Physics Informed Neural Networks and Operator Learning

Davide Murari

Department of Applied Mathematics and Theoretical Physics
University of Cambridge

davidemurari.com/cism

dm2011@cam.ac.uk

Outline

Classical methods for ODEs and PDEs

Simulating ODEs

How do we solve ODEs numerically?

- Solving the initial value problem (IVP)

$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)),\ \mathbf{x}(0) = \mathbf{x}_0 \text{ notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

exactly, is in general impossible. We hence have to approximate $t \mapsto \mathbf{x}(t)$ numerically.

How do we solve ODEs numerically?

- Solving the initial value problem (IVP)

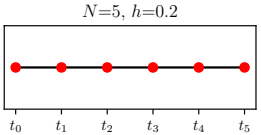
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)),\ \mathbf{x}(0) = \mathbf{x}_0 \text{ notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

exactly, is in general impossible. We hence have to approximate $t \mapsto \mathbf{x}(t)$ numerically.

- $T > 0$, $N \in \mathbb{N}$, and $h = T/N$. A one-step numerical method $\varphi^h_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ is a map

$$\mathbf{y}_{n+1} = \varphi^h_{\mathcal{F}}(\mathbf{y}_n),\ n = 0, ..., N-1, \tag{1}$$

such that $\mathbf{y}_0 = \mathbf{x}_0$ and $\mathbf{y}_n \approx \mathbf{x}(nh)$, $n = 1, ..., N$, for any (regular enough) vector field $\mathcal{F}$.



How do we solve ODEs numerically?

- Solving the initial value problem (IVP)

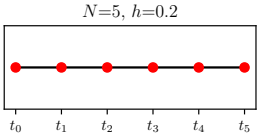
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)),\ \mathbf{x}(0) = \mathbf{x}_0 \text{ notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

exactly, is in general impossible. We hence have to approximate $t \mapsto \mathbf{x}(t)$ numerically.

- $T > 0$, $N \in \mathbb{N}$, and $h = T/N$. A one-step numerical method $\varphi^h_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ is a map

$$\mathbf{y}_{n+1} = \varphi^h_{\mathcal{F}}(\mathbf{y}_n),\ n = 0, ..., N-1, \tag{1}$$

such that $\mathbf{y}_0 = \mathbf{x}_0$ and $\mathbf{y}_n \approx \mathbf{x}(nh)$, $n = 1, ..., N$, for any (regular enough) vector field $\mathcal{F}$.



- Some methods, called implicit, to define the map $\varphi^h_{\mathcal{F}}$ in (1) need to solve a (non-linear) equation. An example is the implicit Euler method: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_{n+1})$.

Structure-preserving methods

- Some of these methods can be designed to preserve desirable properties of the solution. The area studying them is called **geometric numerical analysis**, and those methods are sometimes called structure-preserving.
- Examples are methods that preserve an energy function (such as mass or momentum in a PDE), symmetry properties, or a volume form.
- These methods are often implicit. An example is provided by the implicit midpoint method

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathcal{F}\left(\frac{\mathbf{y}_n + \mathbf{y}_{n+1}}{2}\right),$$

  which conserves all the quadratic energy functions.

Pros and Cons of these methods

- Runge–Kutta methods are a type of such schemes. These, and all the other options, are extremely well studied; they have well-understood stability, convergence, and consistency properties.

Pros and Cons of these methods

- Runge–Kutta methods are a type of such schemes. These, and all the other options, are extremely well studied; they have well-understood stability, convergence, and consistency properties.
- Five of their possible limitations are:
  1. they are sequential: to approximate the solution at $t_n = nh$, we need to apply them n times,
  2. they do not provide the value of the solution outside of the points $\{t_0, t_1, ..., t_N\}$,
  3. for some ODEs, one must use small time-steps or implicit methods to get a stable solution,
  4. to preserve some underlying property, they are generally implicit,
  5. when changing some parameters, we need to solve the equation again.
- The question is whether we can do better than they do with the help of neural networks.

Simulating PDEs

What about PDEs?

- Let us first focus on linear stationary PDEs of the form

$$\mathcal{L}(u)(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} \in \Omega \subset \mathbb{R}^d,\ u : \Omega \to \mathbb{R},\ f : \Omega \to \mathbb{R},$$

  where $\mathcal{L}$ is a differential operator. An example could be $\mathcal{L}(u)(\mathbf{x}) = \partial^2_{x_1} u(\mathbf{x}) + \partial^2_{x_2} u(\mathbf{x})$, with $d = 2$, $f(\mathbf{x}) = 0$, and $\Omega = [0, 1]^2$. This problem needs to be solved under certain boundary conditions.

What about PDEs?

- Let us first focus on linear stationary PDEs of the form

$$\mathcal{L}(u)(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} \in \Omega \subset \mathbb{R}^d,\ u : \Omega \to \mathbb{R},\ f : \Omega \to \mathbb{R},$$

  where $\mathcal{L}$ is a differential operator. An example could be $\mathcal{L}(u)(\mathbf{x}) = \partial^2_{x_1} u(\mathbf{x}) + \partial^2_{x_2} u(\mathbf{x})$, with $d = 2$, $f(\mathbf{x}) = 0$, and $\Omega = [0, 1]^2$. This problem needs to be solved under certain boundary conditions.

- To solve this problem, we can use a numerical method such as finite differences or finite elements. They are both based on considering a suitable set of points $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$ over Ω, defining a mesh.

What about PDEs?

- Let us first focus on linear stationary PDEs of the form

$$\mathcal{L}(u)(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} \in \Omega \subset \mathbb{R}^d,\ u : \Omega \to \mathbb{R},\ f : \Omega \to \mathbb{R},$$

  where $\mathcal{L}$ is a differential operator. An example could be $\mathcal{L}(u)(\mathbf{x}) = \partial^2_{x_1} u(\mathbf{x}) + \partial^2_{x_2} u(\mathbf{x})$, with $d = 2$, $f(\mathbf{x}) = 0$, and $\Omega = [0, 1]^2$. This problem needs to be solved under certain boundary conditions.

- To solve this problem, we can use a numerical method such as finite differences or finite elements. They are both based on considering a suitable set of points $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$ over Ω, defining a mesh.

- The finite differences method approximates the differential operator $\mathcal{L}$ with a matrix $L \in \mathbb{R}^{N \times N}$, and turns the problem into the linear system

$$L\mathbf{u} = \mathbf{f},\ \mathbf{u}_i \approx u(\mathbf{x}_i),\ \mathbf{f}_i = f(\mathbf{x}_i) \implies u(\mathbf{x}_i) \approx (L^{-1}\mathbf{f})_i.$$

What about PDEs?

- The finite differences method does not provide an approximation outside of the grid points. It is also challenging to implement on non-regular grids, and hence on complicated domains.

What about PDEs?

- The finite differences method does not provide an approximation outside of the grid points. It is also challenging to implement on non-regular grids, and hence on complicated domains.
- The finite element method (FEM) changes perspective, and approximates u by u^h belonging to a finite-dimensional space

$$V_h = \text{span}\{\phi_1, ..., \phi_N\}.$$

What about PDEs?

- The finite differences method does not provide an approximation outside of the grid points. It is also challenging to implement on non-regular grids, and hence on complicated domains.

- The finite element method (FEM) changes perspective, and approximates u by u^h belonging to a finite-dimensional space

$$V_h = \text{span}\{\phi_1, ..., \phi_N\}.$$

- Usually, one has $\phi_i(\mathbf{x}_j) = \delta_{ij}$ for $i, j = 1, ..., N$. Writing the approximate solution $u^h \in V_h$ as $u_h = \sum_{i=1}^N u_i^h \phi_i(\mathbf{x})$, and considering the weak formulation of the problem, we still get a linear system

$$L_h \mathbf{u}^h = \mathbf{f},$$

where, in the simplest setup, we could have

$$\mathbf{f}_i = \int_\Omega f(\mathbf{x}) \phi_i(\mathbf{x}) d\mathbf{x}, \ \mathbf{u}_i^h = u_i.$$

What about PDEs?

- The FEM provides an approximate solution on every point $x \in \Omega$. There is much more rigorous analysis than the FDM, and it is very efficiently implemented by several scientific computing libraries.

What about PDEs?

- The FEM provides an approximate solution on every point $x \in \Omega$. There is much more rigorous analysis than the FDM, and it is very efficiently implemented by several scientific computing libraries.
- Both approaches (FD and FE) extend to time-dependent problems $\partial_t u + \mathcal{L}u = f$, where one has to solve a linear ODE rather than a linear system:

$$\frac{d}{dt}\mathbf{u} = -L\mathbf{u} + \mathbf{f}.$$

What about PDEs?

- The FEM provides an approximate solution on every point $x \in \Omega$. There is much more rigorous analysis than the FDM, and it is very efficiently implemented by several scientific computing libraries.
- Both approaches (FD and FE) extend to time-dependent problems $\partial_t u + \mathcal{L}u = f$, where one has to solve a linear ODE rather than a linear system:

$$\frac{d}{dt}\mathbf{u} = -L\mathbf{u} + \mathbf{f}.$$

- Both approaches have problems scaling to very high-dimensional irregular domains. Handling non-linear terms can also be a challenge.

What are we looking for?

- There is a very active field in scientific machine learning working on **building more efficient solvers** for ordinary and partial differential equations. A setup where this becomes extremely important is for parametric ODEs/PDEs, such as

$$\mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  In this case, a numerical method would have to solve it for each set of parameters. Can we learn how to do it more efficiently? The same applies when the BCs are changed.

What are we looking for?

- There is a very active field in scientific machine learning working on **building more efficient solvers** for ordinary and partial differential equations. A setup where this becomes extremely important is for parametric ODEs/PDEs, such as

$$\mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  In this case, a numerical method would have to solve it for each set of parameters. Can we learn how to do it more efficiently? The same applies when the BCs are changed.

- We can distinguish two main approaches:

  1. Data-driven approaches, such as Neural Operators,
  2. Equation-driven methods, such as Physics Informed Neural Networks.

What are we looking for?

- There is a very active field in scientific machine learning working on **building more efficient solvers** for ordinary and partial differential equations. A setup where this becomes extremely important is for parametric ODEs/PDEs, such as

$$\mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  In this case, a numerical method would have to solve it for each set of parameters. Can we learn how to do it more efficiently? The same applies when the BCs are changed.

- We can distinguish two main approaches:

  1. Data-driven approaches, such as Neural Operators,
  2. Equation-driven methods, such as Physics Informed Neural Networks.

- There is a thin line between the two approaches, and a lot of hybrid strategies, together with a lot of different nomenclature, often referring to similar ideas.

What is a PINN and how is it trained?

The main idea behind Physics Informed Neural Networks (PINNs)

- Neural networks are flexible parametric functions which allow for approximating large classes of functions. They should thus be able to approximate the solution of a differential equation as well.

The main idea behind Physics Informed Neural Networks (PINNs)

- Neural networks are flexible parametric functions which allow for approximating large classes of functions. They should thus be able to approximate the solution of a differential equation as well.
- The idea is then to consider an expressive-enough network $\mathcal{N}_\theta$, and train it so that it approximately solves the differential equation at enough points in the domain.

The main idea behind Physics Informed Neural Networks (PINNs)

- Neural networks are flexible parametric functions which allow for approximating large classes of functions. They should thus be able to approximate the solution of a differential equation as well.
- The idea is then to consider an expressive-enough network $\mathcal{N}_\theta$, and train it so that it approximately solves the differential equation at enough points in the domain.
- For example, if we want to solve $\partial_t u = \mathcal{L}u$ over $(t, x) \in [0, T] \times \Omega$, $\Omega \subset \mathbb{R}^d$, we can define $\mathcal{N}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$ and train it so it almost satisfies the initial/boundary conditions and

$$\partial_t \mathcal{N}_\theta(t_i, \mathbf{x}_i) \approx \mathcal{L}(\mathcal{N}_\theta)(t_i, \mathbf{x}_i), \ i = 1, ..., N, \ t_i \in [0, T], \ \mathbf{x}_i \in \Omega.$$

  The derivatives in $\mathcal{L}(\mathcal{N}_\theta)$ and $\partial_t \mathcal{N}_\theta$ can be computed with **automatic differentiation**.

The main idea behind Physics Informed Neural Networks (PINNs)

- Neural networks are flexible parametric functions which allow for approximating large classes of functions. They should thus be able to approximate the solution of a differential equation as well.

- The idea is then to consider an expressive-enough network $\mathcal{N}_\theta$, and train it so that it approximately solves the differential equation at enough points in the domain.

- For example, if we want to solve $\partial_t u = \mathcal{L}u$ over $(t, x) \in [0, T] \times \Omega$, $\Omega \subset \mathbb{R}^d$, we can define $\mathcal{N}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$ and train it so it almost satisfies the initial/boundary conditions and

$$\partial_t \mathcal{N}_\theta(t_i, \mathbf{x}_i) \approx \mathcal{L}(\mathcal{N}_\theta)(t_i, \mathbf{x}_i), \ i = 1, ..., N, \ t_i \in [0, T], \ \mathbf{x}_i \in \Omega.$$

  The derivatives in $\mathcal{L}(\mathcal{N}_\theta)$ and $\partial_t \mathcal{N}_\theta$ can be computed with **automatic differentiation**.

- **Remark:** If we can do so, we do not need to discretise the differential operator, and we can, in principle, also learn how the solution depends on the PDE parameters.

A remark on terminology: Physics-Based/Inspired/Constrained NNs

- Physics enters the *model class / computational graph*: hard constraints, symmetries, conservation laws, or coupling to a solver.
- Examples: HNN/LNN, symplectic & volume-preserving NODEs; equivariant CNNs/GNNs; PDE-Net; differentiable solvers with learned closures; hard-constrained layers.
- Training may be purely data-driven and/or include weak physics regularisers.

Physics-informed neural networks (PINNs)

- Let us start from PINNs trained to solve ODEs, and in particular, the initial value problem

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)) \in \mathbb{R}^d, \\ \mathbf{x}(0) = \mathbf{x}_0. \end{cases}$$

- We introduce a parametric map $\mathcal{N}_\theta(\cdot; \mathbf{x}_0) : [0, T] \to \mathbb{R}^d$, and choose its weights so that

$$\mathcal{L}(\theta) := \frac{1}{C} \sum_{c=1}^{C} \left\| \mathcal{N}'_\theta(t_c; \mathbf{x}_0) - \mathcal{F}\left(\mathcal{N}_\theta(t_c; \mathbf{x}_0)\right) \right\|_2^2 + \gamma \left\| \mathcal{N}_\theta(0; \mathbf{x}_0) - \mathbf{x}_0 \right\|_2^2 \to \min$$

  for some collocation points $t_1, \ldots, t_C \in [0, T]$.

Physics-informed neural networks (PINNs)

- Let us start from PINNs trained to solve ODEs, and in particular, the initial value problem

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)) \in \mathbb{R}^d, \\ \mathbf{x}(0) = \mathbf{x}_0. \end{cases}$$

- We introduce a parametric map $\mathcal{N}_\theta(\cdot; \mathbf{x}_0) : [0, T] \to \mathbb{R}^d$, and choose its weights so that

$$\mathcal{L}(\theta) := \frac{1}{C} \sum_{c=1}^{C} \left\| \mathcal{N}'_\theta(t_c; \mathbf{x}_0) - \mathcal{F}\left(\mathcal{N}_\theta(t_c; \mathbf{x}_0)\right) \right\|_2^2 + \gamma \left\| \mathcal{N}_\theta(0; \mathbf{x}_0) - \mathbf{x}_0 \right\|_2^2 \to \min$$

  for some collocation points $t_1, \ldots, t_C \in [0, T]$.

- Then, $t \mapsto \mathcal{N}_\theta(t; \mathbf{x}_0)$ will solve a different IVP

$$\begin{cases} \dot{\mathbf{y}}(t) = \mathcal{F}(\mathbf{y}(t)) + \left(\mathcal{N}'_\theta(t; \mathbf{x}_0) - \mathcal{F}(\mathbf{y}(t))\right) \in \mathbb{R}^d, \\ \mathbf{y}(0) = \mathcal{N}_\theta(0; \mathbf{x}_0) \in \mathbb{R}^d, \end{cases}$$

  where hopefully the residual $\mathcal{N}'_\theta(t; \mathbf{x}_0) - \mathcal{F}(\mathbf{y}(t))$ is small in some sense.

Connection with classical numerical methods: Collocation methods

Goal: Solve $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)) \in \mathbb{R}^d$ with $\mathbf{x}(0) = \mathbf{x}_0 \in \mathbb{R}^d$, for $t \in [0, \Delta t]$.

Polynomial collocation methods

Pick a set of $s \in \mathbb{N}$ collocation points $c_1, ..., c_s \in [0, 1]$ and define the degree s polynomial $p(\cdot; \mathbf{x}_0) : \mathbb{R} \to \mathbb{R}^d$,

$$p(t; \mathbf{x}_0) = \sum_{i=0}^{s} \mathbf{p}_i \varphi_i(t),$$

such that

$$\begin{aligned} p(0; \mathbf{x}_0) &= \mathbf{x}_0, \\ p'(c_i \Delta t; \mathbf{x}_0) &= \mathcal{F}(p(c_i \Delta t; \mathbf{x}_0)),\ i = 1, ..., s. \end{aligned}$$

PINN

Pick $t_1, ..., t_s \in [0, \Delta t]$ and look for $\mathcal{N}_{\theta^*}(\cdot; \mathbf{x}_0) : \mathbb{R} \to \mathbb{R}^d$

$$\mathcal{N}_{\theta^*}(t; \mathbf{x}_0) = \sum_{i=1}^{h} \mathbf{a}_i^* \sigma(b_i^* t + c_i^*),$$

such that θ^* minimises

$$\gamma \|\mathcal{N}_\theta(0; \mathbf{x}_0) - \mathbf{x}_0\|_2^2 + \sum_{i=1}^{s} \omega_i \left\| \mathcal{N}'_\theta(t_i; \mathbf{x}_0) - \mathcal{F}(\mathcal{N}_\theta(t_i, \mathbf{x}_0)) \right\|_2^2.$$

A-posteriori error estimate

Theorem: Quadrature-based a-posteriori error estimate

Let $\mathbf{x}(t)$ be the solution of the IVP

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)) \in \mathbb{R}^d, \ \mathcal{F} \in \mathcal{C}^{p+1}(\mathbb{R}^d, \mathbb{R}^d), \\ \mathbf{x}(0) = \mathbf{x}_0. \end{cases}$$

Suppose that $\mathcal{N}_\theta(\cdot; \mathbf{x}_0) : [0, \Delta t] \to \mathbb{R}^d$ is smooth and satisfies

$$\left\| \mathcal{N}'_\theta(t_c; \mathbf{x}_0) - \mathcal{F}(\mathcal{N}_\theta(t_c; \mathbf{x}_0)) \right\|_2 \leq \varepsilon, \ c = 1, \ldots, C$$

for C collocation points $0 \leq t_1 < \cdots < t_C \leq \Delta t$ defining a quadrature rule of order p. Then, there exist $\alpha, \beta > 0$ such that

$$\|\mathbf{x}(t) - \mathcal{N}_\theta(t; \mathbf{x}_0)\|_2 \leq \alpha(\Delta t)^{p+1} + \beta\varepsilon, \ t \in [0, \Delta t].$$

Imposing the initial condition

- We will see later that there are situations where we want to enforce the condition $\mathcal{N}_\theta(0; \mathbf{x}_0) = \mathbf{x}_0$ exactly for every $\mathbf{x}_0$.

Imposing the initial condition

- We will see later that there are situations where we want to enforce the condition $\mathcal{N}_\theta(0; \mathbf{x}_0) = \mathbf{x}_0$ exactly for every $\mathbf{x}_0$.
- This can be done in several ways. Two common strategies are:

$$\mathcal{N}_\theta(t; \mathbf{x}_0) = \mathbf{x}_0 + f(t)\widetilde{\mathcal{N}}_\theta(t; \mathbf{x}_0),\ f(0) = 0,\ \text{e.g. } f(t) = t,$$

$$\mathcal{N}_\theta(t; \mathbf{x}_0) = \mathbf{x}_0 + \left(\widetilde{\mathcal{N}}_\theta(t; \mathbf{x}_0) - \widetilde{\mathcal{N}}_\theta(0; \mathbf{x}_0)\right) = \widetilde{\mathcal{N}}_\theta(t; \mathbf{x}_0) + \left(\mathbf{x}_0 - \widetilde{\mathcal{N}}_\theta(0; \mathbf{x}_0)\right).$$

- The second approach is a particular example of a much more general theory, called the Theory of Functional Connections, see Mortari, "The Theory of Connections: Connecting Points".

Is solving a single IVP efficient?

- Solving a single IVP on $[0, T]$ with a neural network can take long training time.
- The obtained solution can not be used to solve the same ordinary differential equation with a different initial condition.

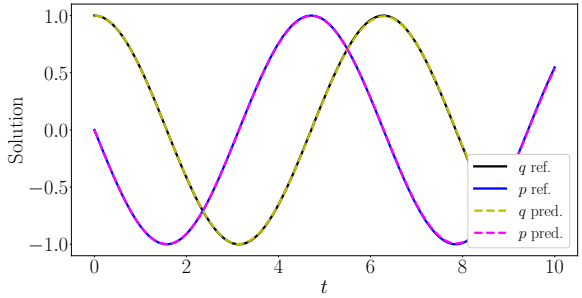


Figure 1: Solution comparison after reaching a loss value of 10^{-5}. The training time is 87 seconds (7500 epochs with 1000 new collocation points randomly sampled at each of them).

Integration over long time intervals

- It is hard to solve initial value problems over long time intervals.

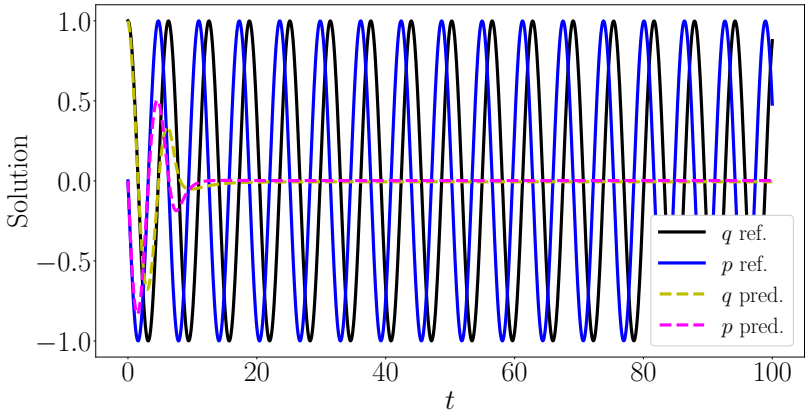


Figure 2: Solution comparison after 10000 epochs.

Forward invariant subset of the phase space

- Consider the vector field $\mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d$, and introduce notation $\phi^t_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ for the time-t flow map of $\mathcal{F}$, which for every $\mathbf{x}_0 \in \mathbb{R}^d$ satisfies

$$\begin{cases} \frac{d}{dt}\phi^t_{\mathcal{F}}(\mathbf{x}_0) = \mathcal{F}(\phi^t_{\mathcal{F}}(\mathbf{x}_0)), \\ \phi^0_{\mathcal{F}}(\mathbf{x}_0) = \mathbf{x}_0. \end{cases}$$

Forward invariant subset of the phase space

- Consider the vector field $\mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d$, and introduce notation $\phi^t_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ for the time-t flow map of $\mathcal{F}$, which for every $\mathbf{x}_0 \in \mathbb{R}^d$ satisfies

$$\begin{cases} \frac{d}{dt}\phi^t_{\mathcal{F}}(\mathbf{x}_0) = \mathcal{F}(\phi^t_{\mathcal{F}}(\mathbf{x}_0)), \\ \phi^0_{\mathcal{F}}(\mathbf{x}_0) = \mathbf{x}_0. \end{cases}$$

- Assume that there exists a set $\Omega \subset \mathbb{R}^d$ such that for every $\mathbf{x}_0 \in \Omega$, $\phi^t_{\mathcal{F}}(\mathbf{x}_0) \in \Omega$ for every $t \geq 0$. This set is then said to be **forward invariant**.

$$\phi^{n\Delta t+\delta t}_{\mathcal{F}} = \phi^{\delta t}_{\mathcal{F}} \circ \phi^{\Delta t}_{\mathcal{F}} \circ ... \circ \phi^{\Delta t}_{\mathcal{F}}, \ n \in \mathbb{N}, \ \delta t \in (0, \Delta t).$$

Forward invariant subset of the phase space

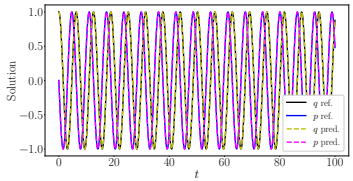
- Consider the vector field $\mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d$, and introduce notation $\phi^t_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ for the time-t flow map of $\mathcal{F}$, which for every $\mathbf{x}_0 \in \mathbb{R}^d$ satisfies

$$\begin{cases} \frac{d}{dt}\phi^t_{\mathcal{F}}(\mathbf{x}_0) = \mathcal{F}(\phi^t_{\mathcal{F}}(\mathbf{x}_0)), \\ \phi^0_{\mathcal{F}}(\mathbf{x}_0) = \mathbf{x}_0. \end{cases}$$

- Assume that there exists a set $\Omega \subset \mathbb{R}^d$ such that for every $\mathbf{x}_0 \in \Omega$, $\phi^t_{\mathcal{F}}(\mathbf{x}_0) \in \Omega$ for every $t \geq 0$. This set is then said to be **forward invariant**.

$$\phi^{n\Delta t+\delta t}_{\mathcal{F}} = \phi^{\delta t}_{\mathcal{F}} \circ \phi^{\Delta t}_{\mathcal{F}} \circ \ldots \circ \phi^{\Delta t}_{\mathcal{F}}, \ n \in \mathbb{N}, \ \delta t \in (0, \Delta t).$$

- Thus, to approximate $\phi^t_{\mathcal{F}} : \Omega \to \Omega$ for any $t \geq 0$, we only approximate it for $t \in [0, \Delta t]$.



PINNs for Hamiltonian ODEs

Canonical Hamiltonian System (recap)

- The equations of motion of canonical Hamiltonian systems write

$$\begin{cases} \frac{d}{dt}\phi_{H,t}(\mathbf{x}_0) = \mathbb{J}\nabla H(\phi_{H,t}(\mathbf{x}_0)) \in \mathbb{R}^{2n} \\ \phi_{H,0}(\mathbf{x}_0) = \mathbf{x}_0 \end{cases}, \quad \mathbb{J} = \begin{bmatrix} 0_n & I_n \\ -I_n & 0_n \end{bmatrix} \in \mathbb{R}^{2n\times 2n}.$$

Canonical Hamiltonian System (recap)

- The equations of motion of canonical Hamiltonian systems write

$$\begin{cases} \frac{d}{dt}\phi_{H,t}(\mathbf{x}_0) = \mathbb{J}\nabla H(\phi_{H,t}(\mathbf{x}_0)) \in \mathbb{R}^{2n} \\ \phi_{H,0}(\mathbf{x}_0) = \mathbf{x}_0 \end{cases}, \quad \mathbb{J} = \begin{bmatrix} 0_n & I_n \\ -I_n & 0_n \end{bmatrix} \in \mathbb{R}^{2n\times 2n}.$$

- The flow $\phi_{H,t} : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ conserves the energy:

$$\frac{d}{dt}H(\phi_{H,t}(\mathbf{x}_0)) = \nabla H(\phi_{H,t}(\mathbf{x}_0))^\top \mathbb{J}\nabla H(\phi_{H,t}(\mathbf{x}_0)) = 0,$$

- and it is symplectic:

$$\left(\frac{\partial \phi_{H,t}(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \mathbb{J} \left(\frac{\partial \phi_{H,t}(\mathbf{x})}{\partial \mathbf{x}}\right) = \mathbb{J}.$$

The SympFlow architecture[1]

- We now build a neural network that approximates $\phi_{H,t} : \Omega \to \Omega$ for a forward invariant set $\Omega \subset \mathbb{R}^{2n}$, and $t \in [0, \Delta t]$, while reproducing the qualitative properties of $\phi_{H,t}$.

[1] Priscilla Canizares et al. "Symplectic neural flows for modeling and discovery". In: *arXiv preprint arXiv:2412.16787* (2024).

The SympFlow architecture[1]

- We now build a neural network that approximates $\phi_{H,t} : \Omega \to \Omega$ for a forward invariant set $\Omega \subset \mathbb{R}^{2n}$, and $t \in [0, \Delta t]$, while reproducing the qualitative properties of $\phi_{H,t}$.
- We rely on two building blocks, which applied to $(\mathbf{q}, \mathbf{p}) \in \mathbb{R}^{2n}$ write:

$$\phi_{\mathbf{p},t}((\mathbf{q},\mathbf{p})) = \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - (\nabla_{\mathbf{q}} V(t,\mathbf{q}) - \nabla_q V(0,\mathbf{q})) \end{bmatrix}, \quad \phi_{q,t}((q,p)) = \begin{bmatrix} \mathbf{q} + (\nabla_p K(t,\mathbf{p}) - \nabla_{\mathbf{p}} K(0,\mathbf{p})) \\ \mathbf{p} \end{bmatrix}.$$

[1] Canizares et al., "Symplectic neural flows for modeling and discovery".

The SympFlow architecture[1]

- We now build a neural network that approximates $\phi_{H,t} : \Omega \to \Omega$ for a forward invariant set $\Omega \subset \mathbb{R}^{2n}$, and $t \in [0, \Delta t]$, while reproducing the qualitative properties of $\phi_{H,t}$.

- We rely on two building blocks, which applied to $(\mathbf{q}, \mathbf{p}) \in \mathbb{R}^{2n}$ write:

$$\phi_{\mathbf{p},t}((\mathbf{q},\mathbf{p})) = \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - (\nabla_{\mathbf{q}} V(t,\mathbf{q}) - \nabla_q V(0,\mathbf{q})) \end{bmatrix}, \quad \phi_{q,t}((q,p)) = \begin{bmatrix} \mathbf{q} + (\nabla_p K(t,\mathbf{p}) - \nabla_{\mathbf{p}} K(0,\mathbf{p})) \\ \mathbf{p} \end{bmatrix}.$$

- The SympFlow architecture is defined as

$$\mathcal{N}_\theta\,(t,(\mathbf{q}_0,\mathbf{p}_0)) = \phi^L_{\mathbf{p},t} \circ \phi^L_{\mathbf{q},t} \circ \cdots \circ \phi^1_{\mathbf{p},t} \circ \phi^1_{\mathbf{q},t}((\mathbf{q}_0,\mathbf{p}_0)),$$

with

$$V^i(t,\mathbf{q}) = \ell_{\theta^i_3} \circ \sigma \circ \ell_{\theta^i_2} \circ \sigma \circ \ell_{\theta^i_1} \left(\begin{bmatrix} \mathbf{q} \\ t \end{bmatrix} \right), \quad K^i(t,\mathbf{p}) = \ell_{\rho^i_3} \circ \sigma \circ \ell_{\rho^i_2} \circ \sigma \circ \ell_{\rho^i_1} \left(\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix} \right)$$

$$\ell_{\theta^i_k}(\mathbf{x}) = A^i_k x + \mathbf{a}^i_k, \quad \ell_{\rho^i_k}(\mathbf{x}) = B^i_k \mathbf{x} + \mathbf{b}^i_k, \quad k = 1,2,3, \quad i = 1,...,L.$$

[1] Canizares et al., "Symplectic neural flows for modeling and discovery".

Properties of the SympFlow

- The SympFlow is symplectic for every time $t \in \mathbb{R}$. The building blocks we compose are exact flows of time-dependent Hamiltonian systems:

$$\begin{aligned}\phi^i_{\mathbf{p},t}((\mathbf{q},\mathbf{p})) &= \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - \left(\nabla_{\mathbf{q}} V^i(t,\mathbf{q}) - \nabla_{\mathbf{q}} V^i(0,\mathbf{q})\right) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - \nabla_q \left(\int_0^t \partial_s V^i(s,\mathbf{q}) ds \right) \end{bmatrix} = \phi_{\widetilde{V}^i,t}((\mathbf{q},\mathbf{p})),\end{aligned}$$

with $\widetilde{V}^i(t,(\mathbf{q},\mathbf{p})) = \partial_t V^i(t,\mathbf{q})$.

Properties of the SympFlow

- The SympFlow is symplectic for every time $t \in \mathbb{R}$. The building blocks we compose are exact flows of time-dependent Hamiltonian systems:

$$\phi^i_{\mathbf{p},t}((\mathbf{q},\mathbf{p})) = \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - \left(\nabla_{\mathbf{q}} V^i(t,\mathbf{q}) - \nabla_{\mathbf{q}} V^i(0,\mathbf{q})\right) \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - \nabla_q \left(\int_0^t \partial_s V^i(s,\mathbf{q}) ds \right) \end{bmatrix} = \phi_{\widetilde{V}^i,t}((\mathbf{q},\mathbf{p})),$$

  with $\widetilde{V}^i(t,(\mathbf{q},\mathbf{p})) = \partial_t V^i(t,\mathbf{q})$.

- The SympFlow is the exact solution of a time-dependent Hamiltonian system.

Training the SympFlow to solve $\dot{\mathbf{x}} = \mathbb{J}\nabla H(\mathbf{x})$

- The SympFlow is based on modelling the scalar-valued potentials $\widetilde{V}^i, \widetilde{K}^i : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ with feed-forward neural networks.
- To train the overall model $\mathcal{N}_\theta$ we minimise the loss function

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left\| \frac{d}{dt} \mathcal{N}_\theta \left(t, \mathbf{x}_0^i\right) \Big|_{t=t_i} - \mathbb{J}\nabla H \left(\mathcal{N}_\theta \left(t_i, \mathbf{x}_0^i\right)\right) \right\|_2^2$$

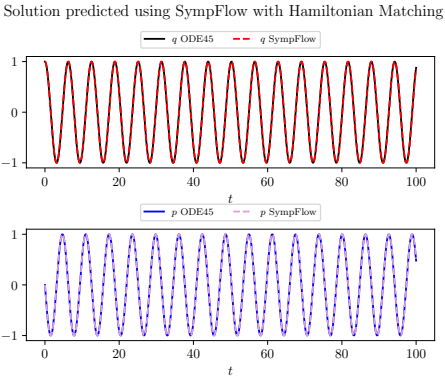
  where we sample $t_i \in [0, \Delta t]$, and $\mathbf{x}_0^i \in \Omega \subset \mathbb{R}^{2n}$.

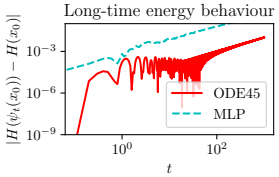
Simple Harmonic Oscillator (unsupervised)

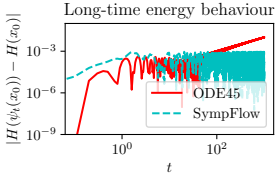
Equations of motion

$$\dot{x} = p, \;\; \dot{p} = -x.$$

Solution predicted using SympFlow with Hamiltonian Matching







Hénon–Heiles (unsupervised)

Equations of motion

$$\dot{x} = p_x,\ \dot{y} = p_y,\ \dot{p}_x = -x - 2xy,\ \dot{p}_y = -y - (x^2 - y^2).$$

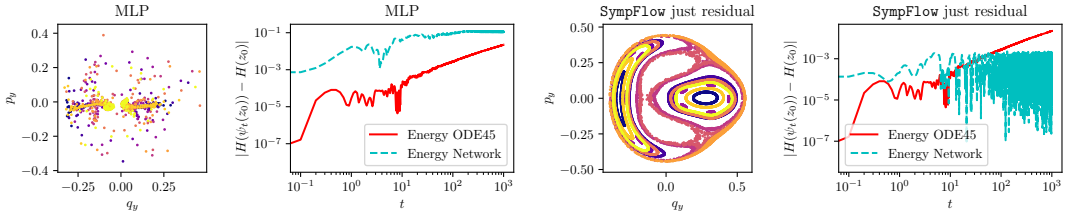


Figure 3: **Unsupervised experiment — Hénon–Heiles:** Comparison of the Poincaré sections and the energy behaviour up to time $T = 1000$.

Extension to PDEs

Extension to PDEs

$$\begin{cases} \partial_t u(t,\mathbf{x}) = \mathcal{L}(u)(t,\mathbf{x}) + f(\mathbf{x}) \in \mathbb{R}, \ (t,\mathbf{x}) \in [0,\Delta t] \times \Omega \subset \mathbb{R} \times \mathbb{R}^d, \\ \mathcal{B}(u)(t,\mathbf{x}) = a(t,\mathbf{x}), \ \mathbf{x} \in \partial\Omega, \ t \in [0,\Delta t], \\ u(0,\mathbf{x}) = b(\mathbf{x}), \ \mathbf{x} \in \overline{\Omega}. \end{cases}$$

Extension to PDEs

$$\begin{cases} \partial_t u(t,\mathbf{x}) = \mathcal{L}(u)(t,\mathbf{x}) + f(\mathbf{x}) \in \mathbb{R}, \ (t,\mathbf{x}) \in [0,\Delta t] \times \Omega \subset \mathbb{R} \times \mathbb{R}^d, \\ \mathcal{B}(u)(t,\mathbf{x}) = a(t,\mathbf{x}), \ \mathbf{x} \in \partial\Omega, \ t \in [0,\Delta t], \\ u(0,\mathbf{x}) = b(\mathbf{x}), \ \mathbf{x} \in \overline{\Omega}. \end{cases}$$

We can define a network $\mathcal{N}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$ and minimise

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} |\partial_t \mathcal{N}_\theta(t_i,\mathbf{x}_i) - \mathcal{L}(\mathcal{N}_\theta)(t_i,\mathbf{x}_i) - f(\mathbf{x}_i)|^2 + \underbrace{\sum_{j=1}^{M} |\mathcal{B}(\mathcal{N}_\theta)(t_j,\mathbf{x}_j) - a(t_j,\mathbf{x}_j)|^2}_{\text{boundary condition}}$$

$$+ \underbrace{\sum_{k=1}^{K} |\mathcal{N}_\theta(0,\mathbf{x}_k) - b(\mathbf{x}_k)|^2}_{\text{initial condition}},$$

where $t_i, t_j \in [0,\Delta t]$, $\mathbf{x}_i \in \Omega$, $\mathbf{x}_j \in \partial\Omega$, $\mathbf{x}_k \in \overline{\Omega}$.

How to enforce the boundary conditions

- In some domains, we can easily enforce the boundary conditions following similar principles to those seen before. For example,

$$\mathcal{N}_\theta(t,x) = \widetilde{\mathcal{N}}_\theta(t,x) + x(u_1 - \widetilde{\mathcal{N}}_\theta(t,1)) + (1-x)(u_0 - \widetilde{\mathcal{N}}_\theta(t,0)),\ (t,x) \in \mathbb{R}\times\mathbb{R},$$

  satisfies $\mathcal{N}_\theta(t,0) = u_0$, and $\mathcal{N}_\theta(t,1) = u_1$ for every $t \in \mathbb{R}$.

How to enforce the boundary conditions

- In some domains, we can easily enforce the boundary conditions following similar principles to those seen before. For example,

$$\mathcal{N}_\theta(t,x) = \widetilde{\mathcal{N}}_\theta(t,x) + x(u_1 - \widetilde{\mathcal{N}}_\theta(t,1)) + (1-x)(u_0 - \widetilde{\mathcal{N}}_\theta(t,0)),\ (t,x) \in \mathbb{R}\times\mathbb{R},$$

  satisfies $\mathcal{N}_\theta(t,0) = u_0$, and $\mathcal{N}_\theta(t,1) = u_1$ for every $t \in \mathbb{R}$.

- The same can be done for some types of Neumann boundary conditions as well, for example

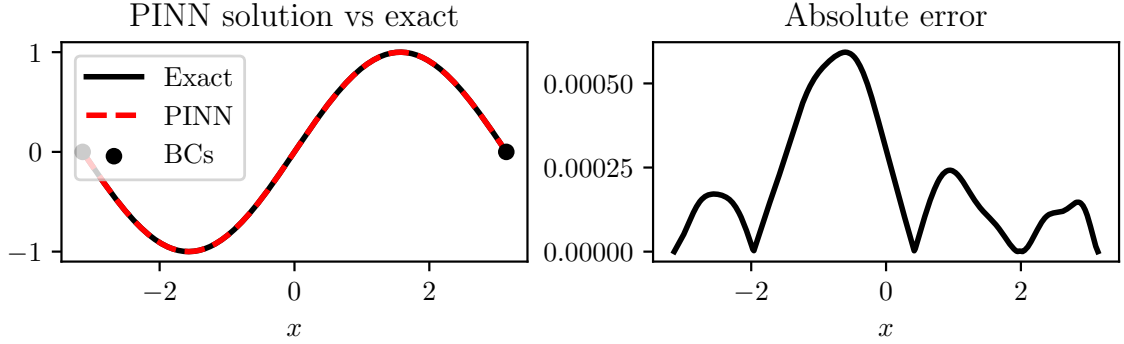
$$\mathcal{N}_\theta(t,x) = \widetilde{\mathcal{N}}_\theta(t,x) + x(u_1' - \partial_x\widetilde{\mathcal{N}}_\theta(t,1)) + (1-x)^2(u_0 - \widetilde{\mathcal{N}}_\theta(t,0)),\ (t,x) \in \mathbb{R}\times\mathbb{R},$$

  satisfies $\mathcal{N}_\theta(t,0) = u_0$, and $\partial_x\mathcal{N}_\theta(t,1) = u_1'$ for every $t \in \mathbb{R}$.

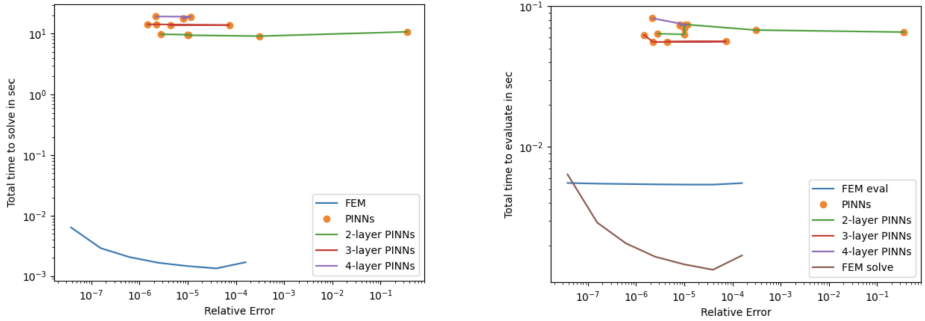
A simple example

$$u''(x) = -\sin(x), \;\; u(-\pi) = u(\pi) = 0.$$

We consider $\mathcal{N}_\theta(x) = \mathbf{a}^\top \tanh(\mathbf{b}x + \mathbf{c})$, $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^{64}$, and train it for 3000 epochs. In each epoch we sample 256 new random collocation points.



PINNs VS Finite Elements



(a) Plot of time to solve FEM and train PINN in sec versus ℓ^2 relative error.

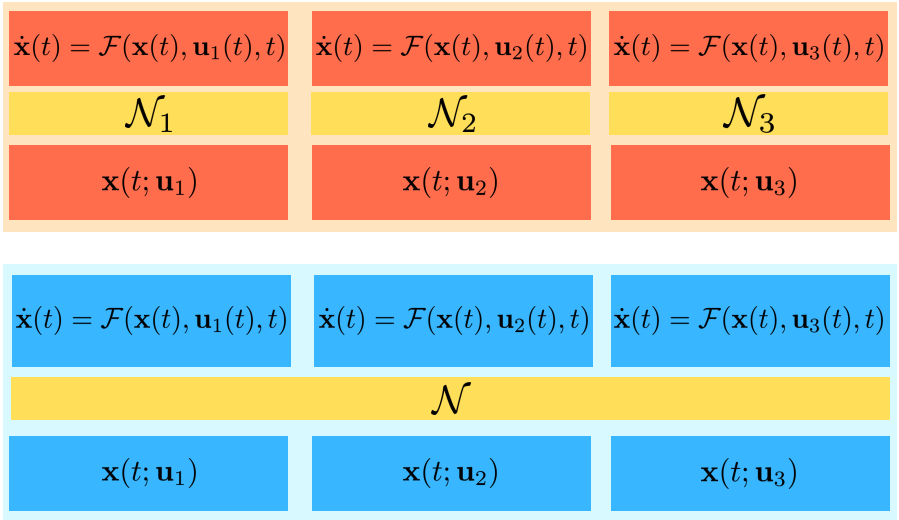
(b) Plot of time to interpolate FEM and evaluate PINN in sec versus ℓ^2 relative error.

Figure 2: Plot for 1D Poisson equation of time in sec versus ℓ^2 relative error.

Figure 4: Source: Grossmann et al., "Can Physics-Informed Neural Networks beat the Finite Element Method?"

Brief introduction to Neural Operators

A paradigm shift



A paradigm shift

- The main idea behind Neural Operators is that if we want to solve the same differential equations several times, but where we change boundary conditions, forcing terms, initial conditions, or controls, we should approximate an operator rather than a single function.

A paradigm shift

- The main idea behind Neural Operators is that if we want to solve the same differential equations several times, but where we change boundary conditions, forcing terms, initial conditions, or controls, we should approximate an operator rather than a single function.

- When solving $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t), \mathbf{u}(t), t) \in \mathbb{R}^d$ where the control/input signal $\mathbf{u} \in \mathcal{C}^1(\mathbb{R}, \mathbb{R}^c)$ can vary, and $\mathbf{x}(0) = \mathbf{x}_0$, we want to approximate a mapping

$$\mathcal{S} : \mathcal{C}^1([0, T], \mathbb{R}^c) \to \mathcal{C}^1([0, T], \mathbb{R}^d),$$

$$[0, T] \ni t \mapsto \mathcal{S}(\mathbf{u})(t) = \mathbf{x}_0 + \int_0^t \mathcal{F}(\mathbf{x}(s), \mathbf{u}(s), s) ds \in \mathbb{R}^d.$$

A paradigm shift

- The main idea behind Neural Operators is that if we want to solve the same differential equations several times, but where we change boundary conditions, forcing terms, initial conditions, or controls, we should approximate an operator rather than a single function.

- When solving $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t), \mathbf{u}(t), t) \in \mathbb{R}^d$ where the control/input signal $\mathbf{u} \in \mathcal{C}^1(\mathbb{R}, \mathbb{R}^c)$ can vary, and $\mathbf{x}(0) = \mathbf{x}_0$, we want to approximate a mapping

$$\mathcal{S} : \mathcal{C}^1([0, T], \mathbb{R}^c) \to \mathcal{C}^1([0, T], \mathbb{R}^d),$$
$$[0, T] \ni t \mapsto \mathcal{S}(\mathbf{u})(t) = \mathbf{x}_0 + \int_0^t \mathcal{F}(\mathbf{x}(s), \mathbf{u}(s), s) ds \in \mathbb{R}^d.$$

- The same can apply to boundary or initial conditions, which would be replaced to the function $\mathbf{u}$.

A paradigm shift

- The main idea behind Neural Operators is that if we want to solve the same differential equations several times, but where we change boundary conditions, forcing terms, initial conditions, or controls, we should approximate an operator rather than a single function.

- When solving $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t), \mathbf{u}(t), t) \in \mathbb{R}^d$ where the control/input signal $\mathbf{u} \in \mathcal{C}^1(\mathbb{R}, \mathbb{R}^c)$ can vary, and $\mathbf{x}(0) = \mathbf{x}_0$, we want to approximate a mapping

$$\mathcal{S} : \mathcal{C}^1([0, T], \mathbb{R}^c) \to \mathcal{C}^1([0, T], \mathbb{R}^d),$$
$$[0, T] \ni t \mapsto \mathcal{S}(\mathbf{u})(t) = \mathbf{x}_0 + \int_0^t \mathcal{F}(\mathbf{x}(s), \mathbf{u}(s), s) ds \in \mathbb{R}^d.$$

- The same can apply to boundary or initial conditions, which would be replaced to the function $\mathbf{u}$.

- The `SympFlow` that we saw before is similar in spirit. The difference is that the initial condition is a fixed vector, not a function: $\mathcal{N}_\theta : \mathbb{R}^d \to \mathcal{C}^1(\mathbb{R}, \mathbb{R}^d)$.

How are they trained?

- Without even knowing how a neural operator can be parametrised, let us first see how they can be trained.

How are they trained?

- Without even knowing how a neural operator can be parametrised, let us first see how they can be trained.
- Assume that there is some neural operator $\mathcal{N}_\theta : \mathcal{X} \to \mathcal{Y}$, for a pair of function spaces $\mathcal{X}$ and $\mathcal{Y}$, that needs to be trained to approximate an operator $\mathcal{S} : \mathcal{X} \to \mathcal{Y}$. Assume that the functions in $\mathcal{X}$ and $\mathcal{Y}$ take inputs in $\Omega \subset \mathbb{R}^d$, and return outputs in $\mathbb{R}^a$ and $\mathbb{R}^b$, respectively.

How are they trained?

- Without even knowing how a neural operator can be parametrised, let us first see how they can be trained.
- Assume that there is some neural operator $\mathcal{N}_\theta : \mathcal{X} \to \mathcal{Y}$, for a pair of function spaces $\mathcal{X}$ and $\mathcal{Y}$, that needs to be trained to approximate an operator $\mathcal{S} : \mathcal{X} \to \mathcal{Y}$. Assume that the functions in $\mathcal{X}$ and $\mathcal{Y}$ take inputs in $\Omega \subset \mathbb{R}^d$, and return outputs in $\mathbb{R}^a$ and $\mathbb{R}^b$, respectively.
- A common way to train $\mathcal{N}_\theta$ is then to do so in a supervised manner, by minimising

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{M_i} \sum_{j=1}^{M_i} \|\mathcal{N}_\theta(\mathbf{u}_i)(\mathbf{z}_{ij}) - \mathcal{S}(\mathbf{u}_i)(\mathbf{z}_{ij})\|_2^2 .$$

Example: DeepONets

$$\dot{x}(t) = \mathcal{F}(x(t), u(t), t) \in \mathbb{R}, \ \mathcal{S}(u)(t) := x_0 + \int_0^t \mathcal{F}(x(s), u(s), s) ds.$$

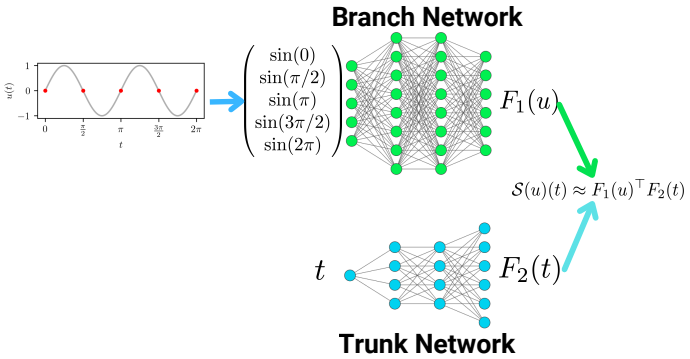


Figure 5: Lu et al., "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators"

Resolution/Discretisation invariance

an operator $G : u \mapsto G(u)$ takes two inputs $[u(x_1), u(x_2), \ldots, u(x_m)]$ 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 $x_1, x_2, \ldots, x_m$. However, we do not enforce any constraints on the number or locations for the evaluation of output functions. (**C**) The stacked DeepONet in Theorem 1 has one trunk network and p stacked branch networks. (**D**) The unstacked DeepONet has one trunk network and one branch network.

Figure 6: Lu et al., "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators"

Discretization-Invariant Models. We formulate a precise mathematical notion of discretization invariance. We require any 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.

Figure 7: Kovachki et al., "Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs"