{bc}}[\theta] + w_{\text{data}}\,\mathcal{L}_{\text{data}}[\theta] = \sum_i w_i\,\mathcal{L}_i[\theta]\,, \tag{4.1}$$

where the coefficients w_i determine the relative importance of the PDE residual, the boundary or initial conditions, and any available data. It is customary to fix $w_{\text{pde}} = 1$, but the remaining weights strongly influence the optimisation landscape.

As discussed in Section 3.3, the training dynamics of PINNs are already intricate due to the evolving alignment of per-sample gradients and the associated transitions between deterministic and stochastic learning regimes. The multi-objective structure of the loss further amplifies this complexity. Different loss terms may operate at different scales, exhibit different curvature properties, or generate gradients that point in conflicting directions. This imbalance can lead to optimisation pathologies such as stiffness, slow convergence, or premature stagnation, phenomena that have been widely documented in the PINN literature [BK25; WTP21; WYP22].

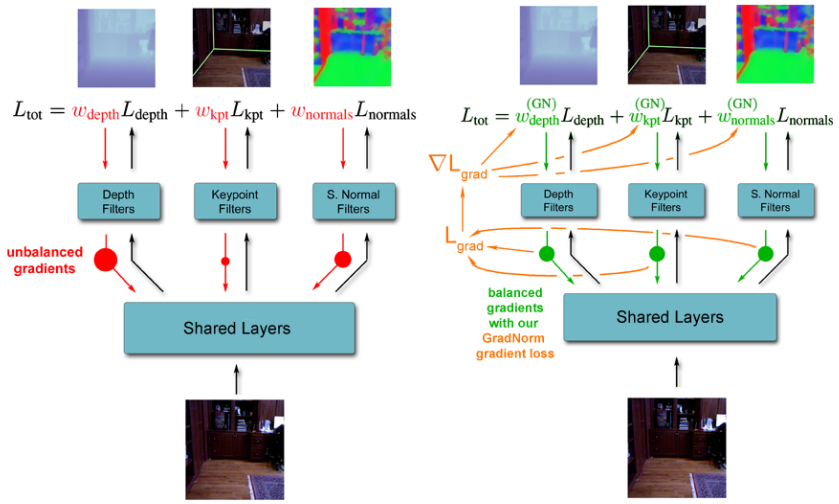


Figure 4.1: Visual representation of the GradNorm algorithm. Imbalanced gradient norms across tasks (left) result in suboptimal training within a multitask network. GradNorm algorithm is implemented through computing a novel gradient loss L_{grad} (right) which tunes the loss weights w_i to fix such imbalances in gradient norms. From [Cra20].

fig:4:1: GradNorm1

These challenges motivate the development of *dynamic hyperparameter optimisation* strategies, where the weights w_i are adapted during training rather than fixed a priori. Several approaches have been proposed, including *GradNorm* [Che+18], *Learning Rate Annealing* [WTP21], *SoftAdapt* [HTM19], *Relative Loss Balancing with Random Lookback* (ReLoBRaLo) [BK25], conflict-aware weighting schemes such as e *Conflict-Free Inverse Gradients* (ConFIG) [LCT24], and NTK-based adaptive weighting inspired by neural tangent kernel theory [JGH18; Wen22; WYP22]. These methods aim to rebalance the contributions of the different loss components in real time, mitigate gradient conflicts, and guide the optimiser toward a regime where all objectives are satisfied coherently and efficiently.

In the following paragraphs we review these approaches, analyse their underlying principles, and discuss their practical impact on the training of PINNs.

4.1.1 GradNorm

Multi-objective learning is not unique to PINNs; on the contrary, it arises quite naturally in many Computer Vision applications, often denoted as deep multi-task learning [Cra20]. Indeed, often, even single computer vision problems can even be framed as multi-task problems, such as in Mask R-CNN for instance segmentation [He+17] or the YOLo family for object detection [Red+16; RF17].

For an explicit example of multi-objective loss, dubbes as "multi-partite" loss function, see Equation (3) in [Red+16].

In [Cra20], the authors defined a dynamic hyperparameter re-weighting for

multi-objective losses in the form

$$\mathcal{L}(\theta(t)) = \sum_i w_i(t)\, \mathcal{L}_i[\theta(t)]\,,$$

where t is the *training time*, by defining $w_i(t)$ with the goals:

(1) to place gradient norms for different tasks on a common scale through which we can reason about their relative magnitudes;

(2) to dynamically adjust gradient norms so different tasks train at similar rates.

The new loss, $\mathcal{L}_{grad}$, is defined in terms of a certain sub-network $W \subset \mathcal{W}$; the weighted norm of its gradient at a certain epoch t

$$G_W^{(i)}(t) \coloneqq \left\| \nabla_W \left(w_i(t) \mathcal{L}_i(\theta(t)) \right) \right\|_2 ;$$

its average across all tasks at step t,

$$\overline{G}_W(t) \coloneqq \mathbb{E}_{task}\left[G_W^{(i)}(t) \right] ,$$

and the relative inverse training rate of task i,

$$r_i(t) \coloneqq \frac{\tilde{\mathcal{L}}_i(\theta(t))}{\mathbb{E}_{task}\left[\tilde{\mathcal{L}}_i(\theta(t)) \right]} , \qquad \tilde{\mathcal{L}}_i(\theta(t)) \coloneqq \frac{\mathcal{L}_i(\theta(t))}{\mathcal{L}_i(\theta(0))} .$$

The goal of GradNorm is to establish a common scale for gradient magnitudes, and also to balance training rates of different tasks. The common scale for gradients is the average gradient norm, $\overline{G}_W(t)$, which establishes a baseline at each training step t by which we can determine relative gradient sizes. The relative inverse training rate of task i, $r_i(t)$, can be used to rate balance our gradients: the higher the value of $r_i(t)$, the higher the gradient magnitudes should be for task i in order to encourage the task to train more quickly. Therefore, our desired gradient norm for each task i is:

$$G_W^{(i)}(t) \mapsto \overline{G}_W(t) \times [r_i(t)]^{\alpha},$$

where α is an additional hyperparameter, which sets the strength of the restoring force which pulls tasks back to a common training rate.

The core idea is that now, we can treat $\{w_i\}$ as *learnable parameters*, adapted at each step, together with the network parameters W, by an optimisation algorithm to minimise a single loss. The above equation gives a target for each task i's gradient norms, and thus a natural choice for the loss is

$$\mathcal{L}_{grad}(\theta(t); w_i(t)) \coloneqq \sum_i \left\| G_W^{(i)}(t) - \bar{G}_W(t) \times [r_i(t)]^{\alpha} \right\|_1 , \tag{4.2}$$

where $\| \cdot \|_1$ is the standard 1-norm, and $\times$ is a normal multiplication operator, added for clarity. We report a schematic representation of GradNorm in Algorithm 3.

alg:4:1:gradnorm

Algorithm 3 Training with GradNorm (from [Che+18])

Initialize $w_i(0) = 1 \; \forall i$
Initialize network weights $\mathcal{W}$
Pick value for $\alpha > 0$ and pick the weights W (usually the final layer of weights which are shared between tasks)
for $t = 0$ **to** N_{epochs} **do**
  Input batch x_i to compute $L_i(t) \; \forall i$ and $L(t) = \sum_i w_i(t) L_i(t)$ [standard forward pass]
  Compute $G_W^{(i)}(t)$ and $r_i(t) \; \forall i$
  Compute $\overline{G}_W(t)$ by averaging the $G_W^{(i)}(t)$
  Compute $L_{\text{grad}} = \sum_i |G_W^{(i)}(t) - \overline{G}_W(t) \times [r_i(t)]^\alpha|_1$
  Compute GradNorm gradients $\nabla_{w_i} L_{\text{grad}}$, keeping targets $\overline{G}_W(t) \times [r_i(t)]^\alpha$ constant
  Compute standard gradients $\nabla_{\mathcal{W}} L(t)$
  Update $w_i(t) \mapsto w_i(t+1)$ using $\nabla_{w_i} L_{\text{grad}}$
  Update $\mathcal{W}(t) \mapsto \mathcal{W}(t+1)$ using $\nabla_{\mathcal{W}} L(t)$ [standard backward pass]
  Renormalize $w_i(t+1)$ so that $\sum_i w_i(t+1) = T$

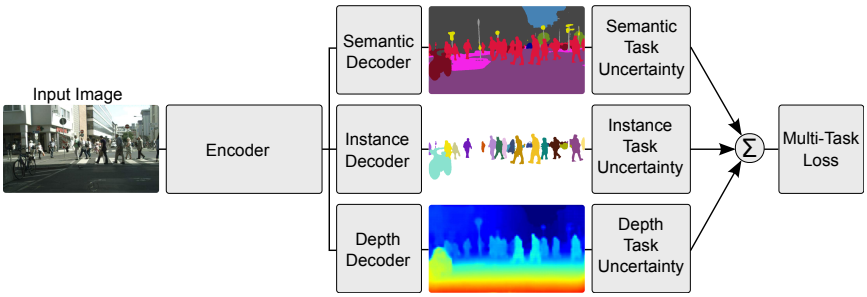


Figure 4.2: Visual example of a Multi-task deep learning model. Homoscedastic Task Uncertainty is introduced as a principled way of combining multiple regression and classification loss functions for multi-task learning. As an example, the architecture takes a single monocular RGB image as input and produces a pixel-wise classification, an instance semantic segmentation and an estimate of per pixel depth. Notice that Multi-task learning can improve accuracy over separately trained models because cues from one task, such as depth, are used to regularize and improve the generalization of another domain, such as segmentation. From [KGC18].

fig:4:1: HomoscedasticModel

4.1.2 Homoscedastic Task Uncertainty

Another approach coming from the field of Computer Vision is *Homoscedastic Task Uncertainty* [KGC18], where the goal is to balance semantic tast, instance task, and depth task uncertainty of a single model (see Figure 4.2).

The idea was to use a probabilistic approach to dynamic loss weighting, interpreting each task-specific loss as arising from a likelihood model with an associated observation noise. In this framework, the relative weight of each loss term is not treated as a free hyperparameter but is instead derived from the *homoscedastic uncertainty* of the corresponding task. This uncertainty is

In statistics, a sequence of random variables is *homoscedastic* if all its random variables have the same finite variance; this is also known as homogeneity of variance.

independent of the input data and reflects the intrinsic noise level of the task itself.

In Bayesian modelling, there are two main types of uncertainty one can model [KG17]: the *Epistemic uncertainty*, descrigint the uncertainty in the model, which captures what the model does not know due to lack of training data; and the *Aleatoric uncertainty*, captureing the uncertainty with respect to information which the data cannot explain. Aleatoric uncertainty can be explained away with the ability to observe all explanatory variables with increasing precision. Furthermore, Aleatoric uncertainty can again be divided into two sub-categories:

- *Data-dependent* or *Heteroscedastic* uncertainty: it is the aleatoric uncertainty which depends on the input data and is predicted as a model output.
- *Task-dependent* or *Homoscedastic* uncertainty: it is the aleatoric uncertainty which is not dependent on the input data. It is not a model output, rather it is a quantity which stays constant for all input data and varies between different tasks. It can therefore be described as task-dependent uncertainty.

In a multi-task setting, the authors [KGC18] show that the task uncertainty captures the relative confidence between tasks, reflecting the uncertainty inherent to the regression or classification task. They thus propose a multi-task loss function based on maximising the Gaussian likelihood with homoscedastic uncertainty.

Additionally, they show that, for a regression task with Gaussian likelihood, the negative log-likelihood takes the form

$$\mathcal{L}_j(\theta, \sigma_j) = \frac{1}{2\sigma_j^2} \|y_j - f_j(\theta)\|^2 + \log \sigma_j,$$

where σ_j is the task-dependent noise parameter. The key observation is that the coefficient $1/\sigma_j^2$ acts as an *automatic loss weight*, while the additional $\log \sigma_j$ term prevents the trivial solution $\sigma_j \to \infty$. When multiple tasks are present, the total loss becomes

$$\mathcal{L}(\theta, \{\sigma_j\}) = \sum_{j=1}^{n_{\text{loss}}} \left[\frac{1}{2\sigma_j^2} \mathcal{L}_j(\theta) + \log \sigma_j \right],$$

and the uncertainty parameters σ_j are learned jointly with the network parameters θ. Tasks with higher intrinsic noise (larger σ_j) receive smaller weights, while tasks with lower noise are emphasised.

In the context of PINNs, this idea can be applied by treating the PDE residual, boundary conditions, and data terms as separate tasks, each with its own homoscedastic uncertainty [Nvi26]. The resulting PINN loss takes the form

$$\mathcal{L}(\theta; \{\sigma_i\}) = \frac{1}{2\sigma_{\text{pde}}^2} \mathcal{L}_{\text{pde}} + \frac{1}{2\sigma_{\text{bc}}^2} \mathcal{L}_{\text{bc}} + \frac{1}{2\sigma_{\text{data}}^2} \mathcal{L}_{\text{data}} + \log \sigma_{\text{pde}} \sigma_{\text{bc}} \sigma_{\text{data}}.$$

This formulation allows the network to learn the appropriate balance between the different components of the PINN loss directly from the data and the PDE structure, without requiring manual tuning of the weights.

An alternative and numerically safer way to define the loss, is to redefine the parameters

$$2\sigma_j^2 \to \exp[\gamma_j]$$

so that

$$\mathcal{L}(\theta;\{\gamma_i\}) = \mathrm{e}^{-\gamma_{\mathrm{pde}}}\,\mathcal{L}_{\mathrm{pde}} + \mathrm{e}^{-\gamma_{\mathrm{bc}}}\,\mathcal{L}_{\mathrm{bc}} + \mathrm{e}^{-\gamma_{\mathrm{data}}}\,\mathcal{L}_{\mathrm{data}} + \frac{1}{2}\left(\gamma_{\mathrm{pde}} + \gamma_{\mathrm{bc}} + \gamma_{\mathrm{data}}\right). \tag{4.3}$$

4.1.3 Learning Rate Annealing

One of the earliest and most widely used dynamic weighting strategies for PINNs is *Learning Rate Annealing*, introduced in [WTP21]. The key idea is to adapt the relative weight between the PDE residual loss and the boundary (or initial) condition loss by monitoring the magnitude of their gradients during training. Since different loss components may operate on different numerical scales, their gradients can differ by several orders of magnitude, which leads to unbalanced optimisation dynamics and poor convergence. Learning Rate Annealing addresses this issue by enforcing that the gradients of the different loss terms contribute comparably to the parameter update.

Given a loss of the form

$$\mathcal{L}(\theta) = \mathcal{L}_{\mathrm{pde}}(\theta) + \lambda^{(i)}\mathcal{L}_{\mathrm{BC}}(\theta),$$

the adaptive weight $\lambda^{(i)}$ at iteration i is computed from the ratio of gradient magnitudes,

$$\bar{\lambda}^{(i)} = \frac{\max\left(\left|\nabla_\theta \mathcal{L}_{\mathrm{pde}}(\theta^{(i)})\right|\right)}{\mathrm{mean}\left(\left|\nabla_\theta \mathcal{L}_{\mathrm{BC}}(\theta^{(i)})\right|\right)},$$

and smoothed through an *exponential moving average*,

$$\lambda^{(i)} = \alpha\,\bar{\lambda}^{(i)} + (1-\alpha)\,\lambda^{(i-1)}.$$

The role of exponential moving average, or exponential smoothing, is to act as low-pass filters to remove high-frequency noise.

This procedure ensures that the optimiser receives balanced gradient contributions from the PDE and boundary losses, preventing one term from dominating the training dynamics. In practice, Learning Rate Annealing aims to mitigates stiffness in the optimisation landscape and improves convergence, especially in problems where the PDE residual and boundary conditions evolve at different rates during training.

Remark 4.1.1. Although the gradient ratio used in Learning Rate Annealing is not directly related to the batch-wise SNR discussed in Section 3.3, both quantities share a similar qualitative purpose. The SNR measures the relative strength of aligned versus misaligned gradient components across training samples, while the annealing ratio compares the magnitudes of gradients arising from different loss terms. In both cases, the goal is to diagnose and correct imbalances in the optimisation dynamics, ensuring that no single component dominates the parameter updates.

4.1.4 SoftAdapt

SoftAdapt is another dynamic loss-balancing strategy that has been successfully applied to PINNs, although it was originally developed in the context of

computer vision and multi-part loss functions [HTM19]. In its original formulation, SoftAdapt was designed to stabilise training in tasks such as image segmentation and multi-branch architectures, where different loss components (for example, pixel-wise reconstruction, perceptual loss, and adversarial loss) evolve at different rates. This makes SoftAdapt particularly interesting for readers with a background in computer vision, since it demonstrates how techniques developed for deep learning in perception tasks can transfer effectively to physics-informed settings, *and vice-versa*.

The core idea of SoftAdapt is to monitor the *relative progress* of each loss term during training. Instead of comparing gradient magnitudes, as in Learning Rate Annealing, SoftAdapt compares the ratio of each loss value at the current iteration to its value at the previous iteration. Loss terms that decrease slowly (or even increase) are assigned larger weights, while those that decrease rapidly are down-weighted. The weights are computed through a softmax transformation, ensuring positivity and smoothness:

$$\lambda_j^{(i)} = \frac{\exp\left(\frac{L_j(i)}{L_j(i-1)}\right)}{\sum_{k=1}^{n_{\text{loss}}} \exp\left(\frac{L_k(i)}{L_k(i-1)}\right)}.$$

This adaptive mechanism encourages the optimiser to focus on the most challenging components of the loss, preventing situations where one term is minimised quickly while others stagnate. In the context of PINNs, SoftAdapt has been shown to mitigate imbalance between PDE residuals, boundary conditions, and data terms, often leading to faster and more stable convergence. Although not originally designed for scientific machine learning, its transfer to PINNs highlights the broader relevance of multi-objective optimisation techniques across different deep learning domains.

4.1.5 ReLoBRaLo

The *Relative Loss Balancing with Random Lookback* (ReLoBRaLo) method [BK25] is a refinement of SoftAdapt that addresses two of its main limitations: (a) sensitivity to noise in the loss ratios, and (b) instability caused by relying exclusively on the most recent loss values. While SoftAdapt reacts immediately to short-term fluctuations in the loss landscape, ReLoBRaLo introduces two stabilising mechanisms: (i) a moving average over past weights, and (ii) a random lookback window that prevents the optimiser from overfitting to transient loss behaviour.

Mathematically speaking, given a multi-objective loss

$$\mathcal{L}(\theta) = \sum_{j=1}^{n_{\text{loss}}} w_j \, \mathcal{L}_j(\theta),$$

ReLoBRaLo computes the adaptive weights through a combination of exponential smoothing and a stochastic lookback:

$$w_j(i) = \alpha\Big(\beta \, w_j(i-1) + (1-\beta)\, \hat{w}_j^{(i;0)}\Big) + (1-\alpha)\, \hat{w}_j^{(i;i')},$$

where β is a Bernoulli random variable with expectation close to one, and i' is a randomly selected past iteration. The quantities $\hat{w}_j^{(i;i')}$ are SoftAdapt-style

weights computed using a temperature parameter τ,

$$\hat{w}_j^{(i;i')} = \frac{\exp\left(\frac{L_j(i)}{\tau\, L_j(i')}\right)}{\sum_{k=1}^{n_{\text{loss}}} \exp\left(\frac{L_k(i)}{\tau\, L_k(i')}\right)}.$$

The temperature τ controls the sharpness of the weighting: large values yield nearly uniform weights, while $\tau \to 0$ recovers an arg max-like behaviour. The random lookback mechanism prevents the method from reacting too aggressively to short-term oscillations, while the moving average smooths the evolution of the weights across iterations.

In practice, ReLoBRaLo has been shown to outperform SoftAdapt in PINN training [WYP22], particularly in stiff problems where different loss components evolve on widely separated scales [BK25]. By combining adaptivity with temporal smoothing and stochasticity, ReLoBRaLo provides a more robust and stable approach to dynamic loss balancing.

4.1.6 ConFIG

The *Conflict-Free Gradient* (ConFIG) method [LCT24] takes a fundamentally different approach to dynamic hyperparameter optimisation. Instead of adapting scalar weights for each loss term, ConFIG directly modifies the *gradient direction* used for the parameter update. The motivation stems from the observation that, in multi-objective PINNs, the gradients associated with different loss components often point in conflicting directions. Such conflicts can severely hinder optimisation, especially in stiff PDEs where the PDE residual, boundary conditions, and data terms may pull the parameters toward incompatible updates.

ConFIG resolves this issue by constructing a unified descent direction that is geometrically consistent across all loss terms. Given the individual gradients $g_1, g_2, \ldots, g_m$ associated with the m loss components, ConFIG first normalises each gradient to extract its direction, then computes a consensus direction g_u by summing the orthogonal components:

$$g_u = \mathcal{U}\Big([\mathcal{U}(g_1), \mathcal{U}(g_2), \ldots, \mathcal{U}(g_m)]^{-T}\, \mathbf{1}_m\Big),$$

where $\mathcal{U}(\cdot)$ denotes normalisation to unit length. The final ConFIG gradient is obtained by projecting each loss-specific gradient onto this consensus direction and summing the contributions,

$$g_{\text{ConFIG}} = \mathcal{G}(g_1, g_2, \ldots, g_m) = \left(\sum_{i=1}^{m} g_i^T g_u\right) g_u.$$

This construction ensures that the update direction is free of destructive interference between loss terms, while still respecting the relative magnitudes of their projections. Unlike weighting-based methods such as SoftAdapt or ReLoBRaLo, ConFIG does not attempt to rebalance the loss values themselves; instead, it enforces geometric coherence at the level of the gradients. This makes ConFIG particularly effective in scenarios where gradient conflicts, rather than scale imbalances, dominate the training dynamics.

In practice, ConFIG has been shown to stabilise optimisation and accelerate convergence in challenging PINN problems, especially those involving multiple competing objectives or highly anisotropic residual landscapes. Its gradient-level formulation also makes it compatible with standard optimisers such as Adam, as demonstrated in the M-ConFIG algorithm proposed in [LCT24].

The ConFIG method can be interpreted as a geometric modification of the standard Adam optimiser Algorithm 4 [KB17]. Both algorithms maintain first and second moment estimates of the gradients, and both update the parameters through a normalised, momentum-driven descent direction. The key difference lies in how the gradient used by the optimiser is constructed. Adam relies directly on the raw gradient of the full loss, while M-ConFIG Algorithm 5 replaces this gradient with a conflict-free direction obtained by projecting the individual loss-specific gradients onto a consensus direction. This substitution leaves the Adam machinery intact but fundamentally alters the descent geometry, enabling the optimiser to avoid destructive interference between competing objectives.

alg:4:1:adam

Algorithm 4 Adam Optimiser

Require: θ_0 (initial parameters), η (learning rate), β_1, β_2 (momentum coefficients), ϵ

$m_0 \leftarrow 0$ ▷ Init first moment
$v_0 \leftarrow 0$ ▷ Init second moment
for $t = 1, 2, \ldots$ **do**
  $g_t \leftarrow \nabla_\theta \mathcal{L}(\theta_{t-1})$ ▷ compute gradient of full loss
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ ▷ Update biased 1° momentum
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ ▷ Update biased 2° raw momentum
  $\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$ ▷ Compute bias-corrected 1° momentum
  $\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$ ▷ Compute bias-corrected 2° raw momentum
  $\theta_t \leftarrow \theta_{t-1} - \eta\, \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update parameters

alg:4:1:ConFIG

Algorithm 5 M-ConFIG Optimiser

Require: θ_0, η, β_1, β_2, ϵ, m_0 (pseudo first momentum)

$\{m_{g_1,0}, \ldots, m_{g_m,0}\} \leftarrow 0$ ▷ Init 1° momentums for each loss term
$v_0 \leftarrow 0$ ▷ Init 2° momentum
$\{t_{g_1}, \ldots, t_{g_m}\} \leftarrow 0$
for $t = 1, 2, \ldots$ **do**
  $i \leftarrow t \bmod m + 1$
  $t_{g_i} \leftarrow t_{g_i} + 1$
  $m_{g_i,t_{g_i}} \leftarrow \beta_1 m_{g_i,t_{g_i}} + (1-\beta_1)\nabla_\theta \mathcal{L}_i(\theta_{t-1})$ ▷ Update 1° momentum of g_i
  $\hat{m}_{g_k} \leftarrow m_{g_k,t_{g_k}} / (1-\beta_1^{t_{g_k}}),\ \forall k$ ▷ Bias corrections for 1° momentum
  $\hat{m}_g \leftarrow \mathcal{G}(\hat{m}_{g_1}, \ldots, \hat{m}_{g_m})$ ▷ ConFIG consensus gradient
  $g_c \leftarrow [\hat{m}_g(1-\beta_1^t) - \beta_1 m_{t-1}]/(1-\beta_1)$ ▷ Obtain the estimated gradient
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_c$ ▷ Update pseudo 1° momentum
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_c^2$ ▷ Update 2° momentum
  $\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$ ▷ Bias correction
  $\theta_t \leftarrow \theta_{t-1} - \eta\, \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update weights

The M-ConFIG algorithm preserves the update structure of Adam but replaces the raw gradient with a conflict-free direction obtained through the ConFIG operator. The red-highlighted lines indicate the only modifications relative to Adam: the computation of the consensus direction, the projection of the loss-specific gradients onto this direction, and the construction of the conflict-free gradient g_c. All remaining steps, including moment estimation and bias correction, follow the standard Adam formulation. This design allows ConFIG to inherit the robustness and adaptivity of Adam while resolving gradient conflicts that arise in multi-objective PINN training.
Computational overhead: Although M-ConFIG introduces additional operations compared to standard Adam, its computational overhead remains modest. The extra cost arises primarily from computing the individual loss-specific gradients and constructing the consensus direction through the ConFIG operator. As noted in [LCT24], these steps scale linearly with the number of loss terms and involve only vector normalisation and inner products, which are negligible compared to the cost of backpropagation. In practice, the overhead is typically less than a few percent of the total training time, while the improvement in optimisation stability and convergence often outweighs this additional cost.

4.1.7 Neural Tangent Kernel-based Adaptive Weighting

subsec:4:1:7:NTK

The Neural Tangent Kernel (NTK) provides a rigorous framework for understanding how neural networks evolve during gradient descent. In particular, in the abstract of [JGH18], the authors write

> « *We prove that the evolution of an ANN during training can also be described by a kernel: during gradient descent on the parameters of an ANN, the network function* $f(\theta;\cdot) \coloneqq f_\theta$ *(which maps input vectors to output vectors) follows the kernel gradient of the functional cost (which is convex, in contrast to the parameter cost) w.r.t. a new kernel: the* Neural Tangent Kernel *(NTK).*»

The starting point is the parameter-space dynamics,

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = -\eta\, \nabla_\theta \mathcal{L}[f_\theta],$$

which expresses how the network parameters change under continuous-time gradient descent. Applying the chain rule to the network output $f_\theta(\mathbf{x})$ yields

$$\frac{\mathrm{d}f(\mathbf{x};\theta)}{\mathrm{d}t} = \nabla_\theta f(\mathbf{x};\theta) \cdot \frac{\mathrm{d}\theta}{\mathrm{d}t} = -\eta\, \nabla_\theta f(\mathbf{x};\theta) \cdot \nabla_\theta \mathcal{L}.$$

For a loss of the form $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \ell(f(\mathbf{x}^{(i)};\theta), y^{(i)})$, this becomes

$$\frac{\mathrm{d}f(\mathbf{x};\theta)}{\mathrm{d}t} = -\frac{\eta}{N}\sum_{i=1}^{N} \nabla_\theta f(\mathbf{x};\theta) \cdot \nabla_\theta f(\mathbf{x}^{(i)};\theta)\, \frac{\delta \ell(f(\mathbf{x}^{(i)};\theta), y^{(i)})}{\delta f}.$$

The inner product

$$K(\mathbf{x},\mathbf{x}';\theta) \equiv \nabla_\theta f(\mathbf{x};\theta) \cdot \nabla_\theta f(\mathbf{x}';\theta)$$

is the *Neural Tangent Kernel*. It encodes how changes in the parameters affect the network outputs at different input locations. Substituting this definition yields the NTK-driven evolution equation,

$$\frac{\mathrm{d}f(\mathbf{x};\theta)}{\mathrm{d}t} = -\frac{\eta}{N}\sum_{i=1}^{N} K(\mathbf{x},\mathbf{x}^{(i)};\theta)\,\frac{\delta\ell(f(\mathbf{x}^{(i)};\theta),y^{(i)})}{\delta f}.$$

Furthermore, still in the abstract, the authors say

> « *We then focus on the setting of least-squares regression and show that in the infinite-width limit, the network function f_θ follows a linear differential equation during training. The convergence is fastest along the largest kernel principal components of the input data with respect to the NTK, hence suggesting a theoretical motivation for early stopping.*»

Indeed, at least in the infinite-width limit, the NTK becomes constant during training [JGH18], and the network output follows a linear differential equation whose convergence is governed by the eigenvalues of the kernel. This provides a spectral interpretation of optimisation and explains why certain components of the solution converge faster than others. The NTK perspective has been applied to PINNs in [WYP22], where it reveals that imbalance in the NTK spectrum between PDE residuals and boundary conditions can lead to stiffness and slow convergence. For an accessible introduction to NTK intuition and its implications for deep learning, see also [Wen22].

The NTK perspective provides not only a theoretical description of training dynamics but also a principled mechanism for rebalancing the loss terms in PINNs. As shown in [WYP22], the convergence rate of each loss component is governed by the eigenvalues of the corresponding NTK blocks. In particular, the PDE residual and boundary-condition losses often exhibit NTK spectra that differ by several orders of magnitude, which leads to stiffness and slow convergence when the loss terms are weighted uniformly. This motivates an adaptive weighting strategy in which each loss term is scaled according to the magnitude of its associated NTK eigenvalues.

Let us move on to how to use NTK to dynamically adapt the loss hyperparameters during training. Let $K(t)$ denote the NTK matrix at iteration t, partitioned according to the PDE residual and boundary-condition samples,

$$K(t) = \begin{bmatrix} K_{uu}(t) & K_{ur}(t) \\ K_{ru}(t) & K_{rr}(t) \end{bmatrix} = J(t)J(t)^T,$$

where $J(t)$ is the Jacobian of the network outputs with respect to the parameters. The eigenvalues of $K_{uu}(t)$ and $K_{rr}(t)$ determine the convergence rates of the boundary and PDE losses, respectively. To ensure uniform convergence across all objectives, [WYP22] proposes to scale the loss terms by the ratio of the total NTK trace to the trace of each block:

$$\lambda_b(t) = \frac{\mathrm{Tr}(K(t))}{\mathrm{Tr}(K_{uu}(t))}, \qquad \lambda_r(t) = \frac{\mathrm{Tr}(K(t))}{\mathrm{Tr}(K_{rr}(t))}.$$

These weights are updated periodically during training and used in the composite loss

$$\mathcal{L}(\theta) = \lambda_b(t)\,\mathcal{L}_b(\theta) + \lambda_r(t)\,\mathcal{L}_r(\theta).$$

This NTK-based reweighting ensures that each loss term evolves at a comparable rate, preventing the optimiser from overfitting to the components associated with larger NTK eigenvalues Algorithm 6. From a spectral viewpoint, the method equalises the effective conditioning of the optimisation problem by compensating for anisotropy in the NTK. In practice, NTK-based weighting has been shown to significantly improve convergence in stiff PINN problems, particularly those where the PDE residual and boundary conditions exhibit highly unbalanced sensitivities.

alg:4:1:NTK

Algorithm 6 Adaptive Weights for Physics-Informed Neural Networks [WYP22]

Require: θ_0 (initial parameters), η_θ (learning rate), S (number of iterations)

1: Initialise $\lambda_b \leftarrow 1$, $\lambda_r \leftarrow 1$

2: **for** $n = 1$ to S **do**

3: 1. Compute the NTK matrix $K(n)$ and its blocks $K_{uu}(n)$ and $K_{rr}(n)$

4: 2. Compute eigenvalues $\{\lambda_i(n)\}$ of $K(n)$

5: 3. Compute eigenvalues $\{\lambda_i^{uu}(n)\}$ of $K_{uu}(n)$

6: 4. Compute eigenvalues $\{\lambda_i^{rr}(n)\}$ of $K_{rr}(n)$

7: 5. Update adaptive weights; Compute λ_b and λ_r by

$$\lambda_b = \frac{\sum_{i=1}^{N_r+N_b} \lambda_i(n)}{\sum_{i=1}^{N_b} \lambda_i^{uu}(n)} = \frac{\mathrm{Tr}(K(n))}{\mathrm{Tr}(K_{uu}(n))} \tag{4.4}$$

$$\lambda_r = \frac{\sum_{i=1}^{N_r+N_b} \lambda_i(n)}{\sum_{i=1}^{N_r} \lambda_i^{rr}(n)} = \frac{\mathrm{Tr}(K(n))}{\mathrm{Tr}(K_{rr}(n))} \tag{4.5}$$

8: where $\lambda_i(n), \lambda_i^{uu}$ and $\lambda_i^{rr}(n)$ are eigenvalues of $K(n), K_{uu}(n), K_{rr}(n)$ at n-th iteration.

9: 6. Form the weighted loss:

$$\mathcal{L}(\theta_n) = \lambda_b \mathcal{L}_b(\theta_n) + \lambda_r \mathcal{L}_r(\theta_n)$$

10: 7. Update parameters via gradient descent:

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta \mathcal{L}(\theta_n)$$

4.2 Advanced Schemes

4.2.1 Sobolev training

Sobolev training, also known as gradient-enhanced training, is an advanced technique that improves the learning of PDE solutions by incorporating derivative information directly into the loss function. The idea is to measure the discrepancy between the predicted solution $\hat{u}_\theta$ and the true solution u not only in an L^p sense, but in a Sobolev norm $W^{p,q}$ (Section 1.3), which also controls derivatives up to order q.

The use of Sobolev norms in neural network training predates PINNs. A foundational contribution is [Cza+17], who introduced *Sobolev Training* for general neural networks. Their motivation was to stabilize training and improve generalization by enforcing agreement not only on function values but also on

derivatives. This idea proved particularly effective in tasks where the target function exhibits sharp variations or where derivative information is available or can be estimated.

The transition of these ideas to physics-informed learning was natural. Since PDEs explicitly involve derivatives, enforcing residuals in Sobolev norms provides a principled way to incorporate the structure of the differential operator into the training process. This led to the development of Sobolev or gradient-enhanced PINNs [Son+21], where the PDE residual and its derivatives are minimized simultaneously.

In standard PINN training, the PDE loss enforces the residual in an L^p norm. Sobolev training generalizes this by defining

$$\mathcal{L}_{PDE} = \|\hat{u}_\theta - u\|_{W^{p,q}(\Omega)} \ ,$$

where

$$\|u\|_{W^{p,q}(\Omega)} = \begin{cases} \left(\sum_{|\alpha|\leq q} \|D^\alpha u\|^p_{L^p(\Omega)}\right)^{\frac{1}{p}}, & 1 \leq p < \infty, \\ \max_{|\alpha|\leq q} \|D^\alpha u\|_{L^p(\Omega)}, & p = \infty, \end{cases}$$

and $D^\alpha u$ denotes the partial derivative of multi-index order α.

In practice, this means that the PINN loss includes not only the PDE residual but also its spatial derivatives:

$$\mathcal{L}_{Sobolev} = \mathcal{L}_{PDE} + \sum_{|\alpha|=1}^{q} \lambda_\alpha \|D^\alpha \mathcal{F}[\hat{u}_\theta]\|^2_{L^2(\Omega)} .$$

In [Son+21], authors found that Sobolev training may lead to faster convergence, and furthermore it may be beneficial for solving high-dimensional PDEs.

However, care must be taken when choosing the relative weights of the additional loss terms with respect to the standard PDE residual and boundary condition loss terms; indeed, Sobolev training may even adversely affect the training convergence and accuracy. Furthermore, Sobolev or gradient-enhanced training increases the training time as the differentiation order will be increased and thus extra backpropagation will be required.

4.2.2 Quasi-random sampling

The choice of collocation points plays a central role in the performance of PINNs. Since the PDE loss is a quadrature approximation of a continuous functional, the sampling strategy directly affects the estimation error $\mathcal{E}_G$. A common baseline is uniform random sampling, which corresponds to Monte Carlo quadrature. While simple and dimension-independent, random sampling suffers from clustering and large gaps, which lead to poor coverage of the domain and slow convergence.

Quasi-random sampling methods originate from the theory of Quasi-Monte Carlo (QMC) integration, where the goal is to approximate high-dimensional integrals using deterministic point sets with low discrepancy. Classical Monte Carlo sampling achieves an error rate of order $\mathcal{O}(N^{-1/2})$, independent of dimension, but suffers from clustering and irregular coverage. In contrast, low-discrepancy sequences such as Halton, Sobol, and R_2 achieve convergence rates of order $\mathcal{O}(N^{-1}(\log N)^d)$ under mild regularity assumptions [Lem09; Nie92].

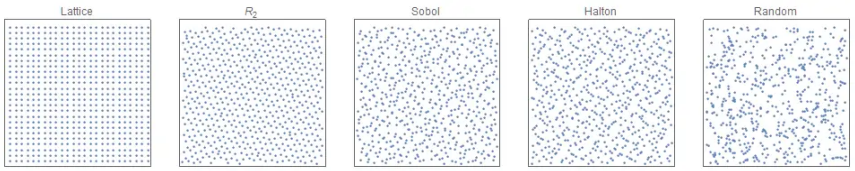


Figure 4.3: Comparison of a regular lattice (left) with 3 different quasirandom functions (middle), and a simple random distribution (right). Notice that the quasirandom distributions appear less regular than a lattice but do not have as many 'clumps' or 'gaps' as the random distribution. From [Vis19]

fig:4:2: Sampling_ strategies

These sequences fill the domain more uniformly, leading to more accurate quadrature and reduced variance. Their use in PINNs is therefore natural, since the PINN loss is itself a quadrature approximation of a continuous PDE residual. A visual comparison of different sampling strategies is shown in Figure 4.3.

Halton sequences: One of the classical low-discrepancy sequences is the Halton sequence [Hal60]. It is based on the one-dimensional van der Corput sequence, defined using the radical inverse function. Let us now denote the N prime numbers as m_1, m_2, ..., m_N. The Halton sequence for the $[0,1]^N$ dimensional space is

$$x_n = \Big(\Phi_{m_1}(n), \ldots, \Phi_{m_i}(n), \ldots, \Phi_{m_N}(n)\Big) \tag{4.6}$$

where $\Phi_{m_i}(n)$ is the i-th *radical inverse function*

$$\Phi_{m_i}(n) \coloneqq \sum_{j=0}^{l(i)} a_j(i,n) m_i^{-(j+1)} \tag{4.7}$$

In the above equation, a_j is an integer coefficient, and it is given as $a_j(i,n) \in [0, m_i - 1]$. Also, the integer terms n and l are given by

$$n = \sum_{j=0}^{l(i)} a_j(i,n) m_i^j, \qquad l(i) = \lceil \log_{m_i} n \rceil \,. \tag{4.8}$$

The samples generated using the Halton sequence are only bounded in between $[0,1]^N$ which are spread more evenly within the design space compared to random sampling.

Sobol sequences: Sobol sequences [JK08; Sob67] are another widely used family of low-discrepancy sequences, based on direction numbers and binary expansion. Their construction ensures good equidistribution properties across all projections, making them suitable for high-dimensional PDEs.

The samples are drawn from a special binary fraction $(v_i^{(j)})$ of length w where $i = 1, 2, \ldots, w$ and $j = 1, 2, \ldots, N$ and where N is the space dimension. The numbers $v_i^{(j)}$ are known as *direction numbers*. To generate direction numbers of dimension j, a primitive polynomial over the field $\mathbb{F}_2$ with elements {0,1} is considered, which is expressed as

$$p_j(x) = x^q + b_1 x^{q-1} + \cdots + b_{q-1} x + 1 \tag{4.9}$$

Thus, the direction numbers, $v_i^{(j)}$ are generated using the following recurrence relation

$$v_i^{(j)} = b_1 v_{i-1}^{(j)} \oplus b_2 v_{i-2}^{(j)} \oplus \cdots \oplus b_{q-1} v_{i-(q-1)}^{(j)} \oplus v_{i-q}^{(j)} \oplus \left(v_{i-q}^{(j)}/2^q\right); \quad i > q\,, \tag{4.10}$$

where $\oplus$ represents the bitwise XOR. Finally, in the dimension j, the Sobol sequence is written as

$$x_n^{(j)} = a_1 v_1^{(j)} \oplus a_2 v_2^{(j)} \oplus \cdots \oplus a_w v_w^{(j)} \tag{4.11}$$

where $n = \sum_{i=0}^{w} a_i 2^i$. The coefficient a_i is the random number in between $\{0,1\}$.

Hammersley point set: The Hammersley point set [Ham60] is a finite low-discrepancy design closely related to the Halton sequence. For a given number of points N and dimension d, the Hammersley set is defined as

$$x_n = \left(\frac{n}{N}, \Phi_{m_1}(n), \ldots, \Phi_{m_i}(n), \ldots, \Phi_{m_{N-1}}(n)\right)$$

where $\Phi_{m_1}(n)$ are the radical inverse functions in pairwise coprime bases defined above. The Hammersley sequence differs from the Halton sequence in that the Hammersley sequence employs (N-1)-Φ sequence; furthermore, while the Halton sequence can generate an infinite number of samples (i.e., infinite N), the Hammersley sequence requires an upper bound on the number of samples. The key idea of Hammersley sequence is that the first coordinate is perfectly stratified (uniform grid), while the remaining coordinates use van der Corput sequences. This hybrid structure yields low discrepancy for finite N and is often used in design of experiments and numerical integration [Ham60].

Faure sequence: The Faure sequence [Fau82] is another classical low-discrepancy sequence based on permutations of digits in a fixed prime base.

Let m is the first prime number such that $m \geq n$, where n is the total number of samples, and let

$$c_{ij} = \binom{i}{j} \bmod m\,, \qquad 0 \leq j \leq i \leq p$$

This implies $\left\{c_{ij} - \binom{i}{j}\right\}$ is a multiple of m. Thus, the base m representation of n is expressed as

$$n = \sum_{i=0}^{p-1} a_i(n) m^i\,, \tag{4.12}$$

where the coefficients $a_i \in [0, m)$ takes integer values. The samples produced by the Faure sequence are denoted by

$$x_n = \sum_{j=0}^{p-1} a'_j(n) m^{-(j+1)} \tag{4.13}$$

and where

$$a'_j = \sum_{l=j}^{p-1} c_{lj} a_l(n) \mod p\,, \qquad j \in \{0, 1, \ldots, p-1\}.$$

The upper bound of the sample size for the Faure sequence is m^p. This is the main difference between the Faure and the Halton sequences. The main advantage of using Faure sampling over Halton sampling is that the Faure sequence is faster to fill the gaps for high dimensional problems. Also, it prevents correlation problems in high dimensions, which are seen in the Halton sequence.

The R_d sequence: The R_d sequence, introduced in [Vis19], is a modern low-discrepancy sequence in d dimensional spaces designed to improve uniformity and reduce correlation artifacts present in classical sequences. It is based on irrational rotations on the unit square and exhibits excellent performance in practice. The sequence is defined by

$$\mathbf{t}_n = (\{n\alpha_1\}, \{n\alpha_2\}, \cdots, \{n\alpha_n\},)$$

where $\{\cdot\}$ denotes the fractional part, and where

$$\alpha_k = \varphi_d^{-k}\,, \qquad k = 1, \ldots, n,$$

while φ_d is the solution of the $d-$generalised golden number, i.e. the positive root of

$$x^{d+1} = x + 1.$$

Empirical studies show that the R_d sequence often outperforms Halton and Sobol sequences in terms of discrepancy and visual uniformity, especially in higher dimensions.

Latin Hypercube Sampling (LHS): Latin Hypercube Sampling [MBC79] is a stratified sampling technique that ensures each coordinate direction is uniformly covered. In d dimensions, the domain is divided into N intervals along each axis, and points are sampled such that each interval is used exactly once per dimension. LHS is not strictly low-discrepancy, but it significantly reduces variance compared to random sampling and is widely used in design of experiments. For each dimension j, one draws a random permutation π_j of $\{1, \ldots, N\}$ and defines the sample points as

$$x_n^{(j)} = \frac{\pi_j(n) - U_{n,j}}{N}, \qquad n = 1, \ldots, N,$$

where $U_{n,j} \sim U(0,1)$ are independent uniform random variables. This guarantees that each interval is used exactly once per dimension, reducing variance compared to pure Monte Carlo sampling [MBC79].

Impact on PINN performance: The effect of sampling strategies on PINN accuracy has been systematically studied in [DT22]. Their results, summarized in Figure 4.4, show that quasi-random sequences such as Sobol, Halton,

alg:4:2:
LatinHypercube

Algorithm 7 Pseudo-code for generation of basic Latin Hypercube Sampling (from [DT22])

1: Aim: Generate n samples over the N-dimensional space.
2: Let x_{ij} is the j-th coordinate of the i-th sample, x_i.
3: **Begin**
4: **for** $j = 1$ to N **do**
5: Generate P_j, a random permutation of the set {1, ..., n}.
6: **for** $j = 1$ to N **do**
7: **for** $i = 1$ to n **do**
8: Generate $x_{ij} = \frac{P_j(i) - U_j(i)}{n}$,

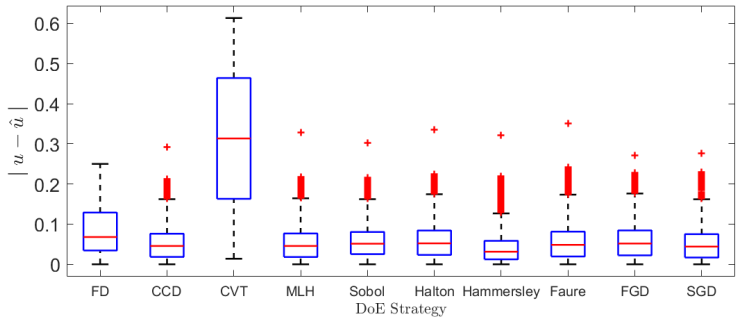


fig:4:2:
Sampling_res

Figure 4.4: Absolute error between original and predicted responses of Heat equation considering different DoE strategies. From [DT22].

Hammersley, and Faure consistently outperform purely random or deterministic designs in terms of prediction error. The improved uniformity of quasi-random sampling leads to more accurate quadrature of the PDE residual, reducing the estimation error $\mathcal{E}_G$ and improving stability during training.

In particular, the study highlights that quasi-random sequences yield lower median error and fewer outliers, and it emphasises the importance of choosing an appropriate sampling strategy in order to obtain accurate predicted responses using a physics-informed neural network.

subsec:4:2:3:
ImportanceSampling

4.2.3 Importance sampling

As we have seen, the standard formulation of PINNs, collocation points are drawn from a uniform distribution over the computational domain. This corresponds to a Monte Carlo approximation of the continuous loss functional and directly influences the estimation error $\mathcal{E}_G$. Uniform sampling is simple and dimension independent, but it is often inefficient when the PDE solution contains sharp gradients, boundary layers, or localized features. In such situations, many points fall in regions where the residual is already small, while too few points are allocated where the solution is difficult to approximate. This imbalance leads to a poor approximation of the continuous risk and slows down convergence.

Importance sampling provides a principled alternative. Instead of sampling from a uniform density $f(\mathbf{x})$, one draws points from a different distribution $q(\mathbf{x})$ that concentrates samples in regions where the PDE residual is expected to be

Please notice that in Section 1.4 we have discussed Monte Carlo integration with generic pdf(x).

large. The loss is then re-weighted to preserve an unbiased estimator of the continuous objective, following the classical importance sampling identity. In the context of PINNs, this leads to the modified empirical risk [NGM21])

$$\theta^* \approx \arg\min_\theta \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{q(\mathbf{x}_i)}\,\ell(\theta;\mathbf{x}_i), \qquad \mathbf{x}_i \sim q(\mathbf{x}),$$

where f is the uniform pdf, and q is the recomputed pdf. The key idea is that the sampling distribution should reflect the local difficulty of the PDE, so that the quadrature approximation of the loss becomes more accurate where it matters most.

The authors suggest, as a means of computing the *importance sampling* pdf, either:

(a) based on results on deep learning [Ala+15; KF18], where authors shown that, in principle, the rate of convergence is maximised if the training samples are drawn with a pdf proportional to the relative magnitude of the Loss gradients,

$$q_j^{(n)} \approx \frac{\|\nabla_{\theta^{(n)}}\ell[\theta^{(n)};x_j]\|_2}{\sum_{j=1}^{N}\|\nabla_{\theta^{(n)}}\ell[\theta^{(n)};x_j]\|_2},$$

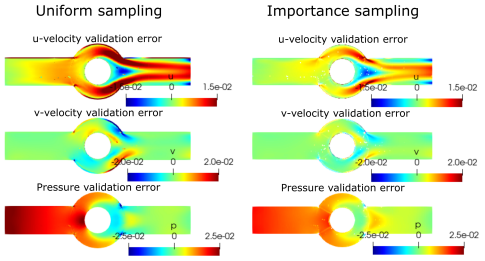
where ℓ are the residuals, n is the epoch, and j is the sample index. However, this formula requires computing the 2-norm of this gradient for all of the collocation point at each iteration, and thus requires extra backpropagations through the computational graph, which can be computationally expensive be slow and computationally inefficient.

(b) A simplified version, relying on the results of [KF17], where it was shown theoretically and numerically that a linear transformation of the loss value at a training example is always greater than the 2-norm of loss gradient at that. For this, the pdf is related directly to the (normalised) residuals:

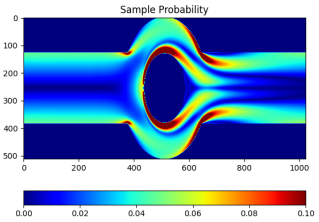
$$q_j^{(n)} \approx \frac{\ell[\theta^{(n)};x_j]}{\sum_{j=1}^{N}\ell[\theta^{(n)};x_j]}.$$

Furthermore, [NGM21]) introduces an adaptive strategy for constructing the sampling distribution Algorithm 8. After a short training phase, the PDE residual is evaluated on a candidate set of points, and the sampling density is updated to be proportional to a power of the residual magnitude. This procedure is repeated periodically during training. The resulting distribution allocates more points to regions where the residual is large, improving the approximation of the continuous loss and accelerating convergence. The authors show that this adaptive refinement leads to a more accurate and stable training process, particularly for problems with localized features or multiscale behaviour.

Although importance sampling introduces additional computational overhead due to the need to estimate and update the sampling distribution, it provides a systematic way to reduce the estimation error $\mathcal{E}_G$ and to improve the robustness of PINN training. It is compatible with any PINN architecture or optimizer and complements other advanced sampling strategies such as quasi-random sequences or residual-based adaptive sampling. For PDEs with strong



(a) A comparison between the uniform and importance sampling validation error results for stationary navier stokes in an annular ring geometry. From [Nvi26].



(b) A visualization of the training point sampling probability at iteration 100K for the annular ring example. From [Nvi26].

fig:4:2: ImportanceSamplingAR

Algorithm 8 Efficient training of PINNs via importance sampling. From [NGM21]

alg:4:2: Importance_sampling

1: Generate N collocation points $\{t_j, \mathbf{x}_j\}_{j=1}^N$ sampled from $[0,T] \times \mathcal{D}$, n boundary points $\{\underline{\mathbf{x}}_j\}_{j=1}^N$ sampled from $\partial\mathcal{D}$, and S seeds $\{t_s, \mathbf{x}_s\}_{s=1}^S$ sampled from $[0,T] \times \mathcal{D}$.
2: For each collocation point, find the nearest seed.
3: Set the model architecture (number of layers, dimensionality of each layer, and nonlinearities). Also specify optimizer hyper-parameters, λ_1, λ_2, batch size m, and error tolerance ϵ.
4: Initialize model parameters $\theta^{(0)}$.
5: **while** $J(\theta) > \epsilon$ **do**
6: Compute $\{J(\theta^{(i)}; \mathbf{x}_s)\}_{s=1}^S$.
7: Compute $\tilde{q}_j^{(i)} = J(\theta^{(i)}; \mathbf{x}_{\rho(j)}) / \sum_{j=1}^N J(\theta^{(i)}; \mathbf{x}_{\rho(j)})\ \forall j \in \{1, \cdots, N\}$.
8: Select a batch of collocation points according to $\mathbf{p}^{(i)} = \{\tilde{q}_1^{(i)}, \cdots, \tilde{q}_N^{(i)}\}$.
9: $\theta^{(i+1)} = \theta^{(i)} - \frac{\eta^{(i)}}{mN} \sum_{j \in M^{(i)}} \frac{1}{\tilde{q}_j^{(i)}} \nabla_\theta J(\theta^{(i)}; \mathbf{x}_j)$.

spatial variability, it seems that importance sampling may offer a significant improvement over uniform sampling and may represent one of the most effective ways to focus computational effort where the PDE is hardest to satisfy.

In Section 4.4 we will see how similar approaches involving some kind of "loss re-weighting" are introduced and generalised.

4.2.4 Approximate distance functions and R-functions for exact boundary conditions

We have seen that, in the PINN framework, the boundary/initial conditions are imposed, as for the PDE, as a weak enforcement via penalty terms in the loss, which is a common source of error and instability: the trained network need not satisfy $\mathcal{B}[u] = g$ on $\partial\Omega$ exactly, and near-boundary errors may pollute the interior solution. A constructive remedy is to build a geometry-aware trial space so that Dirichlet data are satisfied by design [SS22]. Let $g : \partial\Omega \to \mathbb{R}$ denote prescribed boundary data and $\widehat{u}_\theta$ a neural network. If $\phi : \overline{\Omega} \to \mathbb{R}$ is a scalar field with $\phi|_{\partial\Omega} = 0$ and appropriate normal derivative behaviour on $\partial\Omega$,

then the ansatz

$$u(\mathbf{x}) = g(\mathbf{x}) + \phi(\mathbf{x})\,\widehat{u}_\theta(\mathbf{x})$$

enforces $u|_{\partial\Omega} = g$ exactly for any choice of $\widehat{u}_\theta$. The function ϕ is commonly called an *approximate distance function* (ADF): it vanishes on the boundary, is sign-consistent with the interior/exterior of Ω, and is sufficiently smooth so that automatic differentiation yields stable PDE residuals in the interior. The construction of ϕ for complex geometries is most conveniently performed using the algebra of **R-functions**, which provides smooth, differentiable analogues of Boolean set operations on implicit functions and preserves zero level sets under composition [Rva75; Sha07; Sha91].

We will re-encounter ADFs in Section 4.4.

R-functions realize logical operations on level-set representations. Given two implicit scalar functions $r_A(\mathbf{x})$ and $r_B(\mathbf{x})$ whose zero level sets define primitive subdomains A and B, one may form smooth approximations of intersection and union that retain the zero level set structure. Two frequently used binary R-function formulas are

$$\mathcal{R}_\cap(r_A, r_B) \;=\; r_A + r_B - \sqrt{r_A^2 + r_B^2}, \qquad \mathcal{R}_\cup(r_A, r_B) \;=\; r_A + r_B + \sqrt{r_A^2 + r_B^2},$$

where the square root may be regularized as $\sqrt{r_A^2 + r_B^2 + \varepsilon}$ to control differentiability and numerical stability. These expressions are smooth, sign-preserving, and satisfy

$$\mathcal{R}_\cap(r_A, r_B) = 0 \quad\Longleftrightarrow\quad r_A = 0 \text{ or } r_B = 0 \quad \text{on the boundary of } A \cap B,$$

so that the zero level set of the composed function represents the boundary of the composed domain. By iteratively composing primitives with R-function operators one obtains a single implicit function r_Ω whose sign encodes membership in Ω and whose zero level set approximates $\partial\Omega$ with controlled smoothness [Rva75; Sha91].

To convert an implicit representation r_Ω into an ADF ϕ that vanishes on $\partial\Omega$ and has a prescribed normal derivative, one may apply a smooth monotone transform and a local regularization. A practical choice is

$$\phi(\mathbf{x}) = \frac{r_\Omega(\mathbf{x})}{\sqrt{r_\Omega(\mathbf{x})^2 + \delta^2}}\,,$$

with $\delta > 0$ small; this yields $\phi \to \operatorname{sign}(r_\Omega)$ away from the boundary while ensuring differentiability near $\partial\Omega$. If a unit normal derivative is desired on the boundary, one can rescale ϕ locally or apply a further normalization so that $\nabla_n \phi|_{\partial\Omega} = 1$ in a weak or pointwise sense. Signed-distance functions and level-set primitives (for circles, rectangles, half-spaces, etc.) are natural building blocks for r_Ω; standard references on level-set methods provide practical formulas and numerical techniques for these primitives [OFP04].

The PINN workflow with an ADF then proceeds by:

(i) constructing primitive implicit functions r_i for simple geometric pieces;

(ii) composing them with R-function operators to obtain r_Ω;

(iii) regularizing and transforming r_Ω into ϕ;

(iv) training $\widehat{u}_\theta$ using the interior PDE residual only, since Dirichlet conditions are satisfied exactly by construction.

In practice one implements the composition and regularization symbolically or numerically so that ϕ is available as a differentiable function compatible with automatic differentiation frameworks. The use of an ADF reduces the effective search space of the PINN, improves conditioning near $\partial\Omega$, and eliminates the need for Dirichlet penalty terms in the loss; these advantages are demonstrated empirically in [SS22], who provide algorithmic recipes and numerical examples for a variety of two-dimensional geometries.

For clarity, a compact algorithmic description of the ADF construction and PINN integration is given below in pseudocode form (the algorithm is intended as a high-level recipe rather than a line-by-line implementation):

Algorithm 9 Construct ADF and train PINN with exact Dirichlet conditions

1: Define primitive implicit functions $r_i(\mathbf{x})$ for geometric primitives (circles, half-spaces, rectangles, ...).
2: Compose primitives using R-function operators to obtain $r_\Omega(\mathbf{x})$ representing Ω.
3: Regularize and transform: $\phi(\mathbf{x}) \leftarrow r_\Omega(\mathbf{x})/\sqrt{r_\Omega(\mathbf{x})^2 + \delta^2}$.
4: Define trial solution $u(\mathbf{x}) = g(\mathbf{x}) + \phi(\mathbf{x})\,\widehat{u}_\theta(\mathbf{x})$.
5: Minimize the interior PDE residual $\|\mathcal{F}[u]\|_{L^2(\Omega)}$ with respect to θ using automatic differentiation; omit Dirichlet penalty terms.

Implementation notes: when primitives or R-function formulas involve non-differentiable operations (e.g., max, min), replace them with smooth approximations or use analytic signed-distance expressions to preserve differentiability. The regularization parameter δ must be chosen small enough to keep ϕ close to a signed-distance near the boundary but large enough to avoid numerical underflow in gradients; corners and cusps require special care and may benefit from local smoothing or mesh-based refinement.

4.2.5 Curriculum learning

Curriculum learning adapts the training schedule so that the model is first exposed to simpler instances of the task and only later to the full, hard problem. The idea, originally formalized in the machine-learning literature [Ben+09], is to shape the optimization trajectory: early training on easy examples guides the parameters into basins of attraction from which the final, more difficult task can be learned more reliably (for nice recent surveys on the topic, see [Sov+22; WCZ21b]). In the context of PINNs this paradigm is natural because many PDEs admit families of problems parametrised by physical coefficients, forcing terms, and/or boundary/initial data; by starting from a benign parameter regime and progressively moving toward the target regime, one can reduce both the estimation error and the optimization error.

A canonical example where, despite the semplicity of the problem, vanilla PINNs may experience training difficulties, is the linear continuity (advection)

equation on $[0,1] \times [-1,1]$ (see [MA23]),

$$\begin{aligned} \partial_t u(t,x) + \beta\, \partial_x u(t,x) &= 0, \\ u(0,x) &= -\sin(\pi x), \\ u(t,\pm 1) &= 0, \end{aligned}$$

whose exact solution, easily obtainable by the method of characteristics, is

$$u(t,x) = -\sin(\pi(x-\beta t)).$$

As β increases the solution becomes increasingly oscillatory in space and time, and standard PINN training may fail to converge for large β because the residual landscape becomes highly nonconvex and gradients become ill-conditioned. Curriculum learning addresses this by initializing training at a small velocity β_0 and gradually increasing β to the target value, or by using a sequence of intermediate tasks that interpolate between smooth and oscillatory regimes. Empirical studies show that such schedules reduce training time and improve final accuracy for advection-dominated problems [Kri+21; MA23].

Curriculum strategies for PINNs can be implemented in several ways: (i) parameter continuation, where a physical parameter is ramped according to a predefined schedule or an adaptive rule; (ii) data curriculum, where collocation points or boundary/initial data are introduced progressively (for instance, coarse-to-fine in time or space); and (iii) loss curriculum, where loss terms are weighted dynamically so that easier constraints dominate early training. These variants are complementary and can be combined: for example, one may first train on a low-β problem with coarse sampling, then increase β while refining the collocation set and rebalancing loss weights.

From an optimization perspective, curriculum learning acts as a form of informed initialization or regularization: by steering parameters toward favourable regions of the loss landscape, it reduces the probability of becoming trapped in poor local minima and mitigates gradient pathologies. The approach is particularly effective when the family of easier problems is chosen so that their solutions are continuously connected to the target solution; in practice, simple linear schedules or a small number of intermediate steps are often sufficient to obtain substantial improvements [Ben+09; Kri+21].

4.3 Architectures

We have introduced PINN as a general training recipe, without any specification on the model architectures; apparently, everything that is approximating a function, and whose parameter can be learned by solving the multi-task problem, can do the job. This is in principle true; nevertheless, as we will see, in the PINN practice some intervention on the model architecture may be required, if not even *necessary*. In [Cuo+22], authors list a set of standard architectures used in PINNs: fully-connected models, convolutional models, Long-Short Term Memory (LSTM), as well as Bayesian Neural Networks and Generative Adversarial Neural Networks (see Table 4.1), are all have been used to tackle PINNs.

Nevertheless, empirical evidence has suggested that PINNs may need special care when discussing neural network architectures; for example, their *sizes*: as

NN family	NN type	Papers
FF-NN	1 layer / EML	[DS20], [Sch+21]
	2-4 layers	32 neurons/layer [He+20] 50 neurons/layer [Tar+20]
	5-8 layers	250 neurons/layer [ZLY21]
	9+ layers	[CZ21] [Wah+21]
	Sparse	[RR21]
	multi FC-DNN	[Ami+21] [Isl+21]
CNN	plain CNN	[GSW21] [Fan21]
	AE CNN	[Zhu+19], [GZ20] [WCZ21a]
RNN	RNN	[Via+21]
	LSTM	[ZLS20] [YV21]
Other	BNN	[YMK21]
	GAN	[YZK20]

Table 4.1: A list of standard neural network architecture, such as Feedforward neural networks (FFNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNN), utilised in PINN implementations. A publication is reported for each type that either used this type of network first or best describes its implementation. From [Cuo+22]

tab:4:3: NN4PINNv1

already noted in [Cuo+22], and more broadly discussed in [Row26], most of the time, when designing model for PINNs, it is more convenient to use *small* networks.

Furthermore, when dealing with Physics-Informed problem we easily encounter a less known issue with deep learning models: the *spectral bias*, which we will discuss in depth in Section 4.3.3. This issue is relevant also in other fields [Liu+24a], most notably in the context of **Implicit Neural Representations** (INR) [Sit+20], which serve as the foundational building blocks for neural scene representations, where INR are designed to learn continuous functions using a multilayer perception (MLP) that maps coordinates to visual signals, such as images, videos, and 3D scene.

4.3.1 Adaptive Activations and Weight Factorisation

4.3.1.1 Adaptive Activations

subsubsec:4:3:1:1:AdaptiveAct

While designing the neural networks to be used to tackle the differential problem, [JKK20], in addition to leaning the linear transformations, suggest to also learn the non-linear transformations to potentially improve the convergence as well as the accuracy of the model, by using the global adaptive activation functions.

Global adaptive activations consist of a single trainable parameter that is multiplied by the input to the activations in order to modify the slope of activations:

$$\mathcal{N}^{(\ell)}\left(H^{(\ell-1)};\theta^{(\ell)},a\right)=\sigma\left(a\,L^{(\ell)}_{\theta^{(\ell)}}\left(H^{(\ell-1)}\right)\right), \tag{4.14}$$

where $\mathcal{N}^{(\ell)}$ the non-linear transformation at layer ℓ, $H^{(\ell-1)}$ is the output of the $(\ell-1)-$hidden layer, $\theta^{(\ell)}$ is the set of weights and biases of layer ℓ, a is the global adaptive activation parameter, which is one *per-network*, σ is the activation function, and $L^{(\ell)}:x\mapsto W^{(\ell)}x+b^{(\ell)}$ is the linear transformation at layer ℓ.

Similar to the network weights and biases, the global adaptive activation parameter a is also a trainable parameter which needs to be optimised during training.

The intuition of introducing a scalable hyper-parameter in the activation function, which has to be optimised, is that it changes dynamically the topology of the loss function involved in the optimization process. Empirically, in [JKK20] that the adaptive activation function has better learning capabilities with respect to the fixed activation as it improves the convergence rate, especially at early training, as well as the solution accuracy.

Remark 4.3.1. One may think that this is a trivial setting. Indeed, we know since our 101 MLP course that composition of affine transformations are affine transformations, which is the reason we put the non-linear activation after each affine transformations, so that we can have a truly non-linear universa approximator.

And that remains true. What changes here is the way we *parametrise* our neural network. We are tuning down the norm of our parameters $\|\theta\|$ with a global, learnable a, which can tune the *strength* of the training.

Indeed, what we are doing is reparametrising our neural network in a way that:

(i) it *globally* alters the gradient descent, in a *learnable way*:

$$\delta\theta=-\eta\,a\,\nabla_\theta\mathcal{L}^{(\text{unscaled})}[\theta]\,;$$

(ii) it *globally* scales the Neural Tangent Kernel of the network f_θ, under action of the a rescaling:

$$K(\theta;x,x')\coloneqq\nabla_\theta f(x)\cdot\nabla_\theta f(x')\,,$$

so that

$$f(\theta,\cdot)\mapsto f(\theta,a;\cdot)\Rightarrow K\mapsto a^2K\,,$$

the NTK eigenvectors are unchanged, but their eigenvalues are scaled, in a *learnable* way.

(iii) It rescales the curvature of the loss landscape, since the *Hessian* of the Loss

$$H_{ij} \coloneqq \frac{\partial^2 \mathcal{L}[\theta]}{\partial\theta_i \partial\theta_j}\,,$$

get the same scaling behaviour

$$f(\theta,\cdot) \mapsto f(\theta, a;\cdot) \Rightarrow H \mapsto a^2 H\,.$$

4.3.1.2 Random Weight Factorisation

4:2:1:2: WeightFactor

Another similar approach was outlined in [Wan+22], where authors introduced the concept of *Random Weight Factorisation*.

Let $\mathbf{x} \in \mathbb{R}^d$ be the input, $g^{(0)}(\mathbf{x}) = \mathbf{x}$ and $d_0 = d$. They define a standard multi-layer Perceptron (MLP) $f_\theta(\mathbf{x})$ recursively defined by

$$f_\theta^{(l)}(\mathbf{x}) = W^{(l)} \cdot g^{(l-1)}(\mathbf{x}) + b^{(l)}, \quad g^{(l)}(\mathbf{x}) = \sigma(f_\theta^{(l)}(\mathbf{x})), \quad l = 1, 2, \ldots, L, \tag{4.15}$$

with a final layer

$$f_\theta(\mathbf{x}) = W^{(L+1)} \cdot g^{(L)}(\mathbf{x}) + b^{(L+1)}, \tag{4.16}$$

where $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix in l-th layer and σ is an element-wise activation function. Here, the ensemble of trainable parameters are $\theta = \left(W^{(1)}, b^{(1)}, \ldots, W^{(L+1)}, b^{(L+1)}\right)$.

MLPs are commonly trained by minimizing an appropriate loss function $\mathcal{L}(\theta)$ via gradient descent. To improve convergence, we propose to factorize the weight parameters associated with each neuron in the network as follows

$$w^{(k,l)} = s^{(k,l)} \cdot v^{(k,l)}, \quad k = 1, 2, \ldots, d_l, \quad l = 1, 2, \ldots, L+1, \tag{4.17}$$

{eq:4:3:1:2: neuron_fact}

where $w^{(k,l)} \in \mathbb{R}^{d_{l-1}}$ is a weight vector representing the k-th row of the weight matrix $W^{(l)}$, $s^{(k,l)} \in \mathbb{R}$ is a trainable scale factor assigned to each individual neuron, and $v^{(k,l)} \in \mathbb{R}^{d_{l-1}}$. Thus, the proposed weight factorization can be rewritten as

$$W^{(l)} = \operatorname{diag}(s^{(l)}) \cdot V^{(l)}, \quad l = 1, 2, \ldots, L+1. \tag{4.18}$$

with $s \in \mathbb{R}^{d_l}$.

A way to interpret the relevance of this approach is to look its associated gradient updates. We recall that a standard gradient descent update with a learning rate η takes the form

$$\mathbf{w}_{n+1}^{(k,l)} = \mathbf{w}_n^{(k,l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_n^{(k,l)}}. \tag{4.19}$$

In [Wan+22], they prove the following

Theorem 4.3.2. *Under the weight factorization of* (4.17)*, the gradient descent update is given by*

$$\mathbf{w}_{n+1}^{(k,l)} = \mathbf{w}_n^{(k,l)} - \eta \left(\|[s_n^{(k,l)}]^2 + \mathbf{v}_n^{(k,l)}\|_2^2 \right) \frac{\partial \mathcal{L}}{\partial \mathbf{w}_n^{(k,l)}} + \mathcal{O}(\eta^2), \tag{4.20}$$

for $l = 1, 2, \ldots, L+1$ *and* $k = 1, 2, \ldots, d_l$.

We notice that the weight factorization $\mathbf{w} = s \cdot \mathbf{v}$ re-scales the learning rate of $\mathbf{w}$ by a factor of $(s^2 + \|\mathbf{v}\|_2^2)$. Since $\mathbf{s}, \mathbf{v}$ are both trainable parameters, this suggests that weight factorization effectively assigns a self-adaptive learning rate to each neuron in the network.

It is worth pointing out that the weight factorization is the generalisation of the previously introduced Adaptive Activation Section 4.3.1.1.

A major difference is the sampling approach. The random weight factorization is applied as follows; we first initialize the parameters of an MLP network via the Glorot scheme [GB10]. Then, for every weight matrix $\mathbf{W}$, we proceed by initializing a scale vector $\exp(\mathbf{s})$ where $\mathbf{s}$ is sampled from a multivariate normal distribution $\mathcal{N}(\mu, \sigma \mathrm{I})$. Finally, every weight matrix is factorized by the associated scale factor as $\mathbf{W} = \mathrm{diag}(\exp(\mathbf{s})) \cdot \mathbf{V}$ at initialization. We train this network by gradient descent on the new parameters $\mathbf{s}, \mathbf{V}$ directly. See Algorithm 10.

alg:4:3:1:2: RandomWeightAlg

Algorithm 10 Random weight factorization (RWF). From [Wan+22].

1. Initialize a neural network f_θ with $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1}$ (e.g. using the Glorot scheme).

for $l = 1, 2, \ldots, L$ **do**

(a) Initialize each scale factor as $\mathbf{s}^{(l)} \sim \mathcal{N}(\mu, \sigma I)$.

(b) Construct the factorized weight matrices as $\mathbf{W}^{(l)} = \mathrm{diag}(\exp(\mathbf{s}^{(l)})) \cdot \mathbf{V}^{(l)}$.

2. Train the network by gradient descent on the factorized parameters $\{\mathbf{s}^{(l)}, \mathbf{V}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1}$.

The recommended hyper-parameters are $\mu = 1.0, \sigma = 0.1$.

Listing 4.1: Torch implementation of Random Weight Factorisation

```python
import torch
import torch.nn as nn
import math

def factorized_glorot_normal(mean=1.0, stddev=0.1):
    """
    Returns a function that initializes (s, v) such that:
        w = glorot_normal(shape)
        s ~ LogNormal(mean, stddev)  # shape = (out_features,)
        v = w / s
    """
    def init(shape):
        in_features, out_features = shape
        # Glorot normal for w
        fan_in, fan_out = in_features, out_features
        glorot_std = math.sqrt(2.0 / (fan_in + fan_out))
        w = torch.randn(shape) * glorot_std
        # Log-normal scaling vector s (per output unit)
        s = torch.exp(torch.normal(mean=mean, std=stddev, size=(
            out_features,)))
        # Broadcast divide: v[i,j] = w[i,j] / s[j]
        v = w / s
        return s, v
```

```
    return init

class FactorizedLinear(nn.Module):
    def __init__(self, in_features, out_features, mean=1.0, stddev
        =0.1):
        super().__init__()
        self.in_features  = in_features
        self.out_features = out_features
        # Initialize s and v using the factorized initializer
        s, v = factorized_glorot_normal(mean, stddev)((in_features,
            out_features))
        # Register as parameters
        self.s = nn.Parameter(s)              # shape (out_features,)
        self.v = nn.Parameter(v)              # shape (in_features,
            out_features)
        self.bias = nn.Parameter(torch.zeros(out_features))

    def forward(self, x):
        # Recompose kernel
        W = self.v * self.s      # broadcasting: (in, out) * (out,)
        return x @ W + self.bias

# example Model
pinn_model = nn.Sequential(
    FactorizedLinear(3, 16),
    nn.ReLU(),
    FactorizedLinear(16, 16),
    nn.ReLU(),
    FactorizedLinear(16, 1)
)
```

4.3.2 Deep Galerkin Method, Deep Ritz Method, and Variational PINNs

We have seen in Section 2.2.1 that, in finite-element practice, two closely related but conceptually distinct families of weak-form discretisations are commonly used: the *Ritz* (energy) method and the *Galerkin* (projection) method. The Ritz approach begins from an energy or action functional $E[u]$ and seeks the solution as the minimiser of E; discretisation is obtained by restricting E to a finite trial space and minimising there. By contrast, the Galerkin method starts from the weak form of the PDE (often obtained by integration by parts) and enforces that the residual be orthogonal to a chosen test space. Consequently, Ritz methods require that the PDE admit a variational (Euler-Lagrange) formulation (typically self-adjoint and coercive), whereas Galerkin projection is more general and applies even when no energy functional exists. Natural boundary conditions arise automatically from the variational derivative in the Ritz setting, while in Galerkin formulations essential boundary conditions are imposed on the trial space and natural conditions appear in the weak form.

4.3.2.1 The Deep Ritz Method

In [Yu+18], authors aim at solving the Ritz problem

$$\min_{u\in H} I(u), \qquad I(u) \coloneqq \int_\Omega \left(\frac{1}{2}|\nabla u(x)|^2 - f(x)u(x)\right) \mathrm{d}x\,,$$

where H is the set of admissible functions, or trial functions. To solve it, they build on the following ideas:

1. Approximate the trial function with multi-layer perceptron;
2. Use a numerical quadrature rule for the functional;
3. Use an algorithm for solving the final optimization problem.

If you think that these ideas are similar to the PINN concept, well... you are right. This work originate in the same year of PINNs, one month *earlier*.

In particular, they used a 5-layer MLP with two skip-connections

$$x \mapsto y_i = \lambda_1(x) \mapsto y_2 = \lambda_2(y_1) \mapsto y_3 = \lambda_3(y_2 + r_1(x)) \mapsto y_4 = \lambda_4(y_3) \mapsto u = \lambda_5(y_4 + r_2(y_2)))$$

where r_1, r_2 are the two residual connections. In this case, they get a $\hat{u}(\theta; x)$ approximator, and thus defining

$$g(\theta, x) \coloneqq \frac{1}{2}|\nabla \hat{u}(\theta; x)|^2 - f(x)\hat{u}(\theta; x)\,,$$

we get the optimisation problem

$$\min_{\theta\in\Theta} \mathcal{L}(\theta)\,, \quad \mathcal{L}(\theta) = \int_\Omega g(\theta, x)\mathrm{d}x\,.$$

The numerical quadrature rule they use to solve the problem is just Stochastic Gradient Descent over N domain points sampled i.i.d., so that

$$\theta_{n+1} = \theta_n - \eta\nabla_\theta \left[\frac{1}{N}\sum_{j=1}^{N} g(\theta_n, x_{j;n})\right],$$

where they *re-draw* the N points at each epoch n (as denoted by $x_{j;k}$).
In practice, they want to solve the Cauchy problem

$$-\Delta u(x) = 0\,, \quad x \in \Omega\,, \tag{4.21}$$

$$u(x) = r^{\frac{1}{2}} \sin\frac{\theta}{2}\,, \quad x \in \partial\Omega\,; \tag{4.22}$$

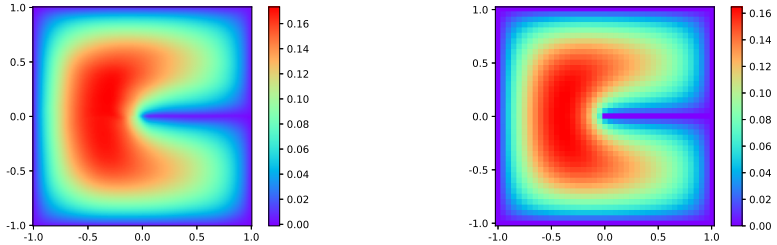
this problem is a well known problem, since its solution suffers from a corner-singularity, caused by the nature of the domain [SF+73]. Additionally, this problem has an exact solution

$$u^*(x) = r^{\frac{1}{2}} \sin\frac{\theta}{2}\,.$$

This allows for error comparison of standard Ritz method and DRM. To handle the B.C., they minimised

$$I(u) = \frac{1}{2}\int_\Omega |\nabla\hat{u}(\theta; x)|^2\mathrm{d}x + \beta\int_{\partial\Omega}[u(x)]^2\mathrm{d}x\,,$$

with β hyper parameter selected to be $\beta = 500$. With a simple MLP with 811 trainable parameters, they got the solution with L_2 error of 0.0072; additional results are shown in Table 4.2; for the visual comparison, see Section 4.3.2.1.



(a) Solution of Deep Ritz method, 811 parameters. From [Yu+18].

(b) Solution with FEM, 1,681 parameters. From [Yu+18].

fig:4:3:DeepRitz

4.3.2.2 The Deep Galerkin Method

In [SS18], instead, the authors introduce the Deep Galerkin Method (DGM); starting from the Cauchy problem

$$\partial_t u(t,x) + L[u(t,x)] = 0\,, \quad x \in [0,T] \times \Omega\,, \tag{4.23}$$

$$u(t=0,x) = u_0(x)\,, \tag{4.24}$$

$$u(t,x) = g(t,x)\,, \qquad x \in \partial\Omega\,, \tag{4.25}$$

where $\Omega \subset \mathbb{R}^d$. The DGM algorithm approximates $u(t,x)$ with a deep neural network $f(t,x;\theta)$ where $\theta \in \mathbb{R}^K$ are the neural network's parameters. The DGM algorithm is

1. Generate random points (t_n, x_n) from $[0,T] \times \Omega$ and (τ_n, z_n) from $[0,T] \times \partial\Omega$ according to respective probability densities ν_1 and ν_2. Also, draw the random point w_n from Ω with probability density ν_3.

2. Calculate the squared error $G(\theta_n, s_n)$ at the randomly sampled points $s_n = \{(t_n, x_n), (\tau_n, z_n), w_n\}$ where:

$$G(\theta_n, s_n) = \left(\frac{\partial f}{\partial t}(t_n, x_n; \theta_n) + \mathcal{L} f(t_n, x_n; \theta_n)\right)^2 + \left(f(\tau_n, z_n; \theta_n) - g(\tau_n, z_n)\right)^2 + \left(f(0, w_n; \theta_n) - u_0(w_n)\right)^2.$$

Table 4.2: Error of Deep Ritz method (DRM) and finite difference. From [Yu+18]. method (FDM)

tab:4:3:3: DRMvsRM

Method	Blocks Num	Parameters	relative L_2 error
DRM	3	591	0.0079
	4	811	0.0072
	5	1031	0.00647
	6	1251	0.0057
FDM		625	0.0125
		2401	0.0063

3. Take a descent step at the random point s_n:

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_\theta G(\theta_n, s_n)$$

4. Repeat until convergence criterion is satisfied.

Do you have a sensation of *déjà vu*? whll, you are (almost) right. This is the same basic recipe we gave for PINNs Section 3.1. Indeed, as outlined in [JKW23], the only major difference between the (original paper version of) PINN and DGM is the *sampling strategy*:

- PINNs draws a large set of points *before* the training process, and, per each epoch, they draw a subsample from there;
- DGM re-draws each sample *per-epoch*.

Furthermore, the DGM method focuses on a specific ANN architecture, similar to the Long-Short Term Memory (LSTM) recurrent neural network [HS97; Sca24; Sta19], that is [SS18]

$$\begin{aligned}
S^1 &= \sigma(W^1 \mathbf{x} + b^1), \\
Z^\ell &= \sigma(U^{z,\ell} \mathbf{x} + W^{z,\ell} S^\ell + b^{z,\ell}), \quad \ell = 1, \dots, L, \\
G^\ell &= \sigma(U^{g,\ell} \mathbf{x} + W^{g,\ell} S^1 + b^{g,\ell}), \quad \ell = 1, \dots, L, \\
R^\ell &= \sigma(U^{r,\ell} \mathbf{x} + W^{r,\ell} S^\ell + b^{r,\ell}), \quad \ell = 1, \dots, L, \\
H^\ell &= \sigma(U^{h,\ell} \mathbf{x} + W^{h,\ell} (S^\ell \odot R^\ell) + b^{h,\ell}), \quad \ell = 1, \dots, L, \\
S^{\ell+1} &= (1 - G^\ell) \odot H^\ell + Z^\ell \odot S^\ell, \quad \ell = 1, \dots, L, \\
f(t, x; \theta) &= W S^{L+1} + b,
\end{aligned} \tag{4.26}$$

{eq:4:3:2:2: DGMArch}

where $\mathbf{x} = (t, x)$, the number of hidden layers is $L + 1$, and $\odot$ denotes element-wise multiplication (i.e., $z \odot v = \big(z_0 v_0, \dots, z_N v_N\big)$). The parameters are

$$\begin{aligned}
\theta = \Bigg\{ & W^1, b^1, \left(U^{z,\ell}, W^{z,\ell}, b^{z,\ell} \right)_{\ell=1}^{L}, \left(U^{g,\ell}, W^{g,\ell}, b^{g,\ell} \right)_{\ell=1}^{L}, \\
& \left(U^{r,\ell}, W^{r,\ell}, b^{r,\ell} \right)_{\ell=1}^{L}, \left(U^{h,\ell}, W^{h,\ell}, b^{h,\ell} \right)_{\ell=1}^{L}, W, b \Bigg\}.
\end{aligned} \tag{4.27}$$

Unfortunately, the PINN folklore took the lead, and nowadays in the literature DGM refers to this LSTM-like architecture for solving PDE with neural networks.

4.3.2.3 Variational PINNs

The variational PINN (vPINN) approach [KZK19] instead, is fully inspired to the weak form of the PDE; so, to follow the nootation of [KZK19], if PINN aim to solve a problem in the form strong form

$$\mathcal{L}^{\mathfrak{q}} u(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{4.28}$$

$$u(\mathbf{x}) = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \tag{4.29}$$

+ using a DNN $u_{NN}(\theta; x)$ to approximate the solution, and optimising the parameters θ by minimising the *strong-form residual*

$$\begin{aligned} residual^{\mathfrak{s}} &= r(\mathbf{x}) - r^b(\mathbf{x}), \\ r(\mathbf{x}) &= \mathcal{L}^{\mathfrak{q}} u_{NN}(\mathbf{x}) - f(\mathbf{x}), \quad \forall \mathbf{x} \in \mathbf{x}_r, \\ r^b(\mathbf{x}) &= u_{NN}(\mathbf{x}) - h(\mathbf{x}), \qquad \forall \mathbf{x} \in \mathbf{x}_u. \end{aligned} \tag{4.30}$$

This give rise to *strong-form loss function* as

$$\begin{aligned} L^{\mathfrak{s}} &= L_r^{\mathfrak{s}} + L_u, \\ L_r^{\mathfrak{s}} &= \frac{1}{N_r} \sum_{i=1}^{N_r} \left| r(\mathbf{x}_{r_i}) \right|^2, \quad L_u = \tau \frac{1}{N_u} \sum_{i=1}^{N_u} \left| r^b(\mathbf{x}_{u_i}) \right|^2, \end{aligned} \tag{4.31}$$

in which τ denotes a penalty parameter. Here, the superscript $\mathfrak{s}$ refers to the loss function associated with the strong form of residual, to distinguish between this from and other considered cases. In this setting, the problem is solved by

$$\text{find } \tilde{u}(\mathbf{x}) = u_{NN}(\mathbf{x}; \mathbf{w}^*, \mathbf{b}^*) \text{ such that } \{\mathbf{w}^*, \mathbf{b}^*\} = \text{argmin}(L^{\mathfrak{s}}(\mathbf{w}, \mathbf{b})). \tag{4.32}$$

vPINNs instead aim to solve the *weak-form* formulation of the differential problem.

In this sense, vPINNs are actually *Deep Galerkin Method*, i.e. the Deep learning approach to Galerkin Methods. But it is what it is.

The goal is thus to find an approximation form $\tilde{u}(\mathbf{x}) = u_{NN}(\mathbf{x}; \mathbf{w}, \mathbf{b})$ with weights and biases $\{\mathbf{w}, \mathbf{b}\}$, respectively. Let $v(\mathbf{x})$ be a properly chosen test function. The differential problem can be recast in the variational form

$$(\mathcal{L}^{\mathfrak{q}} u_{NN}(\mathbf{x}), v(\mathbf{x}))_\Omega = (f(\mathbf{x}), v(\mathbf{x}))_\Omega \tag{4.33}$$

$$u(\mathbf{x}) = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \tag{4.34}$$

where $(\cdot, \cdot)$ denotes the inner product. This variational form leads to the *variational residual*, defined as

$$\begin{aligned} Residual^{\mathfrak{v}} &= \mathcal{R} - F - r^b, \\ \mathcal{R} &= (\mathcal{L}^{\mathfrak{q}} u_{NN}, v)_\Omega, \quad F = (f, v)_\Omega, \end{aligned} \tag{4.35}$$

which is enforced for any admissible test function v_k, $k = 1, 2, \cdots$ and r^b has the same form as in the strong form.

Hence, the authors of [KZK19] construct a discrete finite dimensional space V_K by choosing a finite set of admissible test functions and let

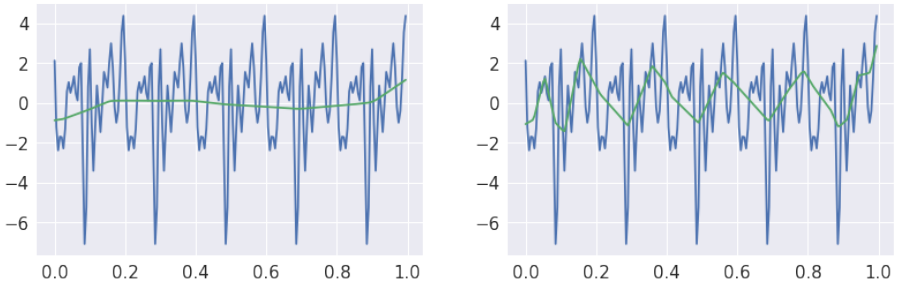
$$V_K = \text{span}\{v_k,\ k = 1, 2, \cdots, K\}.$$

Subsequently, they define the *variational loss function* as

$$\begin{aligned} L^{\mathfrak{v}} &= L_R^{\mathfrak{v}} + L_u, \\ L_R^{\mathfrak{v}} &= \frac{1}{K} \sum_{k=1}^{K} \left| \mathcal{R}_k - F_k \right|^2, \quad L_u = \tau \frac{1}{N_u} \sum_{i=1}^{N_u} \left| r^b(\mathbf{x}_{u_i}) \right|^2, \end{aligned} \tag{4.36}$$

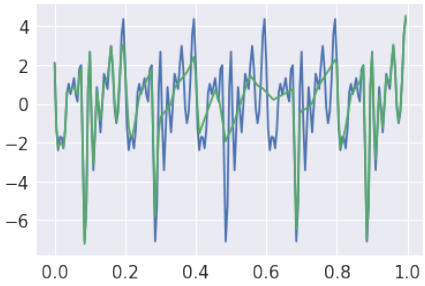
in which τ denotes the penalty parameter. Here, the superscript $\mathfrak{v}$ refers to the loss function associated with the variational form of residual. In this setting, the problem is this

$$\text{find } \tilde{u}(\mathbf{x}) = u_{NN}(\mathbf{x}; \mathbf{w}^*, \mathbf{b}^*) \text{ such that } \{\mathbf{w}^*, \mathbf{b}^*\} = \text{argmin}(L^{\mathfrak{v}}(\mathbf{w}, \mathbf{b})). \tag{4.37}$$

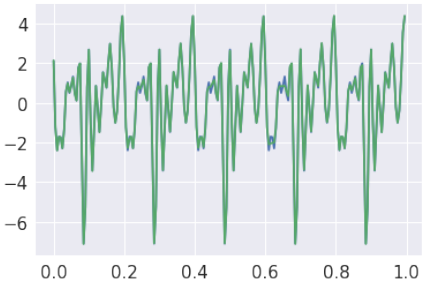


(a) Iteration 100. From [Rah+19].

(b) Iteration 1000. From [Rah+19].



(c) Iteration 10000. From [Rah+19].



(d) Iteration 80000. From [Rah+19].

Figure 4.7: The learnt function (green) overlayed on the target function (blue) as the training progresses. The target function is a superposition of sinusoids of frequencies $\kappa = (5, 10, \ldots, 45, 50)$, equal amplitudes and randomly sampled phases. From [Rah+19].

fig:4:4:
SpectralBiasEx1

4.3.3 Spectral Bias and Fourier Embeddings

subsec:4:4:3:
SpectralBias

An outstanding work [Rah+19] presented[1], using Fourier Analysis, a relevant property of (deep ReLU) neural networks: they are *biased towards low-frequency functions*, i.e. they cannot have local fluctuations without affecting their global behaviour.

A simple, yet illustrative example they propose is the following: Given frequencies $\kappa = (k_1, k_2, ...)$ with corresponding amplitudes $\alpha = (A_1, A_2, ...)$, and phases $\phi = (\varphi_1, \varphi_2, ...)$, define the mapping $\lambda : [0, 1] \to \mathbb{R}$ given by

$$\lambda(z) = \sum_i A_i \sin(2\pi k_i z + \varphi_i).$$

A 6-layer deep 256-unit wide ReLU network f_θ is trained to regress λ with $\kappa = (5, 10, ..., 45, 50)$ and $N = 200$ input samples spaced equally over $[0, 1]$; its spectrum $\tilde{f}_\theta(k)$ in expectation over $\varphi_i \sim U(0, 2\pi)$ is monitored as training progresses. The amplitude are a linear ramp from $A_1 = 0.1$ to $A_{10} = 1$. Figure 4.7 shows the learned function at intermediate training iterations. The result is that lower frequencies (i.e. smaller k_i's) are regressed first, regardless of their amplitudes.

[1] Additional studies [Xu+19; XZX19] help shed light in what they called the *frequency principle* (F-Principle).

But *why* this happens? the point is that it happens because parameters that contribute towards expressing high-frequency components occupy *a small volume* in the parameter space, *regardless* of their amplitude.

Authors also proved this fact. Let us sketch their main line of reasoning. Let us follow [Rah+19] and define *ReLU network* a scalar function $f : \mathbb{R}^d \mapsto \mathbb{R}$ defined by a neural network with L hidden layers of widths $d_1, \cdots d_L$ and a single output neuron:

$$f(\mathbf{x}) = \left(T^{(L+1)} \circ \sigma \circ T^{(L)} \circ \cdots \circ \sigma \circ T^{(1)}\right)(\mathbf{x})$$

where each $T^{(k)} : \mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}$ is an affine function ($d_0 = d$ and $d_{L+1} = 1$) and $\sigma(\mathrm{u})_i = \max(0, u_i)$ denotes the ReLU activation function acting elementwise on a vector $\mathrm{u} = (u_1, \cdots u_n)$. In the standard basis, $T^{(k)}(\mathbf{x}) = W^{(k)}\mathbf{x} + \mathbf{b}^{(k)}$ for some weight matrix $W^{(k)}$ and bias vector $\mathbf{b}^{(k)}$.

The goal is now to study the structure of ReLU networks in terms of their Fourier representation,

$$f(\mathbf{x}) \coloneqq (2\pi)^{\frac{d}{2}} \int \tilde{f}(\mathbf{k})\, \mathrm{e}^{\mathrm{i}\mathbf{k}\cdot\mathbf{x}} \mathrm{d}k,$$

where $\tilde{f}(\mathbf{k}) \coloneqq \int f(\mathbf{x})\, \mathrm{e}^{-\mathrm{i}\mathbf{k}\cdot\mathbf{x}} \mathrm{d}x$ is the Fourier transform. We can now write down the following

For the proofs, see the Appendix C and D of [Rah+19].

Lemma 4.3.3. *The Fourier transform of ReLU networks decomposes as,*

$$\tilde{f}(\mathbf{k}) = i \sum_{\epsilon} \frac{W_\epsilon \mathbf{k}}{k^2}\, \tilde{1}_{P_\epsilon}(\mathbf{k})$$

where $k = \|\mathbf{k}\|$ and $\tilde{1}_P(\mathbf{k}) = \int_P \mathrm{e}^{-\mathrm{i}\mathbf{k}\cdot\mathbf{x}} \mathrm{d}x$ is the Fourier transform of the indicator function of P.

Lemma 4.3.4. *Let P be a full dimensional polytope in $\mathbb{R}^d$. Its Fourier spectrum takes the form:*

$$\tilde{1}_P(\mathbf{k}) = \sum_{n=0}^{d} \frac{D_n(\mathbf{x}) 1_{G_n}(\mathbf{x})}{k^n},$$

where G_n is the union of n-dimensional subspaces that are orthogonal to some n-codimensional face of P, $D_n : \mathbb{R}^d \to \mathbb{C}$ is in $\Theta(1)\,(k \to \infty)$ and 1_{G_n} the indicator over G_n.

These two lemmas give the following

theorem:4:3:4: SpectralThm1

Theorem 4.3.5. *The Fourier components of the ReLU network f_θ with parameters θ is given by the rational function:*

$$\tilde{f}_\theta(\mathbf{x}) = \sum_{n=0}^{d} \frac{C_n(\theta, \mathbf{x}) 1_{H_n^\theta}(\mathbf{x})}{k^{n+1}},$$

where H_n^θ is the union of n-dimensional subspaces that are orthogonal to some n-codimensional faces of some polytope P_ϵ and $C_n(\cdot, \theta) : \mathbb{R}^d \to \mathbb{C}$ is $\Theta(1)\,(k \to \infty)$.

With these tools at hand, let us define what we mean by parameter space volume occupied by a certain frequence:

Definition 4.3.6. Given a ReLU network f_θ of fixed depth, width and weight clip K with parameter vector θ, an $\epsilon > 0$ and $\Theta = B_K^\infty(0)$ a L^∞ ball around 0, we define:

$$\Xi_\epsilon(k) \coloneqq \{\theta \in \Theta : \exists k' > k, |\tilde{f}_\theta(k')| > \epsilon\}$$

as the set of all parameters vectors $\theta \in \Xi_\epsilon(k)$ that contribute more than an ϵ in expressing one or more frequencies k' above a *cut-off frequency* k.

The *relative volume* of the parameter sub-space is thus defined as the integral over the set indicator function:

$$\text{vol}(\Xi_\epsilon(k)) \coloneqq \int_{\theta\in\Theta} 1_k^\epsilon(\theta)\text{d}\theta.$$

lemma:4:3:4: indicatorbound

Lemma 4.3.7. *Let $1_k^\epsilon(\theta)$ be the indicator function on $\Xi_\epsilon(k)$. Then:*

$$\exists\, \kappa > 0 : \forall k \geq \kappa, 1_k^\epsilon(\theta) = 0.$$

Furthermore, this implies that we have $1_k^\epsilon(\theta) \leq |\tilde{f}_\theta(k)|$ for large enough k (i.e. for $k \geq \kappa$), since $|\tilde{f}_\theta(k)| \geq 0$.

Proof. From theorem 4.3.5, we know that $|\tilde{f}_\theta(k)| = \mathcal{O}(k^{-\Delta-1})$ for an integer $1 \leq \Delta \leq d$. In the worse case where $\Delta = 1$, we have that $\exists M < \infty : |\tilde{f}_\theta(k)| < \frac{M}{k^2}$. Now, simply select a $\kappa > \sqrt{\frac{M}{\epsilon}}$ such that $\frac{M}{\kappa^2} < \epsilon$. This yields that $|\tilde{f}_\theta(\kappa)| < \frac{M}{\kappa^2} < \epsilon$, and given that $\frac{M}{\kappa^2} \leq \frac{M}{k^2}\, \forall\, k \geq \kappa$, we find $|\tilde{f}_\theta(k)| < \epsilon\, \forall\, k \geq \kappa$. Now by definition, $\theta \notin \Xi_\epsilon(\kappa)$, and since $\Xi_\epsilon(k) \subseteq \Xi_\epsilon(\kappa)$, we have $\theta \notin \Xi_\epsilon(k)$, implying $1_k^\epsilon(\theta) = 0\, \forall\, k \geq \kappa$. ■

We thus have the main

Spectral Volume Theorem

The relative volume of $\Xi_\epsilon(k)$ w.r.t. Θ is $\mathcal{O}(k^{-\Delta-1})$ where $1 \leq \Delta \leq d$.

Proof. For a large enough k, we have from remark Lemma 4.3.7, the monotonicity of the Lebesgue integral and theorem Theorem 4.3.5 that:

$$\begin{aligned}
\text{vol}(\Xi_\epsilon(k)) &= \int_{\theta\in\Theta} 1_k^\epsilon(\theta) d\theta \\
&\leq \int_{\theta\in\Theta} |\tilde{f}_\theta(k)| d\theta = \mathcal{O}(k^{-\Delta-1})\text{vol}(\Theta) \\
&\implies \frac{\text{vol}(\Xi_\epsilon(k))}{\text{vol}(\Theta)} = \mathcal{O}(k^{-\Delta-1}).
\end{aligned}$$

■

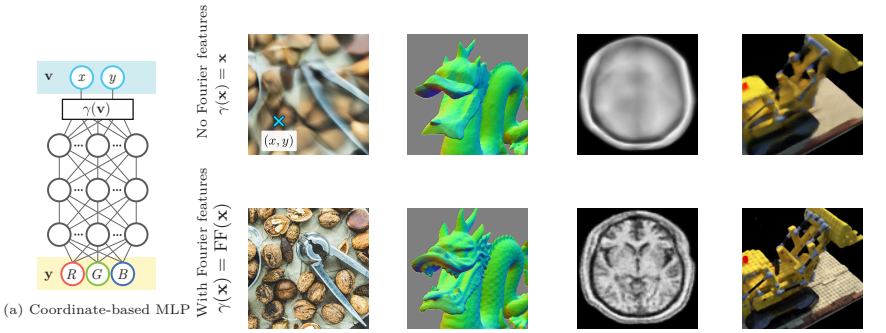
lst:4:3:3: SpectralBias

Listing 4.2: Spectral Bias

```
1 import os
2 import math
3 import numpy as np
4 from typing import List, Tuple
```

```

import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
import ightning as L

# Utility: target generator
def make_target(k_list: List[int], A_list: List[float], seed: int = 0)
     -> Tuple[callable, np.ndarray]:
    rng = np.random.RandomState(seed)
    phases = rng.uniform(0, 2 * np.pi, size=len(k_list))
    def target(z: np.ndarray) -> np.ndarray:
        z = np.asarray(z).reshape(-1)
        out = np.zeros_like(z, dtype=float)
        for k, A, phi in zip(k_list, A_list, phases):
            out += A * np.sin(2 * np.pi * k * z + phi)
        return out
    return target, phases

# MLP model (6 layers, 256)
class MLP(nn.Module):
    def __init__(self, in_dim=1, hidden=256, depth=6, out_dim=1):
        super().__init__()
        layers = []
        layers.append(nn.Linear(in_dim, hidden))
        layers.append(nn.ReLU())
        for _ in range(depth - 2):
            layers.append(nn.Linear(hidden, hidden))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(hidden, out_dim))
        self.net = nn.Sequential(*layers)
        self._init_weights()

    def _init_weights(self):
        for m in self.net:
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)

    def forward(self, x):
        return self.net(x)

# Single LightningModule encapsulating the experiment
class SpectralBiasPL(L.LightningModule):
    def __init__(
        self,
        k_list: List[int],
        A_list: List[float],
        N_train: int = 200,
        x_eval_n: int = 4096,
        hidden: int = 256,
        depth: int = 6,
        lr: float = 1e-3,
```

```
        seed: int = 0,
    ):
        super().__init__()
        self.save_hyperparameters(ignore=["k_list", "A_list"])
        self.k_list = np.array(k_list)
        self.A_list = np.array(A_list)
        self.N_train = N_train
        self.lr = lr
        # target function and phases
        self.target_fn, self.phases = make_target(k_list, A_list, seed
            =seed)
        # training samples (equally spaced)
        self.x_train = np.linspace(0.0, 1.0, N_train, endpoint=False)
        self.y_train = self.target_fn(self.x_train)
        # dense eval grid for spectrum and plotting
        self.x_eval = np.linspace(0.0, 1.0, x_eval_n, endpoint=False)
        # model
        self.model = MLP(in_dim=1, hidden=hidden, depth=depth, out_dim
            =1)
        # loss
        self.criterion = nn.MSELoss()
        # for deterministic behavior
        L.seed_everything(seed)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)

    def train_dataloader(self):
        X = torch.tensor(self.x_train.reshape(-1, 1), dtype=torch.
            float32)
        Y = torch.tensor(self.y_train.reshape(-1, 1), dtype=torch.
            float32)
        ds = TensorDataset(X, Y)
        # full-batch gradient descent: batch_size = N_train
        return DataLoader(ds, batch_size=self.N_train, shuffle=False)

    def configure_optimizers(self):
        # full-batch SGD as in the paper
        opt = torch.optim.SGD(self.parameters(), lr=self.lr)
        return opt

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = self.criterion(y_hat, y)
        # log loss for monitoring
        self.log("train/loss", loss, on_step=True, on_epoch=False,
            prog_bar=True)
        # periodic logging of spectrum and function
        step = int(self.global_step)
        # if wanted to reproduce plot figure, do:
        if (step % self.log_every_steps == 0) or (step == 0):
            self._save_function_and_spectrum(step) # to implement
        return loss
```



(a) Coordinate-based MLP (b) Image regression $(x,y) \to$ RGB (c) 3D shape regression $(x,y,z) \to$ occupancy (d) MRI reconstruction $(x,y,z) \to$ density (e) Inverse rendering $(x,y,z) \to$ RGB, density

Figure 4.8: Fourier features improve the results of coordinate-based MLPs for a variety of high-frequency low-dimensional regression tasks, both with direct (b, c) and indirect (d, e) supervision. We visualize an example MLP (a) for an image regression task (b), where the input to the network is a pixel coordinate and the output is that pixel's color. Passing coordinates directly into the network (top) produces blurry images, whereas preprocessing the input with a Fourier feature mapping (bottom) enables the MLP to represent higher frequency details. From [Tan+20].

fig:4:3:3:1: FourierNetteaser

4.3.3.1 Fourier Feature Embedding and Fourier Network

This spectral property of MLPs is somehow a beneficial property for standard deep learning tasks, since it means that it its training is more stable under noise oscillation in data.

Nevertheless, it also poses a different set of issues: for each tasks where "coordinate-based MLP" (i.e., MLPs eating a coordinate input ($\mathbf{x} = (x, y, z, \dots)$), the regression of high-modes is low, regardless of their importance. This has dramatic consequences in computer vision, scene understanding, and, of course, PINNs [Tan+20; Tos+25].

The solution is to use a *positional encoding* of the input coordinate data, originally introduced in [RR07], and later on popularised by the revolutionary *Neural Radiance Field* (NeRF) paper [Mil+21]:

$$\begin{aligned} \mathbf{x} \mapsto \gamma(\mathbf{x}) = \; & [a_1 \cos(2\pi \mathbf{b}_1 \cdot \mathbf{x}), \dots, a_m \cos(2\pi \mathbf{b}_m \cdot \mathbf{x}), \\ & a_1 \sin(2\pi \mathbf{b}_1 \cdot \mathbf{x}), \dots, a_m \sin(2\pi \mathbf{b}_m \cdot \mathbf{x})] \; , \end{aligned} \tag{4.38}$$

where $(a_i, \mathbf{b}_i)_{i=1,\dots,m}$ are parameters, which can be *fixed* (as in [Mil+21]) or *learnable*.

What we achieve is that, with this positional encoding, we uplift the $\mathbb{R}^d \ni \mathbf{x}$ data space to a $\mathbb{R}^{2m} \ni \gamma(\mathbf{x})$, where we have already either decompose the relevant Fourier modes, i.e. the model starts learning from a set of m Fourier modes, or we *learn* these relevant Fourier modes. The full networks is thus [Tan+20; WWP21b]

$$u_{NN}(\theta; \mathbf{x}) = \mathrm{W}_n \{\lambda_{n-1} \circ \cdots \circ \lambda_1 \circ \gamma\}(\mathbf{x}) + \mathbf{b}_n \, , \quad \lambda_i(\mathbf{x}_i) \coloneqq \phi_i(\mathrm{W}_i \mathbf{x}_i + \mathbf{b}_i) \tag{4.39}$$

4.3.3.2 Modified Fourier Network family

The Fourier Network architecture of Equation (4.39) is just the first step in the design of Fourier-modes-aware neural networks, especially in the PINN domain.

modified Fourier networks: in Fourier network, a standard fully-connected neural network is used as the nonlinear mapping between the Fourier features and the model output.

In *modified Fourier networks* [WTP21], two transformation layers are introduced to project the Fourier features to another learned feature space, and are then used to update the hidden layers through element-wise multiplications:

$$\begin{aligned}\phi_i(\mathbf{x}_i) = &\left[1-\sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\right]\odot\sigma\left(\mathbf{W}_{T_1}\phi_E+\mathbf{b}_{T_1}\right)\\ &+\sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\odot\sigma\left(\mathbf{W}_{T_2}\phi_E+\mathbf{b}_{T_2}\right),\end{aligned} \tag{4.40}$$

$$\tag{4.41}$$

where from now on ϕ_E is the *learnable* Fourier encoding

$$\phi_E(\mathbf{x}) = \left[\sin\left(2\pi\mathbf{B}\cdot\mathbf{x}\right);\cos\left(2\pi\mathbf{B}\cdot\mathbf{x}\right)\right]^T, \tag{4.42}$$

where $\mathbf{B}\in\mathbb{R}^{n_f\times d}$ is a learnable embedding matrix. Furthermore, $\mathbf{W}_{T_{1,2}}$ and $\mathbf{b}_{T_{1,2}}$ are also learnable parameters.

Highway Fourier Network: Inspired by [SGS15], in [Nvi26] it was introduced the *Highway Fourier network*. Highway networks consist of adaptive gating units that control the flow of information, and are originally developed to be used in the training of very deep networks. The hidden layers in this network are

$$\begin{aligned}\phi_i(\mathbf{x}_i) = &\,\sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\odot\sigma_s\left(\mathbf{W}_T\phi_E+\mathbf{b}_T\right)\\ &+\left(\mathbf{W}_P\mathbf{x}+\mathbf{b}_P\right)\odot\left(1-\sigma_s\left(\mathbf{W}_T\phi_E+\mathbf{b}_T\right)\right).\end{aligned} \tag{4.43}$$

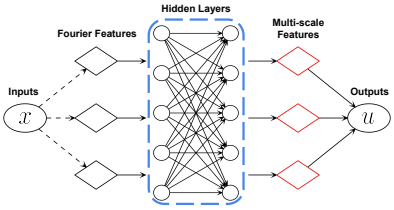
Multi-scale Fourier Feature Network: In [WWP21b], authors proposed a *multi-scale Fourier feature network* architecture that aim to tackle partial differential equations exhibiting multi-scale behaviors.

The key of the proposed architectures is to apply multiple Fourier feature embeddings initialized with different frequencies to input coordinates before passing these embedded inputs through the same fully-connected neural network and finally concatenate the outputs with a linear layer.

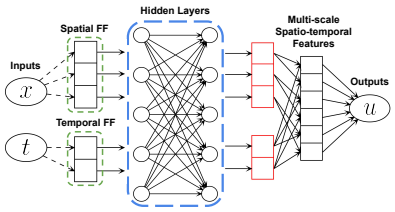
The forward pass of the multi-scale Fourier feature networks is given by

$$\begin{aligned}&\phi_E^{(i)}(\mathbf{x}) = [\sin(2\pi\mathbf{B}^{(i)}\cdot\mathbf{x});\cos(2\pi\mathbf{B}^{(i)}\cdot\mathbf{x})]^T, && \text{for } i=1,2,\ldots,M\\ &\mathbf{H}_1^{(i)} = \sigma(\mathbf{W}_1\cdot\phi_E^{(i)}(\mathbf{x})+\mathbf{b}_1), && \text{for } i=1,2,\ldots,M\\ &\mathbf{H}_\ell^{(i)} = \sigma(\mathbf{W}_\ell\cdot\mathbf{H}_{\ell-1}^{(i)}+\mathbf{b}_\ell), && \text{for } \ell=2,\ldots,L; i=1,2,\ldots,M\\ &\mathbf{u}_{NN}(\mathbf{x},\theta) = \mathbf{W}_{L+1}\cdot\left[\mathbf{H}_L^{(1)},\mathbf{H}_L^{(2)},\ldots,\mathbf{H}_L^{(M)}\right]+\mathbf{b}_{L+1}.\end{aligned} \tag{4.44}$$

For a visual representation of this architecture, see Section 4.3.3.2 (a).



(a) *Multi-scale Fourier feature architecture*: multiple Fourier feature embeddings (initialized with different σ) are applied to input coordinates and then passed through the same fully-connected neural network, before the outputs are finally concatenated with a linear layer. From [WWP21b].



(b) Spatio-temporal multi-scale Fourier feature architecture: Multi-scale Fourier feature architecture: multiple Fourier feature embeddings (initialized with different σ) are separately applied to spatial and temporal input coordinates and then passed through the same fully-connected neural network. Merging of the spatial and temporal outputs is performed using a point-wise multiplication layer, before obtaining the final outputs through a linear layer. From [WWP21b].

fig:4:3:3:1: ModFourNet

Spatio-temporal Fourier Feature Network: For time-dependent problems, multi-scale behaviour may exist not only across spatial directions but also across time.

In the same paper [WWP21b], authors proposed another novel multi-scale Fourier feature architecture to tackle multi-scale problems in spatio-temporal domains, the so-called *Spatio-temporal Fourier Feature Network*. Specifically, the feed-forward pass of the network is now defined as

$$
\begin{aligned}
&\phi_E^{(\mathbf{x})}(\mathbf{x}) = [\sin(2\pi\mathbf{B}^{(i)} \cdot \mathbf{x}); \cos(2\pi\mathbf{B}^{(i)} \cdot \mathbf{x})]^T, \\
&\phi_E^{(t)}(\mathbf{x}) = [\sin(2\pi\mathbf{B}^{(t)} \cdot t); \cos(2\pi\mathbf{B}^{(t)} \cdot t]^T, \\
&\mathbf{H}_1^{(\mathbf{x})} = \sigma(\mathbf{W}_1^{(\mathbf{x})} \cdot \phi_E^{(\mathbf{x})}(\mathbf{x}) + \mathbf{b}_1^{(\mathbf{x})}), \qquad \text{for } i = 1, 2, \ldots, M \\
&\mathbf{H}_1^{(t)} = \sigma(\mathbf{W}_1^{(t)} \cdot \phi_E^{(t)}(\mathbf{x}) + \mathbf{b}_1^{(t)}), \qquad \text{for } i = 1, 2, \ldots, M \\
&\mathbf{H}_\ell^{(\mathbf{x})} = \sigma(\mathbf{W}_\ell^{(\mathbf{x})} \cdot \mathbf{H}_{\ell-1}^{(\mathbf{x})} + \mathbf{b}_\ell^{(\mathbf{x})}), \qquad \text{for } \ell = 2, \ldots, L; i = 1, 2, \ldots, M \\
&\mathbf{H}_\ell^{(t)} = \sigma(\mathbf{W}_\ell^{(t)} \cdot \mathbf{H}_{\ell-1}^{(t)} + \mathbf{b}_\ell^{(t)}), \qquad \text{for } \ell = 2, \ldots, L; i = 1, 2, \ldots, M \\
&\mathbf{H}_i = \mathbf{H}_i^{(\mathbf{x})} \cdot \mathbf{H}_i^{(t)}, \\
&\mathbf{u}_{NN}(\mathbf{x}, \theta) = \mathbf{W}_{L+1} \cdot \mathbf{H}_L + \mathbf{b}_{L+1}.
\end{aligned} \tag{4.45}
$$

For a visual representation of this architecture, see Section 4.3.3.2 (b).

4.3.4 Sinusoidal Representation Networks

subsec:4:4:4: Siren

In [Sit+20] (the webpage, presenting the results, the data and the code, is available at [Sit+].), a simple neural network architecture for implicit neural representations that uses the sine as a periodic activation function called SIREN was proposed. It is defined as

$$
\Phi(\mathbf{x}) = \mathbf{W}_n (\phi_{n-1} \circ \phi_{n-2} \circ \ldots \circ \phi_0)(\mathbf{x}) + \mathbf{b}_n, \quad \mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin(\mathbf{W}_i \mathbf{x}_i + \mathbf{b}_i). \tag{4.46}
$$

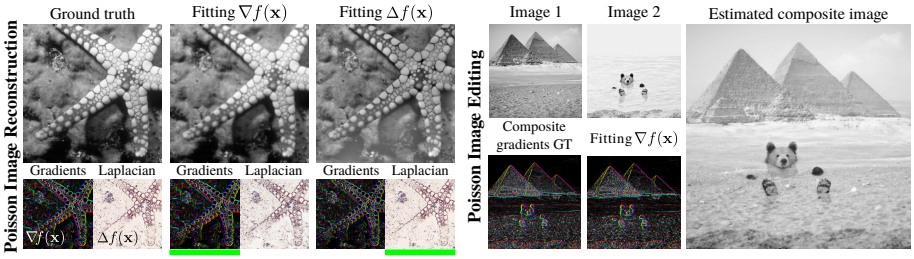


Figure 4.10: **Poisson image reconstruction:** An image (left) is reconstructed by fitting a SiRen, supervised either by its gradients or Laplacians (underlined in green). The results, shown in the centre and right, respectively, match both the image and its derivatives well. **Poisson image editing:** The gradients of two images (top) are fused (bottom left). SiRen allows for the composite (right) to be reconstructed using supervision on the gradients (bottom right). From [Sit+20].

fig:4:4:4:Siren_poisson_results

Here, $\phi_i : \mathbb{R}^{M_i} \mapsto \mathbb{R}^{N_i}$ is the i^{th} layer of the network. It consists of the affine transform defined by the weight matrix $\mathbf{W}_i \in \mathbb{R}^{N_i \times M_i}$ and the biases $\mathbf{b}_i \in \mathbb{R}^{N_i}$ applied on the input $\mathbf{x} \in \mathbb{R}^{M_i}$, followed by the sine nonlinearity applied to each component of the resulting vector.

An interesting property of this architecture is that any derivative of a SIREN *is itself a* SIREN, as the derivative of the sine is a cosine, which can be interpreted as a phase-shifted sine. Therefore, the derivatives of a SIREN inherit the properties of SIRENs, enabling to supervise any derivative of SIREN with "complicated" signals.

This network has similarities to the Fourier networks described in Section 4.3.3, because using a Sin activation function has the same effect as having a learnable Fourier input encoding for the first layer of the network. The main difference is that in SIREN *all* layers have a sinusoidal activation function. Something similar will be seen in Section 4.3.5, where we will introduce the WAVELET activation function.

A key component of SIREN architecture is the initialization scheme. The weight matrices of the network are drawn from a uniform distribution

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{\text{fan\_in}}}, +\sqrt{\frac{6}{\text{fan\_in}}}\right) ,$$

where fan_in is the input size to that layer. The input of each Sin activation has a Gauss normal distribution and the output of each sin activation, an arcsin distribution. This preserves the distribution of activations allowing deep architectures to be constructed and trained effectively.

The first layer of the network is scaled by a factor ω to span multiple periods of the sin function. This was empirically shown to give good performance and is in line with the benefits of the input encoding in the Fourier network. The authors suggest $\omega = 30$ to perform well under many circumstances.

A very interesting fact about SIREN is that, mainly, it was introduced in the context of Computer Vision, to obtain a network capable of learning representation of images (2D data), videos (2+1D data), and 3D representations (see Figure 4.10 and Figure 4.11).

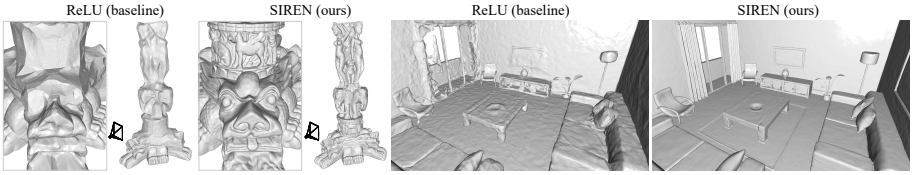


Figure 4.11: Shape representation. The authors of [Sit+20] fit signed distance functions parametrised by implicit neural representations directly on point clouds. Compared to ReLU implicit representations, the sinusoidal activations improve detail of objects (left) and complexity of entire scenes (right). From [Sit+20].

fig:4:4:4: sdf_results

The idea is that shape representation can be modelled by with differentiable *signed distance functions* (SDFs) [Par+19]; thus, it is possible to fit SDFs directly on oriented point clouds using implicit neural representations. This amounts to solving a particular *Eikonal boundary value problem*

$$\begin{aligned} \Phi(\mathbf{x}) &= 0\,, \qquad \mathbf{x} \in \Omega_0\,, \\ |\nabla_{\mathbf{x}}\Phi| &= 1\,, \qquad \mathbf{x} \in \Omega\,. \end{aligned} \tag{4.47}$$

They fit the SIREN with a loss

$$\begin{aligned} \mathcal{L}_{\text{sdf}} = &\int_{\Omega} \big\| \,|\nabla_{\mathbf{x}}\Phi(\mathbf{x})| - 1\big\| \mathrm{d}^d x \\ &+ \int_{\Omega_0} \left[\|\Phi(\mathbf{x})\| + \left(1 - \langle \nabla_{\mathbf{x}}\Phi(\mathbf{x}), \mathbf{n}(\mathbf{x})\rangle\right)\right] \mathrm{d}^d x \\ &+ \int_{\Omega\setminus\Omega_0} \psi\big(\Phi(\mathbf{x})\big) \mathrm{d}^d x, \end{aligned} \tag{4.48}$$

Here, $\psi(\mathbf{x}) = \exp(-\alpha \cdot |\Phi(\mathbf{x})|), \alpha \gg 1$ penalizes off-surface points for creating SDF values close to 0. Ω is the whole domain and Ω_0 is the zero-level set of the SDF. The model $\Phi(x)$ is supervised using oriented points sampled on a mesh, where we require the SIREN to respect $\Phi(\mathbf{x}) = 0$ and its normals $\mathbf{n}(\mathbf{x}) = \nabla f(\mathbf{x})$. During training, each mini-batch contains an equal number of points on and off the mesh, each one randomly sampled over Ω.

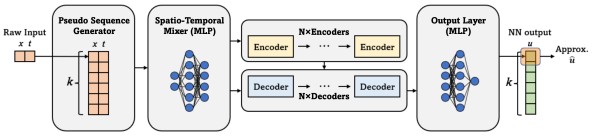
As seen in Fig. 4.11, SIREN significantly increase the details of objects and the complexity of scenes that can be represented by these neural SDFs, parametrising a full room with only a single five-layer fully connected neural network.

4.3.5 Transformers in PINNs

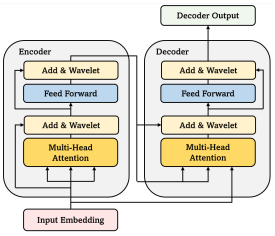
4:4:5: Pinnsformer

In [ZDP23], authors introduce a methodology to use Transformers [Sca24; Vas+17] in the context of PINNs, in particular to attack time-evolution problems. It may seems a complex task, since transformers usually works on (discrete) sequences and/or structured patches (ViT for example [Dos20]). Additionally, they are *permutation equivariant*, and great care must be used to break this property via an appropriate, so-called *Positional Embedding*.

Thus, they introduce three main steps to allow for the usage of a Transformer architecture:



(a) Architecture of the PINNsFormer. PINNsFormer generates a pseudo sequence based on pointwise input features. It outputs the corresponding sequential approximated solution. The first approximation of the sequence is the desired solution $\hat{u}_{NN}$. From [ZDP23].



(b) The architecture of PINNsFormer's Encoder-Decoder Layers. The decoder is **not** equipped with self-attentions. From [ZDP23].

fig:4:3:5: PINNsformer

1. A novel activation function, Wavelet, to achieve Fourier-transforms with forward passing:
$$\texttt{Wavelet}(\mathbf{x}) = \omega_1 \sin \mathbf{x} + \omega_2 \mathbf{x}\,, \tag{4.49}$$
where $\omega_{1,2}$ are learnable parameter.

2. a pseudo-sequence generator, to create *phrases* as a discrete set of time-steps, i.e. a map $\mathbb{R}^{d+1} \to \mathbb{R}^{m\times(d+1)}$:
$$[\mathbf{x}, t] \mapsto \{[\mathbf{x}, t], [\mathbf{x}, t + \Delta t], \dots [\mathbf{x}, t + m\Delta t]\}. \tag{4.50}$$

3. A learnable embedding, the so-called*spatio-temporal mixer* (see Section 4.3.5).

The `Wavelet` activation: The intuition behind `Wavelet` activation simply follows Real Fourier Transform. It can be shown that `Wavelet` can approximate arbitrary functions giving sufficient approximation power, as presented in the following proposition:

Proposition 4.3.8. *Let $\mathcal{N}$ be a two-hidden-layer neural network with infinite width, equipped with `Wavelet` activation function, then $\mathcal{N}$ is a universal approximator for any real-valued target f.*

Proof sketch: The proof follows the Real Fourier Transform. For any given input x and its corresponding real-valued target $f(x)$, we can represent it by its Fourier Integral:
$$f(x) = \int_{-\infty}^{\infty} F_c(\omega)\cos(\omega x)\,\mathrm{d}\omega + \int_{-\infty}^{\infty} F_s(\omega)\sin(\omega x)\,\mathrm{d}\omega$$
where F_c and F_s are the coefficients of Sines and Cosines respectively. Second, by Riemann sum approximation, the integral can be approximated by the infinite sum such that:
$$f(x) \approx \sum_{n=1}^{N} [F_c(\omega_n)\cos(\omega_n x) + F_s(\omega_n)\sin(\omega_n x)] \equiv W_2(\texttt{Wavelet}(W_1 x))$$

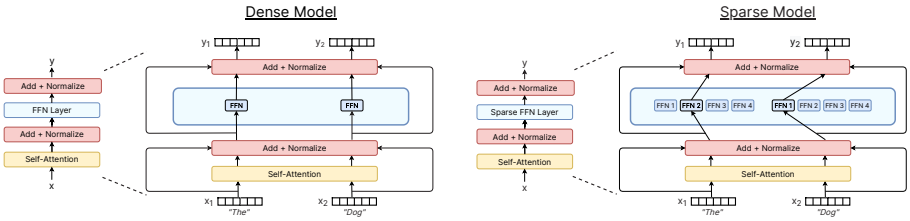


Figure 4.13: **Comparing a dense and sparse expert Transformer.** A dense model **(left)** sends both input tokens to the same feed-forward network parameters (FFN). A sparse expert model **(right)** routes each input token independently among its four experts (FFN1 $\cdots$ FFN4). In this diagram, each model uses a similar amount of computation, but the sparse model has more unique parameters. Note while this figure showcases a specific and common approach of sparse feed-forward network layers in a Transformer [Vas+17], the technique is more general. From [FDZ22]

.

fig:4:3:6:MoE_sparsity_diagram

where W_1 and W_2 are the weights of $\mathcal{N}$'s first and second hidden layer. As W_1 and W_2 are infinite-width, we can divide the piecewise summation into infinitely small intervals, making the approximation arbitrarily close to the true integral. Hence, $\mathcal{N}$ is a universal approximator for any given f.

Notice that `Wavelet` is used in the PINNsformer work just for its expressive power, and, as SIREN before, and Fourier Embedding even before, may have applications in other different contexts.

The Spatio-Temporal Mixer: Most PDEs contain low-dimensional spatial or temporal information. Directly feeding low-dimensional data to encoders may fail to capture the complex relationships between each feature dimension. Hence, in an analogy with Fourier Embedding, the authors of [ZDP23] suggest to embed original sequential data in higher-dimensional spaces such that more information is encoded into each vector, using an MLP on the *phrases*.

Instead of embedding raw data in a high-dimensional space where the distance between vectors reflects the semantic similarity [Vas+17], PINNsFormer constructs a linear projection that maps spatio-temporal inputs onto a higher-dimensional space using a fully-connected MLP. The embedded data enriches the capability of information by mixing all raw spatio-temporal features together, that they call the linear projection Spatio-Temporal Mixer.

4.3.6 Mixture-of-Experts

One powerful concept emerged already at the beginning of the '90s (see, e.g., [Jac+91]) was the possibility of gather a set of *experts* model to perform a certain task, and then aggregate their prediction in a gated manner, for example by training a specialised network to learn how to do that. This approach is increasingly becoming relevant, especially in the context of LLMs (see, e.g., [FDZ22; Zha+25a] and references therein), where it is possible to reduce the computational costs and the latency by using, instead of an extremely large model, a set of *expert* smaller models, of whose only a few are activated to perform a reply, token-by-token, without impacting the performances, as shown,

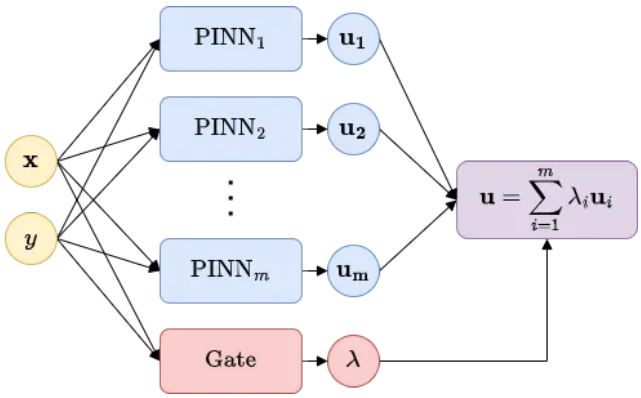


Figure 4.14: Mixture of Experts of PINNs. An arbitrary number m of PINNs, possibly with varying architectures and properties, is initialised together with a gating network. All models receive the same input and the gate produces weights that are used for aggregating the results. From [BK22].

fig:4:3:6: MoE_pinn1

for example, in [Jia+24]. In particular, in [Jia+24], they used eight 7B (pre-trained, and then fine-tuned) transformer model, where the fully-connected layers are replaced by the sparse Mixture-of-Experts, and defined a *router* model to select the TopK models (see Figure 4.13).

The idea was extended in the PINN domain in [BK22], with a few simplification to streamline its interpretability. In particular, MoE-PINN approach uses multiple networks (i.e., the experts) to infer the output multiple time, and then uses an additional network (the gate), which will weight the experts output. The m model may all have a different hyperparameter set (number of layers, activations, number of nodes, different embeddings, etc). Furthermore, by leveraging multiple networks and a gate model to divide the domain, each expert can specialise in solving a distinct part of the problem, allowing for improved accuracy and reduced bias-variance trade-off. Dividing the problem into smaller sub-problems has many advantages:

- By using several learners on distinct subdomains, the complexity of the problem is reduced.
- The gate in MoE-PINNs is a continuous function, resulting in smooth transitions between the domains. Therefore, more complex regions of the domain can be equally divided amongst several learners, whereas simpler regions can be attributed to a single expert.
- The gate can be any type of neural network, from a linear layer to a deep neural network, which allows it to adapt to different types of domains and divide them in an arbitrary way.
- MoE can be easily parallelised, since only the weights lambda need to be shared amongst learners on different devices. In theory, each learner could be placed on a distinct GPU.
- By initialising a large number of PINNs with different architectures, the need for labor-intensive hyperparameter tuning is reduced.

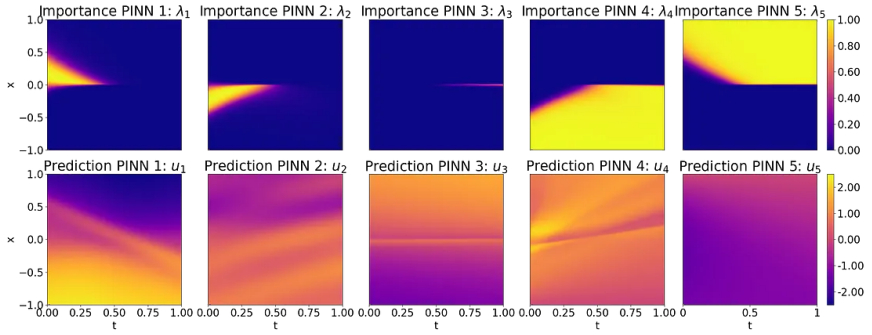


Figure 4.15: Weights lambda produced by the gating network (top row) for each of the expert (columns) as well as the predictions from each expert (bottom row) for Burgers' equation. From [BK22],

fig:4:3:6: MoE_burgers

The set up is the following: we have m networks $u_i(\theta_i) : (t, \mathbf{x}) \mapsto u_i(\theta_i; t, \mathbf{x})$, each of wich, alone, can approximate the solution. Furthermore, we have a gating network $g(\theta_g) : (t, \mathbf{x}) \mapsto g_i(t, \mathbf{x})$, with $i = 1, \dots, m$. Then, using a softmax, we associate to the gating network prediction a probability

$$\lambda_i(t, \mathbf{x}) = \frac{\exp[g_i(t, \mathbf{x})]}{\sum_{j=1}^{m} \exp[g_i(t, \mathbf{x})]} ,$$

which is the *importance* of the expert i. Then, the general approximation of the MoE-PINN is

$$u_{MoE}(\theta; t, \mathbf{x}) = \sum_{i=1}^{m} \lambda_i(t, \mathbf{x}) u_i(\theta_i; t, \mathbf{x}) .$$

Furthermore, MoE-PINNs have a nice *interpretable* feature, if sparsified, i.e. if there is added a sparsifying term to the loss to incentivise the router model $\lambda : (\mathbf{x}, t) \mapsto (\lambda_1(\mathbf{x}, t), \dots, \lambda_m(\mathbf{x}, t))$ the focus of each expert in a region,

$$\mathcal{L}_{sp} = \sum_{i=1}^{m} \left| \frac{1}{|B|} \sum_{(x,t)\in\Omega} \lambda_i(\mathbf{x}, t) \right|^p , \quad p \in (0, 1).$$

In Figure 4.15 is reported an example of MoE-PINNs application to solve the Burgers equation:

$$\begin{aligned} \partial_t u(t, x) + u(t, x)\partial_x u(t, x) &= \nu \partial_x^2 u(t, x) , \quad (t, x) \in [0, T] \times [-1, +1], \\ u(0, x) &= -\sin(\pi x) , \\ u(t, -1) = u(t, +1) &= 0 . \end{aligned}$$

They designed the experiment with $m = 5$ experts network with

- Expert 1: 2 layers with 64 nodes each and tanh activation;
- Expert 2: 2 layers with 64 nodes each and sine activation;
- Expert 3: 2 layers with 128 nodes each and tanh activation;

- Expert 4: 3 layers with 128 nodes each and tanh activation;
- Expert 5: 2 layers with 256 nodes each and swish activation;

Notice how the gating network in MoE-PINNs was able to effectively allocate tasks to each expert based on their respective capacity for modelling different parts of the domain. The experts with fewer layers and nodes were assigned to the smoother regions, which are relatively easier to model, i.e. close to the initial conditions. Meanwhile, the more complex experts, i.e. those with deeper and wider architectures, were utilised in the regions with discontinuities, where a more complex model was necessary for accurate representation. This is particularly evident in the case of expert 3, which was solely dedicated to capturing the discontinuity.

4.4 Optimisation Schemes

sec:4:6:
Optim_schemes

4.4.1 Optimisers: second order over first order

Let us now stop for a moment to look at our optimisation problem. As standard in machine and deep learning, what we are actually dealing with is some form of an optimisation problem, where we need to solve

$$\min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta)\,,$$

or, just to simplify the notation,

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}).$$

The most famous families of algorithms to solve this problem are first order methods, such as the Gradient Descent and its derivatives algorithms (SGD, RMSprop, Adam, etc). Symbolically, an iteration of

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_k)\,,$$

where η is a positive real number called *learning rate*, which controls the step size taken at each iteration. This is clearly a *first-order* optimisation scheme, such it utilise only the first-derivative (the gradient) of our loss. Furthermore remains the issue of picking η; if it is too large, the model diverges out of control; if it is too small, the model becomes highly inefficient, taking unnecessarily long to converge to the minimum. There is no a priori way of choosing a good learning rate, and in practice the optimal learning rate is typically found empirically, by looping through various values and seeing how they perform.

A physicist's point-of-view: Before going on, let us look again at our optimisation problem, and put up our physicist glasses (adapted from [Meh+19]). We need to minimise something, which is estimated via some sort of Monte Carlo integration scheme. Usually, in physical system, we minimise the *energy*

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} e_i(\theta; \mathbf{x}_i),$$

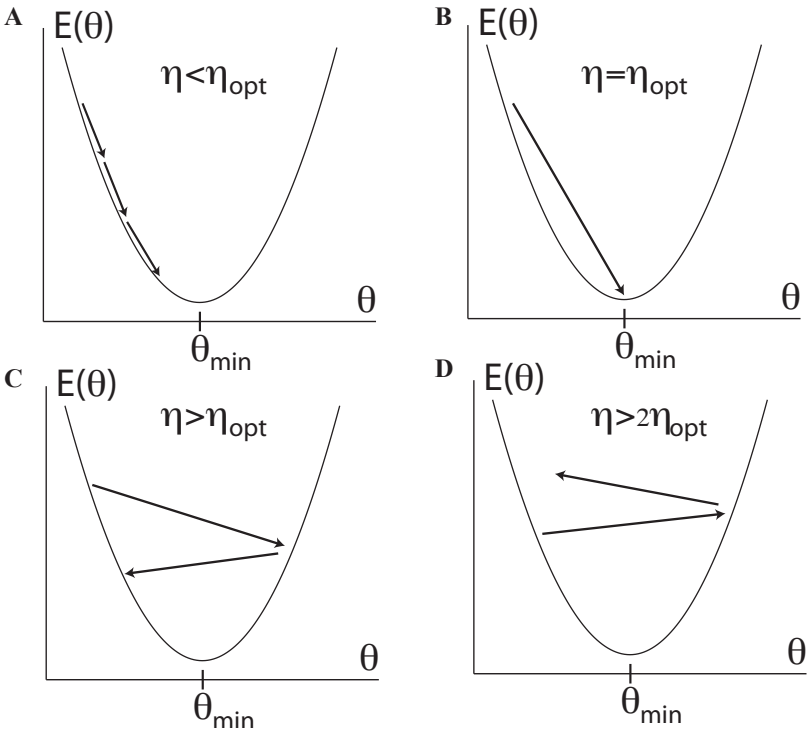


Figure 4.16: **Effect of learning rate on convergence**. For a one dimensional quadratic potential, one can show that there exists four different qualitative behaviours for gradient descent (GD) as a function of the learning rate η depending on the relationship between η and $\eta_{\text{opt}} = [\partial_\theta^2 E(\theta)]^{-1}$. (a) For $\eta < \eta_{\text{opt}}$, GD converges to the minimum. (b) For $\eta = \eta_{\text{opt}}$, GD converges in a single step. (c) For $\eta_{\text{opt}} < \eta < 2\eta_{\text{opt}}$, GD oscillates around the minima and eventually converges. (d) For $\eta > 2\eta_{\text{opt}}$, GD moves away from the minima. From [Meh+19], adapted from [LeC+02].

fig:4:5: GD3regimes

and this is somehow the total energy of a system of n (non-interacting) particles of energy e_i; additionally, if we use the mean squared error (MSE) as a cost function, we have

$$e_i(\theta; \mathbf{x}_i) = (\mathbf{y}_i - f_\theta(\mathbf{x}_i))^2 \,,$$

which is the elastic potential energy of a spring (of elastic constant $k = 1$) extending from $f_\theta(\mathbf{x}_i)$ to $\mathbf{y}_i$. We can thus see the plain gradient descent algorithm

$$\begin{aligned} \mathbf{v}_t &= \eta_t \nabla_\theta E(\theta_t), \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t \,, \end{aligned}$$

where we used the bold symbol to emphasize the vectorial nature of $\boldsymbol{\theta}$. It is clear that for sufficiently small choice of the learning rate η_t this methods will converge to a *local minimum* (in all directions) of the cost function. However, choosing a small η_t comes at a huge computational cost. The smaller η_t, the more steps we have to take to reach the local minimum. In contrast, if η_t is too

large, we can overshoot the minimum and the algorithm becomes unstable; it either oscillates or even moves away from the minimum (see Figure 4.16).

To better understand this behaviour, and highlight some of the shortcomings of plain gradient descent, it is useful to contrast first-order Gradient descent with a second-order method, the *Newton's Method*. In Newton's method, we choose the step $\mathbf{v}$ for the parameters in such a way as to minimize a second-order Taylor expansion to the energy function

$$E(\boldsymbol{\theta} + \mathbf{v}) \approx E(\boldsymbol{\theta}) + \nabla_\theta E(\boldsymbol{\theta})\mathbf{v} + \frac{1}{2}\,\mathbf{v}^T H(\theta)\mathbf{v},$$

where $H(\boldsymbol{\theta})$ is the Hessian matrix of second derivatives. Differentiating this equation respect to $\mathbf{v}$ and noting that for the optimal value $\mathbf{v}_{\text{opt}}$ we expect the minimum, i.e. a stationary point for its derivate $\nabla_\theta E(\boldsymbol{\theta} + \mathbf{v}_{\text{opt}}) = 0$, yields

$$\nabla_\theta E(\boldsymbol{\theta}) + H(\theta)\mathbf{v}_{\text{opt}} = 0,$$

which gives this algorithm

$$\begin{aligned} \mathbf{v}_t &= H^{-1}(\theta_t)\nabla_\theta E(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t. \end{aligned}$$

Since there is no guarantee that the Hessian is well conditioned, in almost all applications of Newton's method, the inverse of the Hessian $H^{-1}(\theta_t)$ is usually replaced by some suitably regularised pseudo-inverse such as $[H(\theta_t) + \epsilon I]^{-1}$ with ϵ a small parameter [Bat92].

We can now use a simple case with Newton's method to develop intuition about the role of the learning rate in first-order methods. Let us first consider the special case of using GD to find the minimum of a quadratic energy function of a single parameter θ [LeC+02]. Given the current value of our parameter θ, we can ask what is the optimal choice of the learning rate η_{opt}, where η_{opt} is defined as the value of η that allows us to reach the minimum of the quadratic energy function in a single step. To find η_{opt}, we expand the energy function to second order around the current value

$$E(\theta + v) = E(\theta_c) + \partial_\theta E(\theta)v + \frac{1}{2}\,\partial_\theta^2 E(\theta)v^2.$$

Now, differentiating with respect to v, and setting $\theta_{\min} = \theta - v$ we get

$$\theta_{\min} = \theta - [\partial_\theta^2 E(\theta)]^{-1}\partial_\theta E(\theta),$$

so that, by looking at these expression, we see that the optimal learning rate is

$$\eta_{\text{opt}} = [\partial_\theta^2 E(\theta)]^{-1}.$$

4.4.1.1 BFGS and its limited memory version

We have seen that Newton's method does not require the learning rate hyperparameter. It seems too good to be true! It turns out that the benefits from Newton's method comes at a cost. There are two main issues with Newton's method [Joh19; NW06] (for a nice blog post, see [Lam20]):

alg:4:5:1:1:BFGS

Algorithm 11 BFGS quasi-Newton method

Require: initial point x_0, tolerance $\varepsilon > 0$, maximum iterations $K_{\max}$, initial inverse Hessian approximation B_0 (symmetric positive definite, e.g. $B_0 = I$)

$k \leftarrow 0$
while $\|\nabla f(x_k)\| > \varepsilon$ and $k < K_{\max}$ **do**
  $p_k \leftarrow -B_k \nabla f(x_k)$ ▷ Compute search direction
  $\alpha_k \leftarrow \text{StrongWolfeLineSearch}(f, x_k, p_k)$ ▷ Pick α_k
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$ ▷ Compute update
  $s_k \leftarrow x_{k+1} - x_k$
  $y_k \leftarrow \nabla f(x_{k+1}) - \nabla f(x_k)$
  if $y_k^T s_k > 0$ **then**
    $\rho_k \leftarrow \dfrac{1}{y_k^T s_k}$
    $B_{k+1} \leftarrow \left(I - \rho_k s_k y_k^T\right) B_k \left(I - \rho_k s_k y_k^T\right)^T + \rho_k s_k s_k^T$ ▷ Update B
  else
    Option: reset $B_{k+1} \leftarrow B_0$ or skip update
  $k \leftarrow k + 1$
return x_k

- Newton's method is sensitive to initial conditions. This is especially apparent if our objective function is non-convex. Unlike gradient descent, which ensures that we're always going downhill by always going in the direction opposite the gradient, Newton's method fits a paraboloid to the local curvature and proceeds to move to the stationary point of that paraboloid. Depending on the local behaviour of our initial point, Newton's method could take us to a maximum or a saddle point instead of a minimum. Thus Newton's method is really only appropriate for minimizing convex objective functions.
- Newton's method is very computationally expensive. While the computation of the gradient scales as $O(n)$, the computation of the inverse Hessian scales as $O(n^3)$. Almost immediately, the benefit of an increased convergence rate will be far outweighed by the large cost of the additional computation time.

computing the Hessian scales as $O(n^2)$, inverting it scales as $O(n^3)$.

Quasi-Newtonian methods aim to handle these issues, still inserting Hessian hints of the landscape while keeping the computational cost low. The most famous method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [Bro70; Fle70; Gol70; Sha70], especially in its Limited-memory/Low storage version (L-BFGS).

The PyTorch version of L-BFGS is inspired to the Matlab version presented in [Sch].

The key idea of quasi-Newton methods is that, instead of computing the actual Hessian, we just approximate it with a positive-definite matrix B, which is updated from iteration to iteration using information computed from previous steps, thus effectively reducing the computational cost of inverting H at each iteration.

To describe the BFGS algorithm (following [NW06]), we start by identifying the objective function at the iteration k

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T H_k p \,,$$

whose minimiser is

$$p_k = H_k^{-1} \nabla_k f \,,$$

which is used as the search direction, and the new iterate is

$$x_{k+1} = x_k + \alpha_k p_k,$$

where α_k can be found performing a one-dimensional optimization (line search) to find an acceptable step-size α_k such that

$$\alpha_k = \arg\min f(x_k + \alpha_k p_k).$$

Practically, an inexact line search usually suffices, with α_k is chosen to satisfy the *Strong Wolfe Condition*

The (not strong) Wolfe Condition is the same, but without the absolute values in the second condition.

$$\begin{aligned} f(x_k + \alpha_k p_k) &\leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \,, \\ |\nabla f(x_k + \alpha_k p_k)^T p_k| &\geq c_2 |\nabla f_k^T p_k| \,, \\ \text{with } 0 < c_1 < c_2 < 1. & \end{aligned} \tag{4.51}$$

The following Listing 4.3 reports a sketch of a plain python implementation of the Strong Wolfe line search.

lst:4:5:1:1: StrongWolfe

Listing 4.3: Line Search with Strong Wolfe Condition

```
def strong_wolf_line_search(f,x,p,nabla):
    a  = 1.0
    c1 = 1e-4 # standard choice
    c2 = 0.9  # standard choices are in [0.1, 0.9]
    fx = f(x)
    x_new = x + a * p
    nabla_new = grad(f,x_new)
    while ( (f(x_new) >= fx + (c1*a*nabla.T @ p) )
            or (abs(nabla_new.T @ p) <= c2*abs(nabla.T @ p) ) ) :
        a *= 0.5
        x_new = x + a * p
        nabla_new = grad(f,x_new)
    return a
```

L-BFGS We notice that applying the BFGS update directly requires $O(n^2)$ storage for H_k^{-1} and $O(n^2)$ work on each step to update H_{k+1}. This is fine for up to a few thousand parameters, but for truly large-scale optimization problems it is prohibitive. Fortunately, BFGS algorithm is made of rank-1 updates, since we use rank-1 matrices (arrays) ρ_k, s_k, y_k to update B_k, there is a solution: store a set of rank-1 updates, not the *entire* matrix! Explicitly, one does not explicitly represents B_k, but computes its contraction $B_k g_k$, $g_k \coloneqq \nabla f(x_k)$ needed for the algorithm, and defines a two-step algorithm Algorithm 12, which replaces the inverse hessian approximation B_k update Algorithm 13.

Relevance for PINN training: Empirically, it was seen that, in some applications, quasi-Newton Methods, mostly the limited memory version of the BFGS algorithm (L-BFGS) [Byr+95], can be used to achieve better performance with fewer iterations with respect to first order methods, though they are more

alg:4:5:1:1:L-BFGS2S

Algorithm 12 L-BFGS two-step algorithm

Require: stored pairs $\{(s_i, y_i)\}_{i=k-m+1}^{k}$, scalars $\rho_i = 1/(y_i^T s_i)$, initial B_0 (often γI), vector $q = g$

for $i = k$ **downto** $k - m + 1$ **do**
  $\alpha_i \leftarrow \rho_i \, s_i^T q$
  $q \leftarrow q - \alpha_i \, y_i$
$r \leftarrow B_0 \, q$ ▷ apply initial matrix
for $i = k - m + 1$ **to** k **do**
  $\beta_i \leftarrow \rho_i \, y_i^T r$
  $r \leftarrow r + s_i \, (\alpha_i - \beta_i)$
return r ▷ this is $B_k g$

alg:4:5:1:1:L-BFGS

Algorithm 13 Limited-memory BFGS (L-BFGS)

Require: initial point x_0, tolerance $\varepsilon > 0$, maximum iterations $K_{\max}$, memory parameter m (number of stored pairs), initial scalar $\gamma_0 > 0$ (for $B_0 = \gamma_0 I$)

$k \leftarrow 0$
Initialize empty lists $\mathcal{S} \leftarrow [], \ \mathcal{Y} \leftarrow [], \ \mathcal{R} \leftarrow []$ ▷ store recent s_i, y_i, ρ_i
while $\|\nabla f(x_k)\| > \varepsilon$ and $k < K_{\max}$ **do**
  $g_k \leftarrow \nabla f(x_k)$
  $r \leftarrow \text{LBFGS2Steps}(g_k, \mathcal{S}, \mathcal{Y}, \mathcal{R}, B_0)$ ▷ Two step algorithm
  $p_k \leftarrow -r$ ▷ search direction
  $\alpha_k \leftarrow \text{StrongWolfeLineSearch}(f, x_k, p_k)$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $s_k \leftarrow x_{k+1} - x_k$
  $y_k \leftarrow \nabla f(x_{k+1}) - g_k$
  if $y_k^T s_k > 0$ **then**
    $\rho_k \leftarrow 1/(y_k^T s_k)$
    Append $\mathcal{S}$.append(s_k), $\mathcal{Y}$.append(y_k), $\mathcal{R}$.append(ρ_k)
    if $|\mathcal{S}| > m$ **then** ▷ drop oldest pair to keep memory m
      $\mathcal{S}$.pop(0), $\mathcal{Y}$.pop(0), $\mathcal{R}$.pop(0) ▷ remove oldest elements
    Optionally update $\gamma_0 \leftarrow \dfrac{s_k^T y_k}{y_k^T y_k}$ ▷ scaling for next iteration
  else
    Option: skip storing the pair or reset memory
  $k \leftarrow k + 1$
return x_k

prone to getting trapped at saddle point [USP25]. To mitigate this, some studies recommend using a first-order optimisation algorithm, mostly Adam, during the initial stages of training, followed by L-BFGS for fine-tuning [RPK17a; Tar+18; USP25].

4.4.1.2 Self-Scaled Broyden (SSBroyden)

In [USP25] was introduced a new, self-scaled quasi-Newton algorithm, which has an huge impact in optimising PINNs [Kho+26; Tos+25].

Self-scaled Broyden (SSBroyden) is a quasi-Newton update that generalises the classical Broyden/BFGS family by introducing *self-scaling* parameters that adapt the inverse-Hessian approximation to improve robustness and global convergence in large-scale, ill-conditioned problems. The method maintains an approximation H_k of the inverse Hessian and updates it using the secant information

$$\begin{aligned} \mathbf{s}_k &= \Theta_{k+1} - \Theta_k, \\ \mathbf{y}_k &= \nabla\mathcal{J}(\Theta_{k+1}) - \nabla\mathcal{J}(\Theta_k). \end{aligned}$$

Introducing the auxiliary vector

$$\mathbf{v}_k = \sqrt{\mathbf{y}_k^\top H_k \mathbf{y}_k}\left(\frac{\mathbf{s}_k}{\mathbf{y}_k^\top \mathbf{s}_k} - \frac{H_k\mathbf{y}_k}{\mathbf{y}_k^\top H_k \mathbf{y}_k}\right),$$

the SSBroyden inverse-Hessian update reads

$$H_{k+1} = \frac{1}{\tau_k}\left[H_k - \frac{H_k\mathbf{y}_k\,\mathbf{y}_k^\top H_k}{\mathbf{y}_k^\top H_k \mathbf{y}_k} + \phi_k\,\mathbf{v}_k\mathbf{v}_k^\top\right] + \frac{\mathbf{s}_k\mathbf{s}_k^\top}{\mathbf{y}_k^\top \mathbf{s}_k},$$

where $\tau_k > 0$ and ϕ_k are scalar scaling/update parameters chosen to enforce stability and (when possible) superlinear convergence. For $\tau_k = \phi_k = 1$ the formula reduces to the standard BFGS update.

Practical parameter choices (SSBroyden variant): A robust choice used in practice sets ϕ_k and τ_k from local curvature diagnostics. Defining

$$\begin{aligned} b_k &= \frac{\mathbf{s}_k^\top H_k^{-1}\mathbf{s}_k}{\mathbf{y}_k^\top \mathbf{s}_k}, \\ h_k &= \frac{\mathbf{y}_k^\top H_k \mathbf{y}_k}{\mathbf{y}_k^\top \mathbf{s}_k}, \\ a_k &= h_k b_k - 1, \\ c_k &= \sqrt{\frac{a_k}{a_k+1}} \\ \rho_k^- &= \min\big(1,\, h_k(1-c_k)\big), \\ \theta_k^- &= \frac{\rho_k^- - 1}{a_k}, \\ \theta_k^+ &= \frac{1}{\rho_k^-}, \end{aligned}$$

$$\theta_k = \max\Big(\theta_k^-,\ \min\big(\theta_k^+,\ \tfrac{1-b_k}{b_k}\big)\Big),$$
$$\sigma_k = 1 + a_k\theta_k.$$

A practical self-scaled choice for τ_k is

$$\tau_k = \begin{cases} \tau_k^{(1)} \min\big(\sigma_k^{-1/(n-1)},\ 1/\theta_k\big) & \text{if } \theta_k > 0, \\ \min\big(\tau_k^{(1)}\sigma_k^{-1/(n-1)},\ \sigma_k\big) & \text{if } \theta_k \le 0, \end{cases}$$

where $\tau_k^{(1)} = \min\{1,\ (\mathbf{y}_k^\top \mathbf{s}_k)/(\mathbf{s}_k^\top H_k^{-1}\mathbf{s}_k)\}$ and $n = \dim(\Theta)$. A recommended choice for ϕ_k is

$$\phi_k = \frac{1-\theta_k}{1+a_k\theta_k}.$$

Algorithm 14 SSBroyden quasi-Newton step

1: **Input:** current iterate Θ_k, inverse-Hessian approx. H_k, objective $\mathcal{J}$, step α_k (line search)
2: Compute gradient $g_k = \nabla\mathcal{J}(\Theta_k)$
3: Compute search direction $p_k = -H_k g_k$ and line search to get α_k
4: Set $\Theta_{k+1} = \Theta_k + \alpha_k p_k$; compute $g_{k+1} = \nabla\mathcal{J}(\Theta_{k+1})$
5: $\mathbf{s}_k = \Theta_{k+1} - \Theta_k, \quad \mathbf{y}_k = g_{k+1} - g_k$
6: Compute scalars $\mathbf{y}_k^\top\mathbf{s}_k,\ \mathbf{y}_k^\top H_k \mathbf{y}_k$
7: Form $\mathbf{v}_k = \sqrt{\mathbf{y}_k^\top H_k \mathbf{y}_k}\big(\mathbf{s}_k/(\mathbf{y}_k^\top\mathbf{s}_k) - H_k\mathbf{y}_k/(\mathbf{y}_k^\top H_k\mathbf{y}_k)\big)$
8: Compute $b_k, h_k, a_k, c_k, \rho_k^-, \theta_k, \sigma_k$ and choose τ_k, ϕ_k as above
9: Update

$$H_{k+1} \leftarrow \frac{1}{\tau_k}\Big[H_k - \frac{H_k\mathbf{y}_k\mathbf{y}_k^\top H_k}{\mathbf{y}_k^\top H_k\mathbf{y}_k} + \phi_k\,\mathbf{v}_k\mathbf{v}_k^\top\Big] + \frac{\mathbf{s}_k\mathbf{s}_k^\top}{\mathbf{y}_k^\top\mathbf{s}_k}$$

10: **Return** Θ_{k+1}, H_{k+1}

4.4.2 Loss functions, residuals, and geometry

4.4.2.1 Lambda Weighting

We have already seen schemes to dynamically adapt the Losses combination coefficients during training in Section 4.1. In a sense, we have seen how to change them during *training time* τ

$$\mathcal{L}(\theta(\tau)) = \sum_{i=1}^{n_{loss}} w_i(\tau)\mathcal{L}_i(\theta(\tau))\,.$$

We have also seen that we can do something similar in spirit (changing how we compute the optimisation risk of out multi-objective task), but different in practice: the importance measure Algorithm 8. Indeed, each individual loss is a (Monte Carlo) integral over some distance

$$\mathcal{L}_i(\theta) = \int_D \ell(\theta;\mathbf{x})\mathrm{d}^d x$$

Such terms should be *strongly zero*, i.e. zero for each point in the domain; the weak formulation may give rise to ill-behaved solutions in negligible-size region (i.e., their contribution to the integral is small w.r.t. the other regions), but giving rise to pathologies due to the non-linear nature of the network, thus affecting the training process towards global convergence.

That fact was behind the intuition of importance sampling; a more general formulation is to change (during training) how we *locally weight* the loss terms,

$$\mathcal{L}_i(\theta) = \int_D \lambda(\mathbf{x})\ell(\theta;\mathbf{x})\mathrm{d}^d x\,.$$

Obviously, is it possible to have the lambda weighting factor dependent on the training time τ (as well as the PDE time t), as $\lambda(\tau; t, \mathbf{x})$.

This is what is called, borrowing the terminology form [Nvi26], *lambda weighting*, or, following [Tos+25], *local weights*. The importance sampling algorithm of Algorithm 8 can thus be seen as a way to specify a certain lambda weighting.

Before moving on to show a few relevant examples of lambda weighting, we first ask: what is the ground of this idea? We recall that the plain loss can be seen as the (Monte Carlo integral of the) distance in the Banach/Hilbert space (e.g., the 2-norm)

$$\mathcal{L}_i(\theta) = d(f_\theta, f) = ||f_\theta(\mathbf{x}) - f(\mathbf{x})||_2 = \int_\Omega (f_\theta(\mathbf{x}) - f(\mathbf{x}))^2 \mathrm{d}^d x\,,$$

where we have, e.g., $f_\theta(\mathbf{x}) = \mathcal{F}[\hat{u}_\theta(\mathbf{x})]$ and $f(\mathbf{x}) = J(\mathbf{x})$. Notice that the $\mathrm{d}^d x$ term represents, on a flat open subset $\Omega \subset \mathbb{R}^d$, simply the standard metric $\mathrm{d}^d x = \mathrm{d}x_1 \mathrm{d}x_2 \cdots \mathrm{d}x_d$. This means that we can interpret the lambda weighting as a *change of measure*,

$$\mathrm{d}^d x \mapsto \lambda(\mathbf{x})\mathrm{d}^d x = \mathrm{d}\omega(\mathbf{x})\,,$$

which, again, can be heuristically interpreted as adding an *intrinsic curvature* to the domain Ω.

4.4.2.2 Signed Distance Function as Lambda Weighting

Sometimes, numerical problems have complex boundary conditions defined on complex geometries, which may have sharp corners; think about a simple squared region, $[-1,+1] \times [-1,+1] \subset \mathbb{R}^2$, with a scalar field having to satisfy mixed boundary conditions on the the $x = -1$ and $y = +1$ boundaries. The corner point $(x, y) = (-1, +1)$ is a *numerically* problematic point. One approach is thus weighting the equation residuals by the *signed distance function* (SDF) of the geometries. If the geometry has sharp corners this often results in sharp gradients in the solution of the differential equation. Weighting by the SDF tends to weight these sharp gradients lower and often results in a convergence speed increase, and sometimes also improved accuracy [Nvi26].

We have already introduced Signed Distance Functions in Section 4.3.4, and the fact that they can be obtained as the solution of the *Eikonal Equation*; let us formalise them a little better.

The signed distance function (sometimes denoted as signed distance field) is the orthogonal distance of a given point $x \in \mathbb{R}^d$ to the boundary of a set $\Omega \subset \mathbb{R}^d$ in a metric space. The sign is determined by whether or not $x \in \mathring{\Omega}$, i.e. it is in the interior of the set. Usually, the convention is that the function has positive

values for $x \in \overset{\circ}{\Omega}$, it decreases in value as $x \to \partial\Omega$, the signed distance function is zero $\forall x \in \partial\Omega$, and it takes negative values $\forall x \notin \overline{\Omega}$. So that

Definition 4.4.1 (Signed Distance Function)**.** Let Ω be a subset of a metric space $\mathscr{X}$, and let $d : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ be a distance function on $\mathscr{X}$. The *unsigned distance function* of $x \in \mathscr{X}$ from Ω is the infimum of all the distances between x and any $y \in \partial\Omega$, i.e.

$$d_{\partial\Omega}(x) \coloneqq \inf_{y \in \partial\Omega} d(x, y),$$

The *signed distance function* sdf is

$$\mathrm{sdf}_{\partial\Omega}(x) \coloneqq \begin{cases} +d_{\partial\Omega}(x)\,, & x \in \overset{\circ}{\Omega}\,, \\ -d_{\partial\Omega}(x)\,, & x \notin \overline{\Omega}\,, \\ 0\,, & x \in \partial\Omega\,. \end{cases} \tag{4.52}$$

An interesting fact is that signed distance functions satisfy a simple Eikonal equation almost everywhere

$$|\nabla \mathrm{sdf}(x)| = 1\,,$$

and, on $\partial\Omega$, we have

$$\nabla \mathrm{sdf}(x)|_{\partial\Omega} = n(x)\,,$$

where n is the orthonormal vector to $\partial\Omega$.

While various algorithm to solve the Eikonal equation to compute the signed distance function exists, sometimes it is beneficial to resort to different approximations of the Eikonal equation, due to the nature of the domain; a relevant example is the *Screened Poisson Equation* Ω [WBR14]:

$$\begin{aligned} v(x) - t\Delta v(x) &= 0\,, & x \in \Omega, \\ v(x) &= 1\,, & x \in \partial\Omega, \end{aligned}$$

where t is a small, positive, real arbitrary parameter. In [Var67] it was shown that

$$\lim_{t \to 0} -\sqrt{t}\,\log[v(x)] = d_{\partial\Omega}(x)\,.$$

Indeed, we can heuristically retrieve that result; if we call $u(x) \coloneqq -\sqrt{t}\,\log|v(x)|$, and placing it inside the Screened Poisson Equation, we express it as a *regularised Eikonal equation*:

Plugging it in $v(x) = \exp[-u(x)/\sqrt{t}]$ as a change of variable.

$$\begin{aligned} (1 - |\nabla u|^2) + \sqrt{t}\Delta u(x) &= 0\,, & x \in \Omega, \\ u(x) &= 0, & x \in \partial\Omega, \end{aligned}$$

which is, in the $t \to 0$ limit, the Eikonal Equation for the SDF. This "Screened Poisson Equation" version is the one implemented in NVIDIA PhysicsNemo [Nvi26].

In Figure 4.17 is reported the comparison of a training with and without SDF lambda weighting in a complex geometry, a 17 fin heat sink. It is evident a convergence speed up, for all four fields (pressure p, fluid velocities u, v, w along x, y, z axis respectively).

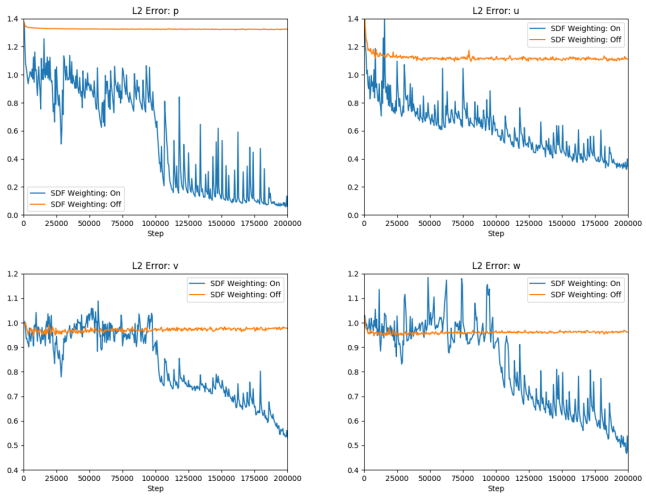


Figure 4.17: L_2 errors for one example of laminar flow (Reynolds number 50) over a 17 fin heat sink in the initial 100,000 iterations. The multiple closely spaced thin fins lead to several sharp gradients in flow equation residuals in the vicinity of the heat sink. Weighting them spatially, the dominance of these sharp gradients during the iterations are essentially minimised, thus achieving a faster rate of convergence. From [Nvi26].

fig:4:5:2:1: SDF_nvidia

4.4.2.3 Residual Based Attention as Lambda Weighting

A relevant, yet numerically simple choice for the lambda weighting function was introduced in [Ana+23]; the authors introduced *residual-based attention* (RBA) weights, where $\lambda_i(\tau; t, \mathbf{x}_i)$ is computed using the exponentially weighted moving average of the residuals $r_i(t, \mathbf{x})$. Since residuals contain information about regions with high error, this method proved to be highly effective, outperforming previous approaches with minimal computational cost.

As we have already noticed, one of the inherent challenges of PINN training is that the residuals of key collocation points can get overlooked by the mean calculation of the objective function. Consequently, despite a decrease in total loss during training, certain spatial (or temporal) characteristics of the searched solution might not be fully captured.

The goal of [Ana+23] was the development of a simple, gradient-less weighting scheme based on the rolling history of the cumulative residuals to extend the "attention" span of the optimizer. The proposed scheme aims to increase attention to the challenging regions of information propagation in both space and time dimensions defining the problem.

Not in the sense of Multi-Head Attention.

The update rule for RBA for any collocation point i at the iteration k is given by:

$$\lambda_i^{(k+1)} \leftarrow \gamma \lambda_i^{(k)} + \eta^* \frac{|r_i|}{\max_i(|r_i|)}, \qquad i \in \{0, 1, ..., N_r\}, \tag{4.53}$$

where N_r is the number of collocation points, γ is the decay parameter, η^* is

the learning rate, and r_i is the PDE residual for point i

$$r_i \coloneqq \left(\mathcal{F}[\hat{u}_\theta(\mathbf{x}_i)] - J(\mathbf{x}_i)\right)^2 .$$

Note that the learning rate of the optimizer η and the learning rate of the weighting scheme η^* are two different hyperparameters.

alg:4:4:2:3: residual-attention

Algorithm 15 Residual-Based Attention Gradient Descent (from [Ana+23])

Require: training set, learning rates η, η^*, decay $\gamma \in [0, 1]$, lower bound $c \geq 0$, batch size, stopping criterion

Initialize w_i with Xavier for all $i \in \{0, \ldots, N_r\}$
Initialize $\lambda_i \leftarrow 0$ for all i
while stopping condition not met **do**
  for each batch of training examples **do**
    $\lambda_i \leftarrow \gamma\, \lambda_i + \eta^* \dfrac{|r_i|}{\max_j |r_j|}$ ▷ RBA weight update
    $\lambda_i \leftarrow \lambda_i + c$ ▷ enforce lower bound shift
    $\mathcal{L}_r^* \leftarrow \dfrac{1}{N_r} \sum_{i=1}^{N_r} \left(\lambda_i\, r_i\right)^2$ ▷ weighted residual loss
    $\mathcal{L} \leftarrow \lambda_{ic}\, \mathcal{L}_{ic} + \lambda_{bc}\, \mathcal{L}_{bc} + \mathcal{L}_r^*$ ▷ total loss
    $w_i \leftarrow w_i - \eta\, \nabla_w \mathcal{L}$ ▷ Update parameters
return optimized parameters $\{w_i\}$

The main advantages of the RBA weighting scheme are:

1. Deterministic operation scheme, bounded by the γ and η^* parameters, which ensure the absence of exploding multipliers.

2. No training or gradient calculation is involved for computing λ, leading to negligible additional computational cost.

3. Scaling with the cumulative residuals guarantees increased attention on the solution fronts where the PDEs are unsatisfied in both spatial and temporal dimensions.

The modified training process for an indicative standard Gradient Descent optimization method is outlined in Algorithm 15. A constant c may be added to the updated weights to raise the lower bound and adjust the ratio $\max(\lambda_i)/\min(\lambda_i)$.

subsubsec:4:4:2:4:CausalLoss

4.4.2.4 Temporal Loss Weighting: causal loss

We have seen methods to change the local weighting of the loss w.r.t. the space dependence. But usually, when dealing with time-dependent problems, a major issue is to ensure a smooth, well-behaved time evolution of the solution. This is especially crucial in auto-regressive models, which treat time evolution as an auto-regressive predictive step (in Chapter 5, we will encounter this kind of behaviour when dealing with Neural Operators). The most important task is to ensure *causality* in the solution, which is quite relevant for non-linear equations which may present chaotic behaviour.

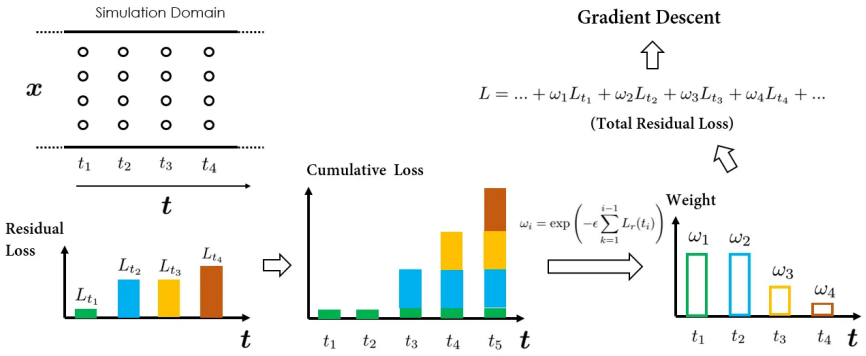


fig:4:4:2:4: CausaLoss

Figure 4.18: Illustration of the causal training algorithm. From [Guo24].

Assume we need to solve a time-dependent evolution problem with a PDE of the form

$$\partial_t u(t,\mathbf{x}) + \mathcal{N}[u(t,\mathbf{x})] = 0\,, \quad (t,\mathbf{x}) \in [0,T] \times \Omega\,.$$

A very simple approach is to blend Temporal loss weighting and a time-marching schedule [Kri+21], a specific case of curriculum learning strategy; a simple example of temporal loss weighting is, e.g., linear decay

$$\lambda(t) = 1 + C_T \left(1 - \frac{t}{T}\right)\,, \quad t \in [0,T].$$

In this way, the learning is penalised more highly when for errors at earlier time. Additionally, a sampling strategy called time-marching schedule can be used to increase the time span of the simulation during training time, for example having a linearly moving time interval

$$t \in [0, T(n)]\,, \quad T(n) \coloneqq T \min\left(1, \frac{n}{N}\right)\,,$$

where n is the training epoch and N is the ending epoch of the time marching schedule (for example, half the whole training epochs total).

A most common approach, called *causal loss*, was proposed in [WSP22]. Indeed, they show that continuous-time PINNs models can violate temporal causality, and hence are susceptible to converge towards erroneous solutions for transient problems. Thus, causal loss was introduced to address this issue and a key source of error by reformulating the PDE residual loss to account explicitly for physical causality during model training. The idea is to split the time interval into chunks $\{[t_i, t_{i+1}]\}_{i=0}^{N-1}$ and then compute the loss for each chunk

$$\mathcal{L}_i(\theta) = \sum_j \left| \frac{\partial u_\theta}{\partial t}(t_j, \mathbf{x}_j) + \mathcal{N}[u(t_j, \mathbf{x}_j)] \right|^2\,, \quad \{t_j, x_j\} \subset [t_{i-1}, t_i] \times \Omega\,.$$

The full PDE loss is then obtained by summing over chunks [Pen+23; WSP22],

$$\mathcal{L}_r(\theta) = \sum_{i=1}^{N} w_i \mathcal{L}_i(\theta)\,,$$

alg4:4:2:4: causalloss

Algorithm 16 Causal Training for Physics-Informed Neural Networks. From [WSP22].

Require: training set $\{t_i\}_{i=0}^{N_t}$, learning rate η, hyperparameters $\{\epsilon_j\}_{j=1}^{k}$, decay threshold δ, loss weights $\lambda_{ic}, \lambda_{bc}$

Initialize network parameters $\boldsymbol{\theta}$ (e.g., Xavier)
Initialize temporal weights $w_i \leftarrow 1$ for all $i = 0, \dots, N_t$
for each ϵ in $\{\epsilon_1, \dots, \epsilon_k\}$ **do** ▷ outer causal loop
  for $n = 1, \dots, S$ **do** ▷ inner gradient descent steps
    Compute losses $\mathcal{L}(t_i, \boldsymbol{\theta})$ for all i
    where
    $\mathcal{L}(t_0, \boldsymbol{\theta}) = \lambda_{ic}\, \mathcal{L}_{ic}(\boldsymbol{\theta})$
    $\mathcal{L}(t_i, \boldsymbol{\theta})$ for $i \geq 1$ is the PDE residual loss
    Update temporal weights:
    for $i = 2, \dots, N_t$ **do**
      $w_i \leftarrow \exp\Big(-\epsilon \sum_{k=1}^{i-1} \mathcal{L}(t_k, \boldsymbol{\theta})\Big)$
    Compute weighted total loss:
    $\mathcal{L}(\boldsymbol{\theta}) = \dfrac{1}{N_t} \displaystyle\sum_{i=0}^{N_t} w_i\, \mathcal{L}(t_i, \boldsymbol{\theta})$
    Gradient descent update:
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$
    if $\min_i w_i > \delta$ **then**
      break ▷ stop inner loop early
return $\boldsymbol{\theta}$

each of which is weighted as

$$w_i = \exp\left[-\epsilon \sum_{k=1}^{i-1} \mathcal{L}_k(\theta)\right], \quad \forall i = 2, 3, \dots, N.$$

In full form, we have

$$\mathcal{L}_{pde}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathrm{e}^{-\epsilon \sum_{k=1}^{i-1} \mathcal{L}_k(\theta)}\, \mathcal{L}_i(\theta)\,. \tag{4.54}$$

{eq:4:4:2:4: CausaLoss}

Note that the causal weight w_i is inversely exponentially proportional to the magnitude of the cumulative residual loss from the previous chunks. As a consequence, the loss at step (chunk) i, $\mathcal{L}_i(\theta)$ will almost not be affected during training unless all previous residuals decrease to some small value such that its weight w_i is large enough. Notice that the w_i must be computed as a parameter, i.e. they have to remain *outside* of the computational graph (e.g., by using DETACH in pytorch.).

Now, notice that Equation (4.54) can be seen as the discretised version of

$$\mathcal{L}_{pde}(\theta) = \int_0^T \mathrm{e}^{-\epsilon \int_0^t \mathcal{L}(\theta; s)\mathrm{d}s}\, \mathcal{L}(\theta; t)\, \mathrm{d}t\,,$$

with the appropriate redefinition of $\epsilon \mapsto \epsilon \Delta t_k$. This form is strikingly similar to the generic solution of the first-order ordinary differential problem

$$\begin{aligned} \dot{y}(t) + \epsilon \mathcal{L}(t) y(t) &= \mathcal{L}(t)\,, \qquad && y(0) = 0\,, \\ \Rightarrow \dot{y}(t) + \epsilon \mathcal{L}(t) \left(y(t) - \frac{1}{\epsilon} \right) &= 0\,, && y(0) = 0\,, \\ \Rightarrow \dot{z}(t) + \epsilon \mathcal{L}(t) z(t) &= 0\,, && z(0) = \frac{1}{\epsilon}\,, \text{ where } z(t) \coloneqq y(t) - \frac{1}{\epsilon}\,. \end{aligned}$$

Heuristically, we can see our causal loss objective $y(t) = \int_0^T \mathrm{e}^{-\epsilon \int_0^t \mathcal{L}(\theta;s)\mathrm{d}s}\, \mathcal{L}(\theta;t)\,\mathrm{d}t$ as a exponentially damped state, relaxed at a rate proportional to $\epsilon \mathcal{L}(t)$.

4.5 Recap-ish: *An Expert's Guide to Training Physics-informed Neural Networks*

In this lecture, we have seen multiple peculiarities, issue, and difficulties experienced when training deep neural networks to solve differential problems of the form

$$\partial_t u(t,\mathbf{x}) + \mathcal{N}[u(t,\mathbf{x})] = 0\,, \qquad (t,\mathbf{x}) \in [0,T] \times \Omega \quad \text{(PDE)}, \tag{4.55}$$ {eq:4:5:PDE}

$$u(0,\mathbf{x}) = g(\mathbf{x})\,, \qquad \mathbf{x} \in \Omega \quad \text{(IC)}, \tag{4.56}$$ {eq:4:5:IC}

$$\mathcal{B}[u(t,\mathbf{x})] = 0\,, \qquad (t,\mathbf{x}) \in [0,T] \times \partial\Omega \quad \text{(BC)}. \tag{4.57}$$ {eq:4:5:BC}

Let us recap the basic steps for training a PINN.

We can define a Monte Carlo sampling of the domain $[0,T]\times\Omega$ as $\{t_r^i, \mathbf{x}_r^i\}_{i=1}^{N_r}$ to impose locally the PDE constraint, and, similarly, $\{\mathbf{x}_{ic}^i\}_{i=1}^{N_{ic}}$ and $\{t_{bc}^i, \mathbf{x}_{bc}^i\}_{i=1}^{N_{bc}}$ for the IC/BC constraint imposition. From here, by defining a deep neural network as an universal approximator of the solution, $u_\theta(t,\mathbf{x}) \sim u(t,\mathbf{x})$, where θ are the learnable parameters describing the neural network, we can compute the PDE residuals as

$$\mathcal{R}_\theta(t,\mathbf{x}) = \frac{\partial \mathbf{u}_\theta}{\partial t}(t_r,\mathbf{x}_r) + \mathcal{N}[\mathbf{u}_\theta(t_r,\mathbf{x}_r)]\,. \tag{4.58}$$ {eq:4:5:pde_residual}

From here, as usual, we have the full multi-objective loss

$$\mathcal{L}(\theta) = \mathcal{L}_{ic}(\theta) + \mathcal{L}_{bc}(\theta) + \mathcal{L}_r(\theta), \tag{4.59}$$ {eq:4:5:PINN_loss}

where each component is

$$\mathcal{L}_{ic}(\theta) = \frac{1}{N_{ic}} \sum_{i=1}^{N_{ic}} \left| \mathbf{u}_\theta(0,\mathbf{x}_{ic}^i) - \mathbf{g}(\mathbf{x}_{ic}^i) \right|^2, \tag{4.60}$$ {eq:4:5:loss_ic}

$$\mathcal{L}_{bc}(\theta) = \frac{1}{N_{bc}} \sum_{i=1}^{N_{bc}} \left| \mathcal{B}[\mathbf{u}_\theta(t_{bc}^i,\mathbf{x}_{bc}^i)] \right|^2, \tag{4.61}$$ {eq:4:5:loss_bc}

$$\mathcal{L}_r(\theta) = \frac{1}{N_r} \sum_{i=1}^{N_r} \left| \mathcal{R}_\theta(t_r^i,\mathbf{x}_r^i) \right|^2. \tag{4.62}$$ {eq:4:5:loss_r}

In [Wan+23], authors propose a baseline training pipeline for PINNs, blending Fourier embeddings Section 4.3.3, Random Weight Factorisation Section 4.3.1.2, causal loss training Section 4.4.2.4, and NTK-based hyperparameter

update Section 4.1.7. The main ingredients of the suggested recipe is what follows (see Algorithm 17):

1. Use a MLP with Fourier embeddings and Random Weight Initialisation;
2. Use causal loss for PDE residuals
3. Every tot epochs, rebalance the multi-objective loss balancing parameters via NTK, using an exponential moving average.

In [Wan+23], the recommended default values for hyper-parameters are as follows: the NTK update every $f = 1,000$ epochs, and its exponential moving average hyperparameter $\alpha = 0.9$; the causal loss hyperparameter $\epsilon = 1.0$. As already noticed in Section 4.4.2.4, it is crucial that the the back-propagation of the weights w_i's and λ_i's with respect to network parameters θ is frozen.

alg:4:5:PINN_training_pipline

Algorithm 17 Training pipeline of physics-informed neural networks. From [Wan+23].

1. Non-dimensionalise the PDE system (4.55).
2. Represent the PDE solution by a multi-layer Perceptron network (MLP) $\mathbf{u}_\theta$ with Fourier feature embeddings and random weight factorization. In general, we recommend using tanh activation and initialized using the *Glorot* scheme.
3. Formulate the weighted loss function according to the PDE system:

$$\mathcal{L}(\theta) = \lambda_{ic}\mathcal{L}_{ic}(\theta) + \lambda_{bc}\mathcal{L}_{bc}(\theta) + \lambda_r\mathcal{L}_r(\theta), \tag{4.63}$$

where $\mathcal{L}_{ic}(\theta)$ and $\mathcal{L}_{bc}(\theta)$ are defined in (4.60), (4.61) respectively, and

$$\mathcal{L}_r(\theta) = \frac{1}{M}\sum_{i=1}^{M} w_i\mathcal{L}_r^i(\theta). \tag{4.64}$$

{eq:4:5:splited_res_loss}

The temporal domain is partitioned into M equal sequential segments and $\mathcal{L}_r^i$ denotes the PDE residual loss within the i-th segment of the temporal domain.
4. Set all global weights $\lambda_{ic}, \lambda_{bc}, \lambda_r$ and temporal weights $\{w_i\}_{i=1}^M$ to 1.
5. Use S steps of a gradient descent algorithm to update the parameters θ as:
for $n = 1, \ldots, S$ **do**

(a) Randomly sample $\{\mathbf{x}_{ic}^i\}_{i=1}^{N_{ic}}$, $\{t_{bc}^i, \mathbf{x}_{bc}^i\}_{i=1}^{N_{bc}}$ and $\{t_r^i, \mathbf{x}_r^i\}_{i=1}^{N_r}$ in the computational domain and evaluated each loss terms $\mathcal{L}_{ic}, \mathcal{L}_{bc}$ and $\{\mathcal{L}_r^i\}_{i=1}^M$.

(b) Compute and update the temporal weights by

$$w_i = \exp\left(-\epsilon\sum_{k=1}^{i-1}\mathcal{L}_r^k(\theta)\right), \text{ for } i = 2, 3, \ldots, M. \tag{4.65}$$

{eq:4:5:temporal_update}

Here $\epsilon > 0$ is a user-defined hyper-parameter that determines the "slope" of temporal weights.

if $n \mod f = 0$ **then**

(c) Compute the global weights by

$$\hat{\lambda}_{ic} = \frac{\|\nabla_\theta\mathcal{L}_{ic}(\theta)\| + \|\nabla_\theta\mathcal{L}_{bc}(\theta)\| + \|\nabla_\theta\mathcal{L}_r(\theta)\|}{\|\nabla_\theta\mathcal{L}_{ic}(\theta)\|}, \tag{4.66}$$

{eq:4:5:lambda_ic_update}

$$\hat{\lambda}_{bc} = \frac{\|\nabla_\theta\mathcal{L}_{ic}(\theta)\| + \|\nabla_\theta\mathcal{L}_{bc}(\theta)\| + \|\nabla_\theta\mathcal{L}_r(\theta)\|}{\|\nabla_\theta\mathcal{L}_{bc}(\theta)\|}, \tag{4.67}$$

{eq:4:5:lambda_bc_update}

$$\hat{\lambda}_r = \frac{\|\nabla_\theta\mathcal{L}_{ic}(\theta)\| + \|\nabla_\theta\mathcal{L}_{bc}(\theta)\| + \|\nabla_\theta\mathcal{L}_r(\theta)\|}{\|\nabla_\theta\mathcal{L}_r(\theta)\|}, \tag{4.68}$$

{eq:4:5:lambda_r_update}

where $\|\cdot\|$ denotes the L^2 norm.

(d) Update the global weights $\lambda = (\lambda_{ic}, \lambda_{bc}, \lambda_r)$ using a moving average of the form

$$\lambda_{\text{new}} = \alpha\lambda_{\text{old}} + (1-\alpha)\hat{\lambda}_{\text{new}}. \tag{4.69}$$

{eq:4:5:lambda_update}

where the parameter α determines the balance between the old and new values

(e) Update the parameters θ via gradient descent

$$\theta_{n+1} = \theta_n - \eta\nabla_\theta\mathcal{L}(\theta_n) \tag{4.70}$$

{eq:4:5:theta_update}

CHAPTER 5

Neural Operators

chap:5:
NeuralOperators

5.1 Learning Operators, Part I: Deep Operator Networks

5.1.1 Why learning *operators*?

We have seen in Section 1.3.2 that the conceptual basis underpinning the entire field on function approximation with neural network is the existence of a theorem, ensuring the representability of any unknown function with a neural network, the universal approximation theorem, either the Cybenko's version [Cyb89], or the Hornik, Stinchcombe, and White version [HSW89].

From here we learn that is possible to devise an universal approximator function, capable of representing any wanted unknown function. In particular, when dealing with differential problems, we have seen that we can use a neural network $u_\theta(\mathbf{x})$ to approximate a function $u \sim u_\theta$ by recasting the differential problem into an optimisation problem.

In this way, we can solve a single differential problem, using the map from the coordinate space $\mathbb{R}^d \ni \mathbf{x}$ to the target space of the function, $u(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^n$ (for scalar functions, $\mathbb{R}$). A very simple case, as a representative, is the Poisson equation:

$$\Delta u(\mathbf{x}) = f(\mathbf{x}) \,, \quad \mathbf{x} \in D \subset \mathbb{R}^d \,.$$

Having the source term f fixed, is it possible to numerically find a solution with an approximating function u_θ.

Now, let us now notice that we can *formally solve*, in an exact manner, this PDE, via the so-called Green Functional method [Sal22]; we recast the problem into a distributional problem

$$\Delta_{\mathbf{x}} G(\mathbf{x}, \mathbf{y}) = \delta(\mathbf{x} - \mathbf{y}) \,,$$

where $G : D \times D \to \mathbb{R}$ is the *Green's Function*, while $\delta(\mathbf{x}, \mathbf{y})$ is the Dirac Delta, defined as a distribution as

$$\int_D \delta(\mathbf{x} - \mathbf{y}) g(\mathbf{y}) \mathrm{d}^d y = g(\mathbf{x}).$$

Heuristically, it can be represented as a continuous generalisation of the Kronecker delta,

$$\delta(\mathbf{x} - \mathbf{y}) = \begin{cases} 0, & \mathbf{x} \neq \mathbf{y} \,, \\ \infty, & \mathbf{x} = \mathbf{y}, \end{cases}$$

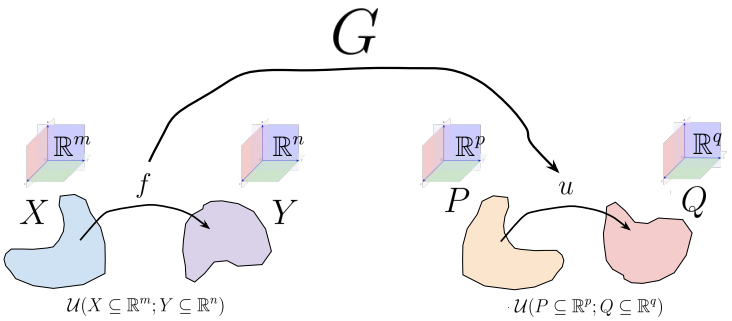


fig:5:1:OperViz

Figure 5.1: Visual representation of an operator.

such that

$$\int_D \delta(\mathbf{y}) \mathrm{d}^d y = 1.$$

Let us, for a moment, suppose that the Green's function can be analytically computed. In that case, we can see that we can express the solution of the Poisson PDE u, as

$$u(\mathbf{x}) = \int_D G(\mathbf{x}, \mathbf{y}) f(\mathbf{y}) \mathrm{d}^d y.$$

Indeed, applying $\Delta_{\mathbf{x}}$, and (with the appropriate conditions satisfied), moving the derivation inside the integral sign,

$$\begin{aligned} \Delta_{\mathbf{x}} u(\mathbf{x}) &= \Delta_{\mathbf{x}} \int_D G(\mathbf{x}, \mathbf{y}) f(\mathbf{y}) \mathrm{d}^d y \\ &= \int_D (\Delta_{\mathbf{x}} G(\mathbf{x}, \mathbf{y})) f(\mathbf{y}) \mathrm{d}^d y \\ &= \int_D \delta(\mathbf{x} - \mathbf{y}) f(\mathbf{y}) \mathrm{d}^d y \\ &= f(\mathbf{x}). \qquad \square \end{aligned}$$

In the following we will show that, indeed, f lives in the dual of the space where u lives, since we will tackle this kind of differential problems in a *weak sense* (Section 1.3).

Notice now that, denoting the (Banach) function space where u lives as $\mathcal{U} \ni u$, and with $\mathcal{U}^* \ni f$ the space where the source function f lives, we can write the Green's function as an operator mapping the two infinite-dimensional, functional spaces, i.e.

$$G : \mathcal{U}^* \to \mathcal{U}\,, \qquad G : f \mapsto G[f] \coloneqq \int_D G(\cdot, \mathbf{y}) f(\mathbf{y}) \mathrm{d}^d y\,.$$

What does it mean? it means that, through the Green's function, is possible to solve *any* differential problem of the Poisson form, not only one specific realisation, simply changing f.

The question now is: is it possible to *learn* such operator? differently stated: is it possible to represent with an *operator approximator*, dependent on a set of parameters θ, which allows us to translate the differential problem *family* to an *optimisation* problem?

The answer is **yes**. Via *Neural Operators*.

5.1.2 The Universal Approximation Theorem for function*als* (and operators)

We have thus seen that underpinning the multi-layer perceptron for function approximation we have the Universal Approximation theorem for functions. The question now is: do we have something similar for function*als* and for operators? the answer is yes [CC93; CC95]. In [CC93; CC95], it was proven that there exists an universal approximation theorem for both functionals and operators. To use the original notation, let denote with X some Banach space with norm $\|\cdot\|$, $K \subset X$ a compact subset, $C(K)$ the Banach space of all continuous functions on K, with the sup-norm. Furthermore, let us have

Definition 5.1.1 (Tauber-Wiener functions)**.** Let $g : \mathbb{R} \to \mathbb{R}$ a function; if every linear combination

$$\sum_{i=1}^{N} c_i g(w_i x + b_i)$$

is dense in every $C[a, b], \forall c_i, w_i, b_i \in \mathbb{R}, i = 1, \ldots, N$, then g is called a Tauber-Wiener function. The ensemble of all Tauber-Wiener functions is denoted with (TW)

Remark 5.1.2. The Tauber-Wiener (TW) requirement on a scalar activation $g : \mathbb{R} \to \mathbb{R}$ is a *sufficient* non-degeneracy condition that guarantees single-hidden-layer networks of the form $\sum_i c_i\, g(w_i \cdot x + b_i)$ are dense in spaces of continuous functions; in other words, TW ensures the classical universal approximation property. It is not a necessary condition (other, weaker hypotheses may also suffice), but it is convenient because it is simple to state and verify. Many commonly used activations (sigmoidal functions and a broad class of nonpolynomial activations; under mild technical hypotheses certain ReLU-type and smooth activations) satisfy the TW/non-degeneracy requirements used in the operator-approximation results of Chen and Chen. Thus, when we assume $g \in$ (TW) we are adopting a standard sufficient hypothesis that covers most practical activation choices while keeping the proofs and statements compact. [CC93; CC95; HSW89]

Indeed, the authors of [CC93; CC95] thus proved

thm:5:1:2: UnivThmFunctions

Theorem 5.1.3 (Universal Approximation Theorem for Functions)**.** *Let $K \subset \mathbb{R}^m$ compact, $U \subset C(K)$ compact, $g \in TW$, and let $f \in U$. Then, $\forall \epsilon > 0$, there exists a positive integer N, $\{b_i\}_{i=1,\ldots,N} \in \mathbb{R}, \{w_i\}_{i=1,\ldots,N} \in \mathbb{R}^m$, independent of f, and $\{c_i(f)\}_{i=1,\ldots,N} \in \mathbb{R}$, depending on f, such that*

$$\left| f(x) - \sum_{i=1}^{N} c_i(f) g(w_i x + b_i) \right| \leq \epsilon\,, \qquad \forall x \in K.$$

Moreover, each $c_i(f)$ is a linear continuous functional defined on U.

Theorem 5.1.3 shows that for a function (continuous or discontinuous) to be qualified as an activation function, a sufficient condition is that it belongs to Tauber-Wiener class. Furthermore, the equiuniform convergence property in Theorem 5.1.3 plays a crucial role in approximation to non-linear operators by neural networks (see the paper for more details).

But the two crucial results are the following

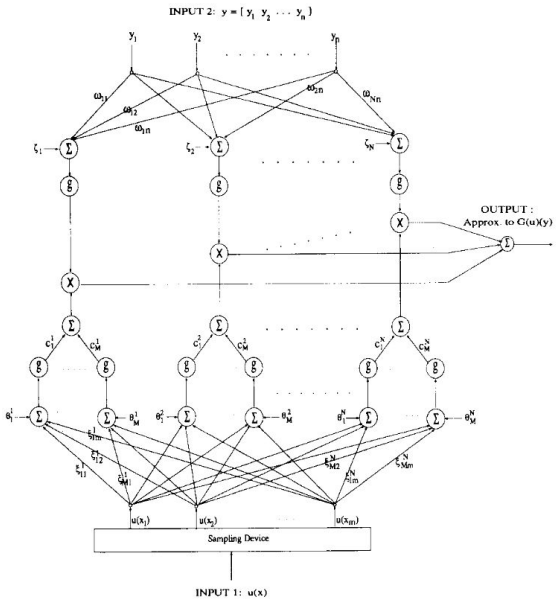


Figure 5.2: A neural network approximation to nonlinear operator $G(u)(y)$ based on theorem 5 of [CC95]. Original Figure 1 of the same paper.

fig:5:1:2: LearningOpOrigImg

thm:5:1:2: UnivThmFunctionals

Theorem 5.1.4 (Universal Approximation Theorem for Functionals)**.** *Suppose that $g \in$ (TW), X is a Banach space, $K \subseteq X$ is a compact set, $V \subset C(K)$ is a compact set, and $f : V \to \mathbb{R}$ is a continuous* functional*. Then for any $\epsilon > 0$ there exist a positive integer N, an integer m, points $x_1, \ldots, x_m \in K$, and real constants c_i, b_i, ξ_{ij} for $i = 1, \ldots, N$ and $j = 1, \ldots, m$, such that*

$$\left| f(u) - \sum_{i=1}^{N} c_i \, g\left(\sum_{j=1}^{m} \xi_{ij} \, u(x_j) + b_i \right) \right| < \epsilon$$

holds for all $u \in V$.

thm:5:1:2: UnivThmOps

Theorem 5.1.5 (Universal Approximation Theorem for Operators)**.** *Suppose that $g \in$ (TW), X is a Banach space, $K_1 \subseteq X$ and $K_2 \subseteq \mathbb{R}^n$ are compact sets, $V \subset C(K_1)$ is a compact set, and $G : V \subset C(K_1) \to C(K_2)$ is a nonlinear continuous* operator*. Then for any $\epsilon > 0$ there exist positive integers M, N, m, real constants $c_i^k, \zeta_k, \xi_{ij}^k$, points $\omega_k \in \mathbb{R}^n$ and $x_j \in K_1$ for $i = 1, \ldots, M$, $k = 1, \ldots, N$, $j = 1, \ldots, m$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{N} \sum_{i=1}^{M} c_i^k \, g\left(\sum_{j=1}^{m} \xi_{ij}^k \, u(x_j) + b_i^k \right) \, g(\omega_k \cdot y + \zeta_k) \right| < \epsilon \qquad (5.1)$$

{eq:5:1:2: UnivThmOpsEq}

holds for all $u \in V$ and all $y \in K_2$.

Remark 5.1.6. Theorem 5.1.5 is existential: it guarantees approximation power but not a constructive procedure for selecting the number of terms or sensor locations;

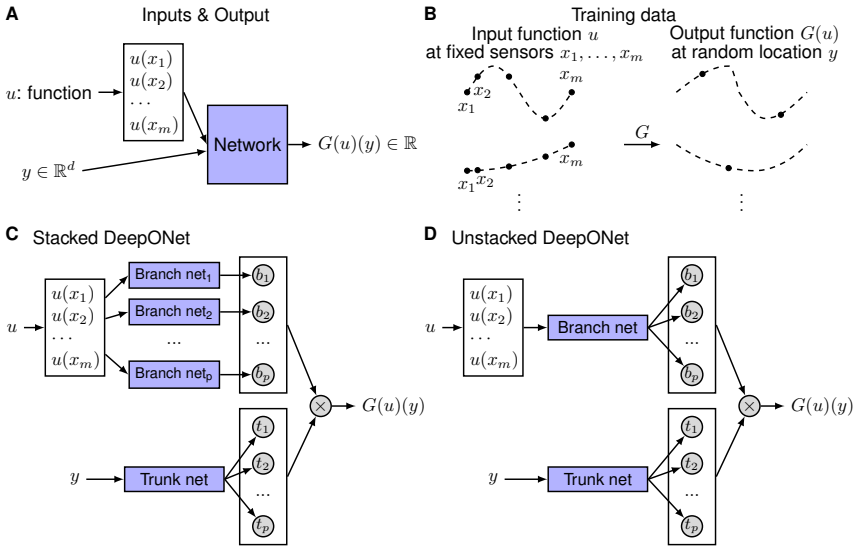


Figure 5.3: **Illustrations of the Operator Learning setup and architectures of DeepONets.** (**A**) The network to learn an operator $G : u \mapsto G(u)$ takes two inputs $[u(x_1), u(x_2), \dots, u(x_m)]$ and y. (**B**) Illustration of the training data. For each input function u, we require that we have the same number of evaluations at the same scattered sensors $x_1, x_2, \dots, x_m$. However, we do not enforce any constraints on the number or locations for the evaluation of output functions. (**C**) The stacked DeepONet in Theorem 5.1.5 has one trunk network and p stacked branch networks. (**D**) The unstacked DeepONet has one trunk network and one branch network. From [Lu+21].

fig:5:1:2: deeponet

Remark 5.1.7. Theorem 5.1.5 guarantees uniform approximation on compact sets (equicontinuity/equiuniform convergence). It's worth noticing that the approximation is uniform in both the input function (over the compact set $V \subset C(K_1)$ and the output function variable $y \in K_2$

Let us now look more deeply at Equation (5.1); We can rewrite the approximator

$$
\begin{aligned}
G_\theta[u](y) &:= \sum_{k=1}^{N} \underbrace{\sum_{i=1}^{M} c_i^k \, g\left(\sum_{j=1}^{m} \xi_{ij}^k \, u(x_j) + b_i^k\right)}_{\text{branch}} \underbrace{g(\omega_k \cdot y + \zeta_k)}_{\text{trunk}} \\
&:= \sum_{k=1}^{N} \sum_{i=1}^{M} c_i^k \, g\left(\sum_{j=1}^{m} \text{branchnet}_{ij}^k(u(x_j))\right) \cdot \text{trunknet}(\omega_k, \zeta_k; y) \,,
\end{aligned}
\tag{5.2}
$$

{eq:5:1:2: DeepONet1}

in terms of two blocks; a so-called *branch*, representing the action of the operator on the functional, and the *trunk*, representing the embedding of the output function's input. The original pictorial representation of this *Operator Network* is reported in Figure 5.2.

To clarify further, we borrow the *physical intuition* from [Lu+21; WWP21a]; suppose we have to learn the operator map of a function $u \in V = L^p(K_1)$ to a

function $G[u] \in C(K_2)$; suppose that we have some "sensors" $x_1, x_2, \dots \in K_1 \subset \mathbb{R}^m$, where we also evaluated $u_i^* = u(x_i)$ (e.g., u is a temperature fields, and x_i are the locations where we have the thermometers, and $G[u]$ is the gas velocity at temperature u (see Figure 5.3(B)).

5.1.3 Deep Operator Networks

We have now at our disposal the needed tools: an universal approximation theorem, furnishing solid grounds, and a way to formulate the approximator model in terms of *learnable parameters*. Indeed, starting from here, [Lu+21] introduced *Deep Operator Networks* (DeepONets), an approach to learning operators. To design and training DeepONets, in this general setting, the network inputs consist of two separate components: $[u(x_1), u(x_2), \dots, u(x_m)]^T$ and y (see Figure 5.3). Now, we can use a multi-layer fully connected networks (as an MLP) to model either the "trunk" network, which takes y as the input and outputs $[t_1, t_2, \dots, t_p]^T \in \mathbb{R}^p$, and the "branch" network. Formally, we should have N independent branch-nets (see Equation (5.1) and especially Equation (5.2)). This is clearly difficult to implement in practice (we should train $m+1$ networks, where m is the number of "sensors"); to overcome this fact, in practice one merges the branch-nets in what is called "Unstacked DeepONet"

$$\begin{aligned} G_\theta[u](y) &= \sum_{k=1}^{N} [\text{branchnet}(u)]_k \, \text{trunknet}(\omega_k, \zeta_k; y) + b_0 \\ &= \langle \text{branchnet}(u(x_1), \dots, u(x_N)) \, , \, \text{trunknet}(y) \rangle_{\mathbb{R}^N} + b_0 \, , \end{aligned} \tag{5.3}$$

{eq:5:1:2: DeepONet2Unstacked}

where we have added a common bias WLOG. The DeepONet version most faithful to Theorem 5.1.5 and to Equation (5.2), is called "Stacked DeepONet". In Figure 5.2 it is reported a pictorial representation of both Stacked (Figure 5.3(C)) and Unstacked DeepONets (Figure 5.3(D)).

Notice that, in the context of Unstacked DeepONets, the $\text{branchnet}_k(u)$ is a single fully connected network branchnet : $\mathbb{R}^m \to \mathbb{R}^N$, mapping $[u(x_1), \dots, u(x_m)] \mapsto \text{branchnet}(u) \sim (N)$.

Here we use the notation of [Sca24], where $X \sim (n, k, l)$ means that, as an array, it has a shape of $[n, k, l]$.

To train DeepONets, we simply start from a training dataset $\mathcal{D} = \{X; Y\} = \{\mathbf{u}_n, \mathbf{y}_n; G(\mathbf{u}_n)(\mathbf{y}_n) =: \mathbf{o}_n\}_{n=1}^{N_{data}}$, where n represents the data index, $\mathbf{u}_n, \mathbf{y}_n \in \mathbb{R}^N$, $\mathbf{u}_n := [u_n(x_1), \dots u_n(x_m)]$, and $G(\mathbf{u}_n)(\mathbf{y}_n) =: \mathbf{o}_n$ is the target value of the unknown function $G(\mathbf{u}_n)$ (mapped by G from $\mathbf{u}_n$) evaluated on $\mathbf{y}_n$. Notice that this dataset is an ensemble of evaluated mapped functions; indeed G maps functions to functions (being an operator) $u \mapsto G(u)$, but, for each simulation n, we have access to u only as a set of its evaluation on certain points, $\mathbf{u}_n := [u_n(x_1), \dots u_n(x_m)]$, and we have access to $G(u)$ again on some of its evaluation points, $G(\mathbf{u}_n)(\mathbf{y}_n)$. Nevertheless, once trained, the DeepONets furnish a *continuous* (approximated) operator mapping for each (evaluated) $\mathbf{u}_p \notin \mathcal{D}$, in the form of

Notice that we *don't* need to know $[x_1, \dots, x_m]$ explicitly.

$$G(\mathbf{u}_p)(\cdot) : \mathbb{R}^m \to C(K_2).$$

A crucial question remains open: *how many sensor points to we need to achieve a certain accuracy*? The authors replied also to this question, at least for ODE systems of the form

$$\frac{\mathrm{d}\mathbf{s}(x)}{\mathrm{d}x} = \mathbf{g}(\mathbf{s}(x), u(x), x)$$

$$\mathbf{s}(a) = \mathbf{s_0}\,.$$

For systems of this kind, we can formally integrate the ODEs as

$$(Gu)(x) = \mathbf{s_0} + \int_a^x \mathbf{g}((Gu)(t), u(t), t)\mathrm{d}t.$$

where $u \in V$(a compact subset of $C[a,b]$) is the input signal, and $\mathbf{s} : [a,b] \to \mathbb{R}^K$ is the solution of system, serving as the output signal.

Now, we choose uniformly $m+1$ points $x_j = a + j(b-a)/m, j = 0, 1, \cdots, m$ from $[a,b]$, and define the function $u_m(x)$ as follows:

$$u_m(x) = u(x_j) + \frac{u(x_{j+1}) - u(x_j)}{x_{j+1} - x_j}(x - x_j),\ x_j \le x \le x_{j+1},\ j = 0, 1, \cdots, m-1.$$

Denote the operator mapping u to u_m by $\mathcal{L}_m$, and let $U_m = \{\mathcal{L}_m(u) | u \in V\}$, which is obviously a compact subset of $C[a,b]$ since V is compact and continuous operator $\mathcal{L}_m$ keeps the compactness. Naturally, $W_m \coloneqq V \cup U_m$ as the union of two compact sets is also compact. Then, set $W \coloneqq \bigcup_{i=1}^{\infty} W_i$, and it can be proven that W is still a compact set. Since G is a continuous operator, $G(W)$ is compact in $C([a,b]; \mathbb{R}^K)$. The subsequent discussions are mainly within W and $G(W)$. For convenience of analysis, we assume that $\mathbf{g}(\mathbf{s}, u, x)$ satisfies the Lipschitz condition with respect to $\mathbf{s}$ and u on $G(W) \times W$, i.e., there is a constant $c > 0$ such that

$$\begin{array}{l} \|\mathbf{g}(\mathbf{s_1}, u, x) - \mathbf{g}(\mathbf{s_2}, u, x)\|_2 \le c\|\mathbf{s_1} - \mathbf{s_2}\|_2 \\ \|\mathbf{g}(\mathbf{s}, u_1, x) - \mathbf{g}(\mathbf{s}, u_2, x)\|_2 \le c|u_1 - u_2| \end{array}.$$

Note that this condition is easy to achieve, for instance, as long as $\mathbf{g}$ is differentiable with respect to $\mathbf{s}$ and u on $G(W) \times W$.

For $u \in V, u_m \in U_m$, there exists a constant $\kappa(m, V)$ depending on m and compact space V, such that

$$\max_{x \in [a,b]} |u(x) - u_m(x)| \le \kappa(m, V), \quad \kappa(m, V) \to 0 \ \ as \ \ m \to \infty. \tag{5.4}$$

Remark 5.1.8. *Discretization versus approximation error:* The total error of a learned DeepONet decomposes into the interpolation (discretization) error $\|u - u_m\|_\infty$ (due to finite sensors), and the neural approximation error of the branch/trunk parametrisation; both terms must be balanced when choosing m and the network sizes.

Based on the these concepts, the authors of [Lu+21] proved the following

Theorem 5.1.9. *Suppose that m is a positive integer making $c(b-a)\kappa(m,V)e^{c(b-a)}$ less than ε, then for any $d \in [a,b]$, there exist $\mathcal{W}_1 \in \mathbb{R}^{n\times(m+1)}, b_1 \in \mathbb{R}^{m+1}, \mathcal{W}_2 \in \mathbb{R}^{K\times n}, b_2 \in \mathbb{R}^K$, such that*

$$\|(Gu)(d) - (\mathcal{W}_2 \cdot \sigma(\mathcal{W}_1 \cdot [u(x_0) \quad \cdots \quad u(x_m)]^T + b_1) + b_2)\|_2 < \varepsilon$$

holds for all $u \in V$.

subsubsec: 5.1.3.1:antider

5.1.3.1 A prototypical application: learning ∂^{-1}

Let us start (following also the original paper [Lu+21]) by try to learn the anti-derivative operator; i.e., suppose we have a (huge) dataset for

$$\frac{\mathrm{d}s(x)}{\mathrm{d}x} = u(x)\,, \qquad u \in [0,1]\,, \tag{5.5}$$

$$s(0) = 0\,. \tag{5.6}$$

The formal solution of this problem is the anti-derivative operator

$$G : u(x) \mapsto s(x) = \int_0^x u(t)\mathrm{d}t.$$

To generate the dataset, it is simple to select random analytical symbolic functions $s(x)$, and computes their derivative $s' = u$, and evaluate it on certain domain point $x \in [0,1]$.

Actually, in their experiments, [Lu+21], they did the opposite: picked u and integrated back to get s. In particular, for the sensors, they used uniformly distributed points

$$x_j = a + j\,\frac{b-a}{m}\,, j = 0, \ldots, m,$$

The above equation is for generating uniformly distributed points in $[a, b]$.

where $a = 0, b = 1$; for the functions, they mainly consider two function spaces: Gaussian random field (GRF) and orthogonal (Chebyshev) polynomials. For the mean-zero GRF

$$u \sim \mathcal{G}(0, k_l(x_1, x_2)),$$

where the covariance kernel $k_l(x_1, x_2) = \exp(-\|x_1 - x_2\|^2/2l^2)$ is the radial-basis function (RBF) kernel with a length-scale parameter $l > 0$. The length-scale l determines the smoothness of the sampled function, and larger l leads to smoother u. For the Chebyshev polynomials (of the first kind), they define the orthogonal polynomials of degree N as:

$$V_{\mathrm{poly}} = \left\{ \sum_{i=0}^{N-1} a_i T_i(x) : |a_i| \le M \right\}.$$

They generate the dataset from V_{poly} by randomly sampling a_i from $[-M, M]$ to get a sample of u.

Thus, after sampling u from the chosen function spaces, they numerically solve the ODE systems by Runge-Kutta (4, 5) and PDEs by a second-order finite difference method to obtain the reference solutions.

Note that one data point is a triplet $(u, y, G(u)(y))$, and thus one specific input u may appear in multiple data points with different values of y. For example, a dataset of size 10000 may only be generated from 100 u trajectories, and each evaluates $G(u)(y)$ for 100 y locations.

5.1.4 Physics-Informed DeepONets

The plain, vanilla DeepONet seen in the previous section is purely *data informed*; the next step is to make it (also) *physics informed*; the step was done by [WWP21a]. This is done, as usual, by leveraging automatic differentiation to impose the underlying physical laws via soft penalty constraints during model training.

To move towards this direction, let us reformulate the problem, in a way which will be useful also in the remainder of the chapter. Let $(\mathcal{U}, \mathcal{V}, \mathcal{S})$ be a

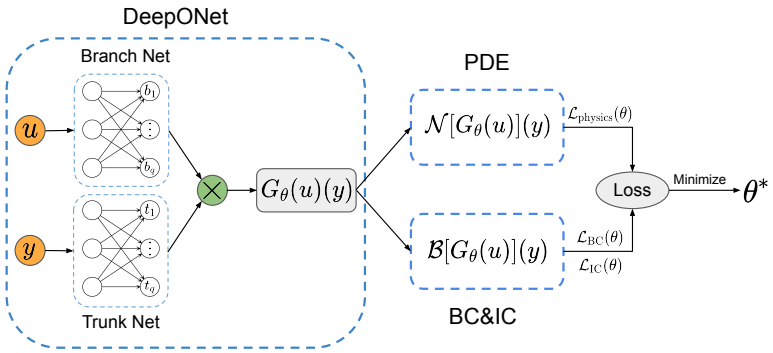


Figure 5.4: *Making DeepONets physics-informed:* We have seen that the DeepONet architecture [Lu+21] consists of two sub-networks: the *branch*, to extract latent representations of input functions; and the *trunk*, to extract latent representations of input coordinates at which the output functions are evaluated. A continuous and differentiable representation of the output functions is then obtained by merging the latent representations extracted by each sub-network via a dot product. To make it physics-informed, automatic differentiation can then be employed to formulate appropriate regularization mechanisms for biasing the DeepONet outputs to satisfy a given system of PDEs. There, the differential operator defining the PDE, $\mathcal{F}$, is denoted as $\mathcal{N}$. From [WWP21a].

fig:5:1:4: physics_ informed_ DeepONet

triplet of Banach spaces and $\mathcal{F} : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$, $\mathcal{B} : \mathcal{S} \to \mathcal{V}$ be (linear or non-linear) differential operators. We consider a differential problem

$$\mathcal{F}(u, s) = 0, \tag{5.7}$$
$$\mathcal{B}(s) = 0 \tag{5.8}$$

where $u \in \mathcal{U}$ denotes the input functions, and $s \in \mathcal{S}$ is the corresponding unknown solutions of the PDE system. In particular, we assume that, for any $u \in \mathcal{U}$, there exists an unique solution $s = s[u] \in \mathcal{U}$ (subject to appropriate initial and boundary conditions). Then, we can define the solution operator $G : \mathcal{U} \to \mathcal{S}$ as

$$G(u) = s(u). \tag{5.9}$$

We then approximate that G with $G_\theta : \mathcal{U} \times \Theta \to \mathcal{S}$, where $\theta \in \Theta$ are the trainable parameters. Usually, G_θ is an unstacked DeepONet,

$$G_\theta[u](y) = \sum_{k=1}^{N} b_k(u(x_1), \ldots, u(x_m))\, t_k(y) + b_0. \tag{5.10}$$

As already noticed, the outputs of a DeepONet model are continuously differentiable with respect to their input coordinates. Therefore, one may use automatic differentiation to formulate an appropriate regularization mechanism for biasing the target output functions to satisfy any given differential constraints. Thus, we can build *physics-informed DeepONets* by formulating the following loss function

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{operator}}(\theta) + \mathcal{L}_{\text{physics}}(\theta), \tag{5.11}$$

where

$$\mathcal{L}_{\text{operator}}(\theta) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_\theta\left[u^{(i)}\right]\left(y_{uj}^{(i)}\right) - s^{(i)}\left(y_{uj}^{(i)}\right)\right|^2, \tag{5.12}$$

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{NQm}\sum_{i=1}^{N}\sum_{j=1}^{Q}\sum_{k=1}^{m}\left|\mathcal{F}\left(u^{(i)}\left(x_k\right), G_\theta\left[u^{(i)}\right]\left(y_{rj}^{(i)}\right)\right)\right|^2. \tag{5.13}$$

Here, $\{u^{(i}\}_{i=1,\dots,N}$ denotes N distinct input functions, sampled from $\mathcal{U}$. For each $u \in \mathcal{U}$, $\{y_{uj}^{(i)}\}_{j=1,\dots,P}$ are P locations where the true field is evaluated. Furthermore, $\{y_{rj}^{(i)}\}_{j=1,\dots,Q}$ are Q random collocation points, sampled in the domain of $G[u^{(i)}]$, and may (or may not) be different from $\{y_{uj}^{(i)}\}_{j=1,\dots,P}$. Furthermore, they *may vary* for different $u^{(i)}$ (thus the subscript u).

In this way, $\mathcal{L}_{\text{operator}}(\theta)$ drives, during training, the approximator towards the available solutions measurements, while $\mathcal{L}_{\text{physics}}(\theta)$ enforces the underlying PDE constraints. We may want also to add the $\mathcal{L}_{\text{boundary}}(\theta)$ loss, as

$$\mathcal{L}_{\text{boundary}}(\theta) = \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{B}\left|\mathcal{B}\left(G_\theta\left[u^{(i)}\right]\left(y_{bj}^{(i)}\right)\right)\right|^2, \tag{5.14}$$

where $\{y_{bj}^{(i)}\}_{j=1,\dots,B}$ are B random collocation points, sampled in the boundary of the domain of $G[u^{(i)}]$.

5.1.4.1 learning ∂^{-1}, the *physics-informed* way

Let us go back to the problem discussed in Section 5.1.3.1, i.e. let us discuss how to learn the anti-derivative operator. We restart from

$$\begin{aligned}\frac{\mathrm{d}s(x)}{\mathrm{d}x} &= u(x)\,, \qquad u \in [0,1]\,,\\ s(0) &= 0\,.\end{aligned}$$

Now, the aim is to learn the solution operator $G_\theta : u \mapsto s$ **without any paired input-output** data. To do so, after modelling the solution operator G_θ with a fully connected neural network (for both branch and trunk net, in an unstacked manner), we recast the general differential problem to an optimisation one with loss

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{boundary}}(\theta) + \mathcal{L}_{\text{physics}}(\theta) \tag{5.15}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left|G_\theta[u^{(i)}](0)\right|^2 + \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|\frac{\mathrm{d}G_\theta[u^{(i)}](y)}{\mathrm{d}y}\Big|_{y=x_j} - u^{(i)}(x_j)\right|^2. \tag{5.16}$$

5.2 Using a different representation theorem for functions: Kolmogorov-Arnold Network

We have seen that our approach to solve various approximating problem is grounded into *universal approximation theorems*. One major result, due to

Kolmogorov and Arnold ([Arn09; Arn60; Kol57]; see [BG09; Sch21] for a modern perspective), directly related to Hilbert 13th problem [Mor21]. The Hilbert 13th problem is to prove the following conjecture

> «*A solution of the general equation of degree 7 cannot be represented as a superposition of continuous functions of two variables.*»

Working on that, Kolmogorov and Arnold proved that *any* multivariate function (i.e., any function whose domain $\Omega \subseteq \mathbb{R}^m, m > 1$) can be written as a *finite* composition of *univariate* functions (i.e., function whose domain is $I \subseteq \mathbb{R}$). The most common version of the Kolmogorov-Arnold Representation Theorem (sometimes abbreviated as KAT, or KART) is the following [Ism25]

Theorem 5.2.1 ((original) Kolmogorov-Arnold Representation Theorem)**.** *Let $I^n = [0,1]^n \subset \mathbb{R}^n, n \geq 2$ be the unit cube. Then, there exists $n(2+1)$ universal, continuous, one-variable functions $\phi_{q,p} : [0,1] \to \mathbb{R}$, with $q = 1, \dots, 2n+1, p = 1, \dots, n$, such that any continuous function $f \in C(I^n)$ admits the precise representation*

$$f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right), \qquad \forall (x_1, \dots, x_n) \in I^n, \tag{5.17}$$

where $\Phi_q : \mathbb{R} \to \mathbb{R}$ are n continuous one-variable functions depending on f.

Notice that the second activation Φ_q is $f-$dependent, as it happened in Theorem 5.1.3 for $c_i(f)$.

Notice also that the original formulation above is about *continuous* function on the unit cube in $\mathbb{R}^n$. Fortunately, it is possible to extend the theorem to approximate (i) a non-continuous function; and (ii) a function with generic bounded domain [II24; Ism23; Ism25]. Furthermore, more recent work have shown that the outer functions Φ_q can be replaced by a single Φ. In particular [Ism25]

Theorem 5.2.2 ((extended) Kolmogorov-Arnold Representation Theorem)**.** *Let $D \subset \mathbb{R}^n, n \geq 2$ be a bounded domain. Then, there exists $n(2+1)$ universal, continuous, one-variable functions $\phi_{q,p} : \mathbb{R} \to \mathbb{R}$, with $q = 1, \dots, 2n+1, p = 1, \dots, n$, such that any function f (not necessarily continuous) admits the precise representation*

$$f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right), \qquad \forall (x_1, \dots, x_n) \in D, \tag{5.18}$$

{eq:5:2:KARTeq}

where $\Phi : \mathbb{R} \to \mathbb{R}$ is a one-variable function depending on f. If f is continuous, then Φ can be chosen to be continuous; if f is discontinuous and bounded, Φ is also discontinuous and bounded; if f is unbounded, Φ is also unbounded.

Furthermore, it has been shown that $\phi_{q,p}$ can be replaced by a family of functions of simpler structure, e.g. [Spr65]

$$\phi_{q,p}(x_p) \to \lambda_p \phi(x_p + aq), \quad \lambda_p, a \in \mathbb{R}.$$

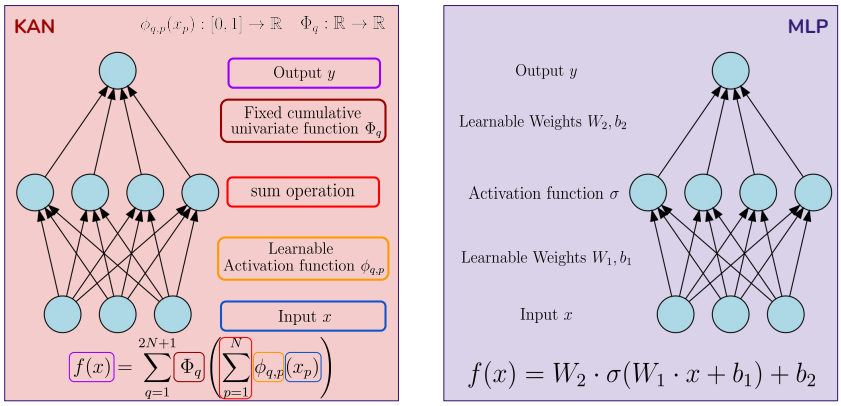


Figure 5.5: Visual comparison between single-hidden layer Kolmogorov-Arnold Networks and Multi-Layer Perceptrons.

fig:5:2:1: KANvsMLP

There have been multiple attempts to use this representation theorems for deep learning purposes, some of which dates back to early 90's and 2000's [Kop02; LU93], up to modern times [II24]. There have been also criticism on the usefulness of KART for this task [GP89; PBL20]. Nevertheless, [Liu+24b] popularised it blending Kolmogorov-Arnold Representation Theorem, symbolic regression, and explainability, giving rise to KAN: Kolmogorov-Arnold Network.

5.2.1 Kolmogorov-Arnold Network

In [Liu+24b], authors introduced the Kolmogorov-Arnold Network (KAN) Figure 5.5. Its shallow version, analogous to the shallow ANN stemming from the Cybenko/Hornik Universal Approximation Theorem (UAT), can be easily done by looking at Equation (5.18):

- **Input Layer:** This layers has n neurons, mapping the input $(x_1, \dots, x_n) \in \mathbb{R}$.
- **First Hidden Layer:** This layer has two ingredients:
  a. a set of $n(2n+1)$ *learnable*, *univariate* activation functions $\phi_{q,p}$, mapping each *individual* input components in $\mathbb{R}$, $x_p \mapsto \phi_{q,p}(x_p)$;
  b. A sum operation on each q node, $(\phi_{q,1}(x_1), \dots, \phi_{q,n}(x_n)) \to \sum_{p=1}^{n} \phi_{q,p}(x_p) =: z_q$.
- **Second hidden layer:** This layers takes the $2n+1$ nodes (neurons), and applies the cumulative activation $\Phi_q : z_q \to \Phi_q(z_q)$;
- **Output Layer:** This layers simply maps the activated nodes of the second hidden layer into the output, $\Phi_q(z_q) \to \tilde{f}$, via the sum $(\Phi_1(z_1), \dots, \Phi_{2n+1}(z_{2n+1})) \mapsto \sum_{q=1}^{2n+1} \Phi_q(z_q)$.

This construction is quite analogous to the standard shallow ANN; furthermore, [Liu+24b] furnished a way to *going deep* with KANs, by stacking

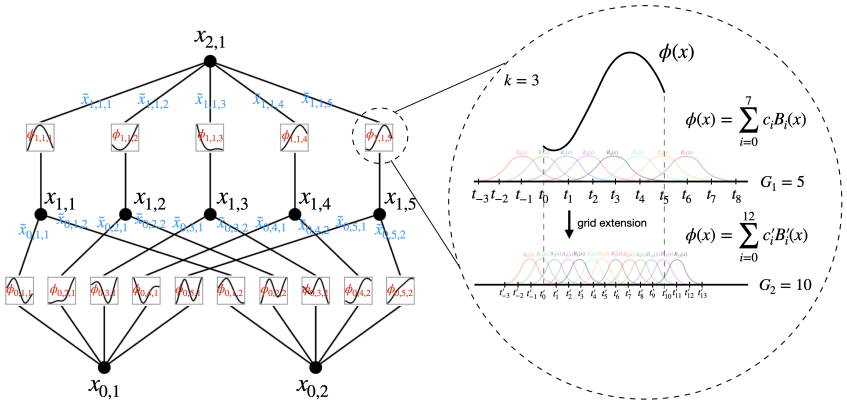


Figure 5.6: *Left*: Notations of activations that flow through the network. *Right*: an activation function is parametrised as a B-spline, which allows switching between coarse-grained and fine-grained grids. From [Liu+24b].

fig:KAN_spline_notation

model	KAN	MLP
shallow	$f(x_1,\dots,x_n) = \sum_{q=1}^{2n+1} \Phi_q\Big(\sum_{p=1}^{n} \phi_{q,p}(x_p)\Big)$	$f(x_1,\dots,x_n) = \sum_{i=1}^{N} c_i\, g(w_i\cdot x + b_i)$
deep	$\mathrm{KAN}(\mathbf{x}) = (\Phi_L \circ \cdots \circ \Phi_1)(\mathbf{x})$	$\mathrm{MLP}(\mathbf{x}) = W_L\, \sigma(W_{L-1}\, \sigma(\cdots \sigma(W_1\mathbf{x} + b_1)\cdots) + b_{n-1}) + b_n$

Table 5.1: Comparison between Kolmogorov–Arnold networks (KAN) and standard multilayer perceptrons (MLP). The *shallow* row shows canonical representation; the *deep* row shows compositional forms.

tab:5:2:1: kan_vs_mlp

multiple layers of the form

$$\Phi_\ell = (\Phi_1^{(n)}, \dots \Phi_{2n+1}^{(\ell)}), \quad \Phi_q^{(\ell)} := \Phi_q\Big(\sum_{p=1}^{n} \phi_{q,p}(x_p)\Big).$$

More crucially, they suggest a way to make $\phi_{q,p}$ *learnable*, by means of B-splines (see Section 2.4, and Figure 2.10). The key trick to define the KAN layer are:

1. **Residual Activation Functions:** They include a basis function $b(x)$ (analogous to residual connection, so to speak) such that the generic activation function is the sum of the basis function $b(x)$ and the spline:

$$\phi(x) = w_b b(x) + w_s\, \mathrm{spline}(x)\,, \tag{5.19}$$

{eq:5:2:1: KANlayer}

where

$$b(x) = \mathrm{silu}(x) := \frac{x}{1+\mathrm{e}^x}\,,$$

and, crucially for [Liu+24b],

$$\mathrm{spline}(x) = \sum_i c_i B_i(x)\,, \tag{5.20}$$

{eq:5:2:1: BsplineKANeq}

where c_i are *trainable* parameters, and also w_b and w_s are trainable parameters.

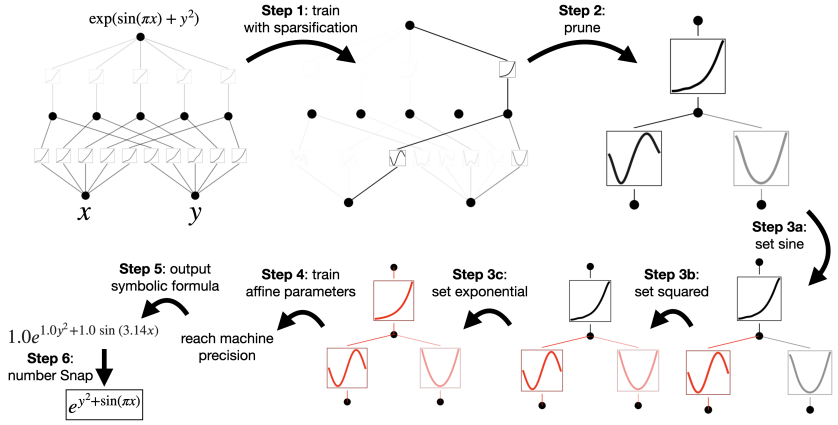


Figure 5.7: An example of how to do symbolic regression with KAN. From [Liu+24b].

fig:5:2:1: KAN_symbolic_ regression

2. **Initialisation scales:** Each activation function is initialized to have $w_s = 1$ and $\text{spline}(x) \approx 0$, while w_b is initialised using Xavier initialisation.

3. **Update of spline grids:** They update each grid on the fly according to its input activations, to address the issue that splines are defined on bounded regions but activation values can evolve out of the fixed region during training.

One of the main, crucial, aspects of KANs, is their potential for interpretability: by sparsifying the training, is it possible to use KANs for *symbolic regression*, i.e., to get a compact, symbol representation of the trained function (see Figure 5.7).

Authors suggest to adapt the standard L1 regularisation for MLP, which favours sparsity. The crucial differences is that (a) in KANs, the norms is defined on the learned activation functions; and (b) L1 is (found empirically to be) insufficient for sparsification of KANs, so entropy regularisation is added.

The total sparsification loss term is

$$\ell_{\text{spars}} = \lambda \left(\mu_1 \sum_{l=1}^{L} |\Phi_l|_1 + \mu_1 \sum_{l=1}^{L} S(\Phi_l) \right) , \tag{5.21}$$

where

$$|\Phi_l|_1 \coloneqq \sum_{i=1}^{n_{in}} \sum_{j=1}^{n_{out}} |\phi_{i,j}|_1 , \qquad |\phi|_1 \coloneqq \frac{1}{N_p} \sum_{s=1}^{N_p} \left| \phi(x^{(s)}) \right| , \tag{5.22}$$

$$S(\Phi) \coloneqq -\sum_{i=1}^{n_{in}} \sum_{j=1}^{n_{out}} \frac{|\phi_{i,j}|_1}{|\Phi|_1} \log \left[\frac{|\phi_{i,j}|_1}{|\Phi|_1} \right] , \tag{5.23}$$

i.e., the L1 norm of an activation function ϕ is its average magnitude over its N_p inputs, while for the entire KAN layer Φ with n_{in} inputs and n_{out} outputs, its norm is simply the sum of the L1 norms of all its activation functions.

The crucial steps, after sparsified training, are *pruning* and *symbolification*;

Pruning: Sparsify KANs on the node level (rather than on the edge level). For each node (say the i^{th} neuron in the l^{th} layer), let us define its *incoming* and *outgoing* score as

$$I_{l,i} \coloneqq \max_k \left(|\phi_{l+1,i,k}|_1 \right) , \tag{5.24}$$

$$O_{l,i} \coloneqq \max_k \left(|\phi_{l-1,i,k}|_1 \right) , \tag{5.25}$$

and consider a node to be important if both incoming and outgoing scores are greater than a threshold hyperparameter (usually, $\theta = 10^{-2}$). Then, all unimportant neurons are pruned.

Symbolification: Symbolification means replacing the (unpruned) activation functions with symbolic expression via per-function fitting. Getting preactivations x and postactivations y from samples, we simplfy fit the affine parameter sets (a, b, c, d) such that, called f the symbolc expression (e.g., $\sin, \exp, \tanh$, etc), we have $y \approx cf(ax + b) + d$. The fitting is done by iterative grid search of (a, b) and linear regression.

5.2.1.1 Critiques to KAN

Following the introduction of Kolmogorov-Arnold Networks, it is essential to critically examine their practical limitations and theoretical foundations. As noted before, early critiques were pointed out on the applicability of the Kolmogorov-Arnold representation theorem in the context of neural networks. These classical theoretical objections, combined with recent systematic empirical evaluations, provide a nuanced perspective on the promise and current limitations of KANs.

Historical Theoretical Critiques: The relevance of the Kolmogorov-Arnold representation theorem for constructing practical neural networks has been questioned since the late 1980s. The core of this critique, articulated by [GP89] and later expanded by [PBL20], rests on several fundamental limitations.

First, a crucial obstacle lies in the regularity, or smoothness, of the functions involved. For a neural network to learn effectively and generalize from noisy data, the functions it represents should ideally be smooth. However, as established by Vitushkin in 1954 [Vit04; Vit54], and highlighted by [GP89], the Kolmogorov-Arnold theorem is a "negative theorem" for smooth representations. Vitushkin proved that if one insists on the inner functions of the representation being smooth (e.g., continuously differentiable), then the theorem's construction breaks down. This means that the "elper functions" in the theorem are inherently non-smooth, or pathological, which is undesirable for learning from data.

Second, the representation is not adaptive. The specific form of the outer function and the inner functions in the Kolmogorov-Arnold theorem are not universal; they depend intimately on the specific target function being represented. As [GP89] argue, this means that for every new function we wish to learn, we would, in theory, require a completely new set of helper functions. This contrasts sharply with the goal of machine learning, where we seek a fixed-architecture model that can be adjusted to approximate many different functions from a given family.

Third, and perhaps most critically, the theorem does not circumvent the curse of dimensionality in practice. [PBL20] frame this by arguing that

the Kolmogorov-Arnold representation merely shifts the complexity into the inner functions, which must be highly unstructured and non-smooth to work universally. Therefore, the representation does not offer a practical advantage for approximating high-dimensional functions, as the complexity is hidden rather than eliminated.

Modern Empirical Findings: regardless of these theoretical concerns, recent systematic and fair comparisons have revealed significant practical challenges regarding computational efficiency, training stability, and scalability of KANs.

A primary critique, as rigorously investigated by [YYW24], concerns the fairness of comparisons often made in favour of KANs. Their study demonstrates that many of the performance gains attributed to KANs can be replicated by simply improving MLP training protocols with modern techniques such as better weight initialization, learning rate schedules, and activation functions. Their extensive experiments across language modelling, vision, and graph learning tasks suggest that, when evaluated under a fair and consistent framework, the superiority of KANs is not as pronounced. They conclude that KANs are particularly sensitive to hyperparameter choices and exhibit a high variance in performance, making them less robust and reliable for general-purpose deep learning tasks compared to their MLP counterparts [YYW24].

Further deepening this critique, [PD24] focus specifically on the convergence dynamics and computational overhead of B-spline-based KANs. Their analysis reveals that the training dynamics of these KANs are fundamentally different from those of MLPs, often leading to slower convergence and difficulties in optimization. The study highlights that the expressive power gained from learnable spline functions comes at a steep computational cost, as the piecewise polynomial evaluations and the associated parameter updates are significantly more expensive than the simple linear transformations in MLPs. This computational inefficiency, as argued by [PD24], severely limits the scalability of KANs to larger datasets and deeper architectures.

In the specific context of scientific machine learning, [Shu+24] provide a comprehensive comparison between MLPs and KANs for solving differential equations and within operator networks. Their findings corroborate and extend the previous critiques. While acknowledging that KANs can achieve high accuracy with a very small number of parameters for certain problems, [Shu+24] demonstrate that this efficiency does not translate to faster training. On the contrary, they report that KANs are consistently slower to train than MLPs by a significant margin. Moreover, their study reveals that KANs can be unstable and highly sensitive to the choice of hyperparameters like grid size and spline order, whereas modern MLPs with appropriate activations and architectures often provide a more robust and computationally efficient baseline for solving partial differential equations and learning operators [Shu+24].

For additional critiques, studies, applications of KANs, see also [Dut+25; Hou+24; Noo+25; Noo25].

5.2.2 Alternative KAN implementations

Following the revival of Kolmogorov-Arnold Networks, the immediate goal was to speed up its training and inference. Furthermore, multiple architectural modification were introduced, replacing B-splines with faster, easier, more

common univariate function interpolators. A nice living review repo on KANs, their implementations, and applications (with link to GitHub code bases and papers) is [min24].

5.2.2.1 EfficientKAN

The first relevant work to speed up the original KAN implementation was Efficient-KAN (eKAN) [Ble24]; as its author stated,

> «*the performance issue of the original implementation is mostly because it needs to expand all intermediate variables to perform the different activation functions. For a layer with* `in_features` *input and* `out_features` *output, the original implementation needs to expand the input to a tensor with shape* (`batch_size, out_features, in_features`) *to perform the activation functions. However, all activation functions are linear combination of a fixed set of basis functions which are B-splines; given that, we can reformulate the computation as activate the input with different basis functions and then combine them linearly. This reformulation can significantly reduce the memory cost and make the computation a straightforward matrix multiplication, and works with both forward and backward pass naturally.*»

Another fact hampering the training speed was the L1 regularisation to foster sparsification. As we have see, the original L1 regularization is defined on the input samples, which requires non-linear operations on the (`batch_size, out_features, in_features`) tensor, and is thus not compatible with the Efficient-KAN reformulation. So, in Efficient-KAN, the L1 regularization of the activation function is replaced with a L1 regularization on the weights, which is more common in neural networks and is compatible with the reformulation.

This kind of implementation was immediately used in the PINN community, as one of the earlier work using KAN for PINNs, [Wan+25], one of the first to introduce the PIKAN nomenclature (together with [Shu+24]), implemented Efficient-KAN for tackling differential problem. The workflow is as easy as simply replacing the approximator function $\hat{u}_\theta(\mathbf{x}) \leftarrow \text{DNN}_\theta(\mathbf{x})$ with $\hat{u}_\theta(\mathbf{x}) \leftarrow \text{eKAN}_\theta(\mathbf{x})$.

5.2.2.2 Using Radial Basis: FastKAN

In [Li24], the author pointed out that

> «*the 3-order B-splines used in Kolmogorov-Arnold Networks (KANs) can be well approximated by Gaussian radial basis functions. Doing so leads to FastKAN, a much faster implementation of KAN which is also a radial basis function (RBF) network.*»

The point is quite easy; in Equation (5.19), instead of using the B-splines of Equation (5.20), FastKAN uses gaussian radial basis function

$$\phi(x) = w_b b(x) + w_s \left(\sum_i k_i \, \mathrm{rbf}(|\mathrm{Lin}(x) - c_i|) \right), \quad k_i, c_i \in \mathbb{R}, \tag{5.26}$$

where $\mathrm{Lin}(x) = Ax + b$ is a linear layer. Notice that this expression can be easily parallelised!

$$\phi(\mathbf{x}) = w_b b(\mathbf{x}) + w_s \left(\mathbf{k} \odot \mathrm{rbf}(|\mathrm{Lin}(\mathbf{x}) - \mathbf{c}|) \right) \cdot 1_d, \tag{5.27}$$

where $\mathbf{k}, \mathbf{c} \sim$ (`in_dim`, `grid_size`), $\odot$ represents the per-element product, and $\cdot 1_d$ denotes the sum over the `grid_size` dimension.

This fact will return in the next approaches.

5.2.2.3 Chebyshev-KAN

As already noticed with FastKAN, the main innovation of KAN is to rely in Kolmogorov-Arnold Theorem, and find a way to learn the univariate activation functions. KAN/eKAN used B-splines, FastKAN uses RBF. But a pletora of common basis functions exists, such as the Chebyshev Polynomials [Čeb53]. Using that, authors of [SAK+24] introduced the Chebyshev-KAN (cKAN). For $x \in [-1, +1]$, Chebyshev polynomials of the first kind are special functions defined iteratively as

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x\,, \\ T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x), \quad \forall n \geq 2. \end{aligned}$$

It can be proven that they admits the compact form

$$T_n(x) = \cos\left(n \arccos(x)\right),$$

which may or may not be more useful from the numerical standpoint. The Chebyshev polynomial of second kind are

$$\begin{aligned} U_0(x) &= 1, \\ U_1(x) &= 2x\,, \\ U_n(x) &= 2xU_{n-1}(x) - U_{n-2}(x), \quad \forall n \geq 2, \end{aligned}$$

i.e., they are identically to the one of the first kind but for the rule for $n = 1$. They can be expressed also as

$$U_n(x) = \frac{\sin((n+1)\arccos(x))}{\sqrt{1-x^2}}\,.$$

They are *orthogonal polynomials*, i.e.

$$\int_{-1}^{+1} \frac{T_m(x)T_n(x)}{\sqrt{1-x^2}}\,\mathrm{d}x = \begin{cases} 0, & n \neq m \\ \frac{\pi}{2}, & n = m = 0, \\ \pi, & n = m \neq 0. \end{cases} \tag{5.28}$$

$$\int_{-1}^{+1} U_m(x)U_n(x)\sqrt{1-x^2}\,\mathrm{d}x = \begin{cases} 0, & n \neq m \\ \frac{\pi}{2}, & n = m, \end{cases} \tag{5.29}$$

The cKAN layer output is computed by performing an Einstein summation operation (einsum) over the Chebyshev polynomials tensor $\mathrm{T} \sim (B, \texttt{input\_dim}, \texttt{degree})$ and the Chebyshev coefficients tensor $\mathrm{C} \sim (B, \texttt{output\_dim}, \texttt{degree})$. This operation effectively combines the polynomial bases with the learned coefficients to produce the final output tensor

$$\mathbf{y}_o = \sum_{i=1}^{\texttt{input\_dim}} \sum_{j=0}^{\texttt{degree}} \mathrm{T}_{bij} \cdot \mathrm{C}_{boj}(\mathbf{x})\,.$$

The $\mathrm{C}_{boj}(\mathbf{x})$ tensor is built easily following three steps:

1. Nornmalise the input $x \mapsto \tilde{x} = \tanh(x)$
2. The Chebyshev polynomials - up to a degree n (hyperparameter) are generated, $\tilde{x} \mapsto [T_0(\tilde{x}), T_1(\tilde{x}), \ldots, T_n(\tilde{x})]$
3. They are combined using a set of learnable parameters $\Theta \sim (d_{in}, d_{out}, n+1)$.

Listing 5.1: Torch implementation of ChebyshevKAN (cKAN). From [Spa24].

```
import torch
import torch.nn as nn

class ChebyKANLayer(nn.Module):
    def __init__(self, input_dim, output_dim, degree):
        super(ChebyKANLayer, self).__init__()
        self.inputdim = input_dim
        self.outdim = output_dim
        self.degree = degree

        self.cheby_coeffs = nn.Parameter(torch.empty(input_dim,
            output_dim, degree + 1))
        nn.init.normal_(self.cheby_coeffs, mean=0.0, std=1/(input_dim
            * (degree + 1)))

    def forward(self, x):
        x = torch.reshape(x, (-1, self.inputdim))  # shape = (
            batch_size, inputdim)
        # Since Chebyshev polynomial is defined in [-1, 1]
        # We need to normalize x to [-1, 1] using tanh
        x = torch.tanh(x)
        # Initialize Chebyshev polynomial tensors
        cheby = torch.ones(x.shape[0], self.inputdim, self.degree + 1,
             device=x.device)
        if self.degree > 0:
            cheby[:, :, 1] = x
        for i in range(2, self.degree + 1):
            cheby[:, :, i] = 2 * x * cheby[:, :, i - 1].clone() -
                cheby[:, :, i - 2].clone()
```

```
        # Compute the Chebyshev interpolation
        y = torch.einsum('bid,iod->bo', cheby, self.cheby_coeffs)  #
            shape = (batch_size, outdim)
        y = y.view(-1, self.outdim)
        return y
```

5.2.2.4 Jacobi-KAN

A more general basis function for the $[-1,+1]$ interval are the Jacobi Polynomials, which, for certain values of their defining parameters, reduces to Chebyshev polynomials.

For parameters $\alpha, \beta > -1$, the *Jacobi polynomials* $\{P_n^{(\alpha,\beta)}(x)\}_{n\geq 0}$ form a family of orthogonal polynomials on $[-1,1]$ with respect to the weight

$$w_{\alpha,\beta}(x) = (1-x)^\alpha (1+x)^\beta .$$

They satisfy the orthogonality relation

$$\int_{-1}^{1} P_m^{(\alpha,\beta)}(x)\, P_n^{(\alpha,\beta)}(x)\, (1-x)^\alpha (1+x)^\beta \,\mathrm{d}x = \delta_{mn} \mathcal{N}_{m,n}^{\alpha,\beta} ,$$

where the normalsiation factor is

$$\mathcal{N}_{m,n}^{\alpha,\beta} = \frac{2^{\alpha+\beta+1}}{2n+\alpha+\beta+1} \frac{\Gamma(n+\alpha+1)\Gamma(n+\beta+1)}{n!\Gamma(n+\alpha+\beta+1)} .$$

Jacobi polynomials obey the recurrence

$$\begin{aligned} P_0^{(\alpha,\beta)}(x) &= 1, \\ P_1^{(\alpha,\beta)}(x) &= \tfrac{1}{2}\big[(2+\alpha+\beta)x + (\alpha-\beta)\big], \end{aligned}$$

and for $n \geq 1$,

$$\begin{aligned} (n+1)(n+\alpha+\beta+1)\, P_{n+1}^{(\alpha,\beta)}(x) = \\ (2n+\alpha+\beta+1)\Big[(2n+\alpha+\beta+2)x + \alpha^2 - \beta^2\Big] P_n^{(\alpha,\beta)}(x) \\ -\ 2(n+\alpha)(n+\beta)(2n+\alpha+\beta+2)\, P_{n-1}^{(\alpha,\beta)}(x). \end{aligned}$$

Jacobi polynomials include several classical families:

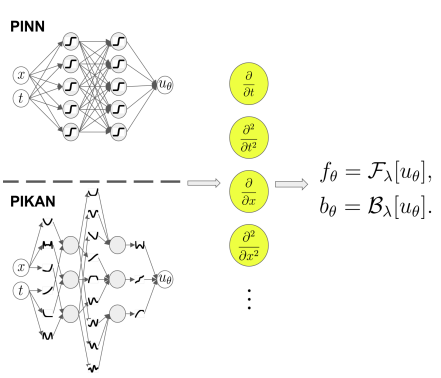
$$\begin{aligned} P_n^{(0,0)}(x) &= P_n(x)\,, && \text{(Legendre polynomials)} \\ P_n^{(-\frac{1}{2},-\frac{1}{2})}(x) &= \frac{\Gamma(n+\frac{1}{2})}{\sqrt{\pi}\,\Gamma(n+1)} T_n(x)\,, && \text{(Chebyshev of the first kind)} \\ P_n^{(\frac{1}{2},\frac{1}{2})}(x) &= \frac{\Gamma(n+\frac{3}{2})}{\sqrt{\pi}\,\Gamma(n+1)} U_n(x)\,, && \text{(Chebyshev of the second kind).} \end{aligned}$$

Thus Chebyshev and Legendre polynomials arise as particular choices of (α, β), making Jacobi polynomials an unifying basis family.

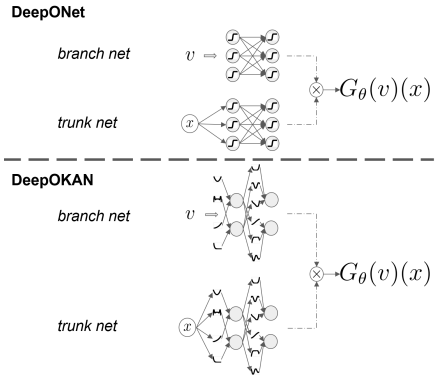
From here, it is immediate to replace Chebyshev polynomials with Jacobi Polynomials, and thus define the JacobiKAN (jKAN).

Listing 5.2: Torch implementation of JacobiKAN (jKAN). From [Spa24].

```python
import torch
import torch.nn as nn

class JacobiKANLayer(nn.Module):
    def __init__(self, input_dim, output_dim, degree, a=1.0, b=1.0):
        super(JacobiKANLayer, self).__init__()
        self.inputdim = input_dim
        self.outdim   = output_dim
        self.a        = a
        self.b        = b
        self.degree   = degree

        self.jacobi_coeffs = nn.Parameter(torch.empty(input_dim,
            output_dim, degree + 1))
        nn.init.normal_(self.jacobi_coeffs, mean=0.0, std=1/(input_dim
             * (degree + 1)))

    def forward(self, x):
        x = torch.reshape(x, (-1, self.inputdim))  # shape = (
            batch_size, inputdim)
        # Since Jacobian polynomial is defined in [-1, 1]
        # We need to normalize x to [-1, 1] using tanh
        x = torch.tanh(x)
        # Initialize Jacobian polynomial tensors
        jacobi = torch.ones(x.shape[0], self.inputdim, self.degree +
            1, device=x.device)
        if self.degree > 0:
            # degree = 0: jacobi[:, :, 0] = 1 (already initialized) ;
            # degree = 1: jacobi[:, :, 1] = x ;
            jacobi[:, :, 1] = ((self.a-self.b) + (self.a+self.b+2) * x
                ) / 2
        for i in range(2, self.degree + 1):
            theta_k  = (2*i+self.a+self.b)*(2*i+self.a+self.b-1) / (2*
                i*(i+self.a+self.b))
            theta_k1 = (2*i+self.a+self.b-1)*(self.a*self.a-self.b*
                self.b) / (2*i*(i+self.a+self.b)*(2*i+self.a+self.b-2)
                )
            theta_k2 = (i+self.a-1)*(i+self.b-1)*(2*i+self.a+self.b) /
                 (i*(i+self.a+self.b)*(2*i+self.a+self.b-2))
            jacobi[:, :, i] = (theta_k * x + theta_k1) * jacobi[:, :,
                i - 1].clone() - theta_k2 * jacobi[:, :, i - 2].clone
                ()  # 2 * x * jacobi[:, :, i - 1].clone() - jacobi[:,
                :, i - 2].clone()
        # Compute the Jacobian interpolation
        y = torch.einsum('bid,iod->bo', jacobi, self.jacobi_coeffs)  #
             shape = (batch_size, outdim)
        y = y.view(-1, self.outdim)
        return y
```



(a) Physics-Informed Networks. Top, PINN; bottom, PIKAN.



(b) Operator Networks. Top, DeepONet; bottom, DeepOKAN.

Figure 5.8: An illustration of MLP and KAN for **(a)** differential equations and **(b)** operator networks (DeepONet [Lu+21] is used as the representation model for operator learning). From [Shu+24].

fig:5:2:3: PIKANvsPINN

5.2.3 KAN for PDEs: PIKAN and DeepOKan

As we have already briefly stated, it is straightforward to incorporate KANs in Physics-Informed task, either for vanilla, inverse, parametric, etc. [Shu+24]. Furthermore, it is possible to replace MLP blocks in Deep Operator Networks to build up a Deep Operator Kolmogorov Arnold Networks (DeepOKAN). In [Shu+24], authors compared performances of PINN, PIKAN on certain benchmark tasks, and DeepONets vs DeepOKANs on certain others.

They have found that both MLPs and KANs achieve high, compared accuracies, but, generally, training KANs is overall slower and less data-efficient. Interestingly, cPIKAN (Chebyshev PIKAN) can achieve the same accuracy with far less network parameters, but it takes more GPU hours to train. Nevertheless, the DeepOKAN has shown competitive performance in operator learning compared to the DeepONet, indicating DeepOKAN as a promising alternative representation model. Furthermore, it is significantly more robust to noisy input functions in the testing stage after being trained with clean data.

5.3 Learning Operators, Part II: Neural Operators - formal theory

sec:5:3: NOsTheory

Let us now go back to Operator Learning. Following [Bha+21; DKA25; Kov+23], we want now to formally introduce the concept of Neural Operators, its formal definition, and its underlying representation properties.

The main properties we want our *Neural Operators* G_θ to obey are, loosely speaking

1. Maps infinite-dimensional (Banach/Hilbert) spaces to infinite-dimensional (Banach/Hilbert) spaces;

2. Admits an explicit realisation for certain input-function values, so that we can use the Neural Operator to approximate a target Banach space

Property \ Model	NNs	DeepONets	Interpolation	Neural Operators
Discretization Invariance	✗	✗	✓	✓
Is the output a function?	✗	✓	✓	✓
Can query the output at any point?	✗	✓	✓	✓
Can take the input at any point?	✗	✗	✓	✓
Universal Approximation	✗	✓	✗	✓

Table 5.2: Comparison of deep learning models. The first row indicates whether the model is discretization invariant. The second and third rows indicate whether the output and input are functions. The fourth row indicates whether the model class is a universal approximator of operators. Neural Operators are discretization invariant deep learning methods that output functions and can approximate any operator. From [Kov+23].

tab:5:4: deeplearning_ comparison

function at any point in its domain;

3. Satisfy an Universal Approximation Theorem of some sort.

We have already sees an Operator Network formalism, the Deep Operator Network (DeepONet), relying on the universal approximation theorem for operators. These are a class of operators that consist of a branch net and a trunk net. The trunk-net allows queries at any point, but the branch-net constrains the input to fixed locations. This fact breaks explicitly the discretisation invariance property (Table 5.2). However, it is possible to modify the branch-net to make the methodology discretization invariant, for example by using the PCA-based approach as used in [Hoo+22].

Let us follow [Kov+23] and require a discretization-invariant model with a fixed number of parameters to satisfy the following:

1. acts on any discretization of the input function, i.e. accepts any set of points in the input domain,

2. can be evaluated at any point of the output domain,

3. converges to a continuum operator as the discretization is refined.

The first two requirements of accepting any input and output points in the domain is a natural requirement for discretization invariance, while the last one ensures consistency in the limit as the discretization is refined.

Before delving into the formalism of Neural Operators, let us stop for a moment and re-focus on a specific task: a pure-boundary problem of the form

$$\mathrm{L}_a[u](x) = f(x)\,, \quad x \in D \subset \mathbb{R}^d, \tag{5.30}$$

$$u(x) = 0\,, \qquad x \in \partial D. \tag{5.31}$$

This may represent, for example, the stationary Advection-Reaction-Diffusion PDE, stationary Navier-Stokes, etc. Relying on intuition from the weak formulation of such a boundary problem, we can see that $u \in \mathcal{U}(D;\mathbb{R})$ of $u : D \to \mathbb{R}$, while $f \in \mathcal{U}^*$ via the map

$$u \mapsto \int_D fu.$$

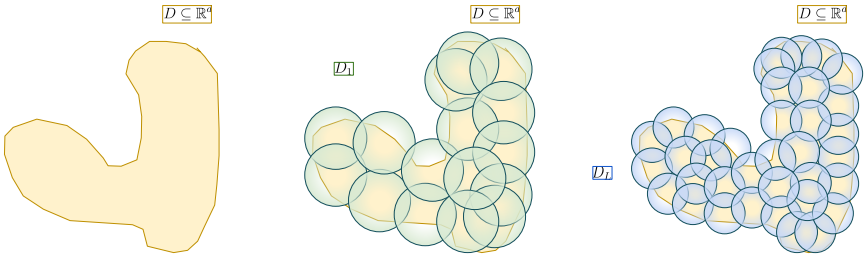


fig:5:3:Domain_refinement

Figure 5.9: Visual representation of the Domain refinement.

The subscript a of the differential operator L_a refers to a function $a \in \mathcal{A}(D' \subset \mathbb{R}^{d_a}; \mathbb{R})$ (for example, the permeability field in the stationary Darcy Flow PDE). In this sense, $\mathrm{L} : \mathcal{A} \times \mathcal{U} \to \mathcal{U}^*$, or, chosen $a \in \mathcal{A}$, $\mathrm{L}_a : \mathcal{U} \to \mathcal{U}^*$. Thus, if such operator can be inverted, we can build

$$G^\dagger \coloneqq \mathrm{L}_a^{-1} f : \mathcal{A} \to \mathcal{U}, \text{ s.t. } G^\dagger : a \mapsto u, \tag{5.32}$$

i.e., $G^\dagger$ maps the *parameter function* a to the solution of the PDE,

$$u_{\mathrm{sol}}(x) = G^\dagger[a](x), \Rightarrow \mathrm{L}_a[G^\dagger[a]](x) = \underbrace{\mathrm{L}_a \mathrm{L}_a^{-1}}_{1} f = f. \tag{5.33}$$

Our goal is to learn this mapping, using a finite collection of observations (input/output). I.e. we aim to build

$$G_\theta : \mathcal{A} \times \Theta \to \mathcal{U}, \tag{5.34}$$

where we want to pick up the optimal $\theta^\dagger \in \Theta$ such that $G_{\theta^\dagger} \approx G^\dagger$.

We sample the parameter functions as i.i.d. samples from a certain probability measure, while the solutions are push-forwards

$$\left\{a^{(i)}, u^{(i)} = G^\dagger[a^{(i)}]\right\}_{i=1,\dots,N}$$

where $a^{(i)} \sim \mu$ are i.i.d. samples drawn from the probability measure μ with support on $\mathcal{A}$. But these functions are defined on a certain domain, $D' \subset \mathbb{R}^{d_a}$ and $D \subset \mathbb{R}^d$. Then

Definition 5.3.1. We call a *discrete refinement* of the domain $D \subset \mathbb{R}^d$ any sequence of nested sets $D_1 \subset D_2 \subset \cdots \subset D$ with $|D_L| = L$ for any $L \in \mathbb{N}$ such that, for any $\epsilon > 0$, there exists a number $L = L(\epsilon) \in \mathbb{N}$ such that

$$D \subseteq \bigcup_{x \in D_L} \{y \in \mathbb{R}^d : \|y - x\|_2 < \epsilon\}.$$

Definition 5.3.2. Given a discrete refinement $(D_L)_{L=1}^\infty$ of the domain $D \subset \mathbb{R}^d$, any member D_L is called a *discretization* of D.

To build our approximation G_θ, we need to cast the problem to an optimisation problem, where we can control the error of the approximation (on average) with respect to the probability measure

$$\|G^\dagger - G_\theta\|_{L^2_\mu(\mathcal{A};\mathcal{U})} = \mathbb{E}_{a\sim\mu}\left[\|G^\dagger - G_\theta\|_{\mathcal{U}}^2\right] = \int_{\mathcal{A}} \|G^\dagger - G_\theta\|_{\mathcal{U}}^2 \,\mathrm{d}\mu(a), \tag{5.35}$$

which, using the Monte Carlo integration approach, give rise to the following minimisation problem

$$\theta^\dagger = \underset{\theta\in\Theta}{\operatorname{argmin}}\, \mathbb{E}_{a\sim\mu}\left[\|G^\dagger - G_\theta\|_{\mathcal{U}}^2\right] \overset{\text{M.C.I.}}{\approx} \underset{\theta\in\Theta}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N}\left\|u^{(i)} - G_\theta[a^{(i)}]\right\|_{\mathcal{U}}^2 . \tag{5.36}$$

Now, since we are treating *functions*, and since we have defined a way to *discrete refine* the domains where these functions are defined on, we can use the *pointwise evaluation* of these functions at a set of L points, giving rise to the (discrete) dataset

$$\{(x_\ell, a(x_\ell))\}_{\ell=1,\dots,L},$$

which may be viewed as a vector in $(\mathbb{R}^d \times \mathbb{R}^m)^L = \mathbb{R}^{d\cdot L} \times \mathbb{R}^{m\cdot L}$. The increase of L to obtain a denser domain is often called a *mesh refinement*.
We have all the needed ingredients to define the optimisation problem:

Definition 5.3.3 (Discretised Uniform Risk)**.** Suppose Å is a Banach space of $\mathbb{R}^m$-valued functions on the domain $D \subset \mathbb{R}^d$. Let $\mathcal{G} : \mathcal{A} \to \mathcal{U}$ be an operator, D_L be an L-point discretization of D, and $\hat{\mathcal{G}} : \mathbb{R}^{Ld} \times \mathbb{R}^{Lm} \to \mathcal{U}$ some map. For any $K \subset \mathcal{A}$ compact, we define the *discretized uniform risk* as

$$R_K(\mathcal{G}, \hat{\mathcal{G}}, D_L) = \sup_{a\in K}\left\|\hat{\mathcal{G}}(D_L, a|_{D_L}) - \mathcal{G}(a)\right\|_{\mathcal{U}} .$$

5.3.1 Neural Operators

With these concept in mind, we can define the Neural Operator as composed by the following operations:

1. **Lifting:** Using a pointwiRse function $\mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}$, map the input $\{a : D \to \mathbb{R}^{d_a}\} \mapsto \{v_0 : D \to \mathbb{R}^{d_{v_0}}\}$ to its first hidden representation. Usually, we choose $d_{v_0} > d_a$ and hence this is a lifting operation performed by *a fully local operator*.

2. **Iterative Kernel Integration**: For $t = 0, \dots, T-1$, map each hidden representation to the next $\{v_t : D_t \to \mathbb{R}^{d_{v_t}}\} \mapsto \{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\}$ via the action of the sum of a local linear operator, a non-local integral kernel operator, and a bias function, composing the sum with a fixed, pointwise nonlinearity. Here we set $D_0 = D$ and $D_T = D'$ and impose that $D_t \subset \mathbb{R}^{d_t}$ is a bounded domain.

3. **Projection**: Using a pointwise function $\mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u}$, map the last hidden representation $\{v_T : D' \to \mathbb{R}^{d_{v_T}}\} \mapsto \{u : D' \to \mathbb{R}^{d_u}\}$ to the output function. Analogously to the first step, we usually pick $d_{v_T} > d_u$ and hence this is a projection step performed by *a fully local operator*.

The outlined structure mimics that of a finite dimensional neural network where hidden representations are successively mapped to produce the final output. In particular, we have

$$\mathcal{G}_\theta := \mathcal{Q} \circ \sigma_T(W_{T-1} + \mathcal{K}_{T-1} + b_{T-1}) \circ \cdots \circ \sigma_1(W_0 + \mathcal{K}_0 + b_0) \circ \mathcal{P} \tag{5.37}$$

where

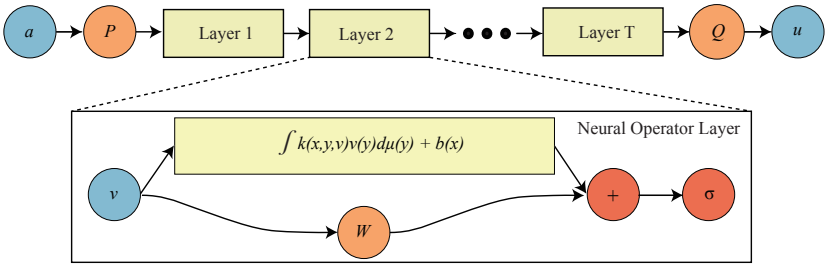


Figure 5.10: Visual representation of the architecture of a generic Neural Operator. Adapted from [Kov+23].

fig:5:3:1: NeurOpSchema

- $\mathcal{P} : \mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}$, $\mathcal{Q} : \mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u}$ are the local lifting and projection mappings respectively;
- $W_t \in \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}}$ are local linear operators (i.e., matrices), $b_t : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}$ are bias functions;
- σ_t are fixed activation functions acting locally as maps $\mathbb{R}^{v_{t+1}} \to \mathbb{R}^{v_{t+1}}$ in each layer;
- crucially, $\mathcal{K}_t : \{v_t : D_t \to \mathbb{R}^{d_{v_t}}\} \to \{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\}$ are the *integral kernel operators*;

The output dimensions $d_{v_0}, \dots, d_{v_T}$ as well as the input dimensions $d_1, \dots, d_{T-1}$ and domains of definition $D_1, \dots, D_{T-1}$ are *hyperparameters* of the architecture.

Notice that, by *local maps*, here we mean that the action is pointwise. In particular, for the *lifting* and *projection* maps, we have $(\mathcal{P}(a))(x) = \mathcal{P}(a(x))$ for any $x \in D$ and $(\mathcal{Q}(v_T))(x) = \mathcal{Q}(v_T(x))$ for any $x \in D'$ and similarly, for the activation, $(\sigma(v_{t+1}))(x) = \sigma(v_{t+1}(x))$ for any $x \in D_{t+1}$. See Figure 5.10 for a visual representation of a Neural Operator.

The crucial difference between this Neural Operator architecture and a standard feed-forward neural network is that all operations are directly defined in function space, and therefore do *not* depend on any discretization of the data.

Intuitively, the lifting step locally maps the data to a space where the non-local part of $\mathcal{G}^\dagger$ is easier to capture. Each integral kernel operator is the function space analogue of the weight matrix in a standard feed-forward network, since they are infinite-dimensional linear operators mapping one function space to another. Furthermore the biases, which are normally vectors, are uplifted to functions and, using intuition from the ResNet architecture, there is also a local linear operator acting on the output of the previous layer before applying the nonlinearity. The final projection step simply gets us back to the space of output function.

But what are those *integral kernel operators*?

In literature we may identify three main versions of the integral kernel operator $\mathcal{K}_t$.

First integral operator family (basic Neural Operator): let $\kappa^{(t)} \in C(D_{t+1} \times D_t; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$ and let ν_t be a Borel measure on D_t. Then we define $\mathcal{K}_t$ by

$$\text{(F1)} \qquad (\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)}(x, y) v_t(y) \, \mathrm{d}\nu_t(y) \,, \qquad \forall x \in D_{t+1}. \tag{5.38}$$

{eq:5:3:1: IntOp1}

This is the easiest family which can be created, with the kernel function κ be dependent only on the layer-input coordinates. By relaxing this, and allowing for multiple dependences, we obtain the other two families.

Second integral operator family: let $\kappa^{(t)} \in C(D_{t+1} \times D_t \times \mathbb{R}^{d_a} \times \mathbb{R}^{d_a}; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$. Then we define $\mathcal{K}_t$ by

$$\text{(F2)} \quad (\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)} \left(x, y; a\left(\Pi^D_{t+1}(x)\right), a\left(\Pi^D_t(y)\right)\right) v_t(y) \mathrm{d}\nu_t(y) \,, \quad \forall x \in D_{t+1}, \tag{5.39}$$

{eq:5:3:1: IntOp2}

where $\Pi^D_t : D_t \to D$ are fixed mappings from the layer-input domain to the $\mathcal{A}$ functions domain. Notice that, now, κ depends on the layer-input coordinates and, thanks to the custom, fixed map Π^D_t, also on the Neural Operator input, a.

Usually, this family is numerically more performant with respect to the first family. A way to look at it is that, if we think to Equation (5.39) as a discrete time dynamical system, then the input $a \in \mathbb{A}$ only enters through the "initial condition"; hence its influence to the Neural Operator forward process diminishes layer by layer. By directly putting in an a-dependence into the kernel, its influence in the entire architecture is ensured.

Third integral operator family - Transformers: let $\kappa^{(t)} \in C(D_{t+1} \times D_t \times \mathbb{R}^{d_{v_t}} \times \mathbb{R}^{d_{v_t}}; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$. Then define $\mathcal{K}_t$ by

$$\text{(F3)} \quad (\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)} \left(x, y; v_t\left(\Pi_t(x)\right)), v_t(y)\right) \, v_t(y) \, \mathrm{d}\nu_t(y) \,, \quad \forall x \in D_{t+1}. \tag{5.40}$$

{eq:5:3:1: IntOp3}

where $\Pi_t : D_{t+1} \to D_t$ are, again, fixed mappings.

Notice that, in contrast to previous families, the integral operator here is *non-linear*, since the kernel κ depends on the input function v_t. With this definition, and a particular choice of kernel κ_t and measure ν_t, we it is possible to show that neural operators are a continuous input/output space generalization of the transformer architecture [Vas+17].

Proposition 5.3.4 (Transformers are Neural Operators)**.** *The Attention Mechanism in transformer models is a special case of Neural Operator layer.*

Proof. Let us start by writing down the single-layer representation of a Neural Operator with Equation (5.40) integral operator, with a simplified settings, i.e., we pick the kernel to *not* explicitly depend on the spatial variables, but only on the input pair :

$$u(x) = \sigma\left(v(x) + \int_D \kappa_v(v(x), v(y)) \, v(y) \, \mathrm{d}y, \right) \quad \forall x \in D.$$

Here, $v : D \to \mathbb{R}^n$ is the input function to the layer, and $u : D \to \mathbb{R}^n$ the output function. κ_v indicates that the kernel depends on the entirety of the function v as well as on its pointwise values $v(x)$ and $v(y)$.

We can now make a choice, and pick a specific form for kernel. In particular, we assume $\kappa_v : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{n\times n}$ does not explicitly depend on the spatial variables (x, y) but only on the input pair $(v(x), v(y))$, and

$$\kappa_v(v(x), v(y)) = g_v(v(x), v(y))\, R,$$

where $R \in \mathbb{R}^{n\times n}$ is a matrix of learnable parameters, and $g_v : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is defined as

$$g_v(v(x), v(y)) = \left(\int_D \exp\left(\frac{\langle Av(s), Bv(y)\rangle}{\sqrt{m}}\right) \mathrm{d}s\right)^{-1} \exp\left(\frac{\langle Av(x), Bv(y)\rangle}{\sqrt{m}}\right).$$

Here $A, B \in \mathbb{R}^{m\times n}$ are again matrices of learnable parameters, $m \in \mathbb{N}$ is a hyperparameter, and $\langle\cdot,\cdot\rangle$ is the Euclidean inner-product on $\mathbb{R}^m$.

From here, it is easy to see that

$$u(x) = \sigma\left(v(x) + \int_D \frac{\exp\left(\frac{\langle Av(x), Bv(y)\rangle}{\sqrt{m}}\right)}{\int_D \exp\left(\frac{\langle Av(s), Bv(y)\rangle}{\sqrt{m}}\right) \mathrm{d}s} R\, v(y)\, \mathrm{d}y\right), \quad \forall x \in D. \tag{5.41}$$

This equation can be seen as a *continuum limit* of the single-head self-attention (plus the skip connection). Indeed, Monte Carlo integration of $\int_D$ gives

$$\int_D \exp\left(\frac{\langle Av(s), Bv(y)\rangle}{\sqrt{m}}\right) \mathrm{d}s \approx \frac{|D|}{k} \sum_{l=1}^{k} \exp\left(\frac{\langle Av_l, Bv(y)\rangle}{\sqrt{m}}\right),$$

where $\{x_1, \ldots, x_k\} \subset D$ is a uniformly-sampled, k-point discretization of D and $v_j = v(x_j) \in \mathbb{R}^n$ and $u_j = u(x_j) \in \mathbb{R}^n$ for $j = 1, \ldots, k$, the consequent discretisations. Plugging it in, and re-doing the Monte Carlo integrations, gives

$$u_j = \sigma\left(v_j + \sum_{q=1}^{k} \frac{\exp\left(\frac{\langle Av_j, Bv_q\rangle}{\sqrt{m}}\right)}{\sum_{l=1}^{k} \exp\left(\frac{\langle Av_l, Bv_q\rangle}{\sqrt{m}}\right)} R\, v_q\right), \quad j = 1, \ldots, k. \tag{5.42}$$

Stated differently, and using

$$\mathrm{softmax}(x)_i = \frac{\mathrm{e}^{x_i}}{\sum_i \mathrm{e}^{x_i}},$$

we have

$$u_j = \sigma\left(v_j + \sum_{q=1}^{k} \mathrm{softmax}\left(\frac{v_q^T B^T\, Av_j}{\sqrt{m}}\right) R\, v_q\right), \quad j = 1, \ldots, k, \tag{5.43}$$

i.e., if we pass to the vectorised form,

$$X : X_j = v_j, \quad K \coloneqq XB \quad Q \coloneqq XA \quad V \coloneqq XR,$$

we have the standard Single-Head Attention (SHA) [Sca24; Vas+17]

$$U = \sigma\Bigg(X + \underbrace{\mathrm{softmax}\left(\frac{Q^T K}{\sqrt{m}}\right) V}_{\mathrm{SHA}(X)}\Bigg). \tag{5.44}$$

■

5.3.2 Universal Approximation Theorems for Neural Operators

In [Kov+23], authors standardise and proved a set of universal approximation theorems for Neural Operators. Here, we briefly reports those results, and we refer the interested reader to the original work for the proofs.

In order to report the universal approximation theorems for neural operators, we first need to fix the notation. First, we can write the hidden single-layer update as

$$v_{t+1}(x) = \sigma\left(W_t v_t\left(\Pi_t(x)\right) + \int_{D_t} \kappa^{(t)}(x,y)\, v_t(y)\, \mathrm{d}\nu_t(y) + b_t(x)\right),$$

valid $\forall x \in D_{t+1}$, and where $\Pi_t : D_{t+1} \to D_t$ are fixed mapping which, since we often consider functions on the same domain, can usually be taken to be the identity. So, to mimic the Cybenko's theorem, we can consider the following two-layer neural operator architecture:

$$\mathcal{G}_\theta[a](x) = \mathcal{Q}\left(\int_D \kappa^{(1)}(x,y)\sigma\left(W_0\mathcal{P}(a(y)) + \int_D \kappa^{(0)}(y,z)\mathcal{P}(a(z))\,\mathrm{d}z + b_0(y)\right)\mathrm{d}y\right), \tag{5.45}$$

{eq:5:3:2: singlehiddenlayer}

for any $x \in D'$. We can now standardise and define the main ingredients of Neural Operators:

(a) Neural Networks;

(b) Activation functions;

(c) Kernel Integral Operators;

Neural Networks (N_n): for any $n \in \mathbb{N}$ and $\sigma : \mathbb{R} \to \mathbb{R}$, we define the set of real-valued n-layer neural networks on $\mathbb{R}^d$ by

$$\begin{aligned}\mathsf{N}_n(\sigma;\mathbb{R}^d) \coloneqq \{f : \mathbb{R}^d \to \mathbb{R} : f(x) = W_n\sigma(\dots W_1\sigma(W_0x + b_0) + b_1 \dots) + b_n,\\ W_0 \in \mathbb{R}^{d_0\times d}, W_1 \in \mathbb{R}^{d_1\times d_0}, \dots, W_n \in \mathbb{R}^{1\times d_{n-1}},\\ b_0 \in \mathbb{R}^{d_0}, b_1 \in \mathbb{R}^{d_1}, \dots, b_n \in \mathbb{R},\ d_0, d_1, \dots, d_{n-1} \in \mathbb{N}\}.\end{aligned}$$

We define the set of $\mathbb{R}^{d'}$-valued neural networks simply by stacking real-valued networks

$$\mathsf{N}_n(\sigma;\mathbb{R}^d,\mathbb{R}^{d'}) \coloneqq \{f : \mathbb{R}^d \to \mathbb{R}^{d'} : f(x) = \big(f_1(x), \dots, f_{d'}(x)\big), f_1, \dots, f_{d'} \in \mathsf{N}_n(\sigma;\mathbb{R}^d)\}.$$

Activation functions (A_m): For any $m \in \mathbb{N}_0$, we define the set of allowable activation functions as the continuous $\mathbb{R} \to \mathbb{R}$ maps which make neural networks dense in $C^m(\mathbb{R}^d)$ on compact set at any fixed depth,

$$\mathsf{A}_m \coloneqq \{\sigma \in C(\mathbb{R}) : \exists n \in \mathbb{N} \text{ s.t. } \mathsf{N}_n(\sigma;\mathbb{R}^d) \text{ is dense in } C^m(K)\ \ \forall K \subset \mathbb{R}^d \text{ compact}\}.$$

Furthermore, we define the set of linearly bounded activations as

$$\mathsf{A}_m^{\mathrm{L}} \coloneqq \left\{\sigma \in \mathsf{A}_m : \sigma \text{ is Borel measurable },\ \sup_{x\in\mathbb{R}} \frac{|\sigma(x)|}{1+|x|} < \infty\right\},$$

noting that any globally Lipschitz, non-polynomial, C^m-function is contained in $\mathsf{A}_m^{\mathrm{L}}$. This definition is quite relevant since most activation functions used in practice fall within this class, for example, ReLU $\in \mathsf{A}_0^{\mathrm{L}}$, ELU $\in \mathsf{A}_1^{\mathrm{L}}$ while $\tanh, \mathrm{sigmoid} \in \mathsf{A}_m^{\mathrm{L}}$ for any $m \in \mathbb{N}_0$.

Finally, saying that the diameter of any set $S \subseteq \mathbb{R}^d$ is defined as, for $|\cdot|_2$ the Euclidean norm on $\mathbb{R}^d$,

$$\mathrm{diam}_2(S) \coloneqq \sup_{x,y \in S} |x - y|_2,$$

Definition 5.3.5 (Bounded Activation)**.** We denote by BA the set of maps $\sigma \in \mathsf{A}_0$ such that, for any compact set $K \subset \mathbb{R}^d$, $\epsilon > 0$, and $C \geq diam_2(K)$, there exists a number $n \in \mathbb{N}$ and a neural network $f \in \mathsf{N}_n(\sigma; \mathbb{R}^d, \mathbb{R}^{d'})$ such that

$$\begin{aligned} |f(x) - x|_2 &\leq \epsilon, \qquad \forall x \in K, \\ |f(x)|_2 &\leq C, \qquad \forall x \in \mathbb{R}^d. \end{aligned}$$

Kernel Integral Operators (IO): Let $D \subset \mathbb{R}^d$ be a domain. For any $\sigma \in \mathsf{A}_0$, we define the set of affine kernel integral operators by

$$\mathsf{IO}(\sigma; D, \mathbb{R}^{d_1}, \mathbb{R}^{d_2}) = \{f \mapsto \int_D \kappa(\cdot, y) f(y) \mathrm{d}y + b : \kappa \in \mathsf{N}_{n_1}(\sigma; \mathbb{R}^d \times \mathbb{R}^d, \mathbb{R}^{d_2 \times d_1}), \\ b \in \mathsf{N}_{n_2}(\sigma; \mathbb{R}^d, \mathbb{R}^{d_2}),\ n_1, n_2 \in \mathbb{N}\},$$

for any $d_1, d_2 \in \mathbb{N}$.

Finally, we have

Definition 5.3.6 (n-layer neural operator)**.** For any $n \in \mathbb{N}_{\geq 2}$, $d_a, d_u \in \mathbb{N}$, $D \subset \mathbb{R}^d$, $D' \subset \mathbb{R}^{d'}$ domains, and $\sigma_1 \in \mathsf{A}_0^{\mathrm{L}}$, $\sigma_2, \sigma_3 \in \mathsf{A}_0$, we define the set of n-layer neural operators by

$$\begin{aligned} \mathsf{NO}_n(\sigma_1, \sigma_2, \sigma_3; D, D', \mathbb{R}^{d_a}, \mathbb{R}^{d_u}) = \{ \\ & f \mapsto \int_D \kappa_n(\cdot, y)\big(S_{n-1}\sigma_1(\dots S_2\sigma_1(S_1(S_0 f)) \dots)\big)(y) \mathrm{d}y : \\ & S_0 \in \mathsf{IO}(\sigma_2, D; \mathbb{R}^{d_a}, \mathbb{R}^{d_1}), \dots S_{n-1} \in \mathsf{IO}(\sigma_2, D; \mathbb{R}^{d_{n-1}}, \mathbb{R}^{d_n}), \\ & \kappa_n \in \mathsf{N}_l(\sigma_3; \mathbb{R}^{d'} \times \mathbb{R}^d, \mathbb{R}^{d_u \times d_n}),\ d_1, \dots, d_n, l \in \mathbb{N}\}. \end{aligned}$$

With this set of definitions, is possible to state the two main results of the approximation theory for neural operators: *discretisation invariance* and *universal approximation theorem.*

5.3.2.1 Discretisation Invariance

First, let us state the following definition:

Definition 5.3.7 (Discretisation invariance)**.** Let $\Theta \subseteq \mathbb{R}^p$ be a finite dimensional parameter space and $\mathcal{G} : \mathcal{A} \times \Theta \to \mathcal{U}$ a map representing a parametric class of operators with parameters $\theta \in \Theta$. Given a discrete refinement $(D_n)_{n=1}^{\infty}$ of the domain $D \subset \mathbb{R}^d$, we say $\mathcal{G}$ is *discretization-invariant* if there exists a sequence

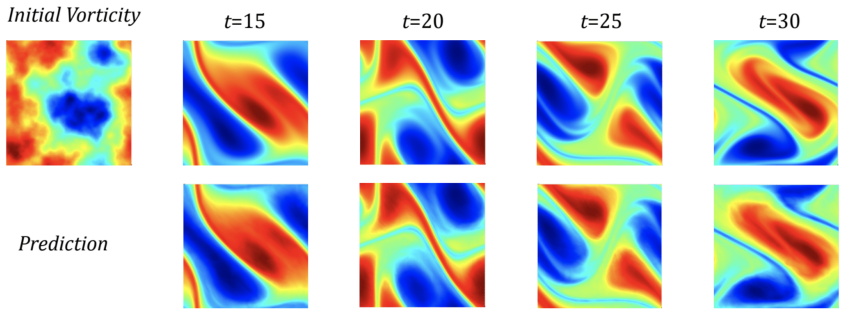


Figure 5.11: Vorticity field of the solution to the two-dimensional Navier-Stokes equation with viscosity $\nu = 10^4$. (Re$\approx$ 200); Ground truth on top and prediction on bottom. The model is trained on data that is discretized on a uniform 64×64 spatial grid and on a 20-point uniform temporal grid. The model is evaluated with a different initial condition that is discretized on a uniform 256×256 spatial grid and a 80-point uniform temporal grid. From [Kov+23].

fig:5:3:2:1: SuperRes

of maps $\hat{\mathcal{G}}_1, \hat{\mathcal{G}}_2, \ldots$ where $\hat{\mathcal{G}}_L : \mathbb{R}^{Ld} \times \mathbb{R}^{Lm} \times \Theta \to \mathcal{U}$ such that, for any $\theta \in \Theta$ and any compact set $K \subset \mathcal{A}$,

$$\lim_{L\to\infty} R_K \left(\mathcal{G}(\cdot, \theta), \hat{\mathcal{G}}_L(\cdot, \cdot, \theta), D_L \right) = 0.$$

From here, it is possible to state the theorem

Theorem 5.3.8 (Discretisation invariance property of Neural Operators)**.** *Let $D \subset \mathbb{R}^d$ and $D' \subset \mathbb{R}^{d'}$ be two domains for some $d, d' \in \mathbb{N}$. Let $\mathcal{A}$ and $\mathcal{U}$ be real-valued Banach function spaces on D and D' respectively. Suppose that $\mathcal{A}$ and $\mathcal{U}$ can be continuously embedded in $C(\bar{D})$ and $C(\bar{D}')$ respectively and that $\sigma_1, \sigma_2, \sigma_3 \in C(\mathbb{R})$. Then, for any $n \in \mathbb{N}$, the set of neural operators $\mathsf{NO}_n(\sigma_1, \sigma_2, \sigma_3; D, D')$ whose elements are viewed as maps $\mathcal{A} \to \mathcal{U}$ is discretization-invariant.*

The proof builds a sequence of finite dimensional maps which approximate the neural operator by Riemann sums and shows uniform converges of the error over compact sets of $\mathcal{A}$. See Appendix E of [Kov+23].

Why do we care about Discretisation invariance? Zero-shot super-resolution

The crucial results from here is that, any Neural Operator, being discretisation invariant, can be trained on a set of numerical results computed on lower-resolution grid, and evaluated at higher resolution, without seeing any higher resolution data! This is the *zero-shot super-resolution*. An example is presented in Figure 5.11, where the Neural Operator is trained on 64x64 data, and evaluated on 256x256. On the top row are presented the real data, and on the botto row the prediction, ad different time steps. The single image on the left is the initial vorticity. The system resolved is the two-dimensional Navier-Stokes

equation for a viscous, incompressible fluid:

$$\begin{aligned} \partial_t u(x,t) + u(x,t)\cdot\nabla u(x,t) + \nabla p(x,t) &= \nu\Delta u(x,t) + f(x), && x\in\mathbb{T}^2, t\in(0,\infty) \\ \nabla\cdot u(x,t) &= 0, && x\in\mathbb{T}^2, t\in[0,\infty) \\ u(x,0) &= u_0(x), && x\in\mathbb{T}^2 \end{aligned} \tag{5.46}$$

where $\mathbb{T}^2$ is the unit torus i.e. $[0,1]^2$ equipped with periodic boundary conditions, and $\nu\in\mathbb{R}_+$ is a fixed viscosity. Here $u:\mathbb{T}^2\times\mathbb{R}_+\to\mathbb{R}^2$ is the velocity field, $p:\mathbb{T}^2\times\mathbb{R}_+\to\mathbb{R}^2$ is the pressure field, and $f:\mathbb{T}^2\to\mathbb{R}$ is a fixed forcing function.

5.3.2.2 Universal Approximation Theorem

Let $\mathcal{A}$ and $\mathcal{U}$ be Banach function spaces on the domains $D\subset\mathbb{R}^d$ and $D'\subset\mathbb{R}^{d'}$, respectively. We are interested in the approximation of non-linear operators $\mathcal{G}^\dagger:\mathcal{A}\to\mathcal{U}$ by neural operators. We will make the following assumptions on the spaces $\mathcal{A}$ and $\mathcal{U}$:

assump:5:3:2:2: input

Hypotesis 5.3.9. *Let $D\subset\mathbb{R}^d$ be a Lipschitz domain for some $d\in\mathbb{N}$. One of the following holds*

1. $\mathcal{A}=L^{p_1}(D)$ *for some* $1\le p_1<\infty$.
2. $\mathcal{A}=W^{m_1,p_1}(D)$ *for some* $1\le p_1<\infty$ *and* $m_1\in\mathbb{N}$,
3. $\mathcal{A}=C(\bar{D})$.

assump:5:3:2:2: output

Hypotesis 5.3.10. *Let $D'\subset\mathbb{R}^{d'}$ be a Lipschitz domain for some $d'\in\mathbb{N}$. One of the following holds*

1. $\mathcal{U}=L^{p_2}(D')$ *for some* $1\le p_2<\infty$, *and* $m_2=0$,
2. $\mathcal{U}=W^{m_2,p_2}(D')$ *for some* $1\le p_2<\infty$ *and* $m_2\in\mathbb{N}$,
3. $\mathcal{U}=C^{m_2}(\bar{D}')$ *and* $m_2\in\mathbb{N}_0$.

From here, we have

Theorem 5.3.11 (Universal Approximation Theorem for Neural Operators)**.** *Let Hypotesis 5.3.9 and Hypotesis 5.3.10 hold and suppose $\mathcal{G}^\dagger:\mathcal{A}\to\mathcal{U}$ is continuous. Let $\sigma_1\in\mathsf{A}_0^{\mathrm{L}}$, $\sigma_2\in\mathsf{A}_0$, and $\sigma_3\in\mathsf{A}_{m_2}$. Then for any compact set $K\subset\mathcal{A}$ and $0<\epsilon\le 1$, there exists a number $N\in\mathbb{N}$ and a neural operator $\mathcal{G}\in\mathsf{NO}_N(\sigma_1,\sigma_2,\sigma_3;D,D')$ such that*

$$\sup_{a\in K}\|\mathcal{G}^\dagger(a)-\mathcal{G}(a)\|_{\mathcal{U}}\le\epsilon.$$

Furthermore, if $\mathcal{U}$ is a Hilbert space and $\sigma_1\in\mathsf{BA}$ and, for some $M>0$, we have that $\|\mathcal{G}^\dagger(a)\|_{\mathcal{U}}\le M$ for all $a\in\mathcal{A}$ then $\mathcal{G}$ can be chosen so that

$$\|\mathcal{G}(a)\|_{\mathcal{U}}\le 4M,\qquad\forall a\in\mathcal{A}.$$

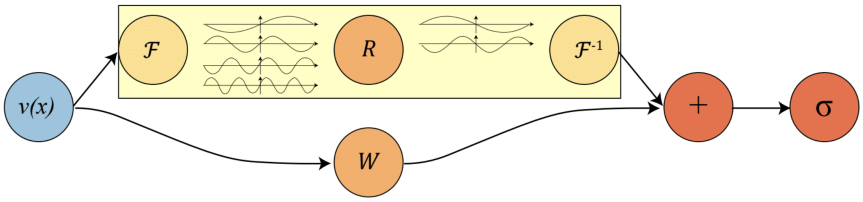


Figure 5.12: **(a) The full architecture of neural operator**: start from input a. 1. Lift to a higher dimension channel space by a neural network $\mathcal{P}$. 2. Apply four layers of integral operators and activation functions. 3. Project back to the target dimension by a neural network $\mathcal{Q}$. Output u. **(b) Fourier layers**: Start from input v. On top: apply the Fourier transform $\mathcal{G}$; a linear transform R on the lower Fourier modes and filters out the higher modes; then apply the inverse Fourier transform $\mathcal{G}^{-1}$. On the bottom: apply a local linear transform W. From [Li+20].

fig:5:4:1: fourier_layer_ fno

5.4 Learning Operators, Part III: Neural Operators - Architectures

5.4.1 Fourier Neural Operator

subsec:5:4:1:FNO

The first example of a Neural Operator architecture is the Fourier Neural Operator (FNO) [Li+20].

The main ingredient of the Fourier Neural Operator is the Fourier representation of the kernel integral operator. Indeed, we have seen in the previous subsection that the Neural Operator layer is written as

$$v_{t+1}(x) = \sigma\left(W_t v_t\left(\Pi_t(x)\right) + \int_{D_t} \kappa^{(t)}(x,y)\, v_t(y)\, \mathrm{d}\nu_t(y) + b_t(x)\right),$$

so, now, let us assume that the integral kernel is translation equivariant and can thus be written as

$$\kappa^{(t)}(x,y) = \kappa^{(t)}(x-y).$$

We can now *Fourier Transform it*, and parametrise its Fourier components with θ_t. The main idea of FNO is to exploit this fact by parameterising κ directly in Fourier space and using the Fast Fourier Transform (FFT) to efficiently compute the integral. Indeed, let FFT denote the Fourier transform of a function $f : D \to \mathbb{R}^{d_v}$ and FFT^{-1} its inverse; then

$$(\mathrm{FFT} f)_j(k) = \int_D f_j(x) \mathrm{e}^{-2i\pi\langle x,k\rangle} \mathrm{d}x, \qquad (\mathrm{FFT}^{-1} f)_j(x) = \int_D f_j(k) \mathrm{e}^{2i\pi\langle x,k\rangle} \mathrm{d}k,$$

for $j = 1, \ldots, d_v$ where $i = \sqrt{-1}$ is the imaginary unit. By assuming $\kappa^{(t)}(x,y) = \kappa^{(t)}(x-y)$, and by using the Convolution Theorem, we find that

$$\big(\mathcal{K}(a)v_t\big)(x) = \mathrm{FFT}^{-1}\big(\mathrm{FFT}(\kappa) \cdot \mathrm{FFT}(v_t)\big)(x), \qquad \forall x \in D.$$

Now, the intuition is to directly parametrise κ in Fourier space;

Definition 5.4.1 (Fourier integral operator $\mathcal{K}$)**.** Define the Fourier integral operator

$$\big(\mathcal{K}(\theta)v_t\big)(x) = \mathrm{FFT}^{-1}\Big(R_\theta \cdot (\mathrm{FFT}[v_t])\Big)(x) \quad \forall x \in D \tag{5.47}$$

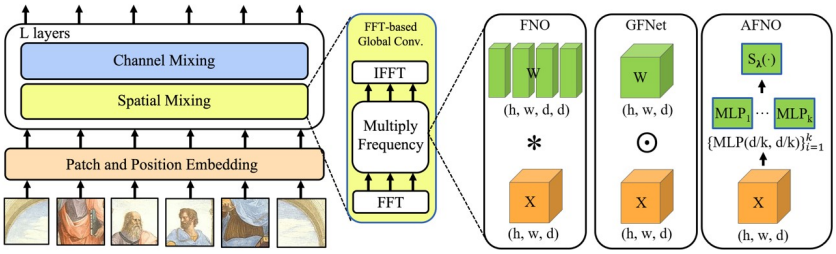


Figure 5.13: The multi-layer transformer network with FNO, GFN, and AFNO mixers. GFNet performs element-wise matrix multiplication with separate weights across channels (k). FNO performs full matrix multiplication that mixes all the channels. AFNO performs block-wise channel mixing using MLP along with soft-thresholding. The symbols h, w, d, and k refer to the height, width, channel size, and block count, respectively. From [Gui+21a].

fig:5:4:2:AFNO

where R_θ is the Fourier transform of a periodic function $\kappa : \bar{D} \to \mathbb{R}^{d_v \times d_v}$, parametrised by $\theta \in \Theta_{\mathcal{K}}$. An illustration is given in Figure Figure 5.12 (b).

Invariance to discretization: The Fourier layers are discretization-invariant because they can learn from and evaluate functions which are discretized in an arbitrary way. Since parameters are learned directly in Fourier space, resolving the functions in physical space simply amounts to projecting on the basis $e^{2\pi i \langle x,k \rangle}$ which are well-defined everywhere on $\mathbb{R}^d$. This allows us to achieve zero-shot super-resolution, as explained before.

Quasi-linear complexity: The weight tensor R contains $k_{\max} < n$ modes, so the inner multiplication has complexity $O(k_{\max})$. Therefore, the majority of the computational cost lies in computing the Fourier transform $\text{FFT}(v_t)$ and its inverse.

General Fourier transforms have complexity $O(n^2)$; however, since the series is truncated, the complexity is in fact $O(nk_{\max})$, while the FFT has complexity $O(n \log n)$.

5.4.2 Adaptive Fourier Neural Operator

subsec:5:4:2:AFNO

One crucial aspect of Self-Attention Mechanism [Vas+17], also in vision tasks [Dos20], is that it imposes *graph structures*, and uses the similarity among the tokens to capture the long-range dependencies.

This makes self-attention parameter efficient and adaptive, but it suffers from a quadratic complexity in the sequence size. To achieve efficient mixing with linear complexity, several approximations have been introduced for self-attention. More recently, alternatives have been introduced for self-attention that relax the graph assumption for efficient mixing. Instead, they leverage the *geometric structures* using Fourier transform.

The idea of [Gui+21a; Gui+21b] was to blend intuition from Fourier Neural Operators, by framing token mixing as operator learning that learns mappings between continuous functions in infinite dimensional spaces. Tokens are treated as continuous elements in the function space, and token mixing is modelled

Step	FNO	AFNO
(i) Token mixing	$X_{m,n:d} \mapsto z_{m,n} = \text{DFT}[X_{m,n:d}]$,	$X_{m,n:d} \mapsto z_{m,n} = \text{DFT}[X_{m,n:d}]$,
(ii) Block mixing	$z_{m,n} \mapsto \tilde{z}_{m,n} = W_{m,n\|m',n'} z_{m',n'}$,	$z_{m,n} \mapsto \tilde{z}_{m,n} = \sum_{\ell=1,\ldots,k} S_\lambda \left(W^{(\ell)}_{m,n\|m',n'} z^{(\ell)}_{m',n'} \right)$,
(iii) Token unmixing	$X_{m,n:d} \mapsto y_{m,n} = \text{IDFT}[\tilde{z}_{m,n}]$,	$X_{m,n:d} \mapsto y_{m,n} = \text{IDFT}[\tilde{z}_{m,n}]$,

Table 5.3: Comparison of the operational steps between FNO and AFNO. The notation d represents the channel dimension, k the number of blocks, and $S_\lambda(\cdot)$ the soft-thresholding operator. In AFNO, the block mixing step incorporates a summation over blocks and the nonlinear shrinkage function.

tab:5:4:2: AFNOvsFNOsteps

as continuous global convolution, which captures global relationships in the geometric space. But, to ease the quadratic complexity in the channel size for channel mixing of standard FNOs, authors impose a block-diagonal structure on the channel mixing weights. Furthermore, to enhance generalisation, inspired by sparse regression, they sparsify the frequencies via soft-thresholding. Finally, for parameter efficiency, the MLP layer shares weights across tokens. These are the ingredients of Adaptive Fourier Neural Operator (see Figure 5.13).

The first step in the architecture involves dividing the input image into a regular grid with $h \times w$ equal sized patches of size $p \times p$. The parameter p is referred to as the *patch size*, as in Vision Transformers [Dos20]. For simplicity, we consider a single channel image. Each patch is embedded into a token of size d , the embedding dimension. The patch embedding operation results in a token tensor $X \sim (h, v, d)$. .

The patch size and embedding dimension are user selected parameters. A smaller patch size allows the model to capture fine scale details better while increasing the computational cost of training the model; on the other hand, a higher embedding dimension also increases the parameter count of the model.

The token tensor is then processed by multiple layers of the transformer architecture performing spatial and channel mixing. The AFNO architecture implements the following operations in each layer.

A crucial element in the AFNO's block mixing step is the application of soft-thresholding and shrinkage. Images are inherently sparse in the Fourier domain, with most of the energy concentrated around low frequency modes. Thus, one can adaptively mask the tokens according to their importance towards the end task. This can use the expressivity towards representing the important tokens. To sparsify the tokens, instead of linear combination, we use the nonlinear LASSO channel mixing as follows:

$$\min \|\tilde{z}_{m,n} - W_{m,n} z_{m,n}\|^2 + \lambda \|\tilde{z}_{m,n}\|_1$$

This can be solved via the soft-thresholding and shrinkage operation

$$\tilde{z}_{m,n} = S_\lambda(W_{m,n} z_{m,n}),$$

that is defined as $S_\lambda(x) = \text{sign}(x) \max\{|x| - \lambda, 0\}$, where λ is a tuning parameter that controls the sparsity. It is also worth noting that the promoted sparsity can also regularize the network and improve the robustness.

lst:5:4:2:afno

Listing 5.3: Pseudocode for AFNO with adaptive weight sharing and adaptive masking. Adapted from [Gui+21a].

```
def AFNO(x):
    bias = x
    x = RFFT2(x)                          # (b, h, w, d) -> (b, h, w//2+1, d)
    x = x.reshape(b, h, w//2+1, k, d//k)  # Split into k blocks
    x = BlockMLP(x)                       # Shared MLP per block
    x = x.reshape(b, h, w//2+1, d)        # Merge blocks
    x = SoftShrink(x)                     # Soft-thresholding
    x = IRFFT2(x)                         # (b, h, w//2+1, d) -> (b, h, w, d)
    return x + bias                       # Residual connection

# Tensor shapes
# x : Tensor[b, h, w, d]                    # Input: batch, height, width, channels
# W_1, W_2 : ComplexTensor[k, d//k, d//k]   # Weights per block
# b_1, b_2 : ComplexTensor[k, d//k]         # Biases per block

def BlockMLP(x):
    """Two-layer MLP with ReLU activation, shared across all tokens"""
    x = MatMul(x, W_1) + b_1                # Linear projection
    x = ReLU(x)                             # Non-linearity
    return MatMul(x, W_2) + b_2             # Second projection
```

lst:5:4:2:afnofull

Listing 5.4: Full pytorch code for AFNO layer. Adapted from [Gui+21a].

```
import math
import torch
import torch.fft
import torch.nn as nn
import torch.nn.functional as F

class AFNO2D(nn.Module):
    """
    hidden_size: channel dimension size
    num_blocks: how many blocks to use in the block diagonal weight matrices (higher => less complexity but less parameters)
    sparsity_threshold: lambda for softshrink
    hard_thresholding_fraction: how many frequencies you want to completely mask out (lower => hard_thresholding_fraction^2 less FLOPs)
    """
    def __init__(
        self,
        hidden_size: int,
        num_blocks : int = 8,
        sparsity_threshold: float = 0.01,
        hard_thresholding_fraction: float = 1,
        hidden_size_factor: float = 1
    ):
        super().__init__()
```

```
        assert hidden_size % num_blocks == 0, f"hidden_size {hidden_size} should be divisble by num_blocks {num_blocks}"

        self.hidden_size = hidden_size
        self.sparsity_threshold = sparsity_threshold
        self.num_blocks = num_blocks
        self.block_size = self.hidden_size // self.num_blocks
        self.hard_thresholding_fraction = hard_thresholding_fraction
        self.hidden_size_factor = hidden_size_factor
        self.scale = 0.02

        self.w1 = nn.Parameter(self.scale * torch.randn(2, self.num_blocks, self.block_size, self.block_size * self.hidden_size_factor))
        self.b1 = nn.Parameter(self.scale * torch.randn(2, self.num_blocks, self.block_size * self.hidden_size_factor))
        self.w2 = nn.Parameter(self.scale * torch.randn(2, self.num_blocks, self.block_size * self.hidden_size_factor, self.block_size))
        self.b2 = nn.Parameter(self.scale * torch.randn(2, self.num_blocks, self.block_size))

    def forward(self, x):
        bias = x

        dtype = x.dtype
        x = x.float()
        B, C, H, W = x.shape
        N = H*W
        x = x.moveaxis(1, -1) # (B, C, H, W) -> (B, H, W, C) # x = x.reshape(B, H, W, C)
        x = torch.fft.rfft2(x, dim=(1, 2), norm="ortho")
        x = x.reshape(B, x.shape[1], x.shape[2], self.num_blocks, self.block_size)

        o1_real = torch.zeros([B, x.shape[1], x.shape[2], self.num_blocks, self.block_size * self.hidden_size_factor], device=x.device)
        o1_imag = torch.zeros([B, x.shape[1], x.shape[2], self.num_blocks, self.block_size * self.hidden_size_factor], device=x.device)
        o2_real = torch.zeros(x.shape, device=x.device)
        o2_imag = torch.zeros(x.shape, device=x.device)

        total_modes = N // 2 + 1
        kept_modes = int(total_modes * self.hard_thresholding_fraction)

        o1_real[:, :, :kept_modes] = F.relu(
            torch.einsum('...bi,bio->...bo', x[:, :, :kept_modes].real, self.w1[0]) - \
            torch.einsum('...bi,bio->...bo', x[:, :, :kept_modes].imag, self.w1[1]) + \
```

```
            self.b1[0]
        )

        o1_imag[:, :, :kept_modes] = F.relu(
            torch.einsum('...bi,bio->...bo', x[:, :, :kept_modes].imag
                , self.w1[0]) + \
            torch.einsum('...bi,bio->...bo', x[:, :, :kept_modes].real
                , self.w1[1]) + \
            self.b1[1]
        )

        o2_real[:, :, :kept_modes] = (
            torch.einsum('...bi,bio->...bo', o1_real[:, :, :kept_modes
                ], self.w2[0]) - \
            torch.einsum('...bi,bio->...bo', o1_imag[:, :, :kept_modes
                ], self.w2[1]) + \
            self.b2[0]
        )

        o2_imag[:, :, :kept_modes] = (
            torch.einsum('...bi,bio->...bo', o1_imag[:, :, :kept_modes
                ], self.w2[0]) + \
            torch.einsum('...bi,bio->...bo', o1_real[:, :, :kept_modes
                ], self.w2[1]) + \
            self.b2[1]
        )

        x = torch.stack([o2_real, o2_imag], dim=-1)
        x = F.softshrink(x, lambd=self.sparsity_threshold)
        x = torch.view_as_complex(x)
        x = x.reshape(B, x.shape[1], x.shape[2], C)
        x = torch.fft.irfft2(x, s=(H, W), dim=(1, 2), norm="ortho")
        x = x.moveaxis(-1, 1) # (B, H, W, C) -> (B, C, H, W)
        x = x.type(dtype)
        return x + bias
```

subsec:5:4:3: PINO

5.4.3 Physics Informed Neural Operator

Up to now we have seen Neural Operators for surrogate modelling. The *Physics Informed Neural Operators* (PINO) [Li+24] approach for surrogate modelling of PDE systems effectively combines the data-informed supervised learning framework of the Fourier Neural Operator with the physics-informed learning framework. PINO incorporates a PDE loss to the Fourier Neural Operator training. This reduces the amount of data required to train a surrogate model, since the PDE loss constrains the solution space. The PDE loss also enforces physical constraints on the solution computed by a surrogate ML model, making it an attractive option as a verifiable, accurate and interpretable ML surrogate modelling tool.

Let us set up the notation, to align with [Li+24]. Suppose we have to face

the pure-boundary differential problem like

$$\begin{aligned} \mathcal{P}(u,a) &= 0, \qquad \text{in } D \subset \mathbb{R}^d \\ u &= g, \qquad \text{in } \partial D \end{aligned} \tag{5.48}$$

where D is a bounded domain, $a \in \mathcal{A} \subseteq \mathcal{V}$ is a PDE coefficient/parameter, $u \in \mathcal{U}$ is the unknown, and $\mathcal{P} : \mathcal{U} \times \mathcal{A} \to \mathcal{F}$ is a possibly non-linear partial differential operator with $(\mathcal{U}, \mathcal{V}, \mathcal{F})$ a triplet of Banach spaces. Usually, the function g is a fixed boundary condition but can also potentially enter as a parameter. This formulation gives rise to the solution operator $\mathcal{G}^\dagger : \mathcal{A} \to \mathcal{U}$ defined by $a \mapsto u$. A prototypical example is the second-order elliptic equation $\mathcal{P}(u,a) = -\nabla \cdot (a\nabla u) + f$. Another class of problem we may want to tackle is the dynamical problem

$$\begin{aligned} \frac{du}{dt} &= \mathcal{R}(u), \qquad \text{in } D \times (0,\infty) \\ u &= g, \qquad \text{in } \partial D \times (0,\infty) \\ u &= a \qquad \text{in } \bar{D} \times \{0\} \end{aligned} \tag{5.49}$$

where $a = u(0) \in \mathcal{A} \subseteq \mathcal{V}$ is the initial condition, $u(t) \in \mathcal{U}$ for $t > 0$ is the unknown, and $\mathcal{R}$ is a possibly non-linear partial differential operator with $\mathcal{U}$, and $\mathcal{V}$ Banach spaces. As before, we take g to be a known boundary condition. We assume that u exists and is bounded for all time and for every $u_0 \in \mathcal{U}$.

This formulation gives rise to the solution operator $\mathcal{G}^\dagger : \mathcal{A} \to C\big((0,T];\mathcal{U}\big)$, mapping $a \mapsto u$.

We have seen that the standard surrogate modelling with neural operators is as follows: given a differential system, and the corresponding solution operator $\mathcal{G}^\dagger$, one can use a neural operator $\mathcal{G}_\theta$ with parameters θ as a surrogate model to approximate $\mathcal{G}^\dagger$. Usually we assume a dataset $\{a_j, u_j\}_{j=1}^N$ is available, where $\mathcal{G}^\dagger(a_j) = u_j$ and $a_j \sim \mu$ are i.i.d. samples from some distribution μ supported on $\mathcal{A}$. In this case, one can optimize the solution operator by minimizing the empirical data loss on a given data pair

$$\mathcal{L}_{\text{data}}(u, \mathcal{G}_\theta(a)) = \|u - \mathcal{G}_\theta(a)\|_{\mathcal{U}}^2 = \int_D |u(x) - \mathcal{G}_\theta(a)(x)|^2 \mathrm{d}x \tag{5.50}$$

The operator data loss is defined as the average error across all possible inputs

$$\begin{aligned} \mathcal{J}_{\text{data}}(\mathcal{G}_\theta) &= \|\mathcal{G}^\dagger - \mathcal{G}_\theta\|_{L^2_\mu(\mathcal{A};\mathcal{U})}^2 \\ &= \mathbb{E}_{a\sim\mu}[\mathcal{L}_{\text{data}}(a,\theta)] \approx \frac{1}{N}\sum_{j=1}^N \int_D |u_j(x) - \mathcal{G}_\theta(a_j)(x)|^2 \mathrm{d}x. \end{aligned} \tag{5.51}$$

We can now define physics-informed loss, e.g.

$$\begin{aligned} \mathcal{L}_{\text{pde}}(a, u_\theta) &= \Big\|\mathcal{P}(a,u_\theta)\Big\|_{L^2(D)}^2 + \alpha\Big\|u_\theta|_{\partial D} - g\Big\|_{L^2(\partial D)}^2 \\ &= \int_D |\mathcal{P}(u_\theta(x), a(x))|^2 \mathrm{d}x + \alpha \int_{\partial D} |u_\theta(x) - g(x)|^2 \mathrm{d}x \end{aligned} \tag{5.52}$$

for the stationary case, or

$$\begin{aligned}\mathcal{L}_{\text{pde}}(a, u_\theta) &= \left\| \frac{du_\theta}{dt} - \mathcal{R}(u_\theta) \right\|^2_{L^2(T;D)} + \alpha \left\| u_\theta|_{\partial D} - g \right\|^2_{L^2(T;\partial D)} + \beta \left\| u_\theta|_{t=0} - a \right\|^2_{L^2(D)} \\ &= \int_0^T \int_D |\frac{du_\theta}{dt}(t,x) - \mathcal{R}(u_\theta)(t,x)|^2 \mathrm{d}x\mathrm{d}t \\ &\quad + \alpha \int_0^T \int_{\partial D} |u_\theta(t,x) - g(t,x)|^2 \mathrm{d}x\mathrm{d}t \\ &\quad + \beta \int_D |u_\theta(0,x) - a(x)|^2 \mathrm{d}x \end{aligned} \tag{5.53}$$

for the dynamic case. These, for the operator case, can be compactly written as

$$\mathcal{J}_{\text{pde}}(\mathcal{G}_\theta) = \mathbb{E}_{a\sim\mu}[\mathcal{L}_{\text{pde}}(a, \mathcal{G}_\theta(a))]. \tag{5.54}$$

The point is that, in general, it is non-trivial to compute the derivatives $d\mathcal{G}_\theta(a)/dx$ and $d\mathcal{G}_\theta(a)/dt$ for model $\mathcal{G}_\theta$. In the following, we discuss the options the authors [Li+24] suggested how to compute these derivatives for Fourier neural operator.

In the FNO framework, the surrogate ML model is given by a the solution operator $\mathcal{G}^\dagger_\theta$, which maps any given coefficient in the coefficient space a to the solution u. The FNO is trained in a supervised fashion using training data in the form of input/output pairs $\{a_j, u_j\}_{j=1}^N$. The training loss for the FNO is given by summing the data loss, $\mathcal{L}_{\text{data}}(\mathcal{G}_\theta) = \|u - \mathcal{G}_\theta(a)\|^2$ over all training pairs $\{a_i, u_i\}_{i=1}^N$.

Indeed, in the PINO framework, the solution operator is optimized also with PDE loss given by $\mathcal{L}_{\text{pde}}(a, \mathcal{G}_\theta(a))$ computed over i.i.d. samples a_j from an appropriate supported distribution in parameter/coefficient space.

In general, the PDE loss involves computing the PDE operator which in turn involves computing the partial derivatives of the Fourier Neural Operator ansatz. In general this is non-trivial. The key set of innovations in the PINO are the various ways to compute the partial derivatives of the operator ansatz. The PINO framework implements the differentiation in four different ways.

1. Numerical differentiation using a finite difference Method (FDM) or spectral derivatives.
2. Pointwise Differentiable programming approach (exact derivatives).
3. Differentiable programming approach in Fourier space.

Let us discuss and compare these three methods;

Numerical differentiation: A simple but efficient approach is to use conventional numerical derivatives such as finite difference and Fourier differentiation. These numerical differentiation methods are fast and memory-efficient: given a n-points grid, finite difference requires $O(n)$, and the Fourier method requires $O(n \log n)$.

However, the numerical differentiation methods face the same challenges as the corresponding numerical solvers: finite difference methods require a fine-resolution uniform grid; spectral methods require smoothness and uniform grids. Furthermore, these numerical errors on the derivatives will be amplified on the output solution.

Pointwise differentiation: in analogy with to PINN, the most general way to compute the exact derivatives is to use the auto-differentiation library of neural networks (autograd). However, it is not straightforward to write out the solution function in the neural operator which directly outputs the numerical solution $u = \mathcal{G}_\theta(a)$ on a grid, especially for FNO which uses FFT. To apply autograd, we design a query function u that input x and output $u(x)$. Recall $\mathcal{G}_\theta := \mathcal{Q} \circ (W_L + \mathcal{K}_L) \circ \cdots \circ \sigma(W_1 + \mathcal{K}_1) \circ \mathcal{P}$ and $u = \mathcal{G}_\theta a = \mathcal{Q} v_L = \mathcal{Q}(W_L + \mathcal{K}_L) v_{L-1} \ldots$. Since $\mathcal{Q}$ is pointwise,

$$u(x) = \mathcal{Q}(v_L)(x) = \mathcal{Q}(v_L(x)) = \mathcal{Q}\big((W_L v_{L-1})(x) + \mathcal{K}_L v_{L-1}(x)\big) \tag{5.55}$$

{eq:autograd}

For both the kernel operator and Fourier operator, we either remove the pointwise residual term of the last layer $(W_L v_{L-1})(x)$ or define the query function as an interpolation function on W_L.

For kernel integral operator, the kernel function can directly take the query points as input. So we can apply auto-differentiation to compute the derivatives

$$u'(x) = \mathcal{Q}'\big(v_L(x)\big) \cdot \sum_{B(x)} \kappa^{(l)'}(x, y, v_{L-1}(y)) \tag{5.56}$$

Similarly, for the Fourier convolution operator, we need to evaluate the Fourier convolution $\mathcal{K}_L v_{L-1}(x)$ on the query points x. It can be done by writing out the output function as Fourier series composing with Q :

$$\begin{aligned} u(x) &= \mathcal{Q} \circ \mathrm{FFT}^{-1}\Big(R \cdot (\mathrm{FFT}[v_{L-1}])\Big)(x) \\ &= \mathcal{Q}\Big(\frac{1}{k_{max}} \sum_{k=0}^{k_{max}} \big(R_k(\mathrm{FFT}[v_{L-1}])_k\big) \exp \frac{i2\pi k}{D}(x)\Big) \end{aligned}$$

Where FFT is the discrete Fourier transform. The inverse discrete Fourier transform is the sum of k_{max} Fourier series with the coefficients $\big(R_k(\mathrm{FFT}[v_{L-1}])_k\big)$ coming from the previous layer.

$$u'(x) = \mathcal{Q}'\big(v_L(x)\big) \cdot \frac{1}{k_{max}} \sum_{k=0}^{k_{max}} \big(R_k(\mathrm{FFT}[v_{L-1}])_k\big) \exp' \frac{i2\pi k}{D}(x) \tag{5.57}$$

Notice $\exp' \frac{i2\pi k}{D}(x) = \frac{i2\pi k}{D} \exp \frac{i2\pi k}{D}(x)$, just as the numerical Fourier method. If the query points x are a uniform grid, the derivative can be efficiently computed with the Fast Fourier transform.

The autograd method is general and exact, however, it is less efficient. Since the number of parameters

is usually much greater than the grid size n, the numerical methods are indeed significantly faster. Empirically, the autograd method is usually slower and memory-consuming.

Function-wise differentiation: while it is expensive to apply the auto differentiation per query point, the derivative can be batched and computed in a function-wise manner. [Li+24] developed an efficient and exact derivatives method based on the architecture of the neural operator that can compute the full gradient field. The idea is similar to the autograd, but the derivatives on the Fourier space are explicitly computed. Given the explicit form, u' can be directly computed on the Fourier space.

$$u' = \mathcal{Q}'\big(v_L\big) \cdot \mathrm{FFT}^{-1}\left[\frac{i2\pi}{D} K \cdot (\mathrm{FFT}[v_L])\right] \tag{5.58}$$

Therefore, to exactly compute the derivative of the Fourier neural operator, one just needs to run the numerical Fourier differentiation. Especially the derivative and be efficiently computed with the Fast Fourier transform when the query is uniform. Similarly, if the kernel function $\kappa^{(l)}$ has a structured form, it is also possible to write out its gradient field explicitly.

5.5 Learning Operators, Part IV: Neural Operators - Applications

5.5.1 Neural operators for Weather forecasting: FourCastNet

One of the most relevant applications of Neural Operator lies in the field of weather forecasting, where Neural Operator zero-shot superresolution property is quite appealing. A set of relevant papers on this matter are the ones introducing *FourCastNet*, a FNO-based neural operator for weather forecasting [Bon+25; Kur+23; Pat+22].

FourCastNet (Fourier Forecasting Neural Network) is a global, data-driven weather forecasting model that leverages the AFNO architecture to deliver accurate short- to medium-range predictions at a high spatial resolution of 0.25°, corresponding to roughly 30 km × 30 km at the equator. By framing token mixing as a continuous global convolution implemented efficiently in the Fourier domain, FourCastNet achieves a quasi-linear computational complexity of $O(N \log N)$ with respect to the sequence length, enabling it to process the 720×1440 global grid with unprecedented efficiency.

The model accurately forecasts high-resolution, fast-timescale variables, including surface wind speed, precipitation, and atmospheric water vapor, quantities that exhibit complex fine-scale structure and have been challenging for previous data-driven approaches. This capability has important implications for planning wind energy resources at onshore and offshore farms, as well as for predicting extreme weather events such as tropical cyclones, extratropical cyclones, and atmospheric rivers.

In terms of forecast skill, FourCastNet matches the accuracy of the ECMWF Integrated Forecasting System (IFS), a state-of-the-art Numerical Weather Prediction (NWP) model, at short lead times for large-scale variables like geopotential height, while outperforming IFS for variables with complex fine-scale structure, including precipitation and near-surface wind speeds. Perhaps most remarkably, FourCastNet generates a week-long forecast in less than 2 seconds, orders of magnitude faster than traditional NWP models, enabling the rapid and inexpensive creation of large-ensemble forecasts with thousands of

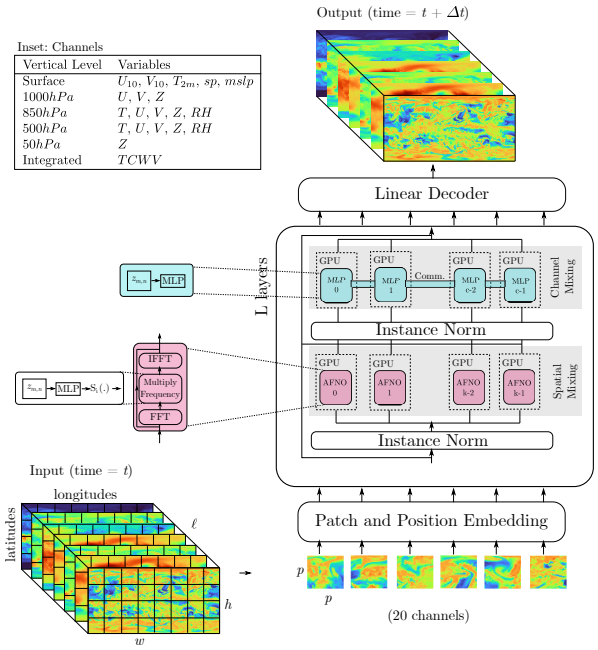


Figure 5.14: The AFNO architecture showing the key operations performed on the the input tensor with dimensions $(20 \times 720 \times 1440)$ to produce a 6 hour single time step forecast with the same dimensions. Model parallelism is implemented by splitting the channels (feature maps) across GPUs. Channel mixing MLP operations require communication across the model parallel ranks, while the FFT based spatial-mixing operates on disjoint blocks that are embarrassingly parallel. From [Kur+23].

fig:5:5:1: afnocastnet

members. This dramatic speedup, quantified as approximately 45.000 times faster on a node-hour basis than IFS, revolutionizes probabilistic forecasting by allowing well-calibrated uncertainty quantification for extreme events that smaller ensembles cannot reliably capture. Furthermore, FourCastNet's training time has been scaled to as little as 67 minutes on 3.072 NVIDIA A100 GPUs, achieving 140.8 petaFLOPS in mixed precision and demonstrating the viability of data-driven deep learning models as a valuable addition to the meteorology toolkit to aid and augment traditional NWP models.

As said, the original implementation of FourCastNet employs an AFNO-based transformer model. As can we see from Figure 5.14, to generate the $t+\Delta t$ prediction it passes the Patch embedded $X \sim (20, 720, 1440)$ tensor to L layers of a transformer-like architecture, where AFNO layers replace the Multi-Head Attention layers to perform the *spatial mixing* part of the Transformer (like in the original AFNO paper, [Gui+21a; Gui+21b]).

As an example of application, we report their result on a global Total Precipitation (TP) forecast using the FourCastNet model, Figure 5.15. The total precipitation (TP) in the ERA5 re-analysis dataset is a variable that represents the the accumulated liquid and frozen water that falls to the Earth's surface through rainfall and snow. It is defined in units of length as the depth of water that would accumulate if spread evenly over a unit grid box of the

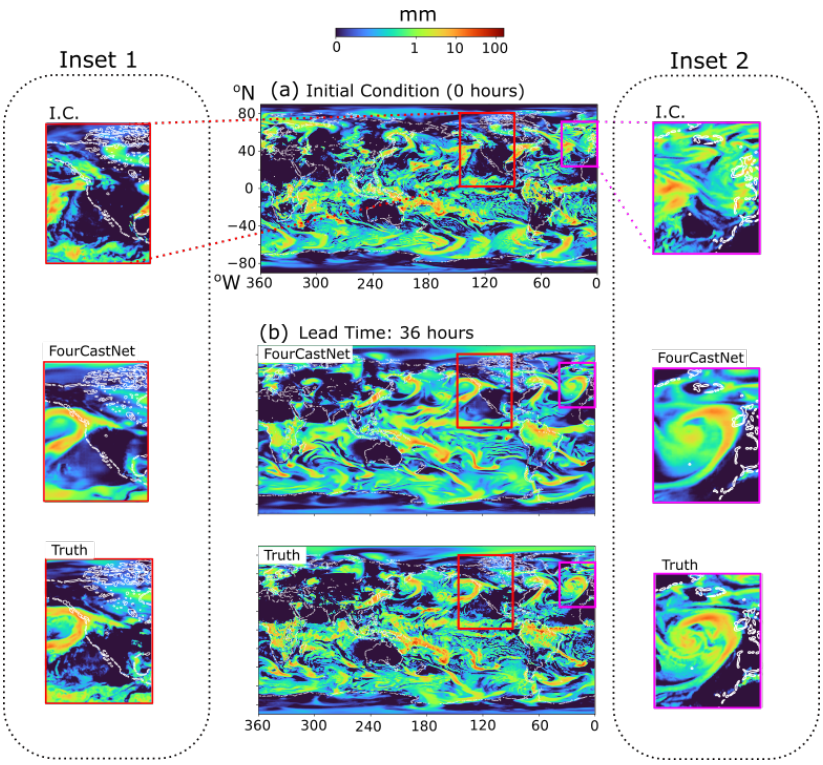


Figure 5.15: Illustration of a global Total Precipitation (TP) forecast using the FourCastNet model. Land-sea borders are shown using a thin white trace. For ease of visualization, the precipitation field is plotted as a log-transformed field in all panels. Panel (a) shows the TP fields at the time of forecast initialization. Panel (b) shows the TP forecast generated by the FourCastNet model (upper panel) over the entire globe at 0.25°-lat-long resolution with the corresponding truth (lower panel). Inset 1 shows the I.C., forecast and true precipitation fields at a lead time of 36 hours over a local region along the western coast of the United States. This highlights the ability of the FourCastNet model to resolve and predict localized regions of high precipitation, in this case due to an atmospheric river. Inset 2 shows the I.C., forecast, and true precipitation fields near the coast of the U.K. and highlights an extreme precipitation event due to an extra-tropical cyclone that is predicted very well by the FourCastNet model. The calendar time-stamp of the initial condition used to generate this forecast was 00:00 UTC on April 4, 2018. The high-resolution FourCastNet model demonstrates excellent skill in capturing small scale features that are key to precipitation forecasting. From [Pat+22].

fig:5:5:1:precip

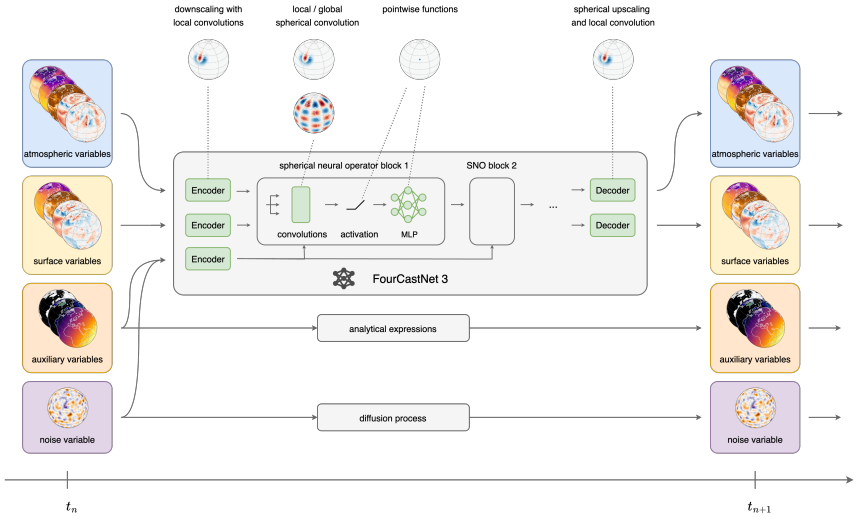


Figure 5.16: Schematic of the FourCastNet 3 model. The model predicts the state of the atmosphere at the next timestep, given the state at the previous timestep. Auxiliary variables such as the cosine zenith angle are computed from analytical expressions for each timestep and appended to the input. A hidden Markov model is obtained by conditioning FourCastNet 3 on a stochastic latent variable whose temporal dynamics are governed by a diffusion process on the sphere. The model itself is formed by an encoder, a decoder and 8 neural operator blocks. Each of these operations can be grouped into local, global and pointwise operations and therefore be formulated on arbitrary grids and resolutions, making FourCastNet 3 discretization independent. Green boxes illustrate learnable operations. From [Bon+25].

fig:5:5:1:1:FCN3

model.

5.5.1.1 FourCastNet3: Spherical Neural Operator Blocks

Building upon the success of the original FourCastNet, the recently introduced FourCastNet 3 (FCN3) advances global weather modeling by implementing a scalable, geometric machine learning approach specifically designed for probabilistic ensemble forecasting [Bon+25]. Unlike its predecessor, which, as we have seen, relied on the Adaptive Fourier Neural Operator (AFNO) within a vision transformer backbone, FCN3 adopts a purely convolutional architecture built around Spherical Neural Operator (SNO) blocks that are tailored to the geometry of the Earth. These blocks leverage two complementary types of spherical convolutions: global convolution filters parameterised in the spectral domain via the spherical harmonic transform (SHT), which resemble classical pseudo-spectral methods such as IFS, and local spherical group convolutions implemented using discrete-continuous (DISCO) convolutions, which enable anisotropic, locally supported filters that are better suited to capturing phenomena like adiabatic flow and orographic effects. The architecture organizes these blocks into an encoder-processor-decoder structure inspired by ConvNeXt [Liu+22], with a carefully calibrated ratio of four local blocks

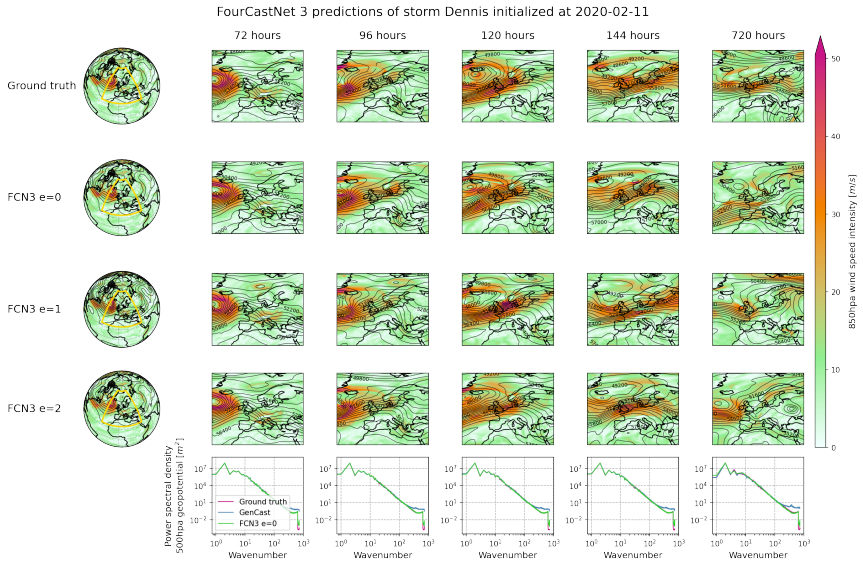


Figure 5.17: FourCastNet 3 prediction of storm Dennis initialized on 2020-02-11 at 00:00:00 UTC. The plot depicts wind-speeds at a pressure level of 850hPa and isohypses (height contours) of the 500hPa geopotential height. FCN3 accurately predicts the storm and its landfall 5 days in advance, with different ensemble members depicting different scenarios. FCN3 predicts global weather phenomena at a spatial resolution of 0.25° and a temporal resolution of 6 hours. From [Bon+25].

fig:enter-label

to one global block per stage, ensuring effective multiscale processing while respecting the spherical geometry and its inherent rotational symmetries. This geometric foundation enables FCN3 to better model the spatially correlated probabilistic nature of atmospheric dynamics, resulting in stable spectra and realistic dynamics across multiple scales.

Another key innovation of FCN3 is its end-to-end probabilistic formulation as a hidden Markov model, trained with a composite loss function that combines the Continuous Ranked Probability Score (CRPS) in both spatial and spectral domains. This approach effectively addresses the common issues of blurring in deterministic models and spurious high-frequency noise in diffusion-based probabilistic models, achieving well-calibrated ensembles with spread-skill ratios approaching unity. In terms of forecast skill, FCN3 surpasses the gold-standard IFS ensemble system and nearly matches the state-of-the-art diffusion-based model GenCast, while being 8 to 60 times faster in inference. A single 15-day global forecast at 0.25° resolution with 6-hourly outputs can be generated in under 60 seconds on a single NVIDIA H100 GPU.

5.5.2 Neural Operators for Reservoir Simulation

5.5.2.1 Fourier Neural Operators for ECHELON software

The application of neural operators has extended beyond weather forecasting into the domain of subsurface energy exploration, where they address critical

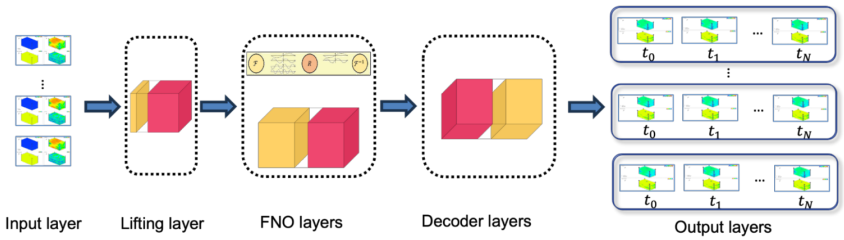


Figure 5.18: Fourier neural operator architecture for generating spatio - temporal reservoir proxies with NVIDIA PhysicsNeMo on AWS. From [Muk+24].

fig:5:5:2: srt_arch

challenges in reservoir simulation. Reservoir simulation involves modelling multiphase flow in porous media to predict oil, gas, and water movement over time. Traditional full-physics numerical simulators, while accurate, are computationally expensive, particularly when performing uncertainty quantification, history matching, or field optimization tasks that require thousands of forward simulations. These tasks often involve complex geological models with hundreds of thousands to millions of active grid cells, making them computationally prohibitive even with high-performance computing resources [Muk+24].

A promising solution to this computational bottleneck lies in the use of full-field proxy models that approximate the solution of primary state variables (such as pressure and saturation) directly in space and time, analogous to a full-physics simulator. Stone Ridge Technology (SRT) has developed a scalable framework that integrates its ECHELON reservoir simulator [Sto25] with NVIDIA PhysicsNeMo [Nvi26] on AWS to generate such full-field proxy models using Fourier neural operators (FNOs) [Muk+24] (see Figure 5.18). The workflow begins with ECHELON generating large volumes of training, validation, and test data, which are stored on Amazon S3 for efficient retrieval. The neural operator model is then trained within the PhysicsNeMo framework to learn the spatio-temporal mapping from reservoir parameters (such as porosity, permeability, and well locations) to the resulting pressure and saturation fields. Once trained, these proxy models can perform inference orders of magnitude faster than forward simulations, achieving speedups of 10x to 100x while maintaining reasonably accurate agreement with the full-physics numerical solutions.

The effectiveness of this approach has been demonstrated on two representative use cases. The first involves well placement optimization in a reservoir with heterogeneous permeability fields. By varying the positions of injectors and producers across 500 training samples, the FNO-based proxy captured the complex topological details of waterfront propagation, showing strong agreement with the ground-truth ECHELON simulations (Figure 5.19). The second case addresses geological uncertainty in a real-field model: the Norne field in the Norwegian Sea, a complex reservoir with faults, heterogeneous anisotropic permeability, and 35 wells. Using 500 realizations with variations in porosity, permeability, and fault transmissibility multipliers, the FNO proxy demonstrated strong agreement with ECHELON for both pressure and water saturation fields across time.

This integration of a high-performance full-physics simulator with neural

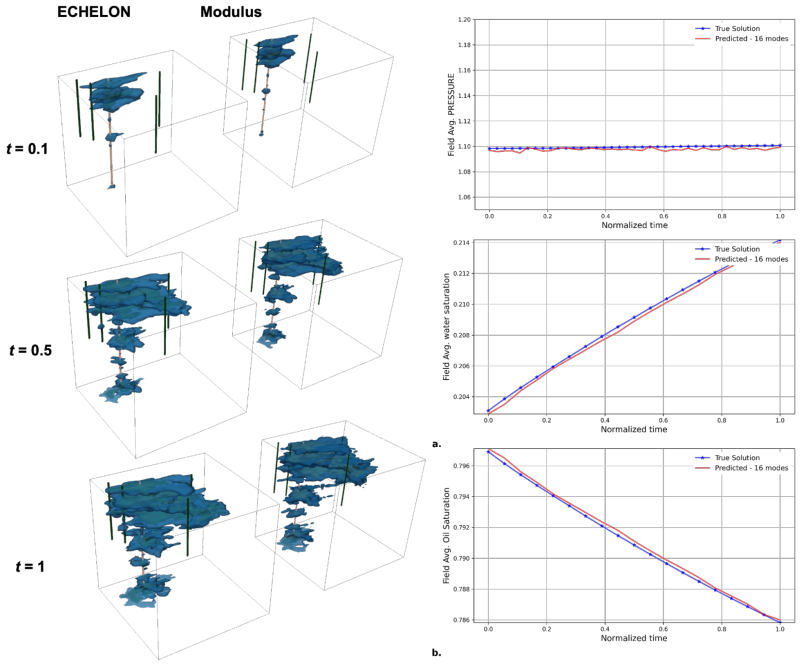


Figure 5.19: Comparison of Fourier neural operator predictions against ground-truth ECHELON simulations for water front propagation in a heterogeneous permeability field under different well placement scenarios. The complex topological details of the waterfront are well captured by the neural operator proxy. Adapted from [Muk+24].

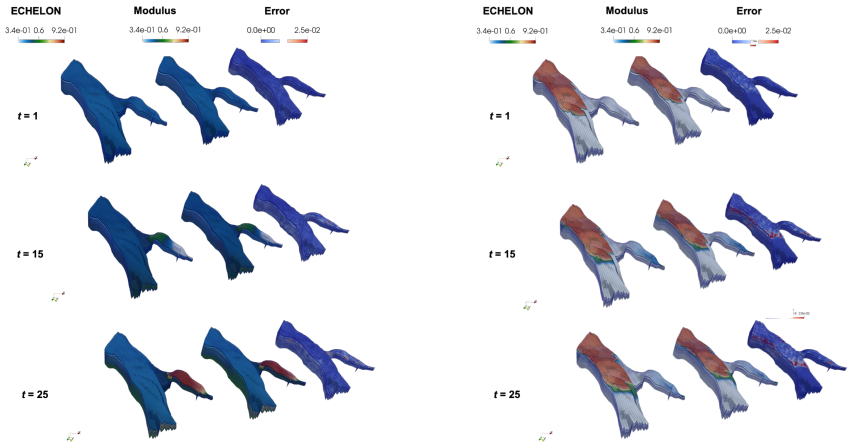
fig:5:5:2: srt_waterfront

operator-based surrogate modeling provides a generalized framework for addressing subsurface challenges, including uncertainty quantification and field development optimization, with enhanced performance and high accuracy. The use of modern GPUs (A100, H100, H200) on AWS enables the computational scale required for training these models, while the resulting proxies open new possibilities for engineering workflows not previously feasible.

5.5.2.2 3D Reservoir Simulation with Physics-Informed Neural Operators

As a complement to the data-driven proxy modeling approach discussed in the previous section, NVIDIA PhysicsNeMo Sym provides a framework for physics-informed machine learning that embeds the governing partial differential equations directly into the training process. A representative example is the 3D reservoir simulation problem, which models two-phase (oil-water) flow in porous media using a physics-informed neural operator (PINO) architecture [EOS23].

Governing Equations: the two-phase black-oil model for reservoir simulation is governed by a system of coupled PDEs describing pressure and saturation evolution. The conservation equations for water and oil phases are given by



(a) Pressure field comparison. (b) Water saturation comparison.

Figure 5.20: Validation of Fourier neural operator (FNO) proxy against full-physics ECHELON simulations for the Norne field. The FNO-based proxy, trained using NVIDIA PhysicsNeMo, demonstrates strong agreement with the ground-truth ECHELON solutions for both (a) pressure fields and (b) water saturation fields across the 3,298-day simulation period. Only small errors are observed, validating the use of neural operators as accurate surrogates for full-physics reservoir simulation. Adapted from [Muk+24].

fig:fno_echelon_validation

[EOS23]:

$$\phi(\mathbf{x})\frac{\partial S_w}{\partial t} - \nabla \cdot [T_w \left(\nabla p_w + \rho_w g \nabla z\right)] = Q_w, \tag{5.59}$$

$$\phi(\mathbf{x})\frac{\partial S_o}{\partial t} - \nabla \cdot [T_o \left(\nabla p_o + \rho_o g \nabla z\right)] = Q_o, \tag{5.60}$$

where $\phi(\mathbf{x})$ is the porosity, S_w and S_o are the water and oil saturations, p_w and p_o are the phase pressures, ρ_w and ρ_o are the phase densities, g is gravitational acceleration, and Q_w, Q_o are source/sink terms representing wells. The phase transmissibilities are defined as:

$$T_w = \frac{K(\mathbf{x})K_{rw}(S_w)}{\mu_w}, \qquad T_o = \frac{K(\mathbf{x})K_{ro}(S_o)}{\mu_o}, \qquad T = T_w + T_o, \tag{5.61}$$

where $K(\mathbf{x})$ is the absolute permeability tensor, $K_{rw}(S_w)$ and $K_{ro}(S_o)$ are the relative permeabilities, and μ_w, μ_o are the phase viscosities. The system is closed by the saturation constraint and capillary pressure relation:

$$S_w + S_o = 1, \qquad p_c(S_w) = p_o - p_w. \tag{5.62}$$

The pressure equation is derived by combining the two phase equations, yielding:

$$-\nabla \cdot [T(\mathbf{x}, S_w)\, \nabla p] = Q, \qquad Q = Q_o + Q_w, \tag{5.63}$$

{eq:pressure}

while the water saturation equation can be expressed as:

$$\phi(\mathbf{x})\frac{\partial S_w}{\partial t} - \nabla \cdot [T_w(\mathbf{x}, S_w)\, \nabla p_w] = Q_w. \tag{5.64}$$

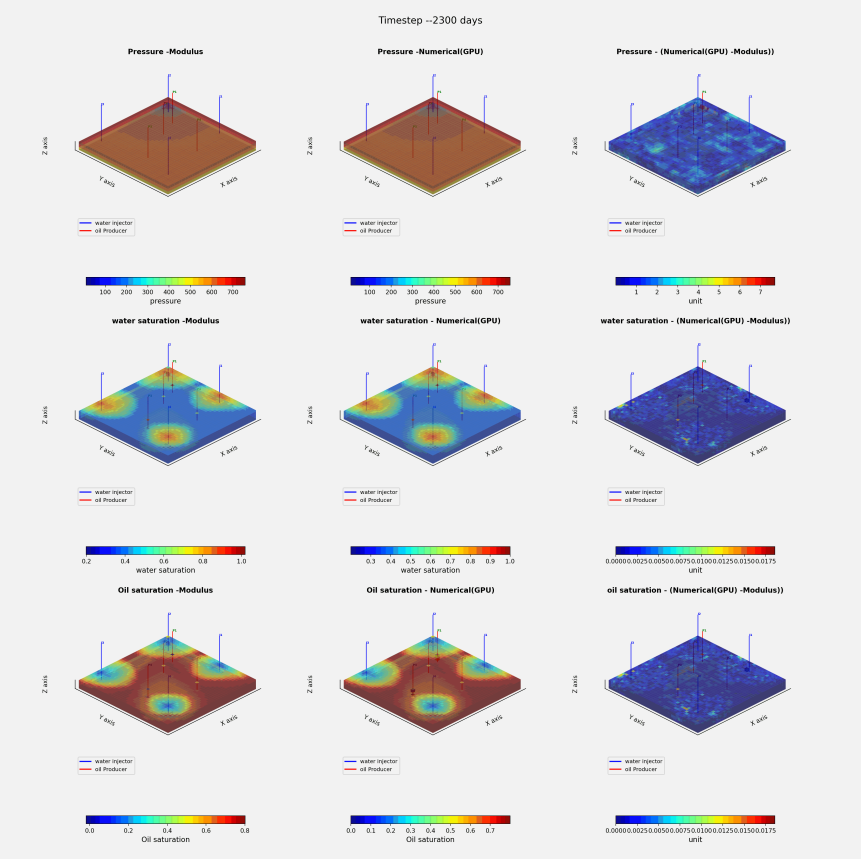


Figure 5.21: Comparison of pressure, water saturation, and oil saturation fields between the PINO surrogate (left column), the finite-volume solver (middle column), and the difference (right column), at the last time step. The results show strong agreement between the physics-informed neural operator and the traditional numerical simulator. Adapted from [EOS23].

fig:5:5:2:2: pino_3d_ reservoir

Physics-Informed Neural Operator Formulation The goal of the surrogate modelling approach is to replace the traditional finite-volume simulator with a neural operator that predicts pressure, saturation, and flux fields given any input permeability and porosity field. Following the approach of [EOS23], an additional vector variable, the flux $\mathbf{F}$, is introduced, which transforms the pressure equation into a first-order system:

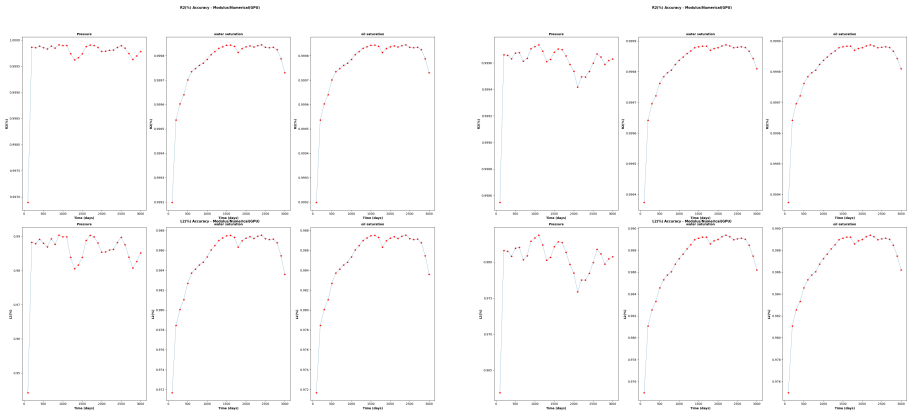
$$\mathbf{u} = \nabla p, \qquad \mathbf{F} = T\nabla p, \qquad -\nabla \cdot \mathbf{F} = Q. \tag{5.65}$$

The pressure equation loss is then formulated as a mixed residual loss:

$$\mathcal{L}_{\text{pressure}} = \frac{1}{n_s}\left(\|\mathbf{F} - T \odot \nabla \mathbf{u}\|_2^2 + \| - \nabla \cdot \mathbf{F} - Q\|_2^2\right), \tag{5.66}$$

while the saturation equation loss is given by:

$$\mathcal{L}_{\text{saturation}} = \frac{1}{n_s}\left\|\phi\frac{\partial S_w}{\partial t} - \nabla \cdot [T_w \nabla \mathbf{u}] - Q_w\right\|_2^2. \tag{5.67}$$



(a) FNO: R^2 and L2 accuracy metrics for pressure and water saturation over time.

(b) PINO: R^2 and L2 accuracy metrics for pressure and water saturation over time.

Figure 5.22: Accuracy metrics for the neural operator surrogates. Both FNO and PINO models achieve high R^2 scores (>0.95) for pressure and saturation predictions throughout the simulation period, demonstrating their capability to accurately surrogate the full-physics reservoir simulator. Adapted from [EOS23].

fig:5:5:2:2: accuracy_metrics

The total physics-informed loss combines both contributions:

$$\mathcal{L}_{\text{PINO}} = \mathcal{L}_{\text{pressure}} + \mathcal{L}_{\text{saturation}}. \tag{5.68}$$

This mixed residual formulation enables the neural operator to learn the underlying physics while maintaining the ability to predict flux variables directly, which are essential for well-based production analysis.

The reservoir model used in this example is a 3D channelized sandstone reservoir with dimensions $40 \times 40 \times 3$ grid cells, containing two lithofacies. The well configuration consists of 4 injectors and 4 producers arranged in an "analogous 5-spot pattern". A total of 2.340 days of simulation are performed, with the model predicting the evolution of pressure, water saturation, and oil saturation fields.

The neural operator is trained on 500 samples generated by the finite-volume simulator, using a combination of data loss and physics-informed loss as described above. Figure 5.21 shows the comparison between the PINO surrogate and the finite-volume reference. The left column displays the PINO predictions, the middle column shows the reference finite-volume solution, and the right column presents the difference between the two. For all panels, the first row corresponds to pressure, the second row to water saturation, and the third row to oil saturation.

The quantitative accuracy of the surrogates is assessed using R^2 and L2 metrics over the simulation time steps, as shown in Figure 5.22. Both the FNO and PINO models achieve high R^2 scores (>0.95) for pressure and saturation predictions throughout the simulation period. Figure 5.23 compares the production profiles between the reference finite-volume solver and the two surrogate models. The panels show bottom-hole pressure for the four injectors, oil production rate, water production rate, and water cut ratio for

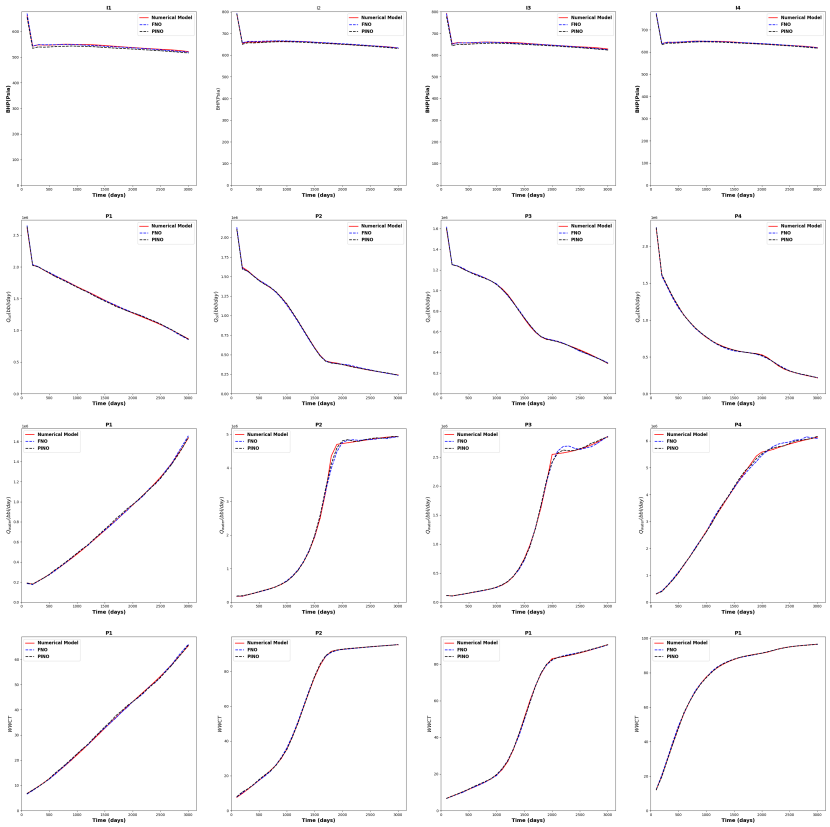


Figure 5.23: Production profile comparison between the true model (red) and the two surrogates (FNO and PINO). The panels show (first row) bottom-hole pressure for injectors I1–I4, (second row) oil production rate for producers P1–P4, (third row) water production rate for producers P1–P4, and (fourth row) water cut ratio for the four producers. The PINO and FNO surrogates closely match the reference production profiles. Adapted from [EOS23].

fig:5:5:2: production_ profiles

the four producers. The excellent agreement between the surrogates and the reference demonstrates that the neural operator models can effectively capture the complex dynamics of two-phase flow in porous media, including well responses critical for reservoir management decisions.

5.5.3 Neural operators for Foundational Models

5.5.3.1 MORPH

While the neural operator architectures discussed in Sections 5.4.1 to 5.4.3 are designed as task-specific surrogate, each tailored to a particular PDE family, fixed grid resolution, and spatial dimensionality, a parallel line of research has emerged with the ambitious goal of developing *foundation models* for scientific machine learning. Inspired by the success of large language models, a PDE foundation model is a task-agnostic generalist, pre-trained on a large and diverse corpus of physics simulations, that can be adapted to new downstream tasks

with minimal data and compute. A step in that direction is MORPH [Rau+25], a modality-agnostic foundation model that exemplifies this paradigm shift, seamlessly handling 1D, 2D, and 3D datasets with mixed scalar and vector fields within a unified architecture.

A central challenge in building a foundation model for PDEs is the heterogeneity of scientific datasets: 1D time series from point probes, 2D weather maps, and 3D turbulence simulations all coexist, often with varying numbers of physical fields (pressure, velocity components, density, etc.). To address this, MORPH introduces the *Unified Physics Tensor Format* (UPTF-7), which standardizes mini-batches into a common 7D shape:

$$\text{UPTF-7} = (N, T, F, C, D, H, W),$$

where N is the number of trajectories, T the number of time steps, F the number of physical fields, C the number of components per field (for vector-valued quantities), and D, H, W the spatial depth, height, and width. This format allows datasets of different modalities to be loaded on-the-fly without costly padding or homogenization, preserving the semantics of physical observations while enabling batched training across diverse sources. Table 5.4 illustrates how datasets from popular benchmarks such as *PDEBench* [Tak+22] and *The Well* [Oha+24] map to UPTF-7.

Dataset	**Native Shape**	**UPTF-7 Mapping**
1D Compressible Navier–Stokes	(B, T, W)	$(B, T, 3, 1, 1, 1, 1024)$
2D Diffusion–Reaction	$(B, T, H, W, 2)$	$(B, T, 2, 1, 1, 128, 128)$
3D Magnetohydrodynamics	$(B, T, D, H, W, 3, 3)$	$(B, T, 3, 3, 64, 64, 64)$

Table 5.4: Examples of datasets represented in the Unified Physics Tensor Format (UPTF-7).

tab:5:5:3:1:
morph_uptf

MORPH's modality-agnostic capability stems from three key architectural mechanisms, which together enable it to learn from heterogeneous data while respecting the computational constraints of high-resolution 3D simulations. A schematic overview is shown in Figure 5.24.

Unlike standard convolutional layers that operate on spatial dimensions, MORPH's first processing stage applies 3D convolutions along the component dimension C of the UPTF-7 tensor. This design jointly processes scalar and vector channels (e.g., the x, y, z components of a velocity field) to capture local interactions in a manner that is independent of the overall spatial dimensionality. This component-wise processing serves as a modality-agnostic alternative to the channel mixing operations in FNO (Section 5.4.1) and the block-diagonal MLP in AFNO (Section 5.4.2), enabling the model to handle inputs ranging from 1D point measurements to 3D volumetric fields without architectural reconfiguration.

After the initial convolution, the model must reason about the relationships between different physical fields: for instance, how pressure and velocity co-evolve in a fluid system. MORPH employs a learned cross-attention mechanism that condenses the F fields into a single fused representation. Given a set of

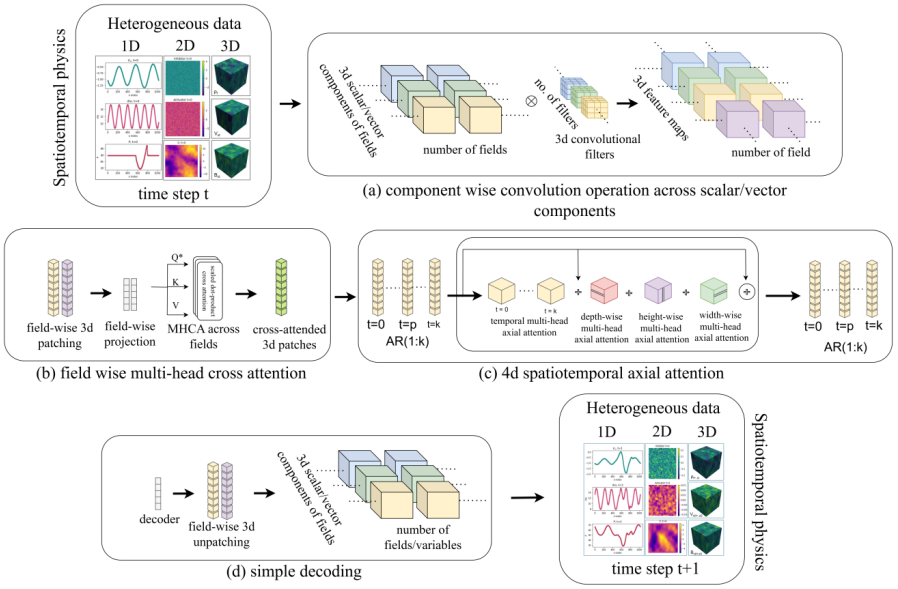


Figure 5.24: Schematic overview of the MORPH architecture. Input data in UPTF-7 format is processed by (a) component-wise convolutions that capture local interactions across scalar and vector channels, (b) inter-field cross-attention that selectively propagates information between physical fields, and (c) 4D axial attention that factorizes spatiotemporal self-attention along individual axes. Adapted from [Rau+25].

fig:5:5:3:1: morph_ architecture

field embeddings $\mathbf{X} \sim (F, E)$, a learned query $\mathbf{q} \sim (E,)$ attends over the fields:

$$\begin{aligned}\alpha^{(h)} &= \text{softmax}\left(\frac{(\mathbf{q}W_Q^{(h)})(\mathbf{X}W_K^{(h)})^\top}{\sqrt{d_h}}\right),\\ \mathbf{z}^{(h)} &= \alpha^{(h)}(\mathbf{X}W_V^{(h)}),\\ \mathbf{z} &= \text{Concat}_h(\mathbf{z}^{(h)})W_O.\end{aligned}$$

This operation is permutation-invariant to the ordering of fields and naturally accommodates a variable number of fields at inference time. This contrasts with the fixed field mixing in FNO and AFNO, which assume a fixed channel count.

The computational bottleneck in applying transformers to high-resolution PDE data is the quadratic cost of full spatio-temporal self-attention: $\mathcal{O}((TDHW)^2)$. To overcome this, MORPH factorizes the 4D space-time tensor into separate 1D attention operations along the time (T), depth (D), height (H), and width (W) axes. For each axis, the remaining dimensions are folded into the batch, attention is applied along the axis length, and the outputs are summed residually:

$$y = x + \tilde{x}^{(t)} + \tilde{x}^{(D)} + \tilde{x}^{(H)} + \tilde{x}^{(W)}.$$

This factorization reduces the computational complexity to $\mathcal{O}(T^2 + D^2 + H^2 + W^2)$, enabling MORPH to process large 3D grids (e.g., 64^3 voxels) that would be infeasible with full attention.

Variant	Conv filters	Attention dim	Heads	Depth	Parameters
MORPH-Ti (Tiny)	8	256	4	4	7M
MORPH-S (Small)	8	512	8	4	30M
MORPH-M (Medium)	8	768	12	8	126M
MORPH-L (Large)	8	1024	16	16	480M

tab:5:5:3:1: morph_variants

Table 5.5: MORPH model variants and their parameter counts.

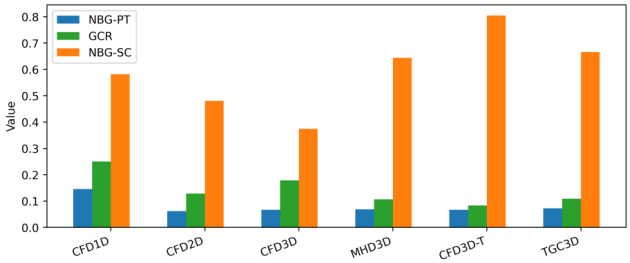


Figure 5.25: Zero-shot Gap-Closure Ratio (GCR) for MORPH pretrained on 2D incompressible Navier–Stokes and evaluated on six out-of-distribution targets. Positive GCR across all targets indicates successful transfer across modalities and physics. Adapted from [Rau+25].

fig:5:5:3:1: morph_gcr

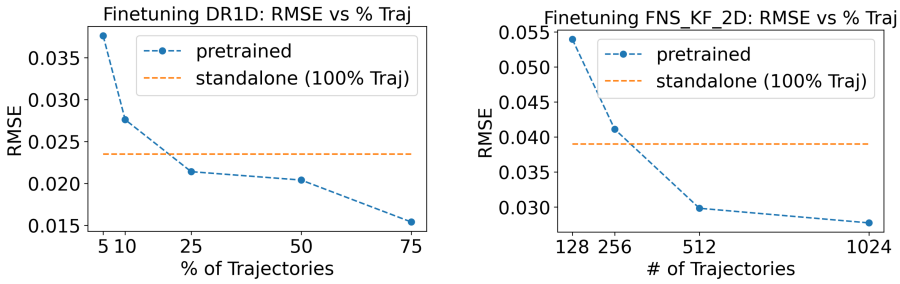
Pretraining and Scaling: MORPH is pretrained on six heterogeneous datasets spanning 1D, 2D, and 3D domains, including compressible Navier–Stokes flows, magnetohydrodynamics (MHD), turbulent self-gravitating flows (TGC), and reaction–diffusion systems. The pretraining corpus comprises datasets from *PDEBench* [Tak+22], *The Well* [Oha+24], and *PDEgym* [Her+24], totalling over 20.000 trajectories. Four model variants are released, scaling from 7 million to 480 million parameters (Table 5.5).

MORPH Results: Perhaps the most striking evidence of MORPH's generalist capability comes from zero-shot transfer experiments. A MORPH model pretrained *only* on 2D incompressible Navier–Stokes data was evaluated on six out-of-distribution targets without any fine-tuning. To quantify transfer, the authors introduced the *Gap-Closure Ratio* (GCR):

$$\mathrm{GCR} = \frac{1 - (E_{\mathrm{PT}}/E_{\mathrm{NB}})}{1 - (E_{\mathrm{SC}}/E_{\mathrm{NB}})},$$

where E_{PT} is the zero-shot error of the pretrained model, E_{SC} is the error of a model trained from scratch on the target, and E_{NB} is the error of a naive baseline (random initialization). A GCR value greater than zero indicates positive transfer. As shown in Figure 5.25, MORPH achieves positive GCR across all six targets, including compressible flows in 1D, 2D, and 3D, as well as magnetohydrodynamics (MHD) and turbulent self-gravitating flows (TGC). This demonstrates that the model learns reusable operators that transfer across both spatial dimensionality and underlying physics.

A foundation model's utility lies in its ability to adapt to new tasks with minimal data. Figure 5.26 compares MORPH's fine-tuning efficiency against a stand-alone MORPH model trained from scratch. On the 1D Diffusion–Reaction



(a) 1D Diffusion–Reaction task.

(b) 2D FNS-KF task.

Figure 5.26: Fine-tuning efficiency of MORPH. Pretrained models (MORPH-FM) achieve better performance with substantially less data than models trained from scratch (MORPH-SS). Adapted from [Rau+25].

fig:5:5:3:1: morph_finetune

task, the pretrained MORPH-FM-Ti outperforms the stand-alone model using only 25% of the training trajectories. On the 2D Forced Navier–Stokes Kolmogorov Flow (FNS-KF) task, the pretrained MORPH-FM-S surpasses the stand-alone model with fewer than 1% of the trajectories (128 samples vs. 20.000). This data efficiency is a direct consequence of the diverse pretraining corpus, which teaches the model the fundamental patterns of PDE evolution.

Connections to Neural Operator Theory and Architecture: MORPH's design reflects the theoretical foundations laid out in Section 5.3, where the kernel integral operator is introduced as the central building block for learning mappings between function spaces. Recall that the kernel operator is defined as:

$$(\mathcal{K}(a))(x) = \int_D \kappa(x, y)\, a(y)\, dy, \qquad \forall x \in D,$$

where $\kappa(x, y)$ is a kernel function that captures the interaction between points x and y in the domain D. In a discretized setting with L points, direct evaluation of this operator incurs a computational cost of $\mathcal{O}(L^2)$, which becomes prohibitive for high-resolution problems where L can be in the millions.

The axial attention mechanism in MORPH provides a practical, factorized approximation to this kernel integral. Instead of learning a full $L \times L$ kernel matrix, axial attention decomposes the interaction along the individual axes of the space-time domain. For a 4D domain with dimensions (T, D, H, W) (time, depth, height, width), the total number of points is $L = T \cdot D \cdot H \cdot W$. A factorised kernel can be expressed as:

$$\kappa(x, y) \approx \kappa_t(t_x, t_y) + \kappa_d(d_x, d_y) + \kappa_h(h_x, h_y) + \kappa_w(w_x, w_y),$$

where each $\kappa_t, \kappa_d, \kappa_h, \kappa_w$ is a 1D kernel operating independently along its respective axis. This factorisation is implemented by applying multi-head self-attention separately along each axis. The resulting computational complexity becomes $\mathcal{O}(T^2 + D^2 + H^2 + W^2)$, which is dramatically smaller than $\mathcal{O}((TDHW)^2)$ for high-dimensional grids. For example, a $64 \times 64 \times 64$ 3D grid would have $L = 262.144$ points, making full attention intractable with $\mathcal{O}(6.9 \times 10^{10})$ operations, while axial attention scales as $\mathcal{O}(3 \times 64^2) = \mathcal{O}(12.288)$, a reduction of over six orders of magnitude.

This factorization aligns with the theoretical insight from Section 5.3 that the kernel integral operator can be approximated by separable kernels without sacrificing the ability to model global correlations. By preserving interactions along each axis while keeping them independent, axial attention retains the expressive power to capture long-range dependencies in the data, while respecting the computational constraints of high-resolution scientific computing.

Compared to the Fourier Neural Operator (Section 5.4.1), which achieves global mixing through spectral convolution with complexity $\mathcal{O}(L \log L)$, axial attention offers a complementary approach that handles non-periodic boundaries and anisotropic features more naturally. Relative to AFNO (Section 5.4.2), which introduced block-diagonal channel mixing for vision tasks, MORPH extends the transformer backbone to arbitrary spatial dimensionalities and adds cross-field attention for multiphysics problems. And in contrast to PINO (Section 5.4.3), which incorporates physics-informed losses for data-efficient learning, MORPH explores a complementary route to data efficiency through large-scale pretraining and parameter-efficient fine-tuning with low-rank adapters (LoRA), achieving similar benefits without requiring explicit PDE residuals.

5.5.4 Neural operators for 3D Scene Reconstruction: PoissonNet

While the preceding applications focused on physics simulation, neural operators have also found a natural home in 3D computer vision, particularly for reconstructing geometric shapes from point cloud measurements. Classical approaches to 3D reconstruction often rely on solving the Poisson equation—a canonical PDE—to recover surfaces from oriented points. *PoissonNet* [And+23] replaces the traditional numerical solver with a Fourier Neural Operator (Section 5.4.1), learning the solution operator that maps point cloud measurements to the reconstructed 3D surface.

Given an oriented point cloud $\mathcal{P} = \{(\mathbf{p}_i, \mathbf{n}_i)\}_{i=1}^N$ representing a 3D surface, where $\mathbf{p}_i \in \mathbb{R}^3$ are point coordinates and $\mathbf{n}_i \in \mathbb{R}^3$ are unit normals, the goal is to reconstruct a watertight mesh that approximates the underlying surface. Poisson surface reconstruction [KBH06] formulates this as solving the Poisson equation:

$$\nabla^2 \phi = \nabla \cdot \mathbf{V}, \qquad \mathbf{V}(\mathbf{x}) = \sum_{i=1}^{N} \mathbf{n}_i \, \delta(\mathbf{x} - \mathbf{p}_i), \tag{5.69}$$

where $\mathbf{V}$ is a vector field derived from the oriented points (the normals extended to the volume), ϕ is an implicit function whose level set $\phi =$ constant defines the reconstructed surface, and δ denotes the Dirac delta distribution. The solution ϕ is then obtained by solving:

$$\phi(\mathbf{x}) = \int_{\mathbb{R}^3} G(\mathbf{x} - \mathbf{y}) \, (\nabla \cdot \mathbf{V})(\mathbf{y}) \, d\mathbf{y}, \tag{5.70}$$

where $G(\mathbf{x}) = 1/(4\pi\|\mathbf{x}\|)$ is the fundamental solution (Green's function) of the Laplace operator in $\mathbb{R}^3$. This formulation highlights the operator nature of the problem: there exists an operator $\mathcal{G}$ that maps the input vector field $\mathbf{V}$ to the implicit function ϕ. Traditional methods discretize this equation on a uniform

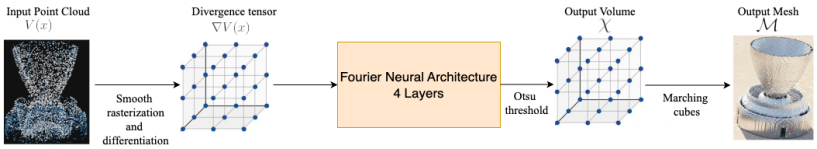


Figure 5.27: PoissonNet architecture: oriented point clouds are converted to a vector field, processed by Fourier Neural Operator layers in the spectral domain, and decoded to the implicit surface representation. The resolution-agnostic property enables training on coarse grids and inference at high resolution. Adapted from [And+23].

fig:5:5:4: poissonnet_arch

grid using finite differences, which limits resolution, requires careful parameter tuning, and does not easily generalize to irregular sampling patterns.

PoissonNet learns the operator $\mathcal{G} : \mathbf{V} \mapsto \phi$ using a Fourier Neural Operator. Recall from Section 5.4.1 that the Fourier Neural Operator parameterises the kernel integral operator in the spectral domain:

$$(\mathcal{K}_\theta(\mathbf{V}))(\mathbf{x}) = \mathrm{FFT}^{-1}\left(R_\theta(\mathbf{k}) \cdot \mathrm{FFT}(\mathbf{V})(\mathbf{k})\right)(\mathbf{x}), \tag{5.71}$$

where FFT denotes the Fourier transform, R_θ is a learnable complex-valued weight function in frequency space, and $\mathbf{k}$ denotes the frequency coordinates. For the 3D Poisson equation, the FNO architecture must learn the mapping from the 3D vector field $\mathbf{V}$ (with 3 channels) to the scalar field ϕ (1 channel). The architecture stacks multiple FNO layers; the final layer outputs the implicit function ϕ, from which the reconstructed surface is extracted via marching cubes.

As we have seen, key theoretical property of FNOs is their **discretization invariance**: they learn a mapping between function spaces that is independent of the discretization used during training. For Poisson surface reconstruction, this means:

- A model trained on coarse grids (e.g., 64^3 resolution) can be evaluated at arbitrarily higher resolutions (e.g., 256^3 or 512^3) without retraining
- The operator can be applied to point clouds with varying sampling densities
- The method naturally handles large-scale scenes by operating at the desired output resolution

Training Methodology: PoissonNet is trained in a supervised manner using synthetic data generated from known 3D shapes. The authors use the ShapeNet dataset [Cha+15], which contains over 50.000 3D models across 55 categories. For each shape:

1. A watertight mesh is converted to an implicit function ϕ_{gt} via *signed distance function* (SDF) computation on a regular grid
2. Oriented point clouds are generated by sampling points on the surface, with normals computed from the mesh

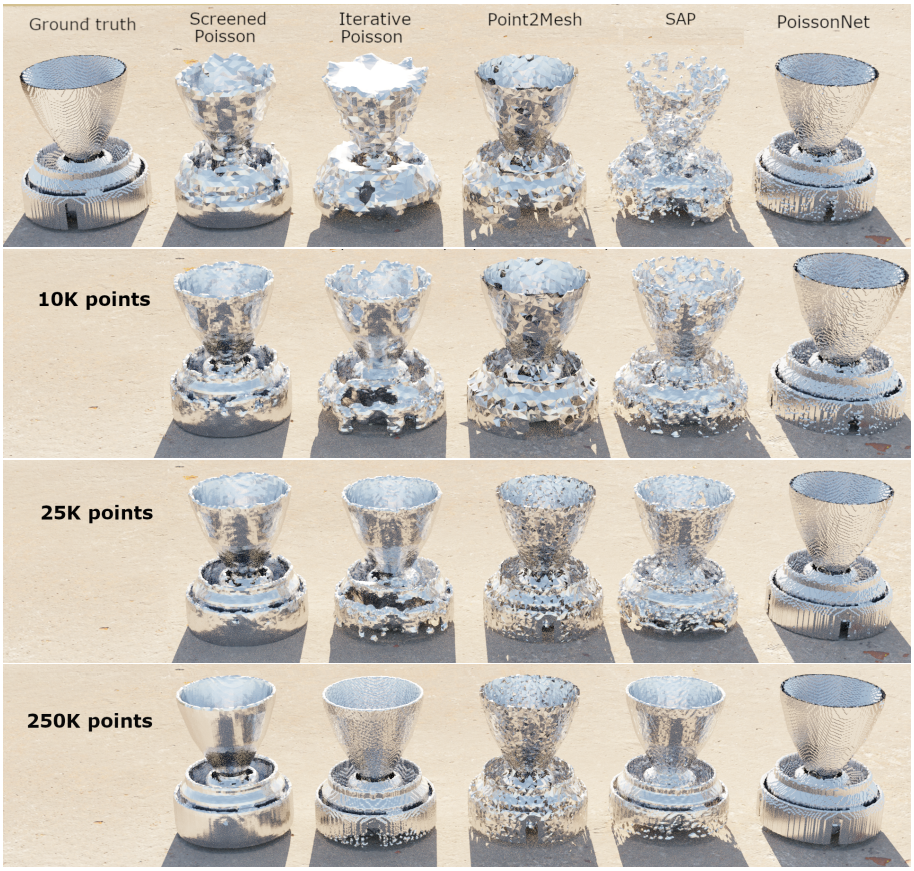


Figure 5.28: Result on a ShapeNet example with different sampling rates, the first row corresponds to 3.000 points. From [And+23].

fig:shapenet-3

3. The vector field $\mathbf{V}_{\text{input}}$ is constructed by spreading the point normals to the volume using a Gaussian kernel

The training data consists of pairs $(\mathbf{V}_{\text{input}}, \phi_{\text{gt}})$ at varying grid resolutions. The authors generate data at 64^3, 128^3, and 256^3 resolutions to evaluate resolution-agnostic properties.

Afterwards, the model is trained to minimize a combined loss function:

$$\mathcal{L} = \mathcal{L}_{\text{SDF}} + \lambda \mathcal{L}_{\text{normal}}, \tag{5.72}$$

where

$$\mathcal{L}_{\text{SDF}} = \|\phi_{\text{pred}} - \phi_{\text{gt}}\|_2^2,$$

is the mean squared error on the implicit function, and

$$\mathcal{L}_{\text{normal}} = \left\| \frac{\nabla \phi_{\text{pred}}}{\|\nabla \phi_{\text{pred}}\|} - \mathbf{n}_{\text{gt}} \right\|_2^2,$$

enforces consistency with the ground truth normals at the surface.

Their FNO architecture has:

- Number of Fourier modes: 32 (keeping the lowest 32 frequencies in each dimension)
- Number of layers: 4
- Hidden channel dimension: 64
- Batch size: 8
- Optimizer: Adam with learning rate 10^{-3}, decayed by factor 0.5 every 50 epochs
- Training epochs: 200 on ShapeNet

The training was performed on a single NVIDIA V100 GPU for approximately 2 days.

The model was trained on 64^3 grids, and tested on different (higher) resolutions producing high-quality reconstructions, achieving comparable quality to models trained directly on high-resolution data. This is a significant advantage over conventional CNNs, which require retraining for each resolution.

Bibliography

ALLEN19751017 [AC75] Allen, S. M. and Cahn, J. W. 'Coherent and incoherent equilibria in iron-rich iron-aluminum alloys'. In: *Acta Metallurgica* vol. 23, no. 9 (1975), pp. 1017–1026. DOI: https://doi.org/10.1016/0001-6160(75)90106-6.

alain2015variance [Ala+15] Alain, G. et al. 'Variance reduction in sgd by distributed importance sampling'. In: *arXiv preprint arXiv:1511.06481* (2015).

Ami2021_PhysicsInformedNeural_HagANHC [Ami+21] Amini Niaki, S. et al. 'Physics-informed neural network for modelling the thermochemical curing process of composite-tool systems during manufacture'. en. In: *Computer Methods in Applied Mechanics and Engineering* vol. 384 (Oct. 2021), p. 113959. DOI: 10.1016/j.cma.2021.113959. URL: https://www.sciencedirect.com/science/article/pii/S0045782521002966.

anagnostopoulos2023residual [Ana+23] Anagnostopoulos, S. J. et al. 'Residual-based attention and connection to information bottleneck theory in PINNs'. In: *arXiv preprint arXiv:2307.00379* (2023).

anagnostopoulos2025learning [Ana+25] Anagnostopoulos, S. J. et al. 'Learning in PINNs: Phase transition, diffusion equilibrium, and generalization'. In: *Neural Networks* (2025), p. 107983.

andrade2023poissonnet [And+23] Andrade-Loarca, H. et al. 'PoissonNet: Resolution-Agnostic 3D Shape Reconstruction using Fourier Neural Operators'. In: *arXiv preprint arXiv:2308.01766* (2023). Submitted to 3DV 2024.

arnold2009collected [Arn09] Arnold, V. I. *Collected Works: Representations of Functions, Celestial Mechanics and KAM Theory, 1957–1965*. Springer, 2009.

arnol1960representation [Arn60] Arnol'd, V. I. 'The representation of functions of several variables'. In: (1960).

aarts2001neural [AV01] Aarts, L. P. and Van Der Veer, P. 'Neural network method for solving partial differential equations'. In: *Neural Processing Letters* vol. 14, no. 3 (2001), pp. 261–271.

bacciu2020gentle [Bac+20] Bacciu, D. et al. 'A gentle introduction to deep learning for graphs'. In: *Neural Networks* vol. 129 (2020), pp. 203–221.

battiti1992first [Bat92] Battiti, R. 'First-and second-order methods for learning: between steepest descent and Newton's method'. In: *Neural computation* vol. 4, no. 2 (1992), pp. 141–166.

bahdanau2014neural [BCB14] Bahdanau, D., Cho, K. and Bengio, Y. 'Neural machine translation by jointly learning to align and translate'. In: *arXiv preprint arXiv:1409.0473* (2014).

BECHET20021 [BCT02] Béchet, E., Cuilliere, J.-C. and Trochu, F. 'Generation of a finite element MESH from stereolithography (STL) files'. In: *Computer-Aided Design* vol. 34, no. 1 (2002), pp. 1–17. DOI: https://doi.org/10.1016/S0010-4485(00)00146-9. URL: https://www.sciencedirect.com/science/article/pii/S0010448500001469.

bengio2009curriculum [Ben+09] Bengio, Y. et al. 'Curriculum learning'. In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

Bertalmio200 [Ber+00] Bertalmio, M. et al. 'Image inpainting'. In: *SIGGRAPH '00*. USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 417–424. DOI: 10.1145/344779.344972. URL: https://doi.org/10.1145/344779.344972.

braun2009constructive [BG09] Braun, J. and Griebel, M. 'On a constructive proof of Kolmogorov's superposition theorem'. In: *Constructive approximation* vol. 30, no. 3 (2009), pp. 653–675.

bhattacharya2021model [Bha+21] Bhattacharya, K. et al. 'Model reduction and neural networks for parametric PDEs'. In: *The SMAI journal of computational mathematics* vol. 7 (2021), pp. 121–157.

BischofMixtureofExperts2025EnsembleMF [BK22] Bischof, R. and Kraus, M. A. 'Mixture-of-Experts-Ensemble Meta-Learning for Physics-Informed Neural Networks'. In: *Forum Bauinformatik*. 2022. URL: https://api.semanticscholar.org/CorpusID:273161693.

bischof2025multi [BK25] Bischof, R. and Kraus, M. A. 'Multi-objective loss balancing for physics-informed deep learning'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 439 (2025), p. 117914.

EfficientKAN2024Git [Ble24] Blealtan. Repository. 2024. URL: https://github.com/Blealtan/efficient-kan/.

belkin2003laplacian [BN03] Belkin, M. and Niyogi, P. 'Laplacian eigenmaps for dimensionality reduction and data representation'. In: *Neural computation* vol. 15, no. 6 (2003), pp. 1373–1396.

bombini_2026_pinnunibz [Bom26] Bombini, A. *Material for the PhD course "Physics Informed Neural Networks and Neural Operators", held at the University of Bozen*. Version 20260324. Mar. 2026. DOI: 10.15161/oar.it/qgy53-gk347. URL: https://doi.org/10.15161/oar.it/qgy53-gk347.

bonev2025fourcastnet [Bon+25] Bonev, B. et al. 'Fourcastnet 3: A geometric approach to probabilistic machine-learning weather forecasting at scale'. In: *arXiv preprint arXiv:2507.12144* (2025).

broyden1970convergence [Bro70] Broyden, C. G. 'The convergence of a class of double-rank minimization algorithms 1. general considerations'. In: *IMA Journal of Applied Mathematics* vol. 6, no. 1 (1970), pp. 76–90.

bruna2013spectral [Bru13] Bruna, J. 'Spectral networks and locally connected networks on graphs'. In: *arXiv preprint arXiv:1312.6203* (2013).

buhmann2000radial [Buh00] Buhmann, M. D. 'Radial basis functions'. In: *Acta numerica* vol. 9 (2000), pp. 1–38.

byrd1995limited [Byr+95] Byrd, R. H. et al. 'A limited memory algorithm for bound constrained optimization'. In: *SIAM Journal on scientific computing* vol. 16, no. 5 (1995), pp. 1190–1208.

Calad2020 [Cal+20] Calad, C. et al. *Combining Machine Learning with Traditional Reservoir Physics for Predictive Modeling and Optimization of a Large Mature Waterflood Project in the Gulf of San Jorge Basin in Argentina*. https://doi.org/10.2118/199048-MS. 2020.

Calad2023 [Cal+23] Calad, C. et al. *Management and Optimization of a Large Waterflood Operation Applying a Physics Embedded Machine Learning Workflow*. https://doi.org/10.3997/2214-4609.202331040. 2023.

chen1993approximations [CC93] Chen, T. and Chen, H. 'Approximations of continuous functionals by neural networks with application to dynamic systems'. In: *IEEE Transactions on Neural networks* vol. 4, no. 6 (1993), pp. 910–918.

chen1995universal [CC95] Chen, T. and Chen, H. 'Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems'. In: *IEEE transactions on neural networks* vol. 6, no. 4 (1995), pp. 911–917.

vcebyvsev1853theorie [Čeb53] Čebyšev, P. L. *Théorie des mécanismes connus sous le nom de parallélogrammes*. Imprimerie de l'Académie impériale des sciences, 1853.

chang2015shapenet [Cha+15] Chang, A. X. et al. 'Shapenet: An information-rich 3d model repository'. In: *arXiv preprint arXiv:1512.03012* (2015).

chen2018gradnorm [Che+18] Chen, Z. et al. 'Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks'. In: *International conference on machine learning*. PMLR. 2018, pp. 794–803.

Chen2020 [Che+20] Chen, Y. et al. 'Physics-informed neural networks for inverse problems in nano-optics and metamaterials'. In: *Optics Express* vol. 28 (2020), pp. 11618–11633.

chung1996lectures [Chu96] Chung, F. R. 'Lectures on spectral graph theory'. In: *CBMS Lectures, Fresno* vol. 6, no. 92 (1996), pp. 17–21.

crawshaw2020multi [Cra20] Crawshaw, M. 'Multi-task learning with deep neural networks: A survey'. In: *arXiv preprint arXiv:2009.09796* (2020).

cuomo2022scientific [Cuo+22] Cuomo, S. et al. 'Scientific machine learning through physics–informed neural networks: Where we are and what's next'. In: *Journal of Scientific Computing* vol. 92, no. 3 (2022), p. 88.

Cybenko1989 [Cyb89] Cybenko, G. 'Approximation by superpositions of a sigmoidal function'. In: *Mathematics of Control, Signals and Systems* vol. 2, no. 4 (Dec. 1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

Che2021_DeepLearningMethod_ZhaCZ [CZ21] Cheng, C. and Zhang, G.-T. 'Deep Learning Method Based on Physics Informed Neural Network with Resnet Block for Solving Fluid Flow Problems'. en. In: *Water* vol. 13, no. 4 (Jan. 2021), p. 423. DOI: 10.3390/w13040423. URL: https://www.mdpi.com/2073-4441/13/4/423.

czarnecki2017sobolev [Cza+17] Czarnecki, W. M. et al. 'Sobolev Training for Neural Networks'. In: *arXiv preprint arXiv:1706.04859* (2017). URL: https://arxiv.org/abs/1706.04859.

defferrard2016convolutiona [DBV16] Defferrard, M., Bresson, X. and Vandergheynst, P. 'Convolutional neural networks on graphs with fast localized spectral filtering'. In: *Advances in neural information processing systems* vol. 29 (2016).

Depina2022 [Dep+22] Depina, I. et al. 'Application of physics-informed neural networks to inverse problems in unsaturated groundwater flow'. In: *Georisk* vol. 16, no. 1 (2022), pp. 21–36. DOI: 10.1080/17499518.2021.1971251.

duruisseaux2025fourier [DKA25] Duruisseaux, V., Kossaifi, J. and Anandkumar, A. 'Fourier Neural Operators Explained: A Practical Perspective'. In: *arXiv preprint arXiv:2512.01421* (2025).

de2021approximation [DLM21] De Ryck, T., Lanthaler, S. and Mishra, S. 'On the approximation of functions by tanh neural networks'. In: *Neural Networks* vol. 143 (2021), pp. 732–750.

de2022error [DM22] De Ryck, T. and Mishra, S. 'Error analysis for physics-informed neural networks (PINNs) approximating Kolmogorov PDEs'. In: *Advances in Computational Mathematics* vol. 48, no. 6 (2022), p. 79.

dosovitskiy2020image [Dos20] Dosovitskiy, A. 'An image is worth 16x16 words: Transformers for image recognition at scale'. In: *arXiv preprint arXiv:2010.11929* (2020).

dissanayake1994neural [DP94] Dissanayake, M. G. and Phan-Thien, N. 'Neural-network-based approximations for solving partial differential equations'. In: *communications in Numerical Methods in Engineering* vol. 10, no. 3 (1994), pp. 195–201.

Dwi2020_PhysicsInformedExtreme_SriDS [DS20] Dwivedi, V. and Srinivasan, B. 'Physics Informed Extreme Learning Machine (PIELM)–A rapid method for the numerical solution of partial differential equations'. en. In: *Neurocomputing* vol. 391 (May 2020), pp. 96–118. DOI: 10.1016/j.neucom.2019.12.099. URL: https://www.sciencedirect.com/science/article/pii/S0925231219318144.

das2022doe [DT22] Das, S. and Tesfamariam, S. 'State-of-the-art review of design of experiments for physics-informed deep learning'. In: *arXiv preprint arXiv:2202.06416* (2022).

dutta2025first [Dut+25] Dutta, A. et al. 'The first two months of Kolmogorov-Arnold Networks (KANs): A survey of the state-of-the-art'. In: *Archives of Computational Methods in Engineering* (2025), pp. 1–12.

eymard02100732 [EGH00] Eymard, R., Gallouët, T. and Herbin, R. 'Finite Volume Methods'. In: *Solution of Equation in $\mathbb{R}^n$ (Part 3), Techniques of Scientific Computing (Part 3)*. Ed. by Lions, J. L. and Ciarlet, P. Vol. 7. Handbook of Numerical Analysis. Elsevier, 2000, pp. 713–1020. DOI: 10.1016/S1570-8659(00)07005-8. URL: https://hal.science/hal-02100732.

Embree2016 [Emb16] Embree, M. *MATH/CS 5466 - Numerical Analysis*. Virginia Tech. 2016. URL: https://personal.math.vt.edu/embree/math5466/lecture10.pdf.

etienam_reservoir_readme [EOS23] Etienam, C., Ovcharenko, O. and Said, I. *Reservoir Simulation Forward Modelling with a Physics Informed Neural Operator (PINO) - 3D Implementation*. NVIDIA PhysicsNeMo Sym Examples. GitHub repository. 2023. URL: https://github.com/NVIDIA/physicsnemo-sym/tree/main/examples/reservoir_simulation/3D.

epstein2005well [Eps05] Epstein, C. 'How well does the finite Fourier transform approximate the Fourier transform?' In: *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences* vol. 58, no. 10 (2005), pp. 1421–1435.

Fan2021_HighEfficientHybrid_Fan [Fan21] Fang, Z. 'A High-Efficient Hybrid Physics-Informed Neural Networks Based on Convolutional Neural Network'. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–13. DOI: 10.1109/TNNLS.2021.3070878.

Fasshauer2014 [Fas14] Fasshauer, G. *Non-Symmetric Kernel Collocation for the Solution of Elliptic Partial Differential Equation*. Illinois Institute of Technology. 2014. URL: https://www.math.iit.edu/~fass/590/notes/Notes590_Ch19Print.pdf.

faure1982faure [Fau82] Faure, H. 'Discrepancy of sequences associated with number systems'. In: *Acta Arithmetica* vol. 41, no. 4 (1982), pp. 337–351.

fedus2022reviewsparseExpertModels [FDZ22] Fedus, W., Dean, J. and Zoph, B. *A Review of Sparse Expert Models in Deep Learning*. 2022. arXiv: 2209.01667 [cs.LG]. URL: https://arxiv.org/abs/2209.01667.

fletcher1970new [Fle70] Fletcher, R. 'A new approach to variable metric algorithms'. In: *The computer journal* vol. 13, no. 3 (1970), pp. 317–322.

Fornberg1988GenerationOF [For88] Fornberg, B. 'Generation of finite difference formulas on arbitrarily spaced grids'. In: *Mathematics of Computation* vol. 51 (1988), pp. 699–706. URL: https://api.semanticscholar.org/CorpusID:119513587.

Gustafson1998 [GA98] Gustafson, K. and Abe, T. 'The third boundary condition—was it robin's?' In: *The Mathematical Intelligencer* vol. 20, no. 1 (Mar. 1998), pp. 63–71. DOI: 10.1007/BF03024402. URL: https://doi.org/10.1007/BF03024402.

glorot2010understanding [GB10] Glorot, X. and Bengio, Y. 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

Goodfellow2016 [GBC16] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

Guillemot2014 [GL14] Guillemot, C. and Le Meur, O. 'Image Inpainting : Overview and Recent Advances'. In: *IEEE Signal Processing Magazine* vol. 31, no. 1 (2014), pp. 127–144. DOI: 10.1109/MSP.2013.2273004.

goldfarb1970family [Gol70] Goldfarb, D. 'A family of variable-metric methods derived by variational means'. In: *Mathematics of computation* vol. 24, no. 109 (1970), pp. 23–26.

girosi1989representation [GP89] Girosi, F. and Poggio, T. 'Representation properties of networks: Kolmogorov's theorem is irrelevant'. In: *Neural Computation* vol. 1, no. 4 (1989), pp. 465–469.

Gao2021_PhygeonetPhysicsInformed_SunGSW [GSW21] Gao, H., Sun, L. and Wang, J.-X. 'PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain'. en. In: *Journal of Computational Physics* vol. 428 (Mar. 2021), p. 110079. DOI: 10.1016/j.jcp.2020.110079. URL: https://www.sciencedirect.com/science/article/pii/S0021999120308536.

guibas2021adaptive [Gui+21a] Guibas, J. et al. 'Adaptive fourier neural operators: Efficient token mixers for transformers'. In: *arXiv preprint arXiv:2111.13587* (2021).

guibas2021efficient [Gui+21b] Guibas, J. et al. 'Efficient token mixing for transformers via adaptive fourier neural operators'. In: *International conference on learning representations*. 2021.

Guo2023causal [Guo24] Guo, S. *Unraveling the Design Pattern of Physics-Informed Neural Networks: Part 06 - Bring causality to PINN training*. Blog. 2024. URL: https://medium.com/data-science/unraveling-the-design-pattern-of-physics-informed-neural-networks-part-06-bcb3557199e2.

Gen2020_ModelingDynamicsPde_ZabGZ [GZ20] Geneva, N. and Zabaras, N. 'Modeling the dynamics of PDE systems with physics-constrained deep auto-regressive networks'. In: *Journal of Computational Physics* vol. 403 (2020), p. 109056. DOI: https://doi.org/10.1016/j.jcp.2019.109056. URL: https://www.sciencedirect.com/science/article/pii/S0021999119307612.

Hadamard1902 [Had02] Hadamard, J. *Sur les problèmes aux dérivées partielles et leur signification physique*. Reprinted in: *Princeton University Bulletin*, 1902. Princeton University Press, 1902.

halton1960efficiency [Hal60] Halton, J. H. 'On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals'. In: *Numerische Mathematik* vol. 2, no. 1 (1960), pp. 84–90.

hammersley1960monte [Ham60] Hammersley, J. M. 'Monte Carlo methods for solving multivariable problems'. In: *Annals of the New York Academy of Sciences* vol. 86, no. 3 (1960), pp. 844–874.

he2017mask [He+17] He, K. et al. 'Mask R-CNN'. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

He2020_PhysicsInformedNeural_BarHBTT [He+20] He, Q. et al. 'Physics-informed neural networks for multiphysics data assimilation with application to subsurface transport'. en. In: *Advances in Water Resources* vol. 141 (July 2020), p. 103610. DOI: 10.1016/j.advwatres.2020.103610. URL: https://www.sciencedirect.com/science/article/pii/S0309170819311649.

Heinlein66 [Hei66] Heinlein, R. A. *The Moon Is a Harsh Mistress*. Book. 1966.

herde2024poseidon [Her+24] Herde, M. et al. 'Poseidon: Efficient foundation models for pdes'. In: *Advances in Neural Information Processing Systems* vol. 37 (2024), pp. 72525–72624.

hoeltgen2019theoretical [Hoe+19] Hoeltgen, L. et al. 'Theoretical foundation of the weighted laplace inpainting problem'. In: *Applications of mathematics* vol. 64, no. 3 (2019), pp. 281–300.

de2022cost [Hoo+22] Hoop, M. V. de et al. 'The cost-accuracy trade-off in operator learning with neural networks'. In: *arXiv preprint arXiv:2203.13181* (2022).

hou2024kolmogorov [Hou+24] Hou, Y. et al. 'Kolmogorov-Arnold Networks: A Critical Assessment of Claims, Performance, and Practical Viability'. In: *arXiv preprint arXiv:2407.11075* (2024).

hochreiter1997long [HS97] Hochreiter, S. and Schmidhuber, J. 'Long short-term memory'. In: *Neural computation* vol. 9, no. 8 (1997), pp. 1735–1780.

hornik1989multilayer [HSW89] Hornik, K., Stinchcombe, M. and White, H. 'Multilayer feedforward networks are universal approximators'. In: *Neural networks* vol. 2, no. 5 (1989), pp. 359–366.

heydari2019softadapt [HTM19] Heydari, A. A., Thompson, C. A. and Mehmood, A. 'Softadapt: Techniques for adaptive loss weighting of neural networks with multi-part loss functions'. In: *arXiv preprint arXiv:1912.12355* (2019).

hu2024survey [Hu+24] Hu, S. et al. 'A survey on information bottleneck'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 46, no. 8 (2024), pp. 5325–5344.

ismayilova2024kolmogorov2 [II24] Ismayilova, A. and Ismailov, V. E. 'On the Kolmogorov neural networks'. In: *Neural Networks* vol. 176 (2024), p. 106333.

WierstrassInst2025 [Ins25] Institute, W. *Numerische Mathematik I*. 2025. URL: https://www.wias-berlin.de/people/john/LEHRE/lehre.html.

iserles2009first [Ise09] Iserles, A. *A first course in the numerical analysis of differential equations*. 2nd. Cambridge texts in applied mathematics. Cambridge: Cambridge University Press, 2009.

Isl2021_ExtractionMaterialProperties_ThaITMH [Isl+21] Islam, M. et al. 'Extraction of material properties through multi-fidelity deep learning from molecular dynamics simulation'. en. In: *Computational Materials Science* vol. 188 (Feb. 2021), p. 110187. DOI: 10.1016/j.commatsci.2020.110187. URL: https://www.sciencedirect.com/science/article/pii/S0927025620306789.

ismailov2023three [Ism23] Ismailov, V. E. 'A three layer neural network can represent any multivariate function'. In: *Journal of Mathematical Analysis and Applications* vol. 523, no. 1 (2023), p. 127096.

ismailov2025addressing [Ism25] Ismailov, V. E. 'Addressing common misinterpretations of KART and UAT in neural network literature'. In: *Neural Networks* (2025), p. 108361.

jacobs1991adaptive [Jac+91] Jacobs, R. A. et al. 'Adaptive mixtures of local experts'. In: *Neural computation* vol. 3, no. 1 (1991), pp. 79–87.

Jagtap2022 [Jag+22] Jagtap, A. D. et al. 'Physics-informed neural networks for inverse problems in supersonic flows'. In: *arXiv preprint arXiv:2202.11821* (2022).

jacot2018neural [JGH18] Jacot, A., Gabriel, F. and Hongler, C. 'Neural tangent kernel: Convergence and generalization in neural networks'. In: *Advances in neural information processing systems* vol. 31 (2018).

jiang2024mixtral [Jia+24] Jiang, A. Q. et al. 'Mixtral of experts'. In: *arXiv preprint arXiv:2401.04088* (2024).

jin2021nsfnets [Jin+21] Jin, X. et al. 'NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations'. In: *Journal of Computational Physics* vol. 426 (2021), p. 109951.

joe2008constructing [JK08] Joe, S. and Kuo, F. 'Constructing Sobol sequences with better two-dimensional projections'. In: *SIAM Journal on Scientific Computing* vol. 30, no. 5 (2008), pp. 2635–2654.

jagtap2020adaptive [JKK20] Jagtap, A. D., Kawaguchi, K. and Karniadakis, G. E. 'Adaptive activation functions accelerate convergence in deep and physics-informed neural networks'. In: *Journal of Computational Physics* vol. 404 (2020), p. 109136.

jentzen2023mathematical [JKW23] Jentzen, A., Kuckuck, B. and Wurstemberger, P. von. 'Mathematical introduction to deep learning: Methods, implementations, and theory'. In: *arXiv preprint arXiv:2310.20360* (2023).

Jin2022 [JMP22] Jin, H., Mattheakis, M. and Protopapas, P. 'Physics-Informed Neural Networks for Quantum Eigenvalue Problems'. In: *arXiv preprint arXiv:2203.00451* (2022).

Johnson2019bfgs [Joh19] Johnson, S. G. *Quasi-Newton Optimization: Origin of the BFGS update*. 18.33 MIT. 2019. URL: https://ocw.mit.edu/courses/18-335j-introduction-to-numerical-methods-spring-2019/1b52b607c223977b35ba1d826e4d8df1_MIT18_335JS19_lec30.pdf.

KANSA1990147 [Kan90] Kansa, E. 'Multiquadrics - A scattered data approximation scheme with applications to computational fluid-dynamics - II solutions to parabolic, hyperbolic and elliptic partial differential equations'. In: *Computers & Mathematics with Applications* vol. 19, no. 8 (1990), pp. 147–161. DOI: https://doi.org/10.1016/0898-1221(90)90271-K.

Kapoor2023 [Kap+23] Kapoor, T. et al. 'Physics-Informed Neural Networks for Solving Forward and Inverse Problems in Complex Beam Systems'. In: *IEEE Transactions on Neural Networks and Learning Systems* vol. 35, no. 5 (2023), pp. 5981–5995. DOI: 10.1109/TNNLS.2023.3310585.

kingma2017adammethodstochasticoptimization [KB17] Kingma, D. P. and Ba, J. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

kazhdan2006poisson [KBH06] Kazhdan, M., Bolitho, M. and Hoppe, H. 'Poisson surface reconstruction'. In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 4. 2006.

katharopoulos2017biased [KF17] Katharopoulos, A. and Fleuret, F. 'Biased importance sampling for deep neural network training'. In: *arXiv preprint arXiv:1706.00043* (2017).

katharopoulos2018not [KF18] Katharopoulos, A. and Fleuret, F. 'Not all samples are created equal: Deep learning with importance sampling'. In: *International conference on machine learning*. PMLR. 2018, pp. 2525–2534.

kendall2017uncertainties [KG17] Kendall, A. and Gal, Y. 'What uncertainties do we need in bayesian deep learning for computer vision?' In: *Advances in neural information processing systems* vol. 30 (2017).

kendall2018multi [KGC18] Kendall, A., Gal, Y. and Cipolla, R. 'Multi-task learning using uncertainty to weigh losses for scene geometry and semantics'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7482–7491.

khodakarami2026spectral [Kho+26] Khodakarami, S. et al. 'Spectral bias in physics-informed and operator learning: Analysis and mitigation guidelines'. In: *arXiv preprint arXiv:2602.19265* (2026).

khanra2026physics [KKB26] Khanra, S., Kukreja, V. K. and Bala, I. 'Physics-informed neural networks for differential equation solutions: A comprehensive review'. In: *Neurocomputing* (2026), p. 133317.

kolmogorov1957representations [Kol57] Kolmogorov, A. N. 'On the representations of continuous functions of many variables by superposition of continuous functions of one variable and addition'. In: *Dokl. Akad. Nauk USSR*. Vol. 114. 1957, pp. 953–956.

koppen2002training [Kop02] Koppen, M. 'On the training of a kolmogorov network'. In: *International Conference on Artificial Neural Networks*. Springer. 2002, pp. 474–479.

kovachki2023neural [Kov+23] Kovachki, N. et al. 'Neural operator: Learning maps between function spaces with applications to pdes'. In: *Journal of Machine Learning Research* vol. 24, no. 89 (2023), pp. 1–97.

krishnapriyan2021characterizing [Kri+21] Krishnapriyan, A. et al. 'Characterizing possible failure modes in physics-informed neural networks'. In: *Advances in neural information processing systems* vol. 34 (2021), pp. 26548–26560.

kurth2023fourcastnet [Kur+23] Kurth, T. et al. 'Fourcastnet: Accelerating global high-resolution weather forecasting using adaptive fourier neural operators'. In: *Proceedings of the platform for advanced scientific computing conference*. 2023, pp. 1–11.

kutyniok2022mathematics [Kut22] Kutyniok, G. 'The mathematics of artificial intelligence'. In: *Proceedings of the International Congress of Mathematicians*. 2022. arXiv: 2203.08890 [cs.LG]. URL: https://arxiv.org/abs/2203.08890.

Korteweg01051895 [KV95] Korteweg, D. J. and Vries, G. de. 'XLI. On the change of form of long waves advancing in a rectangular canal, and on a new type of long stationary waves'. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* vol. 39, no. 240 (1895), pp. 422–443. DOI: 10.1080/14786449508620739.

kumar2011multilayer [KY11] Kumar, M. and Yadav, N. 'Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: a survey'. In: *Computers & Mathematics with Applications* vol. 62, no. 10 (2011), pp. 3796–3811.

kharazmi2019variational [KZK19] Kharazmi, E., Zhang, Z. and Karniadakis, G. E. 'Variational physics-informed neural networks for solving partial differential equations'. In: *arXiv preprint arXiv:1912.00873* (2019).

Lam2020BFGS [Lam20] Lam, A. *BFGS in a Nutshell: An Introduction to Quasi-Newton Methods*. Medium. 2020. URL: https://medium.com/data-science/bfgs-in-a-nutshell-an-introduction-to-quasi-newton-methods-21b0e13ee504.

liu2024config [LCT24] Liu, Q., Chu, M. and Thuerey, N. 'Config: Towards conflict-free training of physics informed neural networks'. In: *arXiv preprint arXiv:2408.11104* (2024).

lecun2002efficient [LeC+02] LeCun, Y. et al. 'Efficient backprop'. In: *Neural networks: Tricks of the trade*. Springer, 2002, pp. 9–50.

lemieux2009monte [Lem09] Lemieux, C. *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer, 2009.

li2020fourier [Li+20] Li, Z. et al. 'Fourier neural operator for parametric partial differential equations'. In: *arXiv preprint arXiv:2010.08895* (2020).

li2024physics [Li+24] Li, Z. et al. 'Physics-informed neural operator for learning partial differential equations'. In: *ACM/IMS Journal of Data Science* vol. 1, no. 3 (2024), pp. 1–27.

li2024kolmogorov [Li24] Li, Z. 'Kolmogorov-arnold networks are radial basis function networks'. In: *arXiv preprint arXiv:2405.06721* (2024).

Linge2020 [Lin20] Linge, S. *Programming for Computations - A Gentle Introduction to Numerical Simulations with Python*. Springer Nature, 2020. DOI: 10.1007/978-3-030-16877-3. URL: http://library.oapen.org/handle/20.500.12657/23103.

liu2022convnet [Liu+22] Liu, Z. et al. 'A convnet for the 2020s'. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 11976–11986.

liu2024finer [Liu+24a] Liu, Z. et al. 'Finer: Flexible spectral-bias tuning in implicit neural representation by variable-periodic activation functions'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 2713–2722.

liu2024kan [Liu+24b] Liu, Z. et al. 'Kan: Kolmogorov-arnold networks'. In: *arXiv preprint arXiv:2404.19756* (2024).

lee1990neural [LK90] Lee, H. and Kang, I. S. 'Neural algorithm for solving differential equations'. In: *Journal of computational physics* vol. 91, no. 1 (1990), pp. 110–131.

Lagaris98 [LLF98] Lagaris, I., Likas, A. and Fotiadis, D. 'Artificial neural networks for solving ordinary and partial differential equations'. In: *IEEE Transactions on Neural Networks* vol. 9, no. 5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

Landau2015 [LPB15] Landau, R. H., Páez, M. J. and Bordeianu, C. C. *Computational Physics: Problem Solving with Python*. 3rd. Wiley-VCH, 2015.

lu2021learning [Lu+21] Lu, L. et al. 'Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators'. In: *Nature machine intelligence* vol. 3, no. 3 (2021), pp. 218–229.

lin1993realization [LU93] Lin, J.-N. and Unbehauen, R. 'On the realization of a kolmogorov network'. In: *Neural Computation* vol. 5, no. 1 (1993), pp. 18–20.

monaco2023training [MA23] Monaco, S. and Apiletti, D. 'Training physics-informed neural networks: One learning to rule them all?' In: *Results in Engineering* vol. 18 (2023), p. 101023.

DeMarchi2018 [Mar18] Marchi, S. de. *Lectures on Radial Basis Functions*. University of Padua. 2018. URL: https://www.math.unipd.it/~demarchi/RBF/LectureNotes.pdf.

mckay1979lhs [MBC79] McKay, M. D., Beckman, R. J. and Conover, W. J. 'A comparison of three methods for selecting values of input variables in the analysis of output from a computer code'. In: *Technometrics* vol. 21, no. 2 (1979), pp. 239–245.

mehta2019high [Meh+19] Mehta, P. et al. 'A high-bias, low-variance introduction to machine learning for physicists'. In: *Physics reports* vol. 810 (2019), pp. 1–124.

mildenhall2021nerf [Mil+21] Mildenhall, B. et al. 'Nerf: Representing scenes as neural radiance fields for view synthesis'. In: *Communications of the ACM* vol. 65, no. 1 (2021), pp. 99–106.

AwesomeKan2024Git [min24] mintisan. Repository. 2024. URL: https://github.com/mintisan/awesome-kan/.

morris2021hilbert [Mor21] Morris, S. 'Hilbert 13: Are there any genuine continuous multivariate real-valued functions?' In: *Bulletin of the American Mathematical Society* vol. 58, no. 1 (2021), pp. 107–118.

FreqNotes [MSR18] Mahmud, S., Snigdha, F. and Rakin, A. 'Development of a Novel Method for Automatic Detection of Musical Chords'. In: *Scientific Modelling and Research* vol. 3 (Jan. 2018), pp. 15–22. DOI: 10.20448/808.3.1.15.22.

srt2024physicsnemo [Muk+24] Mukundakrishnan, K. et al. *Spotlight: Stone Ridge Technology Accelerates Reservoir Simulation Workflows with NVIDIA PhysicsNeMo on AWS*. NVIDIA Technical Blog. Accessed: 2025-03-23. Dec. 2024. URL: https://developer.nvidia.com/blog/spotlight-stone-ridge-technology-accelerates-reservoir-simulation-workflows-with-nvidia-physicsnemo-on-aws/.

Comsol2025 [Mul] Multiphysics, C. *An Introduction to Finite Element Method*. FEM. URL: https://www.comsol.it/multiphysics/finite-element-method.

nabian2021importance [NGM21] Nabian, M. A., Gladstone, R. J. and Meidani, H. 'Efficient training of physics-informed neural networks via importance sampling'. In: *Computer-Aided Civil and Infrastructure Engineering* vol. 36, no. 8 (2021), pp. 962–977.

niederreiter1992random [Nie92] Niederreiter, H. *Random Number Generation and Quasi-Monte Carlo Methods*. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1992.

noorizadegan2025practitioner [Noo+25] Noorizadegan, A. et al. 'A Practitioner's Guide to Kolmogorov-Arnold Networks'. In: *arXiv preprint arXiv:2510.25781* (2025).

GuideToKAN_GitHub [Noo25] Noorizadegan, A. *KAN Review: Companion Repository for "A Practitioner's Guide to Kolmogorov-Arnold Networks"*. https://github.com/AmirNoori68/kan-review. Accessed: 2025-11-01. 2025.

NvidiaPhysicsNemoDocumentation [Nvi26] Nvidia. *NVIDIA PhysicsNeMo Sym Documentation*. Nvidia. 2026. URL: https://docs.nvidia.com/deeplearning/physicsnemo/physicsnemo-sym/.

nocedal2006numerical [NW06] Nocedal, J. and Wright, S. J. *Numerical optimization*. Springer, 2006.

osher2004level [OFP04] Osher, S., Fedkiw, R. and Piechor, K. 'Level set methods and dynamic implicit surfaces'. In: *Appl. Mech. Rev.* vol. 57, no. 3 (2004), B15–B15.

ohana2024well [Oha+24] Ohana, R. et al. 'The Well: A Large-Scale Collection of Diverse Physics Simulations for Machine Learning'. In: *arXiv preprint arXiv:2407.12345* (2024).

park2019deepsdf [Par+19] Park, J. J. et al. 'Deepsdf: Learning continuous signed distance functions for shape representation'. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 165–174.

Patacchio2020 [Pat+20] Patacchio, A. et al. 'Semi-supervised neural networks solve an inverse problem for modeling COVID-19 spread'. In: *arXiv preprint arXiv:2010.05074* (2020).

pathak2022fourcastnet [Pat+22] Pathak, J. et al. 'Fourcastnet: A global data-driven high-resolution weather model using adaptive fourier neural operators'. In: *arXiv preprint arXiv:2202.11214* (2022).

poggio2020theoretica [PBL20] Poggio, T., Banburski, A. and Liao, Q. 'Theoretical issues in deep networks'. In: *Proceedings of the National Academy of Sciences* vol. 117, no. 48 (2020), pp. 30039–30045.

pal2024understanding [PD24] Pal, A. and Das, D. 'Understanding the limitations of B-Spline KANs: Convergence dynamics and computational efficiency'. In: *Dimensions* vol. 21 (2024), p. 22.

penwarden2023unified [Pen+23] Penwarden, M. et al. 'A unified scalable framework for causal sweeping strategies for physics-informed neural networks (PINNs) and their temporal decompositions'. In: *Journal of Computational Physics* vol. 493 (2023), p. 112464.

quarteroni2020numerica [Qua20] Quarteroni, A. *Numerical models for differential problems*. Third. Vol. 16. Modeling, Simulation & Applications. Cham: Springer, 2020. DOI: 10.1007/978-3-319-49316-9. URL: https://www.worldcat.org/title/numerical-models-for-differential-problems/oclc/1204362090&referer=brief_results.

rahaman2019spectral [Rah+19] Rahaman, N. et al. 'On the spectral bias of neural networks'. In: *International conference on machine learning*. PMLR. 2019, pp. 5301–5310.

rautela2025morph [Rau+25] Rautela, M. S. et al. 'Morph: Pde foundation models with arbitrary data modality'. In: *arXiv preprint arXiv:2509.21670* (2025).

redmon2016you [Red+16] Redmon, J. et al. 'You only look once: Unified, real-time object detection'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

redmon2017yolo9000 [RF17] Redmon, J. and Farhadi, A. 'YOLO9000: better, faster, stronger'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

RodriguezMIT [Rod21] Rodriguez, C. *Lecture Notes 18.102: Introduction to Functional Anlysis*. 2021. URL: https://ocw.mit.edu/courses/18-102-introduction-to-functional-analysis-spring-2021/resources/mit18_102s21_full_lec/.

Rowan2026small [Row26] Rowan, C. *On the Possibility of Small Networks for Physics-Informed Learning*. blog post. 2026. URL: https://towardsdatascience.com/on-the-possibility-of-small-networks-for-physics-informed-learning/.

raissi2017physics [RPK17a] Raissi, M., Perdikaris, P. and Karniadakis, G. E. 'Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations'. In: *arXiv preprint arXiv:1711.10561* (2017).

raissi2017physicsinformeddeeplearning [RPK17b] Raissi, M., Perdikaris, P. and Karniadakis, G. E. *Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*. 2017. arXiv: 1711.10566 [cs.AI]. URL: https://arxiv.org/abs/1711.10566.

RAISSI2019686 [RPK19] Raissi, M., Perdikaris, P. and Karniadakis, G. 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations'. In: *Journal of Computational Physics* vol. 378 (2019), pp. 686–707. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

rahimi2007random [RR07] Rahimi, A. and Recht, B. 'Random features for large-scale kernel machines'. In: *Advances in neural information processing systems* vol. 20 (2007).

Ram2021_SpinnSparsePhysics_RamRR [RR21] Ramabathiran, A. A. and Ramachandran, P. 'SPINN: Sparse, Physics-based, and partially Interpretable Neural Networks for PDEs'. en. In: *Journal of Computational Physics* vol. 445 (Nov. 2021), p. 110600. DOI: 10.1016/j.jcp.2021.110600. URL: https://www.sciencedirect.com/science/article/pii/S0021999121004952.

rvachev1975method [Rva75] Rvachev, V. 'Method of R-functions in boundary-value problems'. In: *Soviet Applied Mechanics* vol. 11, no. 4 (1975), pp. 345–354.

sanz2024first [SA24] Sanz-Alonso, D. and Al-Ghattas, O. 'A first course in monte carlo methods'. In: *arXiv preprint arXiv:2405.16359* (2024).

ss2024chebyshev [SAK+24] SS, S., AR, K., KP, A. et al. 'Chebyshev polynomial-based kolmogorov-arnold networks: An efficient architecture for nonlinear function approximation'. In: *arXiv preprint arXiv:2405.07200* (2024).

salsa2016partial [Sal22] Salsa, S. *Partial differential equations in action*. Springer, 2022. DOI: 10.1007/978-3-031-21853-8. URL: https://mate.dm.uba.ar/~jfbonder/libro/(Universitext_)Sandro_Salsa-Partial_Differential_Equations_in_Action_from_Modelling_to_Theory-Springer(2008).pdf.

saxe2019information [Sax+19] Saxe, A. M. et al. 'On the information bottleneck theory of deep learning'. In: *Journal of Statistical Mechanics: Theory and Experiment* vol. 2019, no. 12 (2019), p. 124020.

scardapane2024alice [Sca24] Scardapane, S. 'Alice's Adventures in a Differentiable Wonderland–Volume I, A Tour of the Land'. In: *arXiv preprint arXiv:2404.17625* (2024). URL: https://www.sscardapane.it/alice-book/.

Schmidt2005 [Sch] Schmidt, M. *minFunc: unconstrained differentiable multivariate optimization in Matlab*. code. URL: https://www.cs.ubc.ca/~schmidtm/Software/minFunc.html.

Sch2021_ExtremeTheoryFunctional_FurSFL [Sch+21] Schiassi, E. et al. 'Extreme theory of functional connections: A fast physics-informed neural network method for solving ordinary and partial differential equations'. en. In: *Neurocomputing* vol. 457 (Oct. 2021), pp. 334–356. DOI: 10.1016/j.neucom.2021.06.015. URL: https://www.sciencedirect.com/science/article/pii/S0925231221009140.

Schaback2007 [Sch07] Schaback, R. *A Practical Guide to Radial Basis Functions*. Draft. 2007. URL: https://num.math.uni-goettingen.de/schaback/teaching/sc.pdf.

schmidt2021kolmogorov [Sch21] Schmidt-Hieber, J. 'The Kolmogorov–Arnold representation theorem revisited'. In: *Neural networks* vol. 137 (2021), pp. 119–126.

Schmidhuber92 [Sch92] Schmidhuber, J. 'Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks'. In: *Neural Computation* vol. 4, no. 1 (Jan. 1992), pp. 131–139. DOI: 10.1162/neco.1992.4.1.131. eprint: https://direct.mit.edu/neco/article-pdf/4/1/131/812242/neco.1992.4.1.131.pdf. URL: https://doi.org/10.1162/neco.1992.4.1.131.

strang1973analysis [SF+73] Strang, G., Fix, G. J. et al. *An analysis of the finite element method*. Vol. 212. Prentice-hall, 1973.

srivastava2015training [SGS15] Srivastava, R. K., Greff, K. and Schmidhuber, J. 'Training very deep networks'. In: *Advances in neural information processing systems* vol. 28 (2015).

shapiro2007semi [Sha07] Shapiro, V. 'Semi-analytic geometry with R-functions'. In: *ACTA numerica* vol. 16 (2007), pp. 239–303.

shanno1970conditioning [Sha70] Shanno, D. F. 'Conditioning of quasi-Newton methods for function minimization'. In: *Mathematics of computation* vol. 24, no. 111 (1970), pp. 647–656.

Shapiro1991 [Sha91] Shapiro, V. *Theory of R-functions and applications: A primer*. Tech. rep. Tech. Rep. CPA88-3. Cornell Programmable Automation, Sibley School of Mechanical Engineering, 1991.

shukla2024comprehensive [Shu+24] Shukla, K. et al. 'A comprehensive and FAIR comparison between MLP and KAN representations for differential equations and operator networks'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 431 (2024), p. 117290.

sitzmann2020implicit [Sit+] Sitzmann, V. et al. *Implicit neural representations with periodic activation functions, web page*. public link. URL: https://www.vincentsitzmann.com/siren/.

sitzmann2020implicit [Sit+20] Sitzmann, V. et al. 'Implicit neural representations with periodic activation functions'. In: *Advances in neural information processing systems* vol. 33 (2020), pp. 7462–7473.

slonim2002information [Slo02] Slonim, N. 'The information bottleneck: Theory and applications'. PhD thesis. Hebrew University of Jerusalem Jerusalem, Israel, 2002.

Smith2022 [Smi22] Smith, E. 'Automated Solution of PDEs with FEniCS'. In: *Introduction to the Tools of Scientific Computing*. Cham: Springer International Publishing, 2022, pp. 321–358. DOI: 10.1007/978-3-031-16972-4_15. URL: https://doi.org/10.1007/978-3-031-16972-4_15.

sobol1967distribution [Sob67] Sobol, I. M. 'On the distribution of points in a cube and the approximate evaluation of integrals'. In: *USSR Computational Mathematics and Mathematical Physics* vol. 7, no. 4 (1967), pp. 86–112.

son2021sobolev [Son+21] Son, H. et al. 'Sobolev Training for the Neural Network Solutions of PDEs'. In: *arXiv preprint arXiv:2101.08932* (2021). URL: https://arxiv.org/abs/2101.08932.

soviany2022curriculum [Sov+22] Soviany, P. et al. 'Curriculum learning: A survey'. In: *International Journal of Computer Vision* vol. 130, no. 6 (2022), pp. 1526–1565.

soydaner2022attention [Soy22] Soydaner, D. 'Attention mechanism in neural networks: where it comes and where it goes'. In: *Neural Computing and Applications* vol. 34, no. 16 (2022), pp. 13371–13385.

JacobiKan2024Git [Spa24] SpaceLearner. Repository. 2024. URL: https://github.com/SpaceLearner/JacobiKAN.

sprecher1965structure [Spr65] Sprecher, D. A. 'On the structure of continuous functions of several variables'. In: *Transactions of the American Mathematical Society* vol. 115 (1965), pp. 340–355.

sirignano2018dgm [SS18] Sirignano, J. and Spiliopoulos, K. 'DGM: A deep learning algorithm for solving partial differential equations'. In: *Journal of computational physics* vol. 375 (2018), pp. 1339–1364.

Sukumar2022 [SS22] Sukumar, N. and Srivastava, A. 'Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 389 (2022), p. 114333. eprint: arXiv:2104.08426. URL: https://arxiv.org/pdf/2104.08426.pdf.

shwartz2017opening [ST17] Shwartz-Ziv, R. and Tishby, N. 'Opening the black box of deep neural networks via information'. In: *arXiv preprint arXiv:1703.00810* (2017).

staudemeyer2019understanding [Sta19] Staudemeyer, R. 'Understanding LSTM–a tutorial into long short-term memory recurrent neural networks'. In: *arXiv preprint arXiv:1909.09586* (2019).

srt_echelon_software [Sto25] Stone Ridge Technology. *ECHELON: The Fastest Reservoir Simulation Tool in the World*. High-performance GPU-native reservoir simulator. Developed since 2012; joint partnership with Eni since 2018; runs on NVIDIA and AMD GPUs; supports black oil and compositional models; features CPR-AMG preconditioners and GMRES solvers. 2025. URL: https://stoneridgetechnology.com/ (visited on 23/03/2025).

Suli2021 [Sul21] Suli, E. *B6.1 Numerical Solution of Partial Differential Equations*. 2021. URL: https://people.maths.ox.ac.uk/suli/nspde.2021.2022.pdf.

takamoto2022pdebench [Tak+22] Takamoto, M. et al. 'PDEBench: An Extensive Benchmark for Scientific Machine Learning'. In: *Advances in Neural Information Processing Systems* vol. 35 (2022), pp. 1596–1611.

tancik2020fourier [Tan+20] Tancik, M. et al. 'Fourier features let networks learn high frequency functions in low dimensional domains'. In: *Advances in neural information processing systems* vol. 33 (2020), pp. 7537–7547.

tartakovsky2018learning [Tar+18] Tartakovsky, A. M. et al. 'Learning parameters and constitutive relationships with physics informed deep neural networks'. In: *arXiv preprint arXiv:1808.03398* (2018).

Tar2020_PhysicsInformedDeep_MarTMP [Tar+20] Tartakovsky, A. M. et al. 'Physics-Informed Deep Neural Networks for Learning Parameters and Constitutive Relationships in Subsurface Flow Problems'. en. In: *Water Resources Research* vol. 56, no. 5 (2020), e2019WR026731. DOI: 10.1029/2019WR026731. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/2019WR026731.

toscano2025pinns [Tos+25] Toscano, J. D. et al. 'From pinns to pikans: Recent advances in physics-informed machine learning'. In: *Machine Learning for Computational Science and Engineering* vol. 1, no. 1 (2025), pp. 1–43.

tishby2000information [TPB00] Tishby, N., Pereira, F. C. and Bialek, W. 'The information bottleneck method'. In: *arXiv preprint physics/0004057* (2000).

tishby2015deep [TZ15] Tishby, N. and Zaslavsky, N. 'Deep learning and the information bottleneck principle'. In: *2015 ieee information theory workshop (itw)*. Ieee. 2015, pp. 1–5.

urban2025unveiling [USP25] Urbán, J. F., Stefanou, P. and Pons, J. A. 'Unveiling the optimization process of physics informed neural networks: How accurate and competitive can PINNs be?' In: *Journal of Computational Physics* vol. 523 (2025), p. 113656.

varadhan1967behavior [Var67] Varadhan, S. R. S. 'On the behavior of the fundamental solution of the heat equation with variable coefficients'. In: *Communications on Pure and Applied Mathematics* vol. 20, no. 2 (1967), pp. 431–455.

vaswani2017attention [Vas+17] Vaswani, A. et al. 'Attention is all you need'. In: *Advances in neural information processing systems* vol. 30 (2017).

Via2021_EstimatingModelInadequacy_NasVNDY [Via+21] Viana, F. A. C. et al. 'Estimating model inadequacy in ordinary differential equations with physics-informed neural networks'. en. In: *Computers & Structures* vol. 245 (Mar. 2021), p. 106458. DOI: 10.1016/j.compstruc.2020.106458. URL: https://www.sciencedirect.com/science/article/pii/S0045794920302613.

viswanathan2019unreasonable [Vis19] Viswanathan, R. *The unreasonable effectiveness of quasirandom sequences*. https://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/. 2019.

vitushkin2004hilbert [Vit04] Vitushkin, A. G. 'On Hilbert's thirteenth problem and related questions'. In: *Russian Mathematical Surveys* vol. 59, no. 1 (2004), pp. 11–25.

vitushkin1954hilbert [Vit54] Vitushkin, A. G. 'On Hilbert's thirteenth problem'. In: *Doklady Akademii Nauk SSSR* (1954). (in Russian).

Wah2021_PinneikEikonalSolution_HagWHA [Wah+21] Waheed, U. b. et al. 'PINNeik: Eikonal solution using physics-informed neural networks'. en. In: *Computers & Geosciences* vol. 155 (Oct. 2021), p. 104833. DOI: 10.1016/j.cageo.2021.104833. URL: https://www.sciencedirect.com/science/article/pii/S009830042100131X (visited on 03/11/2021).

wang2022random [Wan+22] Wang, S. et al. 'Random weight factorization improves the training of continuous neural representations'. In: *arXiv preprint arXiv:2210.01274* (2022).

wang2023expert [Wan+23] Wang, S. et al. 'An expert's guide to training physics-informed neural networks'. In: *arXiv preprint arXiv:2308.08468* (2023).

wang2024physics [Wan+24] Wang, F. et al. 'Physics-informed neural network for lithium-ion battery degradation stable modeling and prognosis'. In: *Nature Communications* vol. 15, no. 1 (2024), p. 4332.

wang2025kolmogorov [Wan+25] Wang, Y. et al. 'Kolmogorov–Arnold-Informed neural network: A physics-informed deep learning framework for solving forward and inverse problems based on Kolmogorov–Arnold Networks'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 433 (2025), p. 117518.

whytock2014dynamic [WBR14] Whytock, T., Belyaev, A. and Robertson, N. M. 'Dynamic distance-based shape features for gait recognition'. In: *Journal of Mathematical Imaging and Vision* vol. 50, no. 3 (2014), pp. 314–326.

Wan2021_TheoryGuidedAuto_ChaWCZ [WCZ21a] Wang, N., Chang, H. and Zhang, D. 'Theory-guided Auto-Encoder for surrogate construction and inverse modeling'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 385 (2021), p. 114037. DOI: 10.1016/j.cma.2021.114037. URL: https://www.sciencedirect.com/science/article/pii/S0045782521003686.

wang2021survey [WCZ21b] Wang, X., Chen, Y. and Zhu, W. 'A survey on curriculum learning'. In: *IEEE transactions on pattern analysis and machine intelligence* vol. 44, no. 9 (2021), pp. 4555–4576.

Weng2022 [Wen22] Weng, L. 'Some Math behind Neural Tangent Kernel'. In: *Lil'Log* (Sept. 2022). URL: https://lilianweng.github.io/posts/2022-09-08-ntk/.

westphal2025generalized [WHM25] Westphal, C., Hailes, S. and Musolesi, M. 'A generalized information bottleneck theory of deep learning'. In: *arXiv preprint arXiv:2509.26327* (2025).

Wolchover2017 [Wol17] Wolchover, N. *New Theory Cracks Open the Black Box of Deep Learning*. Blog Post. 2017. URL: https://www.quantamagazine.org/new-theory-cracks-open-the-black-box-of-deep-learning-20170921/.

wang2022respecting [WSP22] Wang, S., Sankaran, S. and Perdikaris, P. 'Respecting causality is all you need for training physics-informed neural networks'. In: *arXiv preprint arXiv:2203.07404* (2022).

wang2021understanding [WTP21] Wang, S., Teng, Y. and Perdikaris, P. 'Understanding and mitigating gradient flow pathologies in physics-informed neural networks'. In: *SIAM Journal on Scientific Computing* vol. 43, no. 5 (2021), A3055–A3081.

wang2021learning [WWP21a] Wang, S., Wang, H. and Perdikaris, P. 'Learning the solution operator of parametric partial differential equations with physics-informed DeepONets'. In: *Science advances* vol. 7, no. 40 (2021), eabi8605.

wang2021eigenvector [WWP21b] Wang, S., Wang, H. and Perdikaris, P. 'On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks'. In: *Computer Methods in Applied Mechanics and Engineering* vol. 384 (2021), p. 113938.

wang2022and [WYP22] Wang, S., Yu, X. and Perdikaris, P. 'When and why PINNs fail to train: A neural tangent kernel perspective'. In: *Journal of Computational Physics* vol. 449 (2022), p. 110768.

xu2019frequency [Xu+19] Xu, Z.-Q. J. et al. 'Frequency principle: Fourier analysis sheds light on deep neural networks'. In: *arXiv preprint arXiv:1901.06523* (2019).

xu2019training [XZX19] Xu, Z.-Q. J., Zhang, Y. and Xiao, Y. 'Training behavior of deep neural network in frequency domain'. In: *International Conference on Neural Information Processing*. Springer. 2019, pp. 264–274.

Yan2021_BPinnsBayesian_MenYMK [YMK21] Yang, L., Meng, X. and Karniadakis, G. E. 'B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data'. en. In: *Journal of Computational Physics* vol. 425 (Jan. 2021), p. 109913. DOI: 10.1016/j.jcp.2020.109913. URL: https://www.sciencedirect.com/science/article/pii/S0021999120306872.

yu2018deep [Yu+18] Yu, B. et al. 'The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems'. In: *Communications in Mathematics and Statistics* vol. 6, no. 1 (2018), pp. 1–12.

Yuc2021_HybridPhysicsInformed_ViaYV [YV21] Yucesan, Y. A. and Viana, F. A. C. 'Hybrid physics-informed neural networks for main bearing fatigue prognosis with visual grease inspection'. en. In: *Computers in Industry* vol. 125 (Feb. 2021), p. 103386. DOI: 10.1016/j.compind.2020.103386. URL: https://www.sciencedirect.com/science/article/pii/S0166361520306205.

yu2024kan [YYW24] Yu, R., Yu, W. and Wang, X. 'Kan or mlp: A fairer comparison'. In: *arXiv preprint arXiv:2407.16674* (2024).

Yan2020_PhysicsInformedGenerative_ZhaYZK [YZK20] Yang, L., Zhang, D. and Karniadakis, G. E. 'Physics-informed generative adversarial networks for stochastic differential equations'. In: *SIAM Journal on Scientific Computing* vol. 42, no. 1 (2020), A292–A317. DOI: 10.1137/18M1225409. eprint: https://doi.org/10.1137/18M1225409. URL: https://doi.org/10.1137/18M1225409.

zhao2023pinnsformer [ZDP23] Zhao, Z., Ding, X. and Prakash, B. A. 'Pinnsformer: A transformer-based framework for physics-informed neural networks'. In: *arXiv preprint arXiv:2307.11833* (2023).

zhang2025mixture [Zha+25a] Zhang, D. et al. 'Mixture of experts in large language models'. In: *arXiv preprint arXiv:2507.11181* (2025).

zhang2025pinnsNOs [Zha+25b] Zhang, Z. et al. 'Physics-Informed Neural Networks and Neural Operators for Parametric PDEs: A Human-AI Collaborative Analysis'. In: *arXiv preprint arXiv:2511.04576* (2025).

ZhangGithubReview [Zha22] Zhang, C. *Neural PDE Solver: Applications*. https://github.com/bitzhangcy/Neural-PDE-Solver. GitHub. 2022.

GitHub2026LivingReview [Zha26] Zhang, C. *Neural PDE Solver - Living Review*. GitHub. 2026. URL: https://github.com/bitzhangcy/Neural-PDE-Solver.

Zhu2019_PhysicsConstrainedDeep_ZabZZKP [Zhu+19] Zhu, Y. et al. 'Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data'. In: *Journal of Computational Physics* vol. 394 (2019), pp. 56–81. DOI: https://doi.org/10.1016/j.jcp.2019.05.024. URL: https://www.sciencedirect.com/science/article/pii/S0021999119303559.

Zha2020_PhysicsInformedMulti_LiuZLS [ZLS20] Zhang, R., Liu, Y. and Sun, H. 'Physics-informed multi-LSTM networks for metamodeling of nonlinear structures'. en. In: *Computer Methods in Applied Mechanics and Engineering* vol. 369 (Sept. 2020), p. 113226. DOI: 10.1016/j.cma.2020.113226. URL: https://www.sciencedirect.com/science/article/pii/S0045782520304114.

Zhu2021_MachineLearningMetal_LiuZLY

[ZLY21] Zhu, Q., Liu, Z. and Yan, J. 'Machine learning for metal additive manufacturing: predicting temperature and melt pool fluid dynamics using physics-informed neural networks'. en. In: *Computational Mechanics* vol. 67, no. 2 (Feb. 2021), pp. 619–635. DOI: 10.1007/s00466-020-01952-9. URL: https://doi.org/10.1007/s00466-020-01952-9.