

# Symplectic Neural Flows and Neural ODEs

Davide Murari

Department of Applied Mathematics and Theoretical Physics  
University of Cambridge

[davidemurari.com/notesunivr2025](https://davidemurari.com/notesunivr2025)

dm2011@cam.ac.uk



# Outline

- 1 Classical methods for ODEs
- 2 What is a PINN and how is it trained?
- 3 PINNs for Hamiltonian ODEs: Symplectic Neural Flows
- 4 Neural ODEs

## Classical methods for ODEs

# How do we solve ODEs numerically?

- Solving the initial value problem (IVP)

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

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)

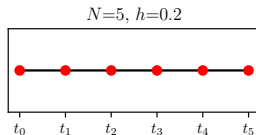
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{x}_0 \quad \text{notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

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  $\varphi_{\mathcal{F}}^h : \mathbb{R}^d \rightarrow \mathbb{R}^d$  is a map

$$\mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^h(\mathbf{y}_n), \quad n = 0, \dots, N-1, \quad (1)$$

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  $\mathcal{F}$ .



# How do we solve ODEs numerically?

- Solving the initial value problem (IVP)

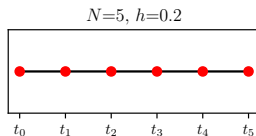
$$\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{x}_0 \quad \text{notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t),$$

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  $\varphi_{\mathcal{F}}^h : \mathbb{R}^d \rightarrow \mathbb{R}^d$  is a map

$$\mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^h(\mathbf{y}_n), \quad n = 0, \dots, N-1, \quad (1)$$

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  $\varphi_{\mathcal{F}}^h$  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  $\{t_0, t_1, \dots, t_N\}$ ,
  - 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

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

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

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

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

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

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  $(t, \mathbf{x}) \in [0, T] \times \Omega$ ,  $\Omega \subset \mathbb{R}^d$ , we can define  $\mathcal{N}_\theta : \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}$  and train it so it almost satisfies the initial/boundary conditions and

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

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  $(t, \mathbf{x}) \in [0, T] \times \Omega$ ,  $\Omega \subset \mathbb{R}^d$ , we can define  $\mathcal{N}_\theta : \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}$  and train it so it almost satisfies the initial/boundary conditions and

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

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  $\mathcal{N}_\theta(\cdot; \mathbf{x}_0) : [0, T] \rightarrow \mathbb{R}^d$ , and choose its weights so that

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

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

# 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  $\mathcal{N}_\theta(\cdot; \mathbf{x}_0) : [0, T] \rightarrow \mathbb{R}^d$ , and choose its weights so that

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

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

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

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  $c_1, \dots, c_s \in [0, 1]$  and define the degree  $s$  polynomial  $p(\cdot; \mathbf{x}_0) : \mathbb{R} \rightarrow \mathbb{R}^d$ ,

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

such that

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

## PINN

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

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

such that  $\theta^*$  minimises

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

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

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

for  $C$  collocation points  $0 \leq t_1 < \dots < t_C \leq \Delta t$  defining a quadrature rule of order  $p$ .

Then, there exist  $\alpha, \beta > 0$  such that

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

# 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:

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

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

- 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.

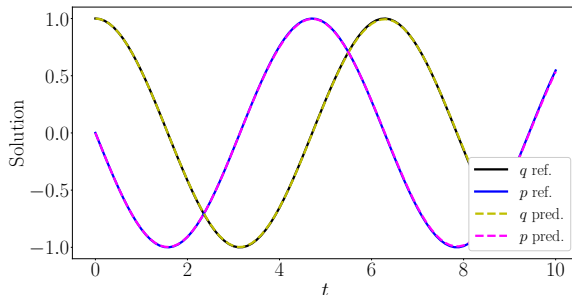


Figure 1: Solution comparison after reaching a loss value of  $10^{-5}$ . The training time is 87 seconds (7500 epochs with 1000 new collocation points randomly sampled at each of them).

# Integration over long time intervals

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

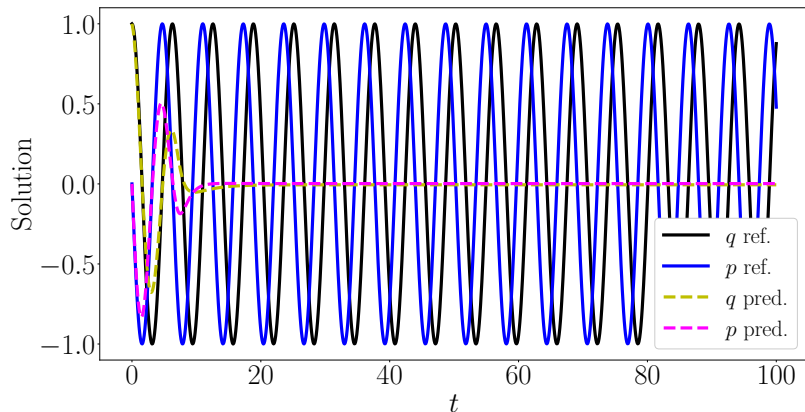


Figure 2: Solution comparison after 10000 epochs.

## Forward invariant subset of the phase space

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

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

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

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

# Forward invariant subset of the phase space

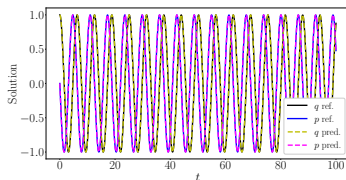
- Consider the vector field  $\mathcal{F} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ , and introduce notation  $\phi_{\mathcal{F}}^t : \mathbb{R}^d \rightarrow \mathbb{R}^d$  for the time- $t$  flow map of  $\mathcal{F}$ , which for every  $\mathbf{x}_0 \in \mathbb{R}^d$  satisfies

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

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

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

- Thus, to approximate  $\phi_{\mathcal{F}}^t : \Omega \rightarrow \Omega$  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  $\phi_{H,t} : \mathbb{R}^{2n} \rightarrow \mathbb{R}^{2n}$  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<sup>1</sup>

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

---

<sup>1</sup>Priscilla Canizares et al. “Symplectic neural flows for modeling and discovery”. In: *arXiv preprint arXiv:2412.16787* (2024).

# The SympFlow architecture<sup>1</sup>

- We now build a neural network that approximates  $\phi_{H,t} : \Omega \rightarrow \Omega$  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:

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

---

<sup>1</sup>Canizares et al., “Symplectic neural flows for modeling and discovery”.

# The SympFlow architecture<sup>1</sup>

- We now build a neural network that approximates  $\phi_{H,t} : \Omega \rightarrow \Omega$  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:

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

- The SympFlow architecture is defined as

$$\mathcal{N}_{\theta}(t, (\mathbf{q}_0, \mathbf{p}_0)) = \phi_{\mathbf{p},t}^L \circ \phi_{\mathbf{q},t}^L \circ \dots \circ \phi_{\mathbf{p},t}^1 \circ \phi_{\mathbf{q},t}^1((\mathbf{q}_0, \mathbf{p}_0)),$$

with

$$V^i(t, \mathbf{q}) = \ell_{\theta_3^i} \circ \sigma \circ \ell_{\theta_2^i} \circ \sigma \circ \ell_{\theta_1^i} \left( \begin{bmatrix} \mathbf{q} \\ t \end{bmatrix} \right), \quad K^i(t, \mathbf{p}) = \ell_{\rho_3^i} \circ \sigma \circ \ell_{\rho_2^i} \circ \sigma \circ \ell_{\rho_1^i} \left( \begin{bmatrix} \mathbf{p} \\ t \end{bmatrix} \right)$$
$$\ell_{\theta_k^i}(\mathbf{x}) = A_k^i \mathbf{x} + \mathbf{a}_k^i, \quad \ell_{\rho_k^i}(\mathbf{x}) = B_k^i \mathbf{x} + \mathbf{b}_k^i, \quad k = 1, 2, 3, \quad i = 1, \dots, L.$$

---

<sup>1</sup>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:

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

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

# 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:

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

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

- 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  $\tilde{V}^i, \tilde{K}^i : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}$  with feed-forward neural networks.
- To train the overall model  $\mathcal{N}_\theta$  we minimise the loss function

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

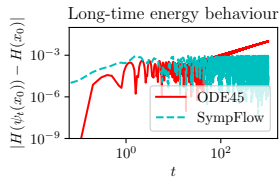
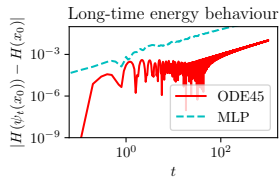
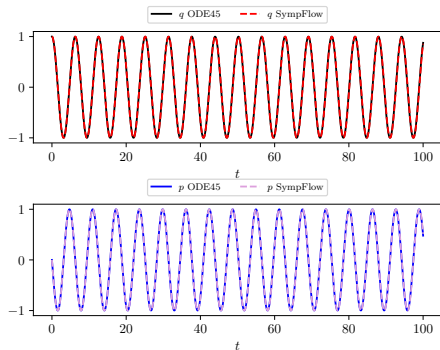
where we sample  $t_i \in [0, \Delta t]$ , and  $\mathbf{x}_0^i \in \Omega \subset \mathbb{R}^{2n}$ .

# Simple Harmonic Oscillator (unsupervised)

## Equations of motion

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

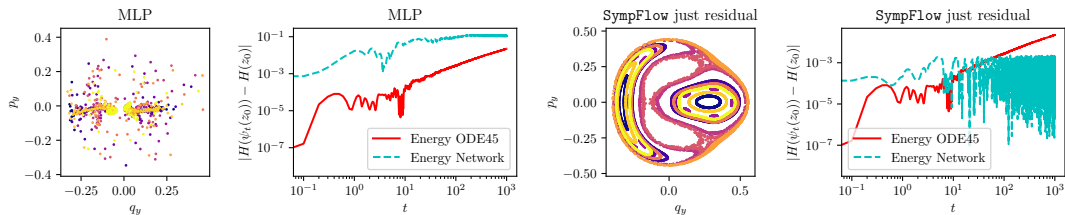
Solution predicted using SympFlow with Hamiltonian Matching



# Hénon–Heiles (unsupervised)

## Equations of motion

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



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



# Neural ODEs

# 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

$$\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(t, \mathbf{x}(t)), \quad \theta \in \Theta,$$

where  $\mathcal{F}_\theta : \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}^d$  is parametrised by a neural network.

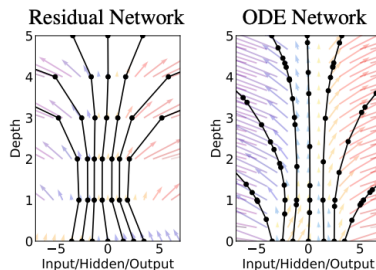


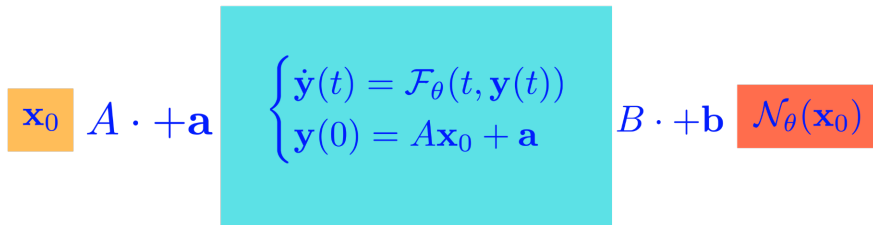
Figure 4: Source: **chen2018neural**.

# Neural ODEs

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

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

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  $\mathbf{b} \in \mathbb{R}^c$ .


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

# How to train them: discrete backpropagation vs. adjoint method

- There are two main strategies to train Neural ODEs:
  - ① Discretise backpropagation, and
  - ② Adjoint method.
- The first, corresponds to the conventional backpropagation algorithm, where the forward pass is defined through a numerical method:

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

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

# The adjoint sensitivity method

- For simplicity, fix  $d = c$ , and consider Neural ODEs of the form

$$\mathcal{N}_\theta(\mathbf{x}_0) = \mathbf{x}(T) = \mathbf{x}_0 + \int_0^T \mathcal{F}_\theta(t, \mathbf{x}(t)) dt, \quad A = B = I_d, \quad \mathbf{a} = \mathbf{b} = 0.$$

- Let us introduce a loss function  $L : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+$ , and study the gradient  $\nabla_\theta L(\mathcal{N}_\theta(\mathbf{x}_0), \mathbf{y})$ .
- First, we introduce the so-called adjoint variable

$$\mathbf{a}(t) = \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t)} \in \mathbb{R}^d.$$

Assuming to know  $\mathbf{x}(T)$ , we see that  $\mathbf{a}(T)$  is known as well. What about a generic  $\mathbf{a}(t)$  for  $t \in [0, T)$ ?

# The adjoint sensitivity method

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

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

Furthermore, by the chain rule we get

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

- This allows us to obtain that

$$\frac{d}{dt} \mathbf{a}(t) = \lim_{\varepsilon \rightarrow 0} \frac{\mathbf{a}(t + \varepsilon) - \mathbf{a}(t)}{\varepsilon} = \dots = - \left( \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)} \right)^\top \mathbf{a}(t).$$

- It follows that, for  $t \in [0, T]$ :

$$\mathbf{a}(t) = \mathbf{a}(T) - \int_T^t \left( \frac{\partial \mathcal{F}_\theta(s, \mathbf{x}(s))}{\partial \mathbf{x}(s)} \right)^\top \mathbf{a}(s) ds.$$

# The adjoint sensitivity method

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

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

- 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

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

# 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 **choromanski2020ode**, the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

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



# 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 **choromanski2020ode**, the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

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

- In **norcliffe2020second**, the authors consider second order Neural ODEs

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

# 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

simpleH0 = 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(simpleH0, 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()

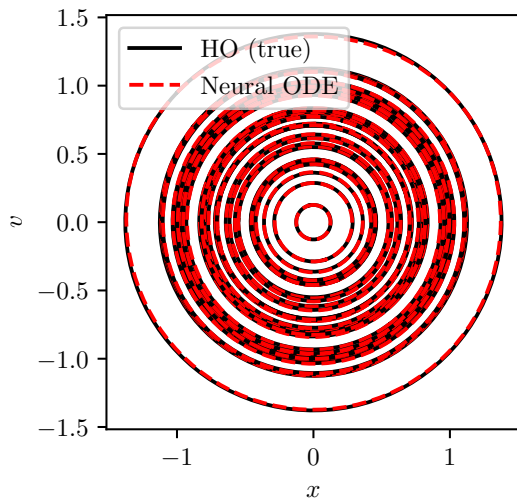
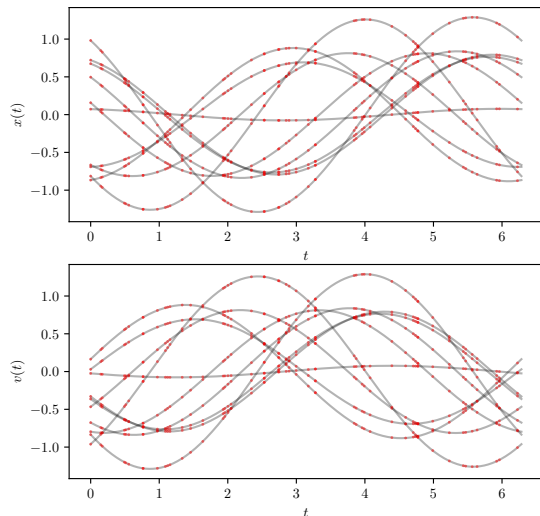
class ODEFunc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(2, 32), nn.Tanh(), nn.Linear(32, 2))
    def forward(self, t, y):
        return self.net(y)

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



# 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.

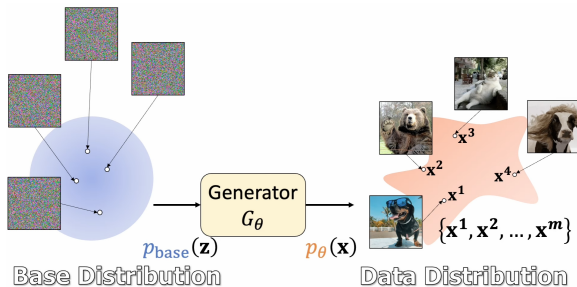


Figure 6: Source: [https://www.youtube.com/watch?v=DDq\\_pIfHqLs&ab\\_channel=Jia-BinHuang](https://www.youtube.com/watch?v=DDq_pIfHqLs&ab_channel=Jia-BinHuang).

## Second application of Neural ODEs: Generative Modelling

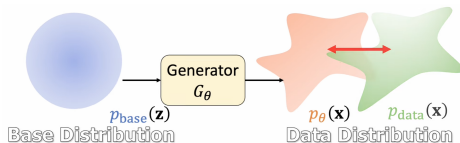


Figure 7: Source: [https://www.youtube.com/watch?v=DDq\\_pIfHqLs&ab\\_channel=Jia-BinHuang](https://www.youtube.com/watch?v=DDq_pIfHqLs&ab_channel=Jia-BinHuang).

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

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

$$\text{Empirically: } \arg \min_{\theta} -\frac{1}{N} \sum_{i=1}^N \log(p_\theta(\mathbf{x}_i)), \quad \mathbf{x}_1, \dots, \mathbf{x}_N \sim p_{\text{data}}, \text{ iid.}$$

# Generative modelling with continuous normalising flows

Consider a neural ODE

$$\begin{cases} \frac{d}{dt}\phi_{t,\theta}(z) = \mathcal{F}_t^\theta(\phi_{t,\theta}(z)), & \mathcal{F}^\theta : [0, 1] \times \mathbb{R}^d \rightarrow \mathbb{R}^d, \\ \phi_0(z) = z, \end{cases}$$

and an easy-to-sample probability measure with density  $p_{\text{init}}$ . We then set  $x = \phi_1(z)$ .

Continuous normalising flows define  $p_t^\theta = (\phi_{t,\theta})_* p_{\text{init}}$ ,  $t \in [0, 1]$ , as

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

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