Symplectic Neural Flows and Neural ODEs

Davide Murari

Department of Applied Mathematics and Theoretical Physics
University of Cambridge

davidemurari.com/notesunivr2025

dm2011@cam.ac.uk

Outline

Symplectic Neural Flows

Classical methods for 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)

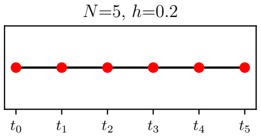
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)),\ \mathbf{x}(0) = \mathbf{x}_0 \text{ notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

  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)

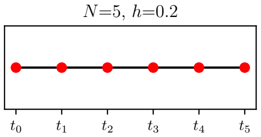
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)),\ \mathbf{x}(0) = \mathbf{x}_0 \text{ notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

  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

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

  such that $\mathbf{y}_0 = \mathbf{x}_0$ and $\mathbf{y}_n \approx \mathbf{x}(nh)$, $n = 1, \dots, N$, 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.

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

$$\ddot{u}(t) = -ku(t) + \gamma\dot{u}(t),\ \mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  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

$$\ddot{u}(t) = -ku(t) + \gamma \dot{u}(t), \ \mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  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

$$\ddot{u}(t) = -ku(t) + \gamma\dot{u}(t),\ \mathcal{L}_\alpha u(\mathbf{x}) = f_\beta(\mathbf{x}).$$

  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 $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t))$ over $t \in [0, T]$, we can define $\mathcal{N}_\theta : \mathbb{R} \to \mathbb{R}^d$ and train it so it almost satisfies the initial condition and

$$\frac{d}{dt}\mathcal{N}_\theta(t_i) \approx \mathcal{F}(\mathcal{N}_\theta(t_i)),\ i = 1, ..., N,\ t_i \in [0, T].$$

The time derivative $\frac{d}{dt}\mathcal{N}_\theta$ 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 $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t))$ over $t \in [0, T]$, we can define $\mathcal{N}_\theta : \mathbb{R} \to \mathbb{R}^d$ and train it so it almost satisfies the initial condition and

$$\frac{d}{dt}\mathcal{N}_\theta(t_i) \approx \mathcal{F}(\mathcal{N}_\theta(t_i)), \ i = 1, ..., N, \ t_i \in [0, T].$$

  The time derivative $\frac{d}{dt}\mathcal{N}_\theta$ 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 ODE 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 consider 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 $\mathcal{N}_\theta\left(\cdot;\mathbf{x}_0\right) : [0, T] \to \mathbb{R}^d$, and choose its weights so that

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

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

Physics-informed neural networks (PINNs)

- Let us consider 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 $t_1, \dots, t_C \in [0,T]$.

- 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

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

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

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

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 $\mathcal{N}_\theta\left(0;\mathbf{x}_0\right)=\mathbf{x}_0$ exactly for every $\mathbf{x}_0$.
- This can be done in several ways. Two common strategies are:

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

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

- 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.

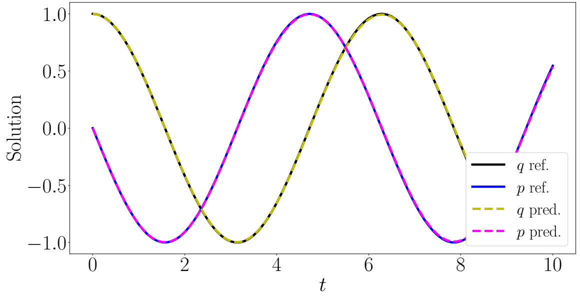


Figure 1: Solution comparison after reaching a loss value of 10^{-5}. The training time is 87 seconds (7500 epochs with 1000 new collocation points randomly sampled at each of them).

Integration over long time intervals

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

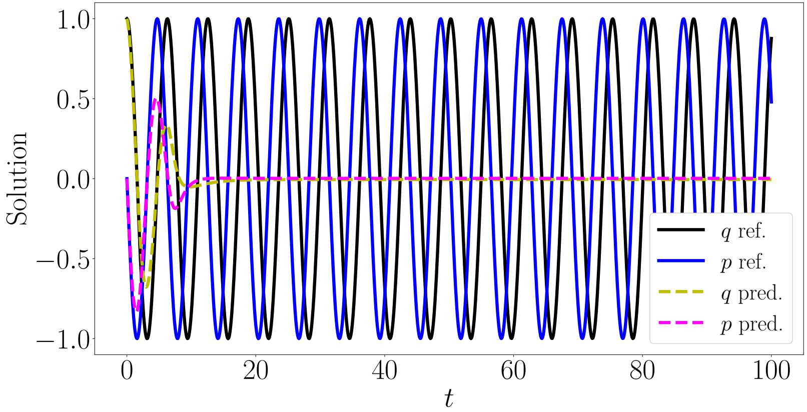


Figure 2: Solution comparison after 10000 epochs.

Forward invariant subset of the phase space

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

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

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**.

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

Forward invariant subset of the phase space

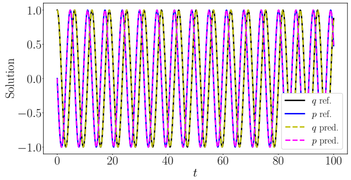
- Consider the vector field $\mathcal{F}:\mathbb{R}^d\to\mathbb{R}^d$, and introduce notation $\phi^t_{\mathcal{F}}:\mathbb{R}^d\to\mathbb{R}^d$ for the time-t flow map of $\mathcal{F}$, which for every $\mathbf{x}_0\in\mathbb{R}^d$ satisfies

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

- 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: Symplectic Neural Flows

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, Davide Murari, Carola-Bibiane Schönlieb, Ferdia Sherry, and Zakhar Shumaylov. "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:

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

[1]Canizares, Murari, Schönlieb, Sherry, and Shumaylov, "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:

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

- The SympFlow architecture is defined as

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

with

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

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

[1]Canizares, Murari, Schönlieb, Sherry, and Shumaylov, "Symplectic neural flows for modeling and discovery".

Properties of `SympFlow`

- `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 SympFlow

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

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

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

- `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

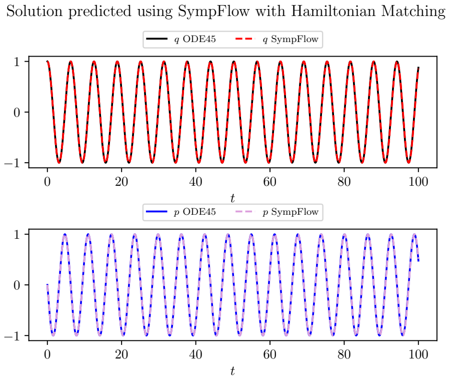
$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \left\| \frac{d}{dt}\mathcal{N}_\theta\left(t, \mathbf{x}_0^i\right)\Big|_{t=t_i} - \mathbb{J}\nabla H\left(\mathcal{N}_\theta\left(t_i, \mathbf{x}_0^i\right)\right) \right\|_2^2$$

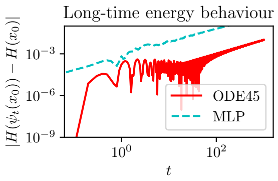
where we sample $t_i \in [0, \Delta t]$, and $\mathbf{x}_0^i \in \Omega \subset \mathbb{R}^{2n}$.

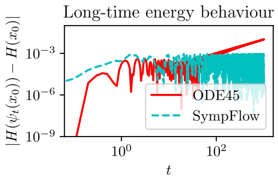
Simple Harmonic Oscillator (unsupervised)

Equations of motion

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







Hénon–Heiles (unsupervised)

Equations of motion

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

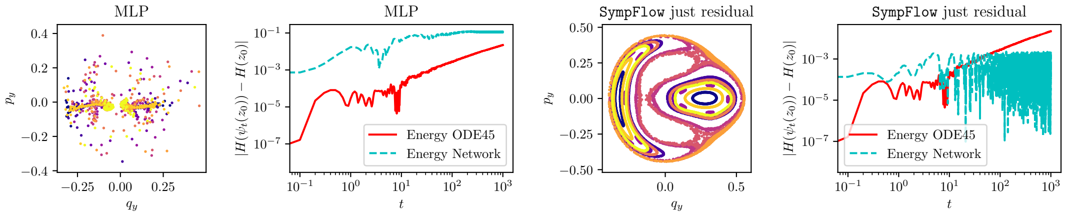


Figure 3: **Unsupervised experiment — Hénon–Heiles:** Comparison of the Poincaré sections and the energy behaviour up to time $T = 1000$.

Neural ODEs

What are they?

Neural ODEs: The Continuous-Depth Limit

- ResNet layers can be interpreted as discretisations of parametric ODEs.
- If we go to the limit as the time step goes to zero, we can recover a dynamical system
$$\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(t, \mathbf{x}(t)),\ \theta \in \Theta,$$
where $\mathcal{F}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is parametrised by a neural network.

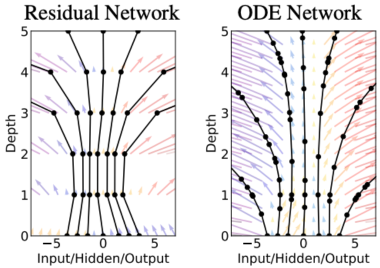


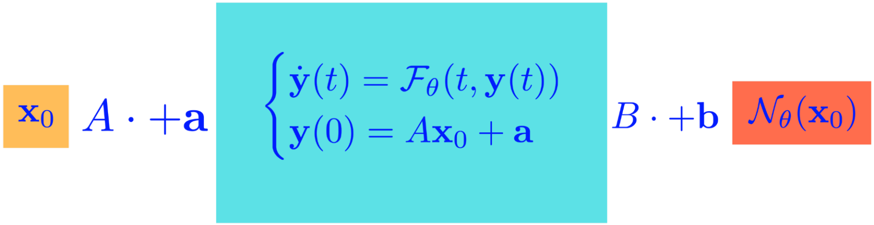
Figure 4: Source: Chen, Rubanova, Bettencourt, and Duvenaud, "Neural Ordinary Differential Equations".

Neural ODEs

More explicitly, a Neural ODE is a parametric map $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^c$ of the form

$$\mathcal{N}_\theta(\mathbf{x}_0) = B\mathbf{y}(T) + \mathbf{b} \in \mathbb{R}^c, \quad \begin{cases} \dot{\mathbf{y}}(t) = \mathcal{F}_\theta(t, \mathbf{y}(t)), \ \mathcal{F}_\theta : \mathbb{R} \times \mathbb{R}^h \to \mathbb{R}^h, \\ \mathbf{y}(0) = A\mathbf{x}_0 + \mathbf{a} \in \mathbb{R}^h, \end{cases}$$

for an $h \in \mathbb{N}$. Here, $A \in \mathbb{R}^{h\times d}$, $\mathbf{a} \in \mathbb{R}^h$, $B \in \mathbb{R}^{c\times h}$, and $\mathbf{b} \in \mathbb{R}^c$.


$\mathbf{x}_0$
$A \cdot + \mathbf{a}$
$\begin{cases} \dot{\mathbf{y}}(t) = \mathcal{F}_\theta(t, \mathbf{y}(t)) \\ \mathbf{y}(0) = A\mathbf{x}_0 + \mathbf{a} \end{cases}$
$B \cdot + \mathbf{b}$
$\mathcal{N}_\theta(\mathbf{x}_0)$

How can we train them?

How to train them: discrete vs continuous adjoint method

- There are two main strategies to train Neural ODEs:
  1. Discrete backpropagation/adjoint, and
  2. Continuous adjoint.

How to train them: discrete vs continuous adjoint method

- There are two main strategies to train Neural ODEs:
  1. Discrete backpropagation/adjoint, and
  2. Continuous adjoint.

- The first, corresponds to the conventional backpropagation algorithm, where the forward pass is defined through a numerical method:

$$\begin{aligned} \mathbf{y}_0 &= A\mathbf{x}_0 + \mathbf{a} \\ \mathbf{y}_{k+1} &= \varphi^{h_k}_{\mathcal{F}_\theta}(t_k, \mathbf{y}_k),\ t_{k+1} = t_k + h_k,\ k = 0, \dots, K-1,\ T = t_K, \\ \mathcal{N}_\theta(\mathbf{x}_0) &= B\mathbf{y}_K + \mathbf{b}. \end{aligned}$$

  As long as the numerical method φ is differentiable, we can backpropagate through it and minimise the loss function to find a good set of weights.

- This approach could be memory inefficient because we have to store all the intermediate solutions $\mathbf{y}_1, \dots, \mathbf{y}_{K-1}$ (and the Runge–Kutta stages).

The adjoint variable

- For simplicity, fix $d = h = c$, and consider Neural ODEs of the form
$$\mathcal{N}_\theta(\mathbf{x}_0) = \mathbf{x}(T) = \mathbf{x}_0 + \int_0^T \mathcal{F}_\theta(t, \mathbf{x}(t))dt, \ A = B = I_d, \mathbf{a} = \mathbf{b} = 0.$$
We assume an oracle (a numerical method) has provided this output.

The adjoint variable

- For simplicity, fix $d = h = c$, and consider Neural ODEs of the form
$$\mathcal{N}_\theta(\mathbf{x}_0) = \mathbf{x}(T) = \mathbf{x}_0 + \int_0^T \mathcal{F}_\theta(t, \mathbf{x}(t))dt, \; A = B = I_d, \, \mathbf{a} = \mathbf{b} = 0.$$
We assume an oracle (a numerical method) has provided this output.

- We introduce the loss $L : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, and inspect $\nabla_\theta L(\mathcal{N}_\theta(\mathbf{x}_0), \mathbf{y})$. We see that
$$\boxed{\nabla_\theta L(\mathcal{N}_\theta(\mathbf{x}_0), \mathbf{y}) = \left(\frac{\partial \mathbf{x}(T)}{\partial \theta}\right)^\top \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(T)}.}$$

- Given the expression above, we introduce the so-called adjoint variable
$$\mathbf{a}(t) = \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t)} \in \mathbb{R}^d.$$
What is the value of $\mathbf{a}(t)$ for $t \in [0, T]$?

The adjoint equation

- For any $t \in \mathbb{R}$ and $\varepsilon > 0$, we see that

$$\mathbf{x}(t+\varepsilon) - \mathbf{x}(t) = \int_t^{t+\varepsilon} \dot{\mathbf{x}}(s)ds = \int_t^{t+\varepsilon} \mathcal{F}_\theta(s, \mathbf{x}(s))ds.$$

  Furthermore, by the chain rule we get

$$\frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t)} = \left(\frac{\partial \mathbf{x}(t+\varepsilon)}{d\mathbf{x}(t)}\right)^\top \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t+\varepsilon)}, \text{ i.e., } \mathbf{a}(t) = \left(\frac{\partial \mathbf{x}(t+\varepsilon)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t+\varepsilon).$$

The adjoint equation

- For any $t \in \mathbb{R}$ and $\varepsilon > 0$, we see that

$$\mathbf{x}(t+\varepsilon) - \mathbf{x}(t) = \int_t^{t+\varepsilon} \dot{\mathbf{x}}(s)ds = \int_t^{t+\varepsilon} \mathcal{F}_\theta(s, \mathbf{x}(s))ds.$$

  Furthermore, by the chain rule we get

$$\frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t)} = \left(\frac{\partial \mathbf{x}(t+\varepsilon)}{d\mathbf{x}(t)}\right)^\top \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t+\varepsilon)}, \text{ i.e., } \mathbf{a}(t) = \left(\frac{\partial \mathbf{x}(t+\varepsilon)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t+\varepsilon).$$

- Called $J(\varepsilon) = \frac{\partial \mathbf{x}(t+\varepsilon)}{\partial \mathbf{x}(t)}$, one can see that $J'(0) = \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)}$ (exercise). Differentiating at $\varepsilon = 0$ the equality $a(t) = J(\varepsilon)^\top a(t+\varepsilon)$ we then recover

Adjoint equation

$$\dot{\mathbf{a}}(t) = -\left(\frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t), \;\; \mathbf{a}(T) = \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(T)}.$$

Computing the desired gradient

- Let $\theta \in \mathbb{R}^p$. Call $J_\theta(t) = \frac{\partial \mathbf{x}(t)}{\partial \theta} \in \mathbb{R}^{d\times p}$, the matrix which satisfies the ODE

$$\frac{d}{dt} J_\theta(t) = \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \theta} + \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)} J_\theta(t), \;\; J_\theta(0) = \frac{\partial \mathbf{x}_0}{\partial \theta} = 0_{d\times p}.$$

Computing the desired gradient

- Let $\theta \in \mathbb{R}^p$. Call $J_\theta(t) = \frac{\partial \mathbf{x}(t)}{\partial \theta} \in \mathbb{R}^{d\times p}$, the matrix which satisfies the ODE

$$\boxed{\frac{d}{dt} J_\theta(t) = \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \theta} + \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)} J_\theta(t), \;\; J_\theta(0) = \frac{\partial \mathbf{x}_0}{\partial \theta} = 0_{d\times p}.}$$

- We see that $\mathbb{R}^p \ni \nabla_\theta L(\mathbf{x}(T), \mathbf{y}) = (J_\theta(T))^\top \mathbf{a}(T)$ Why do we even need the adjoint equation if we can just integrate the matrix ODE above and find $J_\theta(T) \in \mathbb{R}^{d\times p}$, and we already have $a(T)$?

Computing the desired gradient

- Let $\theta \in \mathbb{R}^p$. Call $J_\theta(t) = \frac{\partial \mathbf{x}(t)}{\partial \theta} \in \mathbb{R}^{d\times p}$, the matrix which satisfies the ODE

$$\frac{d}{dt} J_\theta(t) = \frac{\partial \mathcal{F}_\theta(t,\mathbf{x}(t))}{\partial \theta} + \frac{\partial \mathcal{F}_\theta(t,\mathbf{x}(t))}{\partial \mathbf{x}(t)} J_\theta(t),\;\; J_\theta(0) = \frac{\partial \mathbf{x}_0}{\partial \theta} = 0_{d\times p}.$$

- We see that

$$\mathbb{R}^p \ni \nabla_\theta L(\mathbf{x}(T),\mathbf{y}) = (J_\theta(T))^\top \mathbf{a}(T) = (J_\theta(0))^\top \mathbf{a}(0) + \int_0^T \frac{d}{dt}\left((J_\theta(t))^\top \mathbf{a}(t)\right) dt.$$

The desired expression follows from the derivation below

$$\frac{d}{dt}((J_\theta(t))^\top \mathbf{a}(t)) = -(J_\theta(t))^\top \left(\partial_{\mathbf{x}(t)} \mathcal{F}_\theta(t,\mathbf{x}(t))\right)^\top \mathbf{a}(t)$$
$$+ \left(\partial_\theta \mathcal{F}_\theta(t,\mathbf{x}(t)) + \partial_{\mathbf{x}(t)} \mathcal{F}_\theta(t,\mathbf{x}(t)) J_\theta(t)\right)^\top \mathbf{a}(t) = (\partial_\theta \mathcal{F}_\theta(t,\mathbf{x}(t)))^\top \mathbf{a}(t).$$
$$\implies \nabla_\theta L(\mathbf{x}(T),\mathbf{y}) = \int_0^T (\partial_\theta \mathcal{F}_\theta(t,\mathbf{x}(t)))^\top \mathbf{a}(t) dt.$$

Summary of the Adjoint method to compute $\nabla_\theta L(\mathbf{x}(T),\mathbf{y})$

Consider the Neural ODE $\dot{\mathbf{x}}(t)=\mathcal{F}_\theta(t,\mathbf{x}(t))$. Solve the coupled system of ODEs

$$\begin{cases}\dot{\mathbf{x}}(t)=\mathcal{F}_\theta(t,\mathbf{x}(t)) \ \text{forward},\\ \dot{\mathbf{a}}(t)=-\partial_{\mathbf{x}(t)}\mathcal{F}_\theta(t,\mathbf{x}(t))^\top\mathbf{a}(t) \ \text{backward},\\ \mathbf{x}(0)=\mathbf{x}_0\in\mathbb{R}^d,\ \mathbf{a}(T)=\partial_{\mathbf{x}(T)}L(\mathbf{x}(T),\mathbf{y}).\end{cases}$$

Compute then the desired gradient as

$$\nabla_\theta L(\mathbf{x}(T),\mathbf{y})=\int_0^T(\partial_\theta\mathcal{F}_\theta(t,\mathbf{x}(t)))^\top\mathbf{a}(t)dt.$$

Remark: Notice that $\nabla_\theta(\mathcal{F}_\theta(t,\mathbf{x}(t))^\top\mathbf{a}(t))=(\partial_\theta\mathcal{F}_\theta(t,\mathbf{x}(t)))^\top\mathbf{a}(t)$ and this can be efficiently computed as a vector-Jacobian product (VJP).
Updates the weights, for example with GD.

Variations of the conventional Neural ODE

There are a lot of research papers considering alternative design strategies for Neural ODEs. We include here a couple:

- In Krzysztof M Choromanski, Jared Quincy Davis, Valerii Likhosherstov, Xingyou Song, Jean-Jacques Slotine, Jacob Varley, Honglak Lee, Adrian Weller, and Vikas Sindhwani. "Ode to an ODE". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 3338–3350, the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

$$\begin{cases} \dot{\mathbf{x}}(t) = \sigma(W(t)\mathbf{x}(t)) \in \mathbb{R}^d, \ (\text{e.g. } \sigma(x) = |x|) \\ \dot{W}(t) = W(t)\Omega(t, W(t)) \in \mathbb{R}^{d\times d}, \ \Omega(t, W) \in \mathrm{Skew}(d) \\ \mathbf{x}(0) = \mathbf{x}_0, W(0) = W_0 \in \mathcal{O}(d). \end{cases}$$

Variations of the conventional Neural ODE

There are a lot of research papers considering alternative design strategies for Neural ODEs. We include here a couple:

- In Choromanski, Davis, Likhosherstov, Song, Slotine, Varley, Lee, Weller, and Sindhwani, "Ode to an ODE", the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

$$\begin{cases} \dot{\mathbf{x}}(t) = \sigma(W(t)\mathbf{x}(t)) \in \mathbb{R}^d, \ (\text{e.g. } \sigma(x) = |x|) \\ \dot{W}(t) = W(t)\Omega(t, W(t)) \in \mathbb{R}^{d\times d}, \ \Omega(t, W) \in \mathrm{Skew}(d) \\ \mathbf{x}(0) = \mathbf{x}_0, W(0) = W_0 \in \mathcal{O}(d). \end{cases}$$

- In Norcliffe, Bodnar, Day, Simidjievski, and Liò, "On Second Order Behaviour in Augmented Neural ODEs", the authors consider second order Neural ODEs

$$\ddot{\mathbf{x}}(t) = \mathcal{F}_\theta(\mathbf{x}(t), \dot{\mathbf{x}}(t), t, \theta) \in \mathbb{R}^d \iff \begin{cases} \dot{\mathbf{x}}(t) = \mathbf{v}(t) \\ \dot{\mathbf{v}}(t) = \mathcal{F}_\theta(\mathbf{x}(t), \mathbf{v}(t), t, \theta). \end{cases}$$

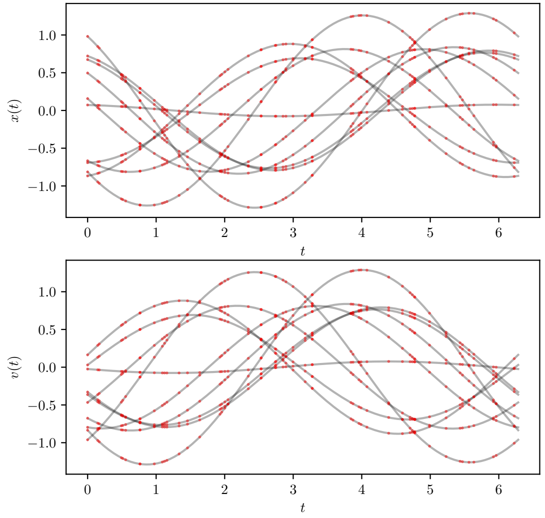
The first application: Data-driven discovery and simulation

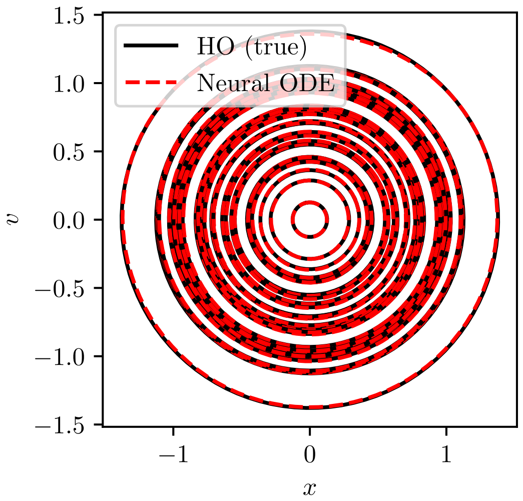
Implementation with PyTorch

- There are several libraries that allow for the quick implementation of these models.
- An example is https://github.com/rtqichen/torchdiffeq:

```
import numpy as np; from scipy.integrate import odeint as sp_odeint
import torch, torch.nn as nn, torch.optim as optim; from torchdiffeq import odeint_adjoint as odeint

simpleHO = lambda y, t: [y[1], -y[0]]

T = np.linspace(0., 2*np.pi, 50); Y0_np = np.random.randn(1000, 2)
Y_star_np = np.stack([sp_odeint(simpleHO, y0, T) for y0 in Y0_np], axis=1)

T_t = torch.from_numpy(T).float(); Y0  = torch.from_numpy(Y0_np).float(); Y_star = torch.from_numpy(Y_star_np).float()

class ODEFunc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(2, 32), nn.Tanh(), nn.Linear(32, 2))
    def forward(self, t, y):
        return self.net(y)

f = ODEFunc(); opt = optim.Adam(f.parameters(), lr=1e-2)

for _ in range(1000):
    Y = odeint(f, Y0, T_t)
    loss = (Y - Y_star).pow(2).mean()
    opt.zero_grad(); loss.backward(); opt.step()
```

Simulation with irregular time-sampling

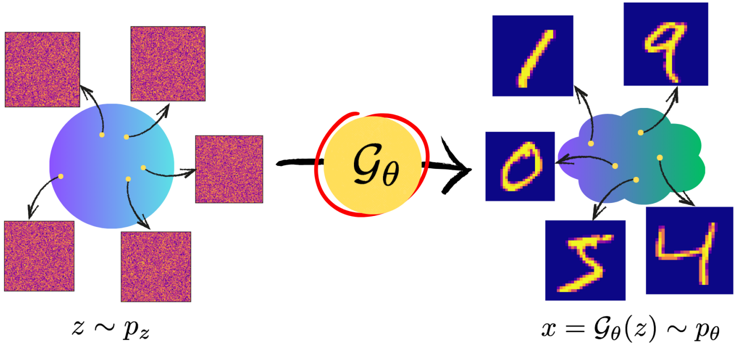




Generative modelling

What do we mean by generative modelling?

A generative model is a machine learning model designed to create new data that is similar to its training data. Generative models learn the distribution of the training data, then apply those understandings to generate new content in response to new input data.



How can we train the generator $\mathcal{G}_\theta$?

Before specifying a modelling strategy for $\mathcal{G}_\theta$, let us understand how we can train its weights θ so that p_θ resembles p_{data}.

A way to get p_θ as close as possible to the correct distribution p_{data} is to maximise the log-likelihood:

$$\arg\max_\theta \mathbb{E}_{x\sim p_{\text{data}}}[\log(p_\theta(x))] = \arg\min_\theta D_{\text{KL}}(p_{\text{data}}||p_\theta),\ D_{\text{KL}}(P||Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right].$$

Exercise: Why is D_{KL} not a metric?

$$\text{Empirically: } \arg\min_\theta -\frac{1}{N}\sum_{i=1}^N \log(p_\theta(x_i)),\ x_1,...,x_N \sim p_{\text{data}},\ \text{iid.}$$

Continuous normalising flows

Generative modelling with continuous normalising flows

Consider a neural ODE

$$\begin{cases} \frac{d}{dt}\phi_{t,\theta}(z) = \mathcal{F}_t^\theta(\phi_{t,\theta}(z)),\ \mathcal{F}^\theta : [0,1]\times\mathbb{R}^d \to \mathbb{R}^d, \\ \phi_{0,\theta}(z) = z, \end{cases}$$

and an easy-to-sample probability measure with density p_{init}. We then set $x = \phi_{1,\theta}(z)$. We assume that $\mathcal{F}^\theta$ is C^1 and both $\mathcal{F}^\theta$ and $\partial_z\mathcal{F}^\theta$ are Lipschitz in z.

Continuous normalising flows define

$$p_t^\theta(x) = p_{\text{init}}(\phi_{t,\theta}^{-1}(x))\left|\det \partial_x\left(\phi_{t,\theta}^{-1}(x)\right)\right|,\ t\in[0,1].$$

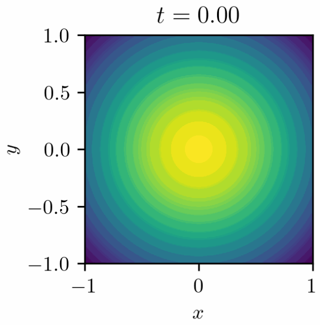
This leads to $\log p_1^\theta(x) = \log(p_{\text{init}}(\phi_{1,\theta}^{-1}(x))) - \int_0^1 \operatorname{div}(\mathcal{F}_s^\theta)(\phi_{s,\theta}^{-1}(x))ds$.

An example

For the ODEs $\dot{x} = y$, $\dot{y} = -2x$, with exact flow

$$\phi^t(x,y) = \left(x\cos(\sqrt{2}t) + \frac{y}{\sqrt{2}}\sin(\sqrt{2}t), y\cos(\sqrt{2}t) - \sqrt{2}x\sin(\sqrt{2}t) \right),$$

we have $\det \partial_z(\phi^t(z)) = 1$, since the equations are Hamiltonian, and hence $p_t(z) = p_0(\phi^{-t}(z))$.



Derivations of the previous formulas

Consider the probability measure π_0 with density function p_0, and let π_t^θ be defined as the pushforward $(\phi_{t,\theta})_*\pi_0$. Then its density can be characterised as follows:

$$\pi_t^\theta(A) = \pi_0(\phi_{t,\theta}^{-1}(A)) = \int_{\phi_{t,\theta}^{-1}(A)} p_0(z)dz = \int_A p_0(\phi_{t,\theta}^{-1}(x))\,\mathrm{det}\,\partial_x\left(\phi_{t,\theta}^{-1}(x)\right)dx =: \int_A p_\theta(x)dx.$$

Derivations of the previous formulas

Consider the probability measure π_0 with density function p_0, and let π_t^θ be defined as the pushforward $(\phi_{t,\theta})_*\pi_0$. Then its density can be characterised as follows:

$$\pi_t^\theta(A) = \pi_0(\phi_{t,\theta}^{-1}(A)) = \int_{\phi_{t,\theta}^{-1}(A)} p_0(z)dz = \int_A p_0(\phi_{t,\theta}^{-1}(x))\det\partial_x\left(\phi_{t,\theta}^{-1}(x)\right)dx =: \int_A p_\theta(x)dx.$$

Since we need $\log p_1^\theta(x)$ for the KL divergence, we now look for a more efficient way to compute it than by taking the determinant of the Jacobian. More precisely, we now look for an ODE solved by $\log p_t(x)$, and solve it up to time $t=1$ to compute $\log p_1^\theta$.

Derivations of the ODE for the log probability

Let

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

and, for $h > 0$, define the map $T_h : \mathbb{R}^d \to \mathbb{R}^d$ as $T_h\mathbf{x}(t) = \mathbf{x}(t+h)$. We see that[2]

$$\begin{aligned}
\frac{d}{dt}\log p_t(\mathbf{x}(t)) &= \lim_{h\to 0^+} \frac{\log p_{t+h}(T_h(\mathbf{x}(t))) - \log p_t(\mathbf{x}(t))}{h} \\
&= \lim_{h\to 0^+} \frac{\log\left(p_t(T_h^{-1}T_h(\mathbf{x}(t)))|\det \partial_x T_h^{-1}(\mathbf{x}(t))|\right) - \log p_t(\mathbf{x}(t))}{h} \\
&= -\lim_{h\to 0^+} \frac{\log \det \partial_x T_h(\mathbf{x}(t))}{h} \\
&= -\lim_{h\to 0^+} \frac{\partial_h \det \partial_x T_h(\mathbf{x}(t))}{\det \partial_x T_h(\mathbf{x}(t))} \text{ (L'Hôpital's rule) } = -\lim_{h\to 0^+} \partial_h \det \partial_x T_h(\mathbf{x}(t))
\end{aligned}$$

[2] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems* 31 (2018).

Derivations of the ODE for the log probability

Recall Jacobi's formula

$$\frac{d}{dh}\det(A(h)) = \mathrm{trace}(\mathrm{adj}(A(h))\frac{d}{dh}A(h)).$$

Since $\mathrm{adj}(I_d) = I_d$ and $T_h(\mathbf{x}(t)) = \mathbf{x}(t) + h\mathcal{F}_t(\mathbf{x}(t)) + \mathcal{O}(h^2)$, we can derive the final ODE:

Derivations of the ODE for the log probability

Recall Jacobi's formula

$$\frac{d}{dh}\det(A(h)) = \mathrm{trace}(\mathrm{adj}(A(h))\frac{d}{dh}A(h)).$$

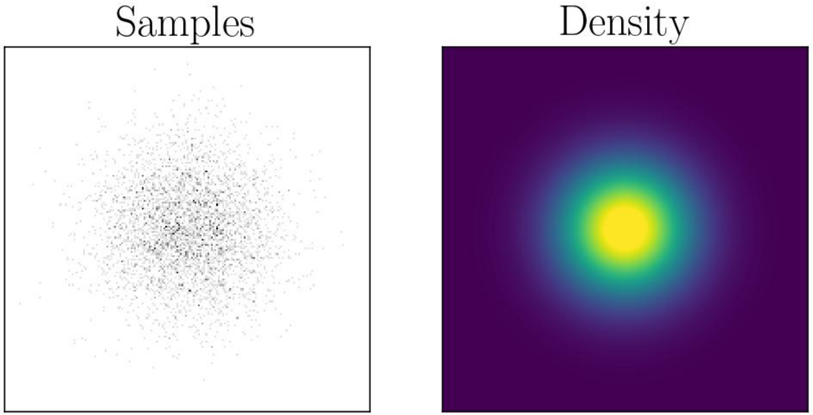
Since $\mathrm{adj}(I_d) = I_d$ and $T_h(\mathbf{x}(t)) = \mathbf{x}(t) + h\mathcal{F}_t(\mathbf{x}(t)) + \mathcal{O}(h^2)$, we can derive the final ODE:

$$\begin{aligned}\frac{d}{dt}\log p_t(\mathbf{x}(t)) &= -\mathrm{trace}\left(\lim_{h\to 0^+}\partial_h\partial_x T_h(\mathbf{x}(t))\right)\\ &= -\mathrm{trace}\left(\lim_{h\to 0^+}\partial_h\left(I + h\partial_x\mathcal{F}_t(\mathbf{x}(t)) + \mathcal{O}(h^2)\right)\right)\\ &= -\mathrm{trace}(\partial_x\mathcal{F}_t(\mathbf{x}(t))) = -\mathrm{div}(\mathcal{F}_t(\mathbf{x}(t))).\end{aligned}$$

Thus, we have that

$$\log p_t^\theta(\mathbf{x}(t)) = \log p_0(\phi_{t,\theta}^{-1}(\mathbf{x}(t))) - \int_0^t \mathrm{div}(\mathcal{F}_s(\phi_{s,\theta}^{-1}(\mathbf{x}(t))))ds.$$

A simulation



A nice and convenient trick: Hutchinson Trace Estimation[3]

If d is large, it is expensive to compute

$$\mathrm{div}(\mathcal{F}_s^\theta)(x) = \sum_{i=1}^d \partial_{x_i}(e_i^\top \mathcal{F}_s^\theta(x)) = \sum_{i=1}^d \frac{d}{d\lambda}\Big|_{\lambda=0} \left(e_i^\top \mathcal{F}_s^\theta(x+\lambda e_i)\right).$$

[3] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models". In: *arXiv preprint arXiv:1810.01367* (2018).

A nice and convenient trick: Hutchinson Trace Estimation[3]

If d is large, it is expensive to compute

$$\mathrm{div}(\mathcal{F}_s^\theta)(x) = \sum_{i=1}^{d} \partial_{x_i}(e_i^\top \mathcal{F}_s^\theta(x)) = \sum_{i=1}^{d} \frac{d}{d\lambda}\Big|_{\lambda=0} \left(e_i^\top \mathcal{F}_s^\theta(x+\lambda e_i)\right).$$

Hutchinson Trick

Consider $\varepsilon \sim \mathcal{N}(0, I_d)$, $\mathbb{E}(\varepsilon\varepsilon^\top) = I_d$. Fix $A \in \mathbb{R}^{d\times d}$. Then we have that

$$\mathrm{trace}(A) = \mathrm{trace}(A\mathbb{E}(\varepsilon\varepsilon^\top)) = \mathbb{E}(\mathrm{trace}(A\varepsilon\varepsilon^\top)) = \mathbb{E}(\mathrm{trace}(\varepsilon^\top A\varepsilon)) \approx \frac{1}{k}\sum_{i=1}^{k} \varepsilon_i^\top A \varepsilon_i$$

$$\mathrm{div}(\mathcal{F}_s^\theta)(x) = \mathrm{trace}(\partial_x \mathcal{F}_s^\theta(x)) \approx \frac{1}{k}\sum_{i=1}^{k} \varepsilon_i^\top (\partial_x \mathcal{F}_s^\theta)(x)\varepsilon_i.$$

[3] Grathwohl, Chen, Bettencourt, Sutskever, and Duvenaud, "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models".

Main drawback of log-likelihood maximisation

Let $p_\theta := p_1^\theta$. To train the continuous normalising flow, via the empirical estimation of the log-likelihood, we have to compute

$$-\frac{1}{N}\sum_{i=1}^{N}\log(p_\theta(x_i)) = -\frac{1}{N}\sum_{i=1}^{N}\left(\log(p_{\text{init}}(\phi_{1,\theta}^{-1}(x_i))) - \int_0^1 \mathrm{div}(\mathcal{F}_s^\theta)(\phi_{s,\theta}^{-1}(x_i))ds\right),$$

where $x_1, ..., x_N \sim p_{\text{data}}$, iid. **This requires simulating the ODE backwards (and forward for the backpropagation) for each forward pass.**

A recent improvement: Flow matching

Assume that there exists a vector field $\mathcal{F} : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ with flow $\phi_t : \mathbb{R}^d \to \mathbb{R}^d$ such that $(\phi_t)_* p_{\text{init}} = p_t$ with $p_1 = p_{\text{data}}$. Then, it would be enough to match it with our network, i.e., to minimise

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{\substack{t \sim \mathcal{U}[0,1], \\ x_t \sim p_t}} \left[\left\| \mathcal{F}_t(x_t) - \mathcal{F}_t^\theta(x_t) \right\|_2^2 \right] \approx \frac{1}{N} \sum_{i=1}^N \left\| \mathcal{F}_{t_i}(x_{t_i}) - \mathcal{F}_{t_i}^\theta(x_{t_i}) \right\|_2^2 .$$

The goal now will be to figure out how $\mathcal{F}_t$ and p_t are defined, and how to get a computationally practical loss function.

See Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky TQ Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. "Flow matching guide and code". In: *arXiv preprint arXiv:2412.06264* (2024) for a very extensive description of this method.