Solving differential equations using neural networks

Emmanuel Skoufris

July 29, 2024

Introduction

Partial differential equations (PDEs) play a central role in science and engineering. Traditionally, analytic techniques or numerical techniques are used to solve PDEs. Neither approach however is without drawbacks. Analytic techniques have thus far only yielded solutions for a very small class of PDEs. Numerical techniques allow for the estimation of a system at discrete points of a domain, and find wide use in applications. Nonetheless, numerical techniques can be slow, and need to be reemployed for every different instance of a PDE. Neither approach therefore has thus far supplied an adaptive, let alone universal, and PDE solver, one that can solve near instantaneously a PDE over a broad class of domains, differential operators, and boundary and initial conditions.

Deep neural networks could provide such a solver. They are known for their ability to approximate functions, as a result of numerous Universal Approximation Theorems, and have been applied to a diverse range of problems. Promising work applying deep learning techniques to solving PDEs suggests that neural networks are a viable approach to constructing more robust PDE solvers.

This project examines the burgeoning area of solving differential equations using neural networks and summarises research and experimentation conducted over the course of one month in this area, and suggests some extensions of current work in the literature. In addition, some of the ideas discussed herein were personally implemented[1] and are included in this report.

Related work

Throughout this project, research was conducted on the current state of this field. The dominant frameworks used in applications are physics-informed neural networks, and neural operators. Both ideas will be explored in this report. The code personally developed implements physics-informed neural networks.

Physics-informed neural network

Automatic differentiation provides an efficient way to differentiate neural networks with respect to either the parameters of the network or its inputs. Whilst differentiation of a

[1]The github repository is to be found at `https://github.com/ESkoufris/PhysicsInformedMachineLearning-WinterResearchProject.git`

neural network with respect to its parameters enables gradient descent, the key approach to optimizing a neural network's parameters during training, differentiation of a neural network with respect to its inputs makes possible the action of differential operators on neural networks. Universal approximation and automatic differentiation thus enabled the development of a new, data-efficient method for approximating PDE solutions using neural networks, termed *physics-informed neural networks* (PINNs). PINNs were first formulated by Raissi et al. [7], and the following approach is based on their pivotal work.

Problem setup

Consider the general problem of solving

$$\begin{cases} u_t(x,t) + \mathcal{N}[u](x,t) = 0, & (x,t) \in \mathcal{D} \times (0,T) \\ u(x,t) = g(x,t), & (x,t) \in \partial\mathcal{D} \times (0,T) \\ u(x,0) = f(x), & x \in \mathcal{D} \cup \partial\mathcal{D}, \end{cases}$$

where $\mathcal{N}$ is an operator (linear or non-linear), and $\mathcal{D} \subseteq \mathbb{R}^n$ is some bounded domain. Although theoretically the initial conditions can be incorporated into the boundary conditions, the distinction between these is made to emphasize that the PINN approach can be used to solve dynamic differential equations.

Let u_Θ be a multi-layer perceptron (MLP), with parameters given by Θ. Let $\{(x_b^i, t_b^i, u_b^i)\}_{i=1}^{N_b}$ and $\{(x_0^i, t_0^i, u_0^i)\}_{i=1}^{N_0}$ be the known boundary and initial state data, respectively, obtained either through a predefined function or from physical observation. Let $P = \{(x_p^i, t_p^i)\}_{i=1}^{N_p}$ be a set of *collocation points*. We define the *residual* of the network at (x,t) by

$$r_\Theta(x,t) = \frac{\partial u_\Theta(x,t)}{\partial t} + \mathcal{N}[u_\Theta](x,t).$$

This is computed via automatic differentiation of the network with respect to the input point (x,t). The total loss of the network u_Θ is given by

$$L_\Theta = \underbrace{\frac{1}{N_b}\sum_{i=1}^{N_b}\left[u_\Theta(x_b^i, t_b^i) - u_b^i\right]^2}_{\text{boundary loss}} + \underbrace{\frac{1}{N_0}\sum_{i=1}^{N_0}\left[u_\Theta(x_0^i, t_0^i) - u_0^i\right]^2}_{\text{initial loss}} + \underbrace{\frac{1}{N_p}\sum_{i=1}^{N_p} r_\Theta(x_p^i, t_p^i)^2}_{\text{physics loss}}.$$

Adding the physics loss can be seen as regularising the solution so that it obeys known physical laws. It also allows for a data-efficient approach to solving a PDE, since only collocation data are needed to compute it, whilst the boundary and initial losses ensure that the network conforms to physical observations. The loss functional is minimised in a standard way using gradient descent methods.

Extensions

To enhance the basic PINN, one could allow the differential operator to be an input for the network. For example, let

$$\Lambda = \text{span}\{\mathcal{L}_1, \ldots, \mathcal{L}_n\},$$

where $\mathcal{L}_1, \ldots, \mathcal{L}_n$ are linearly independent differential operators. In this case, we can represent any $\mathcal{L} \in \Lambda$ as a vector $a = (a_1, \ldots, a_n)$ representing the basis coefficients of

$\mathcal{L}$. Then, adjusting the parameters of the input layer as needed, the input to the PINN would be the concatenation (x, t, a), where a is sampled uniformly from some hypercube.

The PINN would thus be trained to solve

$$\begin{cases} (a_1\mathcal{L}_1 + \cdots + a_n\mathcal{L}_n)[u](x,t) = 0, & (x,t) \in \mathcal{D} \times (0,T) \\ u(x,t) = g(x,t), & (x,t) \in \partial\mathcal{D} \times (0,T) \\ u(x,0) = f(x), & x \in \mathcal{D} \cup \partial\mathcal{D}, \end{cases}$$

Ideally, one would devise more sophisticated architectures to deal with basis coefficients, but exploring the use of a basic MLP in this extension would perhaps be instructive as a baseline. In the repository, this idea is implemented for the one-dimensional heat equation. Specifically, an MLP is trained to solve

$$\begin{cases} u_t(x,t) - c^2 u_{xx}(x,t) = 0, & (x,t) \in (0,1) \times (0,T) \\ u(x,t) = 0, & (x,t) \in \partial[0,1] \times (0,T) \\ u(x,0) = \sin(x), & x \in [0,1], \end{cases} \tag{1}$$

for $c \in [1, 5]$. The MLP is composed of 4 hidden layers with 100 hidden units per layer. The input dimension for the input layer is 4, whilst the output dimension of the network is 1. In addition, tanh activation is used to ensure a non-vanishing second-derivative almost everywhere, owing to the form of the heat equation. The approximated solution to (1) is displayed in Figure 1 for two different values of c.

Neural operators

To improve upon traditional methods and PINNs, it would be desirable to have a solver that is trained to solve a PDE problem for a range of boundary conditions.

This poses a problem for traditional neural networks, as the space of boundary/initial conditions is *infinite* dimensional, so the solver would need to learn a map from some infinite dimensional space to another - namely, the space of possible solutions.

Problem setup

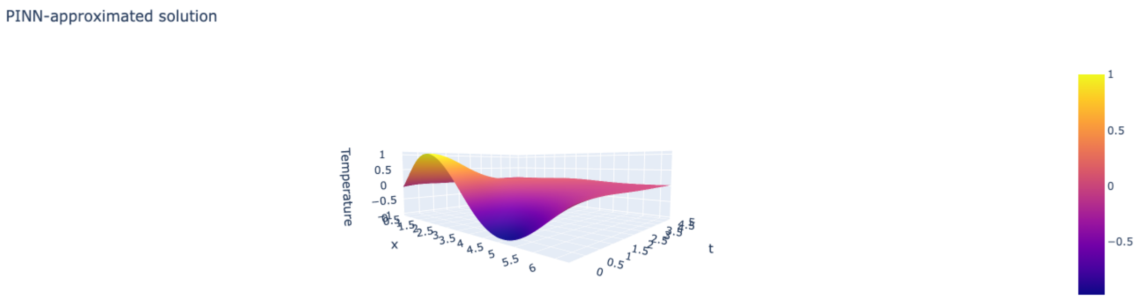
The following setup is based on [3]. Consider the PDE

$$\begin{cases} (\mathcal{L}_a u)(x) = f(x), & x \in \mathcal{D} \\ u(x) = 0, & x \in \partial\mathcal{D}, \end{cases}$$

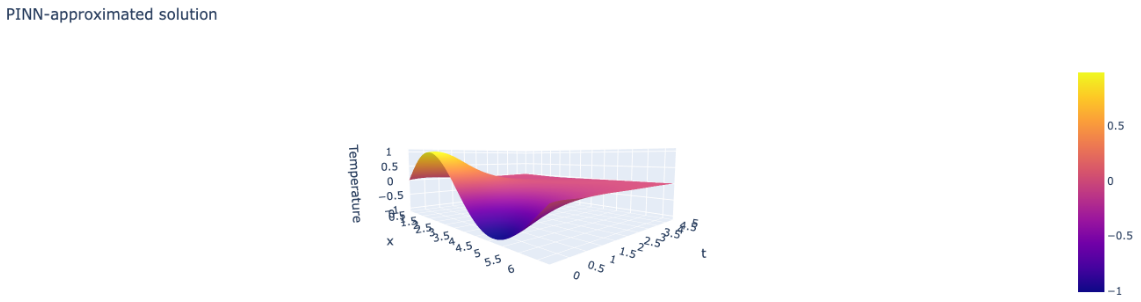
where $a \in L^2(\mathcal{D}_0; \mathbf{R}^{d_a})$ and $u \in L^2(\mathcal{D}_0; \mathbf{R}^{d_u})$, and $\mathcal{D}_0$ is a bounded domain (open, connected) subset of $\mathbf{R}^n$. Here, a is some parametric function describing some physical property of the system, whilst u is the solution. As an example, one can consider the 2D Darcy Flow equation

$$\begin{cases} -\nabla \cdot (a(x)\nabla u(x)) = f(x), & x \in (0,1)^2 \\ u(x) = 0, & x \in \partial(0,1)^2, \end{cases}$$

where a is the diffusion coefficient.



(a) PINN approximated solution to (1) with $c = 1$.



(b) PINN approximated solution to (1) with $c = 3$.

Figure 1: PINN approximated solution to (1)

For each $x, y \in \mathcal{D}_0$, let $G_a(x,y)$ be the *Green's function* satisfying

$$(\mathcal{L}_a G_a)(x,y) = \delta_x(y) \cong \begin{cases} 0, \; y \neq x \\ \infty, \; y = x, \end{cases}$$

assuming of course that such a G_a exists. Under mild conditions on $\mathcal{L}_a$, the solution is given by $u(x) = \int_{\mathcal{D}_0} G_a(x,y) f(y) dy$, since

$$(\mathcal{L}_a u)(x) = \int_{\mathcal{D}_0} (\mathcal{L}_a G_a)(x,y) f(y) dy = \int_{\mathcal{D}_0} \delta_x(y) f(y) dy = f(x).$$

One can similarly check that the boundary condition is satisfied.

Inspired by this, [3] builds a neural network to approximate the solution operator $\mathcal{G} : L^2(\mathcal{D}_0, \mathbf{R}^{d_a}) \to L^2(\mathcal{D}_0, \mathbf{R}^{d_u})$ mapping between two *infinite* dimensional vector (Hilbert) spaces. If such a map can be successfully learned over a range of input functions a, fully retraining a model for each different set of parameters/parameter functions will not be necessary, in contrast to traditional numerical solvers.

We approximate $\mathcal{G}$ using a parametric function $\mathcal{G}_\theta : L^2(\mathcal{D}_0, \mathbf{R}^{d_a}) \to L^2(\mathcal{D}_0, \mathbf{R}^{d_u})$ for a vector of parameters $\theta \in \mathbf{R}^p$. The architecture is given by

$$\mathcal{G}_\theta = Q \circ \sigma(W_L + K_L + b_L) \circ \cdots \circ \sigma(W_1 + K_1 + b_1) \circ P,$$

where $P \in \mathbf{R}^{d_0 \times d_a}$, $Q \in \mathbf{R}^{d_u \times d_L}$ are lifting and projection operators, respectively; $b_\ell : \mathcal{D}_\ell \to \mathbf{R}^{d_{\ell+1}}$ for $\ell = 1, \ldots, L$ are the bias functions; $W_\ell \in \mathbf{R}^{d_{\ell+1} \times d_\ell}$ for $\ell = 1, \ldots, L-1$; and σ is the activation function, acting pointwise on the inputs.

Given $a \in L^2(\mathcal{D}_0, \mathbf{R}^{d_a})$, we let $v_0 = Pa$, where $v_0(x) = (Pa)(x) = Pa(x)$, and Q works similarly. We thereby get a sequence of functions

$$v_0 \mapsto v_1 \mapsto \cdots \mapsto v_L \mapsto Qv_L = \hat{u},$$

with $v_\ell : \mathcal{D}_\ell \to \mathbf{R}^{d_{\ell+1}}$ for $\ell = 0, 1, \ldots, L-1$, defined recursively by

$$v_\ell(x) = \sigma\left[W_\ell v_{\ell-1}(\Pi_\ell(x)) + (K_\ell v_{\ell-1})(x) + b_\ell(x)\right], \, x \in D_\ell,$$

where $\Pi_\ell : \mathcal{D}_\ell \to \mathcal{D}_{\ell-1}$ are fixed mappings. $K_\ell : L^2(\mathcal{D}_{\ell-1}; \mathbf{R}^{d_\ell}) \to L^2(D_\ell; \mathbf{R}^{d_{\ell+1}})$ is a *linear integral kernel operator*, defined by

$$(K_\ell v_{\ell-1})(x) = \int_{\mathcal{D}_{\ell-1}} \kappa_\ell(x,y) v_{\ell-1}(y) \, dy \quad \text{for } x \in \mathcal{D}_\ell, \tag{2}$$

where $\kappa_\ell : \mathcal{D}_\ell \times \mathcal{D}_{\ell-1} \to \mathbf{R}^{d_{\ell+1} \times d_\ell}$ is continuous. The basic neural operator architecture is displayed in Figure 2. There are multiple ways to estimate the kernels κ_ℓ, as described in the following sections.

Importantly, it is also possible for the neural operator to map between the space of parametric functions to the space of solutions, by simply replacing the above formulation of the input function space with the space of boundary conditions. Therefore, the neural operator is a solver that can account either for a variety of different parametric functions in the PDE or for different boundary conditions, which makes it a versatile tool for solving PDEs.

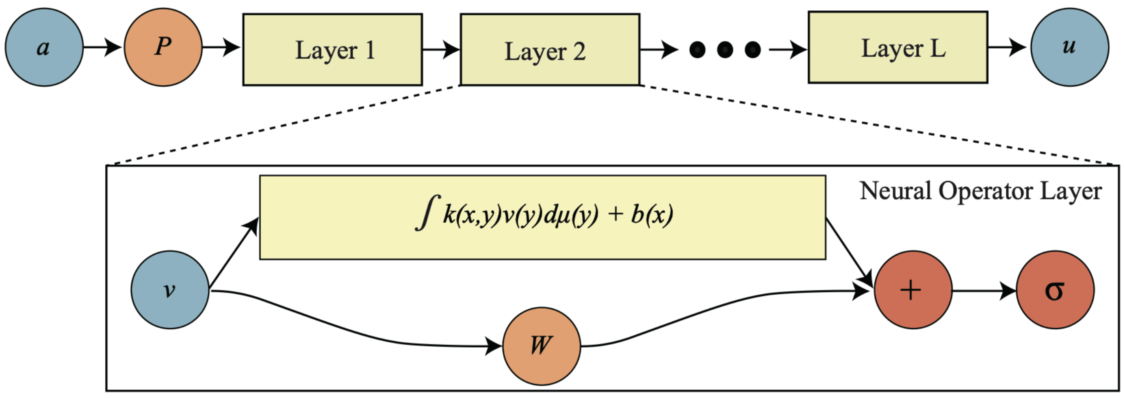


Figure 2: Neural operator architecture [3].

Graph neural operator

One way to estimate the kernel in (2) is to use a graph neural network (GNN), as is done in [2]. To simplify calculations, for each $x \in \mathcal{D}_\ell$, we reduce the integral to be defined over a smaller ball $B_r(x)$:

$$(K_\ell v_{\ell-1})(x) = \int_{B_r(x)} \kappa_\ell(x,y) v_{\ell-1}(y) dy,\ x \in \mathcal{D}_\ell$$

for some uniform $r > 0$. Given N-point discretizations of $\mathcal{D}_\ell$ for each ℓ, for every x and y in these discretizations, we have $\kappa_\ell(x,y) \in \mathbf{R}^{d_{\ell+1} \times d_\ell}$. Therefore, in discrete form, $\kappa_{\ell-1} \in \mathbf{R}^{Nd_{\ell+1} \times Nd_\ell}$, i.e. $\kappa_{\ell-1}$ is a $N \times N$ block matrix, the entries of which are to be estimated. To ensure that the GNO is discretization invariant, the matrix entries are kept constant across the N^2 blocks of $\kappa_{\ell-1}$.

We then form a graph G that has nodes given by the selected discrete points of the domain, with vertex features $v_{\ell-1} \in \mathbf{R}^{N \times d_{\ell+1}}$, and edge weights $e(x,y) = (x, y, a(x), a(y)) \in \mathbf{R}^{N \times (d_\ell + d_{\ell-1} + 2d_a)}$. The neighbourhoods of each node are given by $\mathcal{N}(x) = B_r(x) \cap (\text{discretization})$. Then

$$(K_\ell v_{\ell-1})(x) = \sum_{y \in \mathcal{N}(x)} \kappa_\ell(x,y) v_{\ell-1}(y) \mu(y).$$

Thus, the GNO's layers are message passing GNNs [1] with average aggregation:

$$v_\ell(x) = \sigma\left(W v_{\ell-1}(x) + \sum_{y \in \mathcal{N}(x)} \kappa_\ell(x,y) v_{\ell-1}(y) \mu(y) \right), \ell = 1, 2, \ldots, L.$$

Fourier neural operator

Another way to estimate the kernels κ_ℓ is to do so in Fourier space, which yields the Fourier neural operator (FNO), as proposed in [4]. This allows for the application of the Fast Fourier Transform (FFT) algorithm. Moreover, imposing that K_ℓ is a convolution operator, so that

$$\kappa_\ell(x,y) = \kappa(x-y),$$

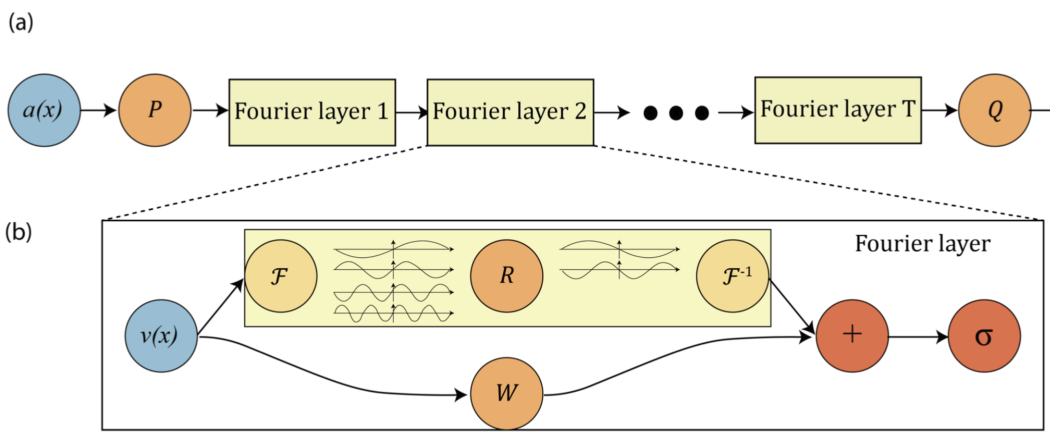


Figure 3: FNO architecture [4].

for some function κ, allows for the convolution theorem to be used.

For a function $f : \mathbf{R}^n \to \mathbf{R}^k$, its Fourier transform is given by

$$\mathcal{F}\{f\}(\boldsymbol{\xi}) = \int_{\mathbf{R}^n} f(\mathbf{x}) e^{-2\pi i \boldsymbol{\xi}\cdot\mathbf{x}} d\mathbf{x}.$$

The inverse Fourier transform is given by

$$\mathcal{F}^{-1}\{f\}(\boldsymbol{\xi}) = \int_{\mathbf{R}^n} f(\mathbf{x}) e^{2\pi i \boldsymbol{\xi}\cdot\mathbf{x}} d\mathbf{x}.$$

There also holds the convolution theorem:

$$\mathcal{F}\{f * g\}(\boldsymbol{\xi}) = \mathcal{F}\{f\}(\boldsymbol{\xi}) \cdot \mathcal{F}\{g\}(\boldsymbol{\xi}),$$

where $*$ and $\cdot$ (convolution and multiplication respectively) are performed component-wise.

Thus, for the Fourier neural operator, by applying the convolution theorem, the integral operator is given by

$$(K_\ell v_{\ell-1})(x) = \mathcal{F}^{-1}(\mathcal{F}\{\kappa_\ell\}\mathcal{F}\{v_{\ell-1}\})(x) = \mathcal{F}^{-1}(R_\phi \cdot \mathcal{F}\{v_{\ell-1}\})(x),$$

where R_ϕ is now the object to be estimated. The FNO architecture is depicted in Figure 3.

These operations have been defined abstractly, without regard to how to actually compute them. In practice, the discrete Fourier transform (DFT) is used:

$$\mathbf{X}_{k_1,\ldots,k_d} = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} \mathbf{x}_{n_1,\ldots,n_d} \exp\left(-2\pi\left(\frac{n_1 k_1}{N_1} + \cdots + \frac{n_d k_d}{N_d}\right)\right),$$

where $\{\mathbf{x}_\mathbf{k}\}_{\mathbf{k}\in I}$ is a sequence of vectors multi-indexed by $\mathbf{k}$. Let the domain $\mathcal{D}_0$ be discretized with n points, so that $v_{\ell-1} \in \mathbf{R}^{n\times d_\ell}$, and $\mathcal{F}\{v_{\ell-1}\} \in \mathbf{C}^{n\times d_\ell}$, where $\mathcal{F}\{v_{\ell-1}\}$ is the DFT at n points of the domain - the result is n vectors of length d_ℓ. We assume that κ_ℓ is periodic, so that it admits a Fourier expansion, with integer frequencies for each component. Given some integer $k_{\max}$, we let

$$N = |\{k \in \mathbb{Z}^{d_\ell} : \|k\|_\infty \leq k_{\max}\}|.$$

Usually, in applying the discrete Fourier transform, the low-frequency modes are determined by the ℓ_1-norm, but the ℓ_∞-norm is used in practice for computational efficiency.

For each mode $\mathbf{k} \in \mathcal{D}_\ell$, we have $R(\mathbf{k}) = \mathcal{F}\{\kappa_\ell\}(\mathbf{k}) \in \mathbf{C}^{d_{\ell+1} \times d_\ell}$, so $R \in \mathbf{C}^{n \times d_{\ell+1} \times d_\ell}$. We truncate the higher modes to get $R \in \mathbf{C}^{N \times d_{\ell+1} \times d_\ell}$. Since we convolve $v_{\ell-1}$ with a function with only the lower frequencies, we truncate the higher modes to get $\mathcal{F}\{v_{\ell-1}\} \in \mathbf{C}^{N \times d_\ell}$. We then get that

$$(R \cdot \mathcal{F}\{v_{\ell-1}\})_{k,l} = \sum_{j=1}^{d_\ell} R_{k,l,j} (\mathcal{F}\{v_\ell\})_{k,j},$$

where $k = 1, \dots, N, l = 1, \dots, d_{\ell+1}$. The entries of R are the numbers estimated during training. When computing the DFTs, the FFT is used, although this means that a uniform grid must be used as the input discretization.

Training a neural operator

Given a neural operator u_Θ, the goal during training is to attain

$$\begin{aligned}\min_{\theta \in \mathbf{R}^p} \mathbb{E}_{a \sim \mu} C[\mathcal{G}_\theta(a), \mathcal{G}(a)] &\approx \min_{\theta \in \mathbf{R}^p} \frac{1}{N} \sum_{j=1}^{N} \|\mathcal{G}_\theta(a^j) - \mathcal{G}(a^j)\|^2_{L^2(\mathcal{D}_0; \mathbf{R}^{d_u})} \\ &= \min_{\theta \in \mathbf{R}^p} \frac{1}{N} \sum_{j=1}^{N} \int_{\mathcal{D}_0} |\mathcal{G}_\theta(a^j)(x) - \mathcal{G}(a^j)(x)| \, dx.\end{aligned}$$

These integrals are evaluated numerically. The functions a^j are sampled from some distribution μ, supported on a compact subset of $L^2(\mathcal{D}_0; \mathbf{R}^{d_a})$. In the original formulation of the neural operator, the numbers $\mathcal{G}(a^j)(x)$ are obtained via numerical approximation, which hampers the efficiency of this method.

Physics informed neural operator

To circumvent numerical approximation of PDE solutions during training, the physics-informed neural operator combines the architecture of a neural operator, containing either an FNO or GNO kernel, with the physics-informed training process of the PINN. The idea is depicted in Figure 4. This allows us to avoid numerical estimation of the solutions, and instead rely on the PDE residuals for training.

Geometry-informed neural networks

The geometry-informed neural operator (GINO) extends the neural operator by coupling the space of boundary conditions with the space of geometries over some domain, allowing the network to learn how the geometry over which a PDE is considered determines the solution.

Problem setup

The following setup is based on [5]. Feeding different geometries into the GINO requires encapsulating the geometry via a so-called *distance function*. To see this, consider some

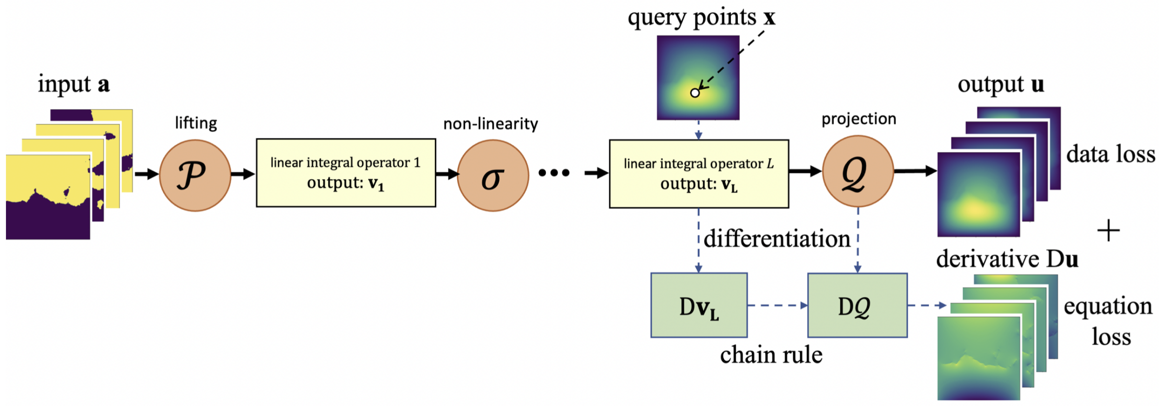


Figure 4: Physics-informed neural operator [6].

Lipschitz domain[2] $\mathcal{D} \subseteq \mathbf{R}^d$, and some Banach space of real functions $\mathcal{A}$ defined on $\mathcal{D}$. We let $\mathcal{T} \subseteq \mathcal{A}$ be a subset of functions such that, for each $T \in \mathcal{T}$, the set

$$S_T = T^{-1}(\{0\}) = \{x \in \mathcal{D} : T(x) = 0\}$$

is a $(d-1)$-dimensional sub-manifold. S_T is the "surface" of interest in our PDE. One can think of S_T as the zero-contour of a real-valued function. For example, the unit sphere $\mathbb{S}^2$ is the preimage of zero under the function $(x, y, z) \mapsto x^2 + y^2 + z^2 - 1$. Thus, it is not difficult to encode different common geometries using some appropriately constrained function T.

Further assume that for each T, S_T is simply-connected, closed (compact with trivial *geometric* boundary), smooth, and that there exists some $\epsilon > 0$ such that $B_\epsilon(x) \cap \partial\mathcal{D} = \varnothing$ for every $x \in S_T$, for every $T \in \mathcal{T}$.

Let Q_T be the open volume[3] enclosed by the hypersurface S_T, so that $\partial Q_T = S_T$. Finally, define $\Omega_T = D \setminus \bar{Q}_T$, meaning that $\partial\Omega_T = \partial\mathcal{D} \cup S_T$. Let $\mathcal{L}$ be a differential operator and consider the problem

$$\begin{aligned} \mathcal{L}(u)(x) &= f(x),\ x \in \Omega_T \\ u(x) &= g(x),\ x \in \partial\Omega_T, \end{aligned}$$

for some $f \in \mathcal{F}$ and $g \in \mathcal{B}$, where $\mathcal{F}$ and $\mathcal{B}$ are Banach spaces of functions defined on $\mathbf{R}^d$. Let $\mathcal{U}$ denote a Banach space of functions on $\mathcal{D}$ and U_T a Banach space of functions on Ω_T. Assume that $\mathcal{L}$ is such that for any such triplet (T, f, g), the PDE has a unique solution. Let $\{E_T : \mathcal{U}_T \to \mathcal{U}\}$ denote a family of extension operators that are linear and bounded (i.e. continuous). The operator we wish to estimate is

$$\Psi : \mathcal{T} \times \mathcal{F} \times \mathcal{B} \to \mathcal{U},$$

where $\Psi(T, f, g) = E_T(u)$.

In the forward pass, the input geometry is discretised and a GNO acts on the grid, yielding a function defined on a uniformly spaced, latent grid. More precisely, given a set

[2]Meaning that, at each point x of $\partial\mathcal{D}$, $\mathcal{D}$ is locally the set of points located above some Lipschitz function.

[3]We also assume Q_T to be a Lipschitz domain

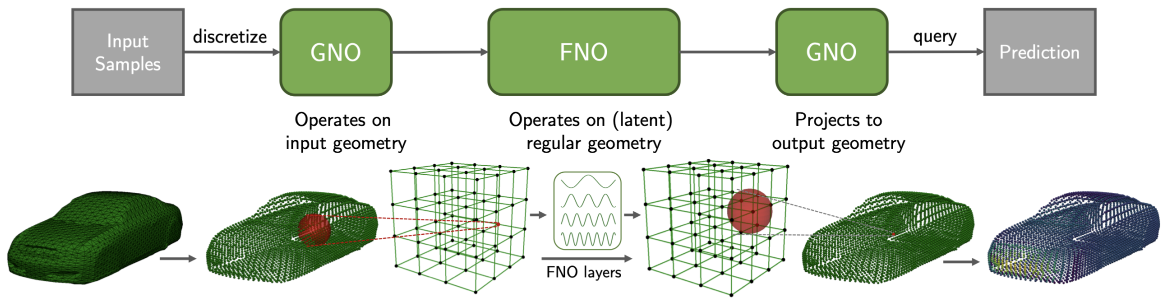

Input Samples
discretize
GNO
FNO
GNO
query
Prediction
Operates on input geometry
Operates on (latent) regular geometry
Projects to output geometry
FNO layers

Figure 5: GINO architecture [5]

of points $\{x_1^{\text{in}}, \dots, x_N^{\text{in}}\} \subseteq S_T \subseteq \mathcal{D}$, we let $\{x_1^{\text{grid}}, \dots, x_S^{\text{grid}}\} \subseteq D$ represent the latent grid, for a fixed resolution S. Then we compute:

$$v_0(x^{\text{grid}}) = \sum_{y^{\text{in}} \in B_r(x^{\text{grid}})} \kappa(x^{\text{grid}}, y^{\text{in}})\mu(y^{\text{in}}),$$

where the weights $\mu(y^{\text{in}})$ are also fine-tuned during training. This network has the same representation as a message-passing GNN as an ordinary GNO.

Figure 5 depicts the processes composing the GINO. Note that the grid forming the graph for the first GNO is in fact coincident with the input geometry, so that the open balls in the grid intersect points in the input discretization.

The function v_0, defined on a uniformly space grid, is then fed through an FNO block, resulting in a function v defined on the latent regular grid.

Finally, during the decoding phase, we randomly sample points $\{x_1^{\text{out}}, \dots, x_N^{\text{out}}\} \subseteq \Omega_T$, and compute

$$u(x^{\text{out}}) = \sum_{y^{\text{grid}} \in B_r(x^{\text{out}}) \cap \text{grid}} \kappa(x^{\text{out}}, y^{\text{grid}}) v(y^{\text{grid}}) \mu(y^{\text{grid}}),$$

where we set $\mu(y^{\text{grid}}) = 1/S$ since the grid is uniform. This yields the estimated solution u to the PDE, which is now defined on Ω_T.

Conclusion

Thus far, machine learning techniques have proven effective in solving PDEs. Although convergence to the solution is not guaranteed, owing to the stochastic nature of the training process, these methods are used in various applications to solve PDEs. The basic PINN approach is a data-efficient method in this endeavour. It nonetheless suffers in accuracy, and, at inference time, is only employed for a one instance of a PDE, at least using the original formulation of a PINN. Neural operators solve this problem by learning a map between two function spaces, enabling a more robust solver. The PINO then makes the neural operator data-efficient by incorporating the physics loss. Finally, the GINO is so far the most universal solver, and can be trained to map between geometries and boundary conditions to the solution space.
Further work in this area would ideally further universalise such solvers, by perhaps incorporating differential operators into the input space of a neural network, using either a basic PINN or neural operator approach. Theoretical aspects of these solvers would

also be important to practitioners, as guaranteed convergence to the soluton of a PDE is crucial. Thus, advancing the theory of neural networks and deep learning as a whole would promote the use and development of deep learning methods for solving PDEs, and could consequently revolutionise fields that require PDE solvers, in the same way that deep learning has transformed other areas like computer vision.

References

[1] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017. URL `https://arxiv.org/abs/1704.01212`.

[2] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Graph kernel network for partial differential equations, 2020. URL `https://arxiv.org/abs/2003.03485`.

[3] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces with applications to pdes. *arXiv preprint arXiv:2003.03485*, 2020.

[4] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations, 2021. URL `https://arxiv.org/abs/2010.08895`.

[5] Zongyi Li, Nikola Borislavov Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Prakash Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli, and Anima Anandkumar. Geometry-informed neural operator for large-scale 3d pdes, 2023. URL `https://arxiv.org/abs/2309.00583`.

[6] Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations, 2023. URL `https://arxiv.org/abs/2111.03794`.

[7] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017. URL `https://arxiv.org/abs/1711.10561`.