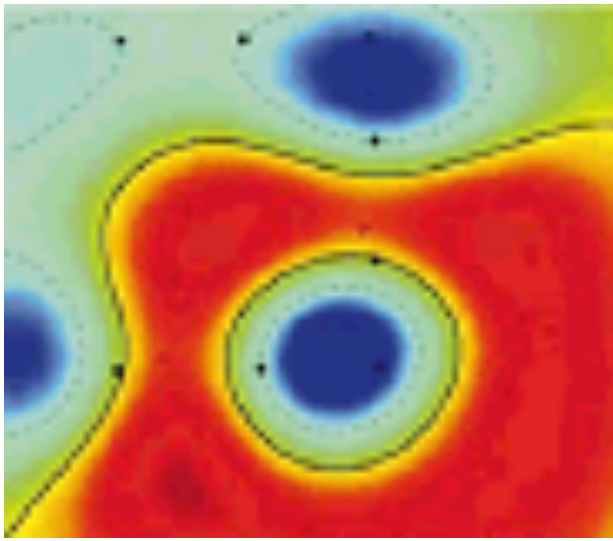
SoSe 2023

Deep Learning 2

Technische Universität Berlin



Lecture 7 | Neural ODEs

Overview

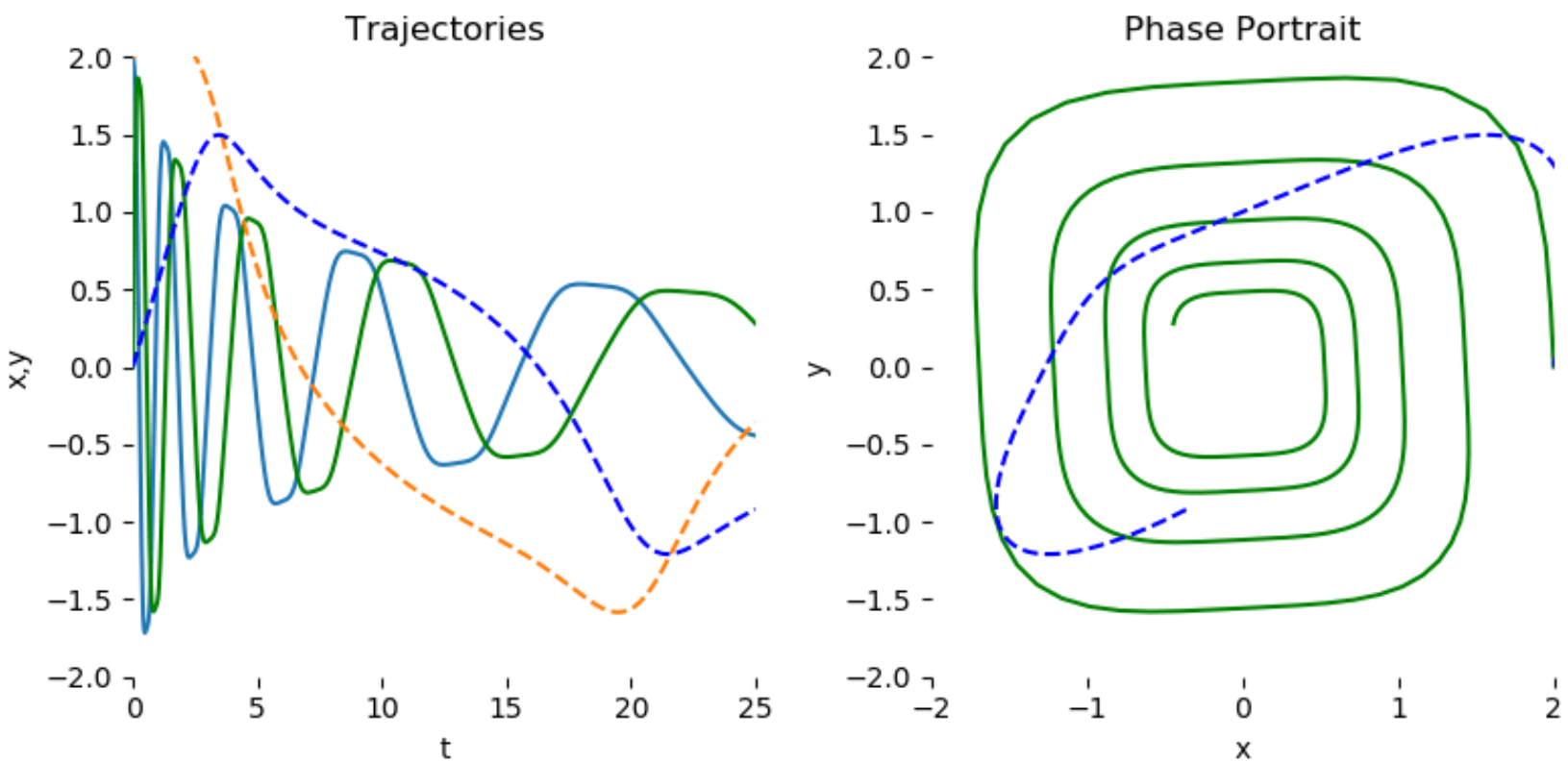
Introduction to ODEs

Neural ODEs

Sensitivity methods

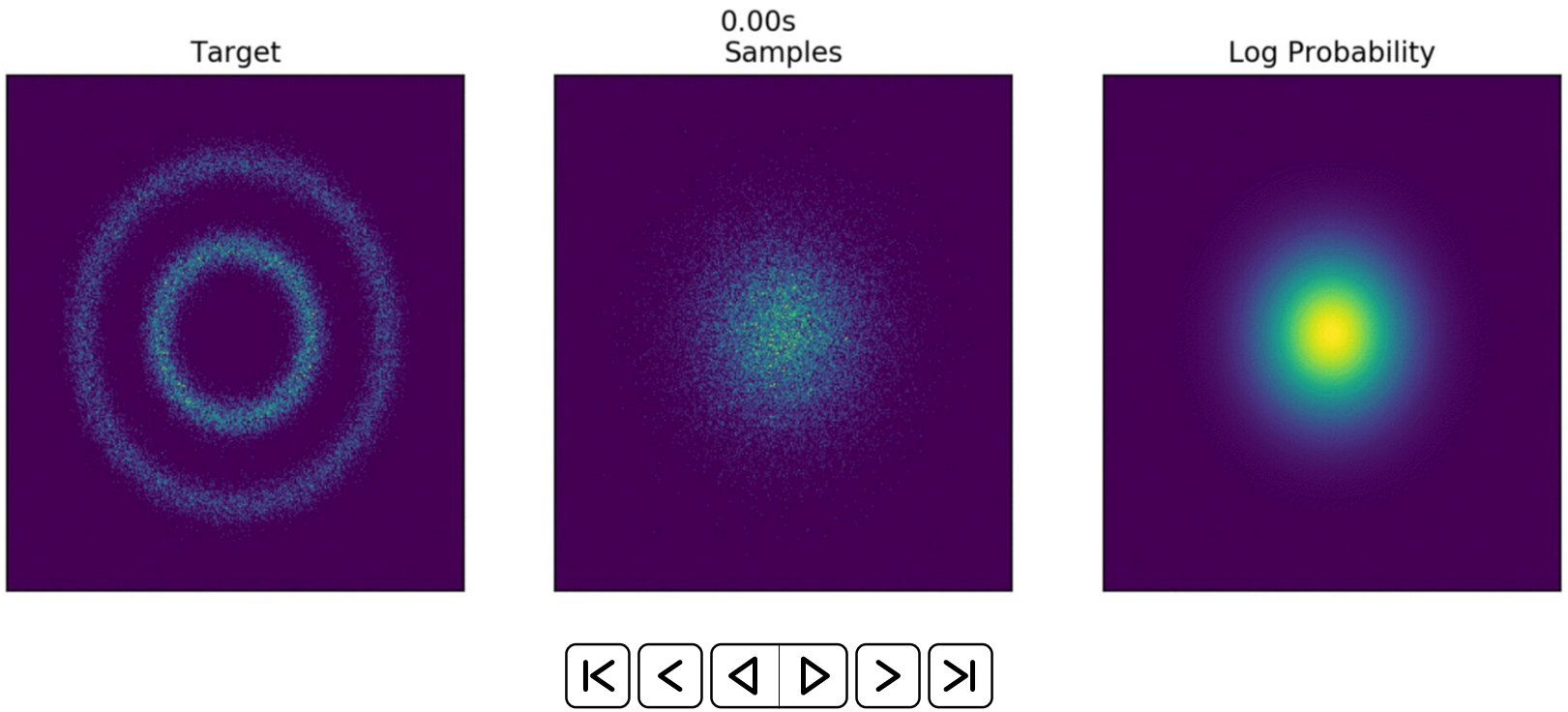
Applications

Motivation: Trajectory optimization



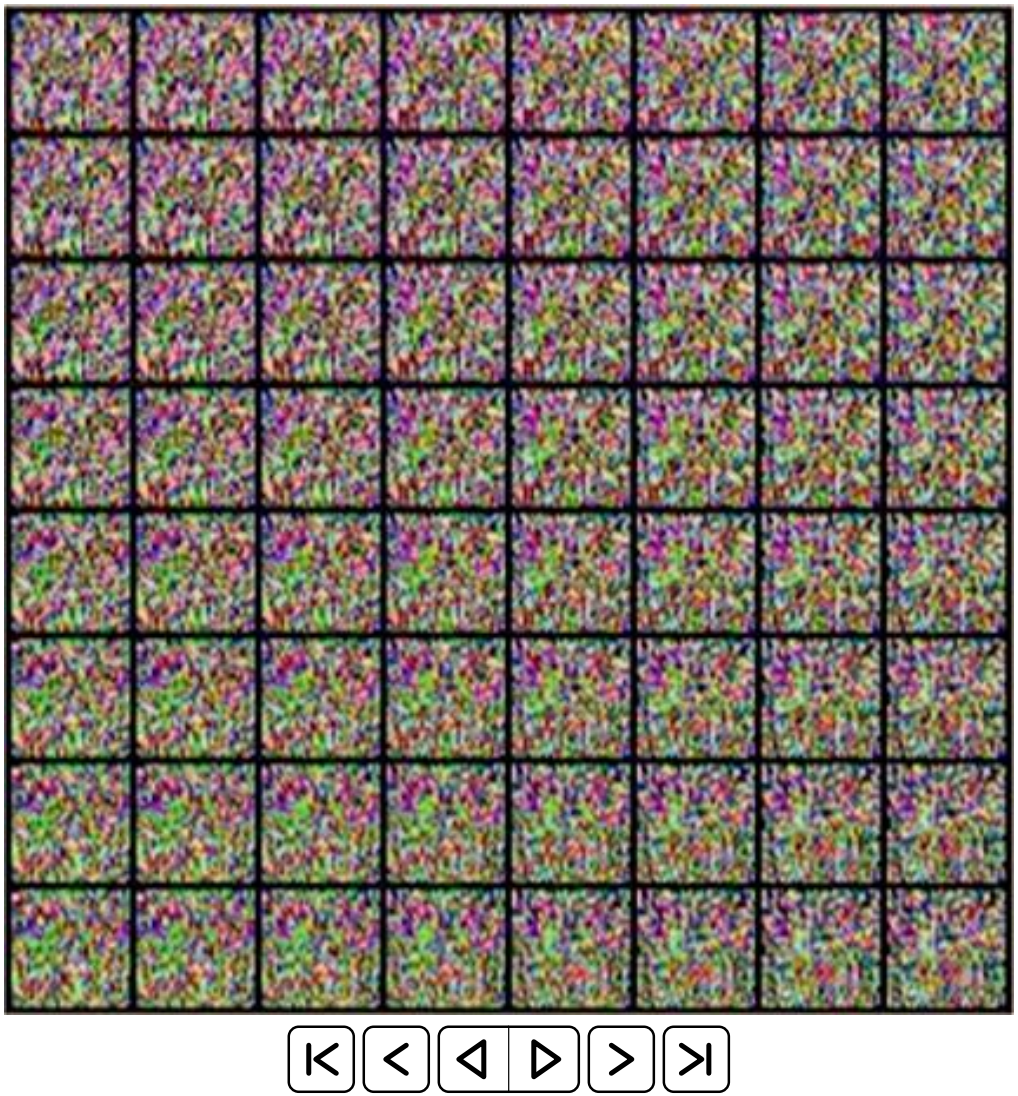
Source: [Che18]

Motivation: Continuous Normalizing Flows



Source: [CRBD18, Che18]

Motivation: Continuous Normalizing Flows



Source: [CRBD18]

Introduction to ODEs

What are Ordinary Differential Equations?

Differential Equations relate functions and their derivatives.

In this lecture: relate states $\underline{u}$ and their rate of change over time t

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad t \in \mathcal{T}$$

- $\mathcal{T} \subset \mathbb{R}$ is the time domain, usually $\mathcal{T} = [0, T]$
- $\underline{u} : \mathcal{T} \to \mathbb{R}^n$ is the unknown function of the *state* over time
- $\underline{f} : \mathcal{T} \times \mathbb{R}^n \to \mathbb{R}^n$ is the *vector field*

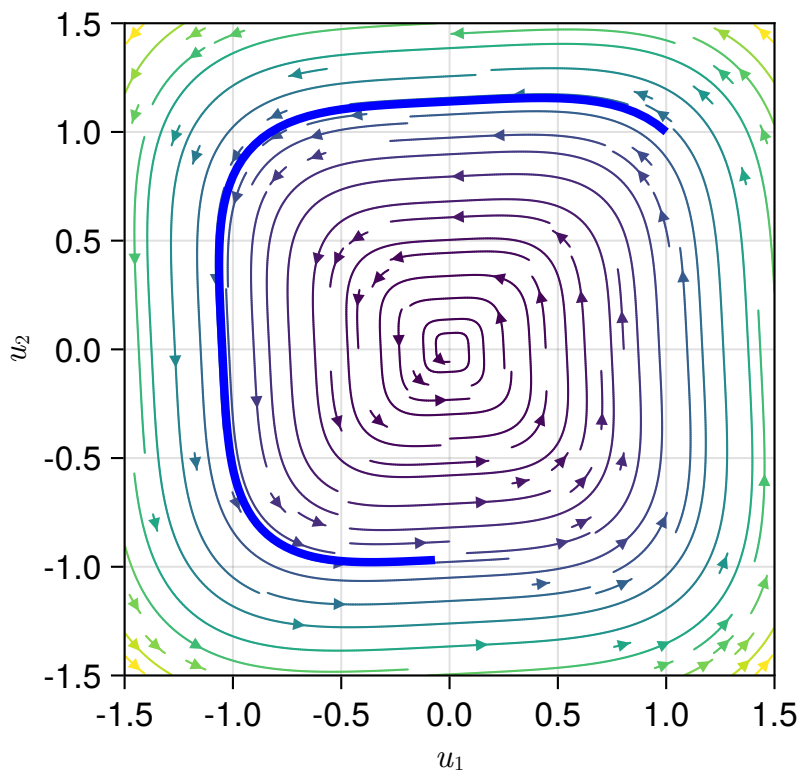
What are Ordinary Differential Equations?

Differential Equations relate functions and their derivatives.

In this lecture: relate states $\underline{u}$ and their rate of change over time t

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad t \in \mathcal{T}$$

- $\mathcal{T} \subset \mathbb{R}$ is the time domain, usually $\mathcal{T} = [0, T]$
- $\underline{u} : \mathcal{T} \to \mathbb{R}^n$ is the unknown function of the *state* over time
- $\underline{f} : \mathcal{T} \times \mathbb{R}^n \to \mathbb{R}^n$ is the *vector field*



Types of Differential Equations

Ordinary Differential Equations (ODEs)

Relate functions of one independent variable (e.g. t) and their derivatives:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right)$$

Partial Differential Equations (PDEs)

Relate multivariate functions and their partial derivatives, e.g. for $\underline{u}(t, x)$:

$$\frac{\partial \underline{u}(t,x)}{\partial t} + c\frac{\partial \underline{u}(t,x)}{\partial x} = 0$$

Stochastic Differential Equations (SDEs)

Diff. eq. that contains a stochastic process, e.g. *Wiener process* W:

$$\mathrm{d}\underline{u} = \underline{f}\left(t, \underline{u}(t)\right)\mathrm{d}t + \underline{g}\left(t, \underline{u}(t)\right)\mathrm{d}W$$

- Basis of diffusion models [SSDK+20]

Types of Differential Equations

Ordinary Differential Equations (ODEs)

Relate functions of one independent variable (e.g. t) and their derivatives:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right)$$

Partial Differential Equations (PDEs)

Relate multivariate functions and their partial derivatives, e.g. for $\underline{u}(t, x)$:

$$\frac{\partial \underline{u}(t,x)}{\partial t} + c\frac{\partial \underline{u}(t,x)}{\partial x} = 0$$

Stochastic Differential Equations (SDEs)

Diff. eq. that contains a stochastic process, e.g. *Wiener process* W:

$$\mathrm{d}\underline{u} = \underline{f}\left(t, \underline{u}(t)\right)\mathrm{d}t + \underline{g}\left(t, \underline{u}(t)\right)\mathrm{d}W$$

- Basis of diffusion models [SSDK+20]

Types of Ordinary Differential Equations

Ordinary Differential Equations can be classified further:

- linear / non-linear
- autonomous / non-autonomous
- homogeneous / inhomogeneous

Autonomous ODE

Function $\underline{f}$ doesn't *explicitly* depend on the independent variable:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(\underline{u}(t)\right)$$

Also called *time-invariant* if the independent variable is the time.

Types of Ordinary Differential Equations

Ordinary Differential Equations can be classified further:

- linear / non-linear
- autonomous / non-autonomous
- homogeneous / inhomogeneous

Autonomous ODE

Function $\underline{f}$ doesn't *explicitly* depend on the independent variable:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(\underline{u}(t)\right)$$

Also called *time-invariant* if the independent variable is the time.

Initial Value Problems

An initial value problem (IVP) consists of a differential equation and an initial condition:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad \underline{u}(t_0) = \underline{u}_0$$

Initial Value Problems

An initial value problem (IVP) consists of a differential equation and an initial condition:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad \underline{u}(t_0) = \underline{u}_0$$

The existence and uniqueness of a solution is given by:

Picard-Lindelöf theorem

Let $\underline{f} : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t and uniformly Lipschitz in $\underline{u}$. Let $\underline{u}_0 \in \mathbb{R}^d$. Then there exists a unique differentiable $\underline{u} : [0, T] \to \mathbb{R}^d$ satisfying

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

IVP implicitly defines a trajectory $\underline{u}(t)$ through its solution using the fundamental theorem of calculus:

$$\underline{u}(t) = \underline{u}_0 + \int_{t_0}^{t} \underline{f}\left(\tau, \underline{u}(\tau)\right) \mathrm{d}\tau$$

Initial Value Problems

An initial value problem (IVP) consists of a differential equation and an initial condition:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad \underline{u}(t_0) = \underline{u}_0$$

The existence and uniqueness of a solution is given by:

Picard-Lindelöf theorem

Let $\underline{f} : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t and uniformly Lipschitz in $\underline{u}$. Let $\underline{u}_0 \in \mathbb{R}^d$. Then there exists a unique differentiable $\underline{u} : [0, T] \to \mathbb{R}^d$ satisfying

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

IVP implicitly defines a trajectory $\underline{u}(t)$ through its solution using the fundamental theorem of calculus:

$$\underline{u}(t) = \underline{u}_0 + \int_{t_0}^{t} \underline{f}\left(\tau, \underline{u}(\tau)\right) \mathrm{d}\tau$$

➔ Requires numerical integration from $\underline{u}_0$

Numerical Integration: Euler method

The Taylor series expansion of u around t_0 corresponds to

$$u(t_0 + \Delta t) = u(t_0) + u'(t_0)\Delta t + \frac{1}{2}u''(t_0)(\Delta t)^2 + \ldots = \sum_{n=0}^{\infty} \frac{u^{(n)(t_0)}}{n!}(\Delta t)^n \quad .$$

Ignoring higher-order terms, we obtain

$$\begin{aligned} u(t_0 + \Delta t) &\approx u(t_0) + u'(t_0)\Delta t \\ &\approx u(t_0) + \Delta t \cdot f\left(t_0, u(t_0)\right) \quad . \end{aligned}$$

Numerical Integration: Euler method

The Taylor series expansion of u around t_0 corresponds to

$$u(t_0 + \Delta t) = u(t_0) + u'(t_0)\Delta t + \frac{1}{2}u''(t_0)(\Delta t)^2 + \ldots = \sum_{n=0}^{\infty} \frac{u^{(n)(t_0)}}{n!}(\Delta t)^n \quad .$$

Ignoring higher-order terms, we obtain

$$\begin{aligned} u(t_0 + \Delta t) &\approx u(t_0) + u'(t_0)\Delta t \\ &\approx u(t_0) + \Delta t \cdot f\left(t_0, u(t_0)\right) \quad . \end{aligned}$$

Discretizing the dynamics and extrapolating step by step, we obtain:

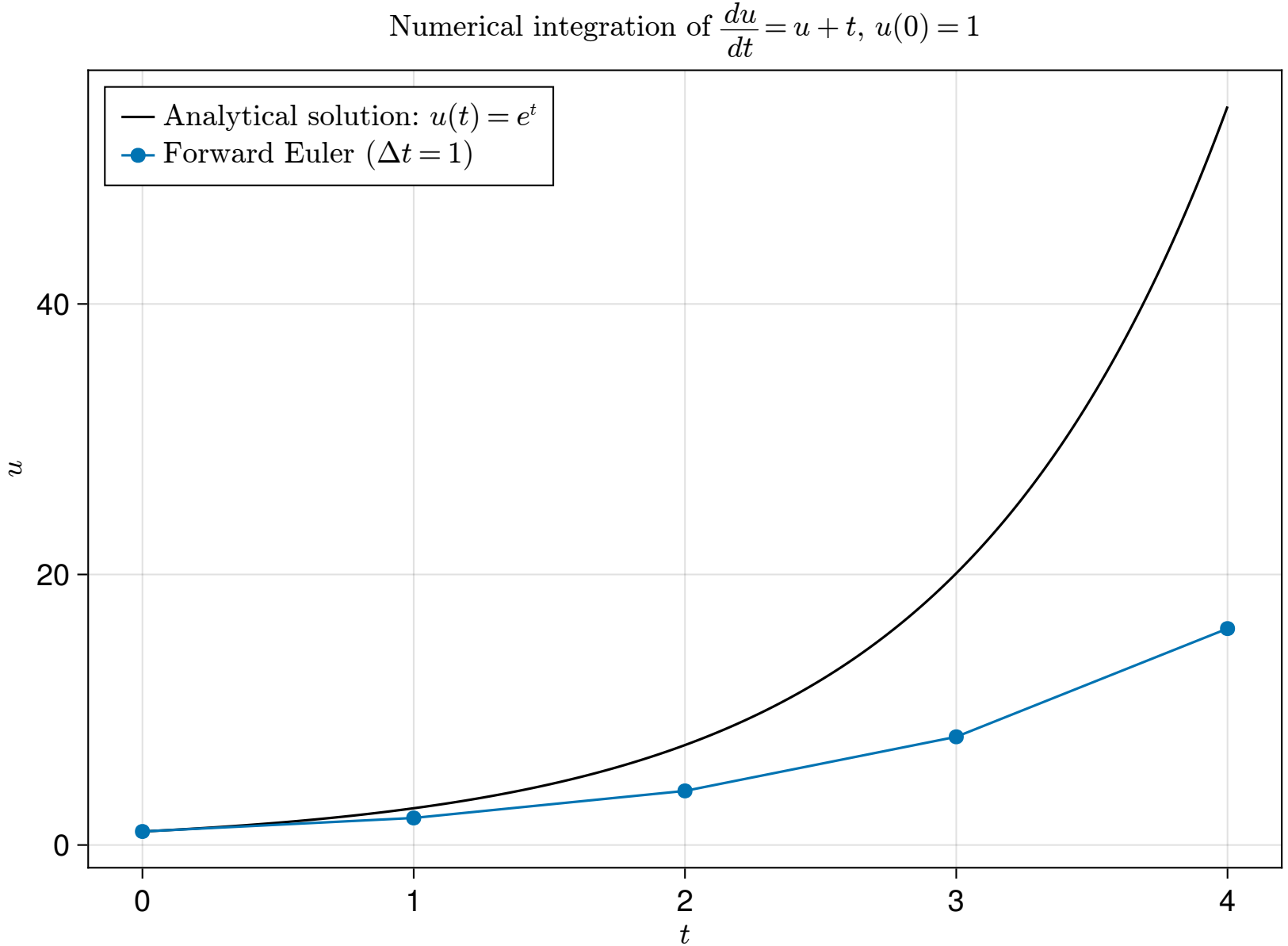
Forward Euler method

Using a constant step size Δt:

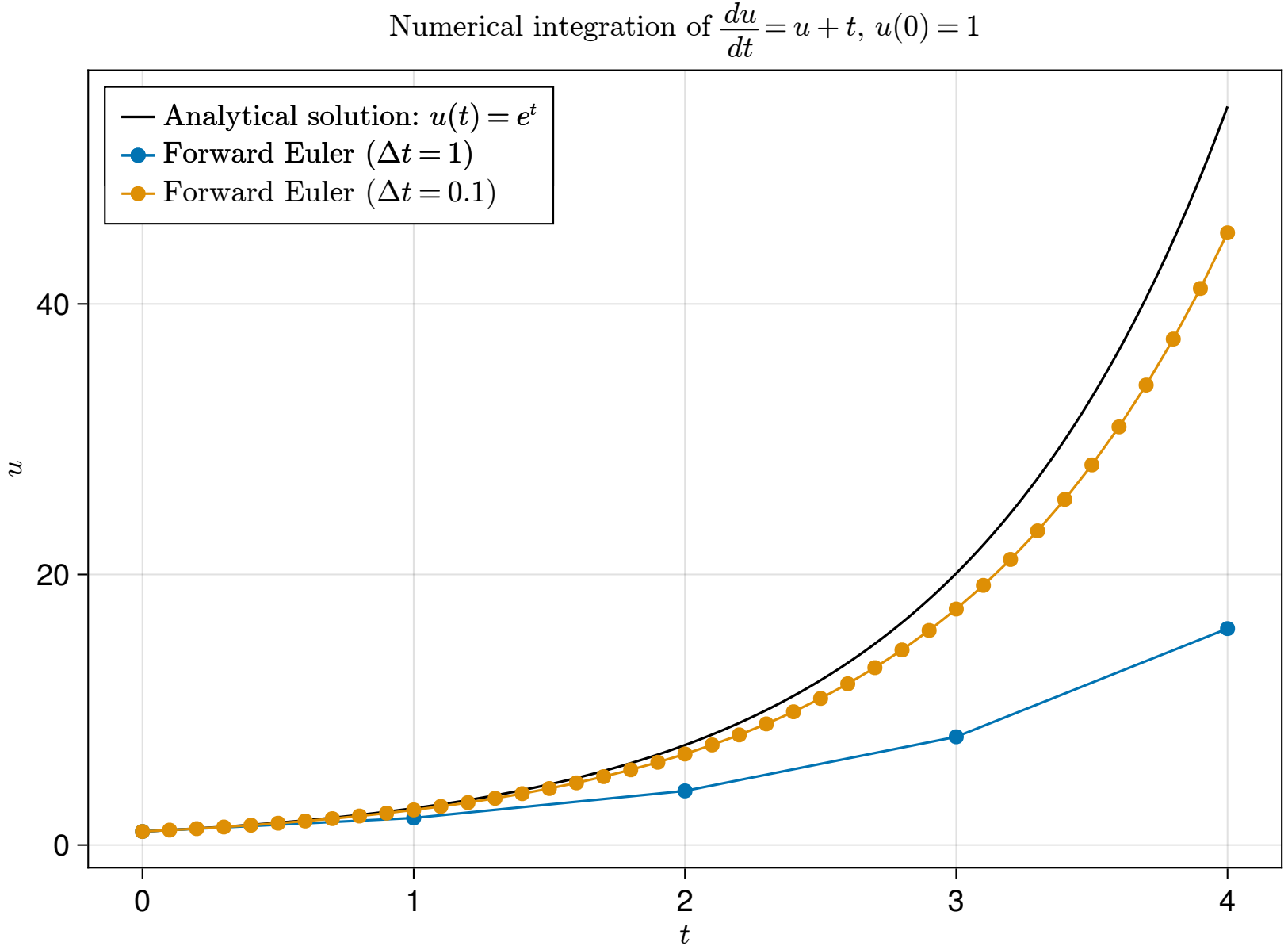
$$\begin{aligned} t_{k+1} &= t_k + \Delta t \\ \underline{u}_{k+1} &= \underline{u}_k + \Delta t \cdot \underline{f}\left(t_k, \underline{u}_k\right) \end{aligned}$$

➔ Total accumulated error of order $\mathcal{O}(\Delta t)$

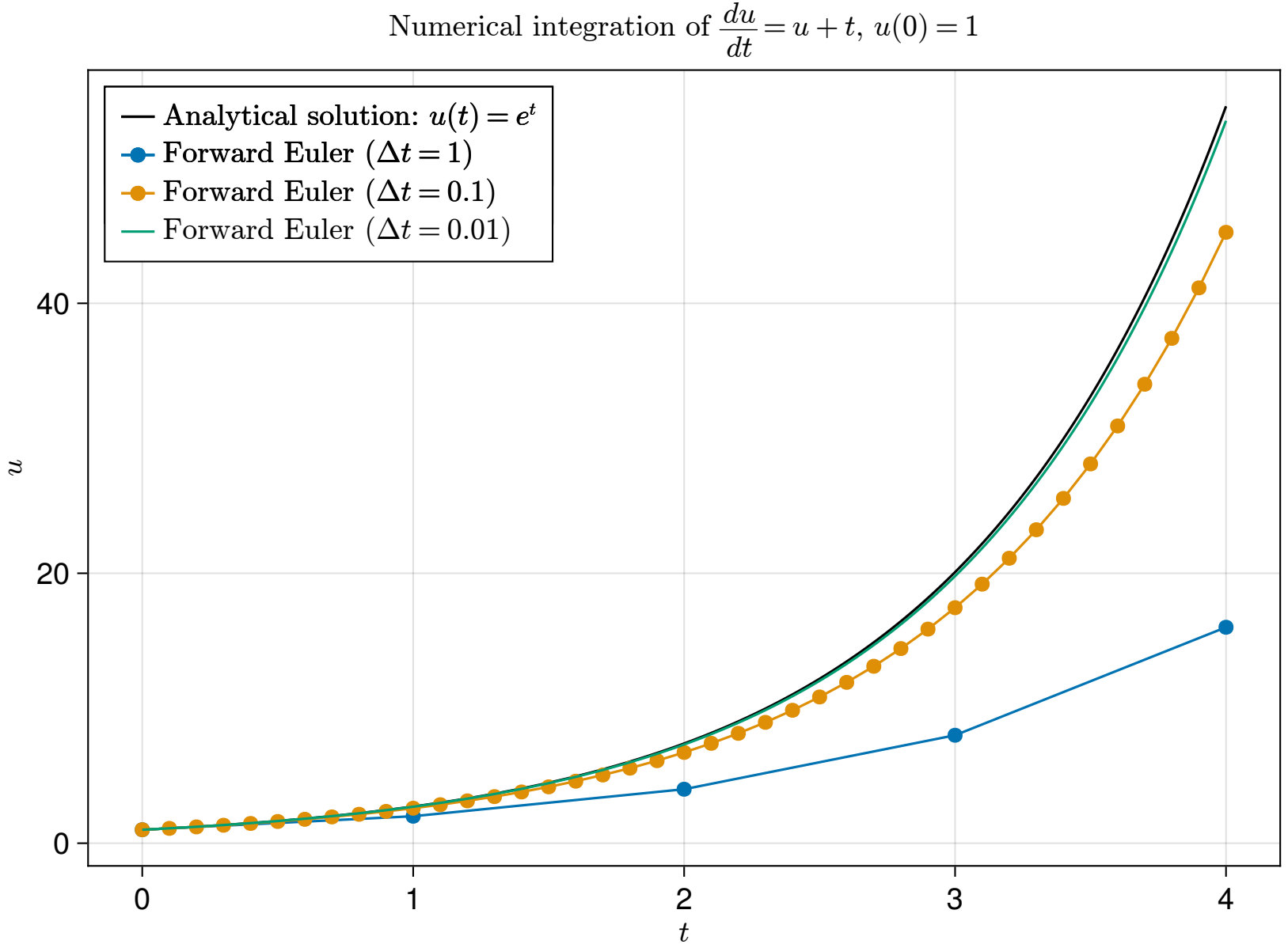
Numerical Integration: Step size Δt



Numerical Integration: Step size Δt



Numerical Integration: Step size Δt



Numerical Integration: Runge-Kutta methods

Runge-Kutta methods are a family of solvers, the most famous of which is the 4th order method.

Runge–Kutta 4

$$
\begin{aligned}
t_{k+1} &= t_k + \Delta t \\
k_1 &= \underline{f}\left(t_k,\, \underline{u}_k\right) \\
k_2 &= \underline{f}\left(t_k + \tfrac{1}{2}\Delta t,\, \underline{u}_k + \tfrac{1}{2}\Delta t\, k_1\right) \\
k_3 &= \underline{f}\left(t_k + \tfrac{1}{2}\Delta t,\, \underline{u}_k + \tfrac{1}{2}\Delta t\, k_2\right) \\
k_4 &= \underline{f}\left(t_k + \Delta t,\, \underline{u}_k + \Delta t\, k_3\right) \\
\underline{u}_{k+1} &= \underline{u}_k + \tfrac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

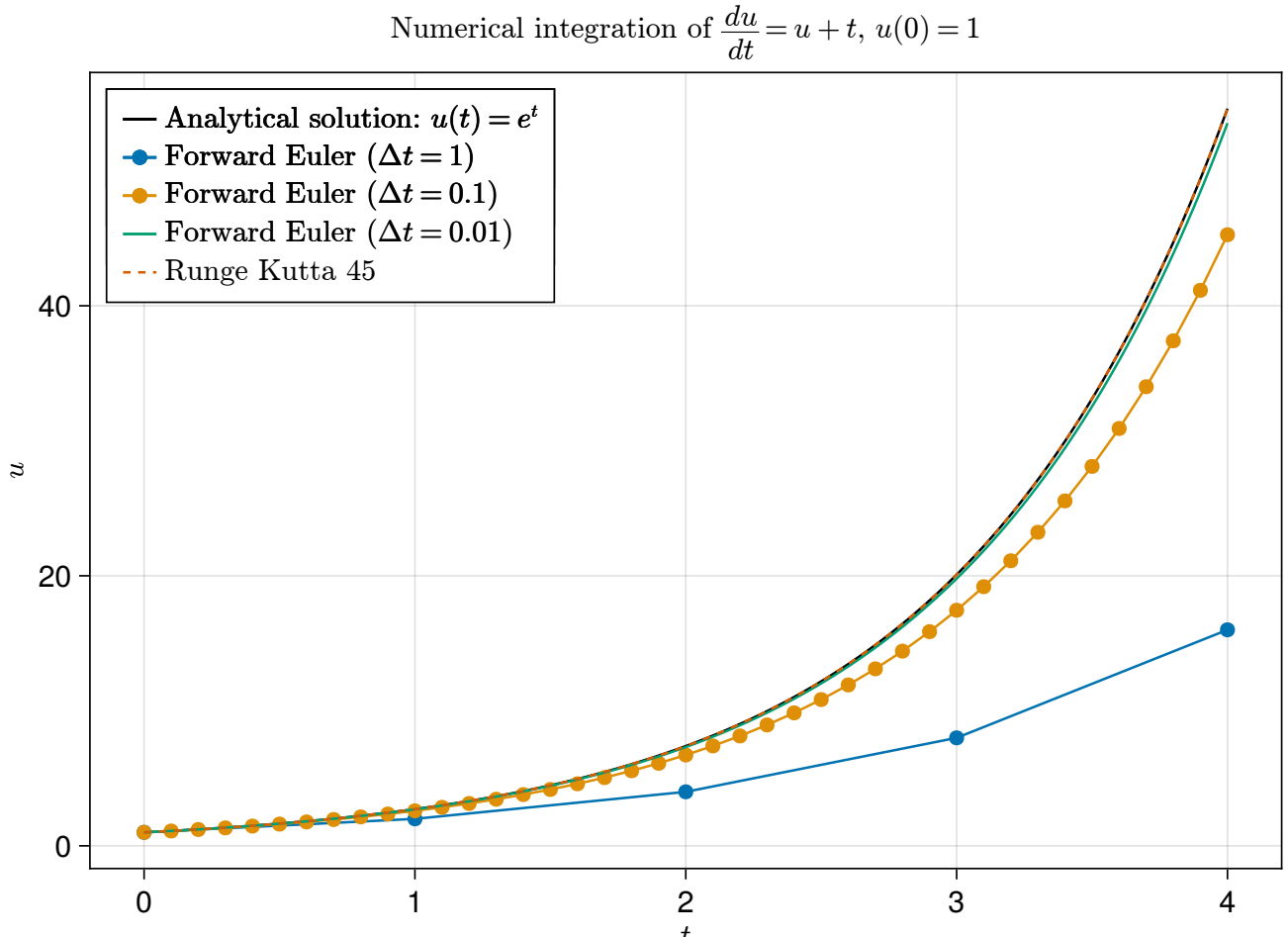
The coefficients can be derived from a 4th order Taylor series expansion.
➔ Total accumulated error of order $\mathcal{O}(\Delta t^4)$

Numerical Integration: Runge-Kutta methods

Runge-Kutta methods can be extended to use adaptive step-sizes

Example: Runge-Kutta 45 method, also called *Dormand–Prince method*

1. apply Runge-Kutta method of 4th and 5th order
2. use 4th order solution to estimate error and set step size
3. step with 5th order solution



Neural ODEs

Neural ODEs

Most common form of Neural ODE [CRBD18]:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}_\theta\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

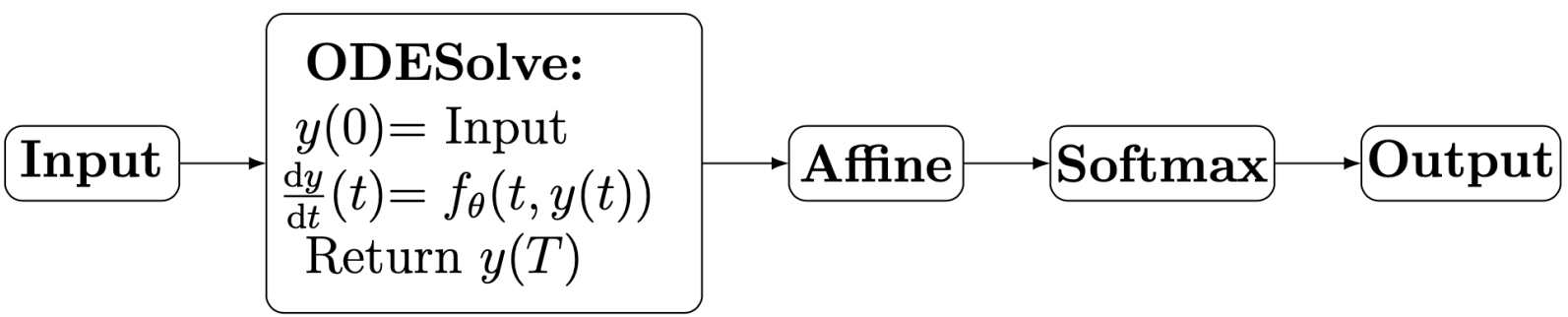
- $\underline{f}_\theta$: neural network parametrized by θ
  - describes vector field
  - typically small and simple architecture
- $\underline{u}_0$: input tensor
- Bounds of integration:
  - given when fitting Neural ODE to time series data
  - otherwise t can be chosen, usually set to $t \in [0, 1]$
- IVP defines a trajectory $\underline{u}_\theta(t)$

Neural ODEs

Most common form of Neural ODE [CRBD18]:

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}_\theta\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

- $\underline{f}_\theta$: neural network parametrized by θ
  - describes vector field
  - typically small and simple architecture
- $\underline{u}_0$: input tensor
- Bounds of integration:
  - given when fitting Neural ODE to time series data
  - otherwise t can be chosen, usually set to $t \in [0, 1]$
- IVP defines a trajectory $\underline{u}_\theta(t)$



Example architecture [Kid22]

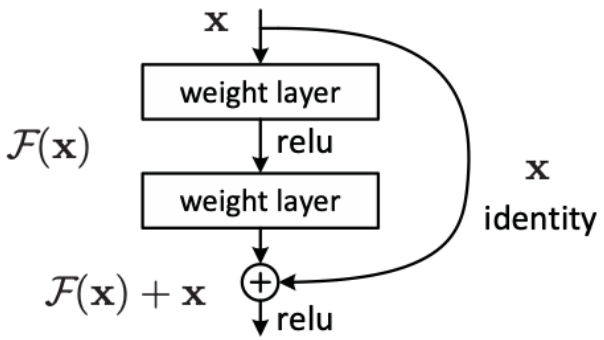
Neural ODEs as Implicit layer models

Connection to ResNets [HR17]:

- from Euler discretization

$$\underline{u}_{k+1} = \underline{u}_k + \Delta t \cdot \underline{f}_\theta \left(t_k, \underline{u}_k\right)$$

- merging the constant term Δt into f_θ, this looks exactly like a ResNet!
- each evaluation of f_θ in the integrator can be seen as a call to an *implicit layer*



ResNet block [HZRS16]

[1] NeurIPS 2020 Tutorial: `https://implicit-layers-tutorial.org`

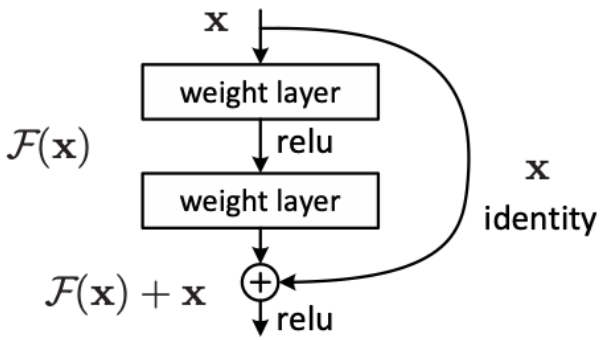
Neural ODEs as Implicit layer models

Connection to ResNets [HR17]:

- from Euler discretization

$$\underline{u}_{k+1} = \underline{u}_k + \Delta t \cdot \underline{f}_\theta \left(t_k, \underline{u}_k\right)$$

- merging the constant term Δt into f_θ, this looks exactly like a ResNet!
- each evaluation of f_θ in the integrator can be seen as a call to an *implicit layer*



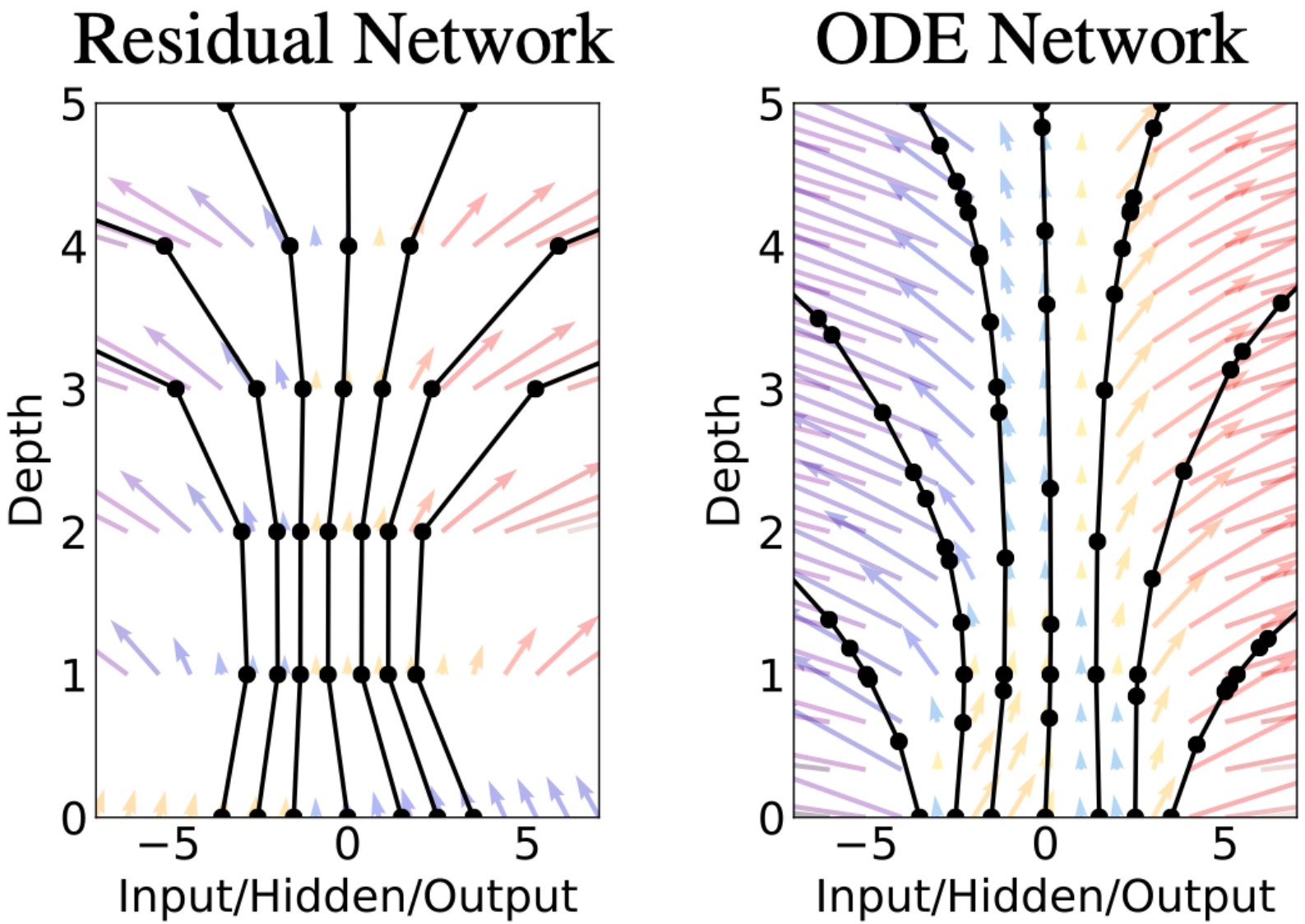
ResNet block [HZRS16]

Other *implicit layer models* [1]:

- Deep Equilibrium models [BKK19, BKK20]
- Differentiable optimization layers [Amo19]

[1] NeurIPS 2020 Tutorial: `https://implicit-layers-tutorial.org`

Neural ODEs as Implicit layer models



Source: [CRBD18]

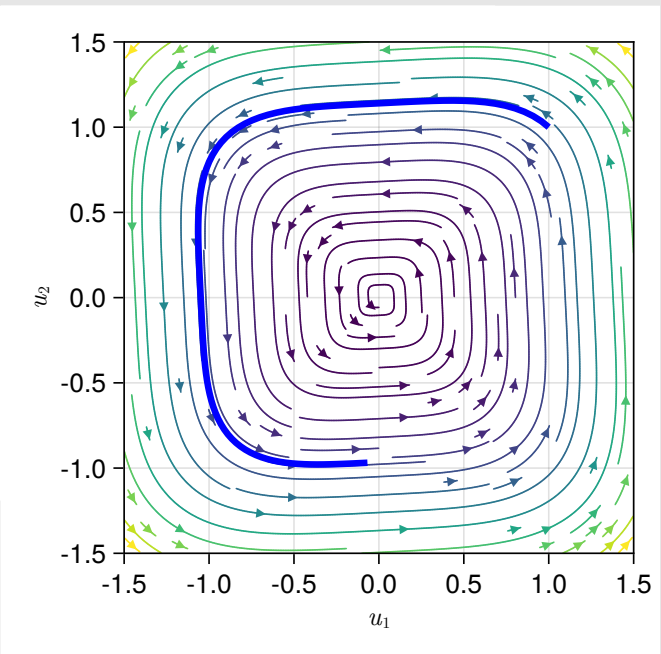
Properties of Neural ODEs: Invertability

Neural ODEs are invertible:

- Neural ODEs describe vector fields
- Trajectories in vector fields never intersect

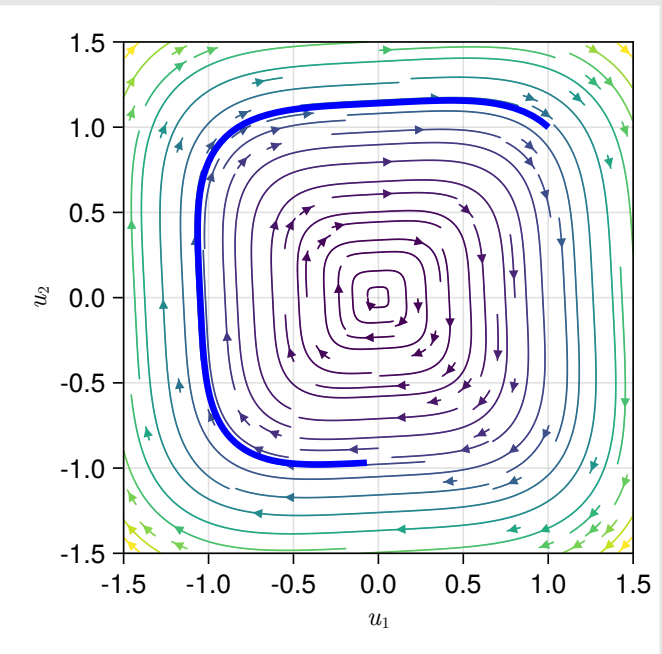
Forward

$$\underline{u}(t) = \underline{u}(0) + \int_0^t \underline{f}\left(\tau, \underline{u}(\tau)\right) \mathrm{d}\tau$$



Reverse

$$\underline{u}(t) = \underline{u}(T) + \int_T^t \underline{f}\left(\tau, \underline{u}(\tau)\right) \mathrm{d}\tau$$



Augmented Neural ODEs

Trajectories never intersect: functions such as $f(u) = -u$ can't be learned

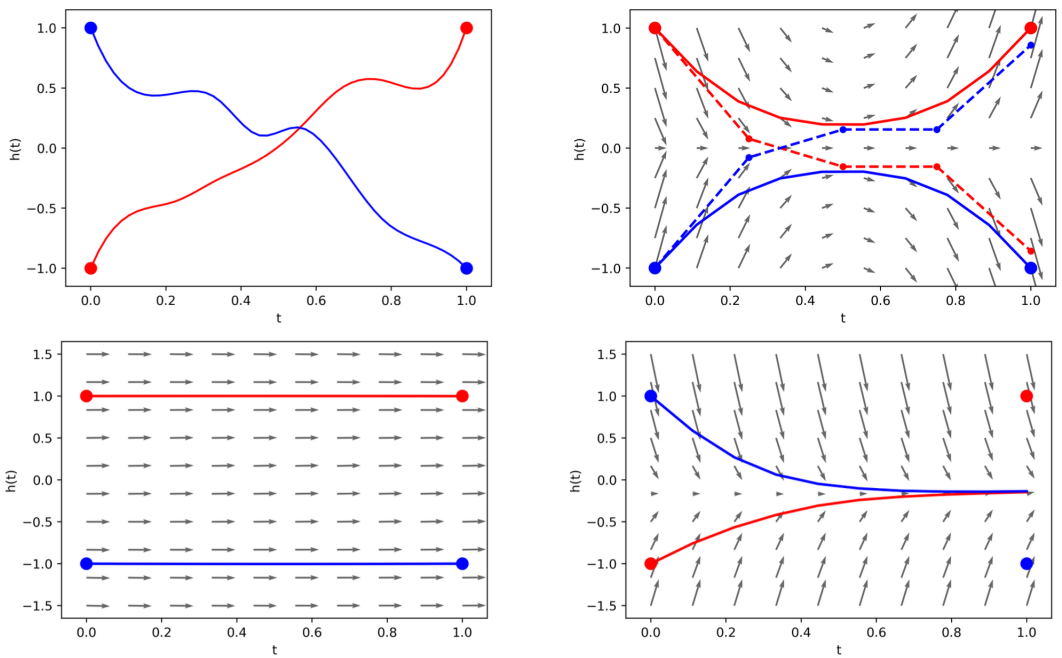
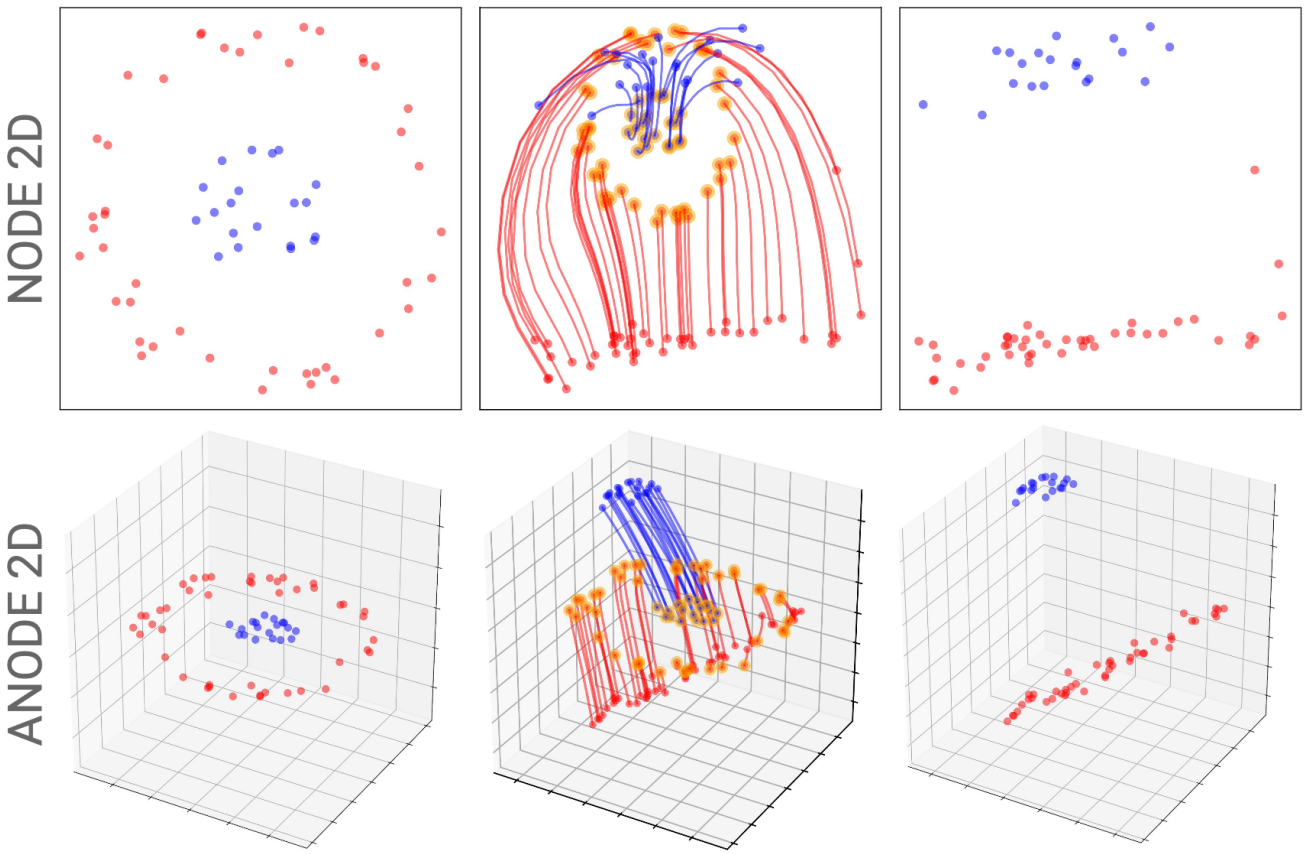


Figure 3: (Top left) Continuous trajectories mapping -1 to 1 (red) and 1 to -1 (blue) must intersect each other, which is not possible for an ODE. (Top right) Solutions of the ODE are shown in solid lines and solutions using the Euler method (which corresponds to ResNets) are shown in dashed lines. As can be seen, the discretization error allows the trajectories to cross. (Bottom) Resulting vector fields and trajectories from training on the identity function (left) and $g_{1\mathrm{d}}(x)$ (right).

Source: [DDT19]

Augmented Neural ODEs

- Neural ODEs **aren't** universal function approximators
- ANODEs add extra dimension(s) to the state $\underline{u}$ [DDT19]
- ANODEs **are** universal function approximators [DDT19, Kid22]



Source: [DDT19]

Sensitivity methods

Backpropagating through NODEs

We now know how to compute a forward-pass through our model, but **how do we optimize the parameters** θ of our Neural ODE?

To minimize a loss function $\mathcal{L}(\theta)$ using gradient-based optimization, we need to compute gradients $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta}$.

We are going to look at the two most commonly used approaches:[2]

1. Discretize-then-optimize
2. Optimize-then-discretize

[2] For a summary and comparison of more sensitivity methods, refer to [MDI+21, SFP14] and `https://docs.sciml.ai/SciMLSensitivity/`.

Discretize-then-optimize

Idea: Differentiate through steps of numerical solver

- Implement solver in an automatic differentiation (AD) framework (e.g. JAX, PyTorch, Julia)
- Solver internally performs differentiable operations (e.g. additions, multiplications)
- Solve is a composition of differentiable operations and therefore differentiable
- Use reverse- or forward-mode AD

[3] same memory consumption as a regular neural network

Discretize-then-optimize

Idea: Differentiate through steps of numerical solver

- Implement solver in an automatic differentiation (AD) framework (e.g. JAX, PyTorch, Julia)
- Solver internally performs differentiable operations (e.g. additions, multiplications)
- Solve is a composition of differentiable operations and therefore differentiable
- Use reverse- or forward-mode AD

Advantages:

- accurate gradients for discretized model
- usually the fastest method
- easy to implement

Disadvantages:

- memory inefficient[3]

[3] same memory consumption as a regular neural network

Optimize-then-discretize

Idea: First solve neural ODE forwards

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}_\theta\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

then solve the *adjoint equations* backwards in time

$$\frac{\mathrm{d}\underline{a}_u(t)}{\mathrm{d}t} = -\underline{a}_u(t)^T \frac{\partial \underline{f}_\theta\left(t, \underline{u}(t)\right)}{\partial \underline{u}}, \quad \underline{a}_u(T) = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\underline{u}(T)}$$

$$\frac{\mathrm{d}\underline{a}_\theta(t)}{\mathrm{d}t} = -\underline{a}_u(t)^T \frac{\partial \underline{f}_\theta\left(t, \underline{u}(t)\right)}{\partial \theta}, \quad \underline{a}_\theta(T) = 0$$

to obtain the gradient $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \underline{a}_\theta(0)$.

Optimize-then-discretize

Idea: First solve neural ODE forwards

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}_\theta\left(t, \underline{u}(t)\right), \quad \underline{u}(0) = \underline{u}_0$$

then solve the *adjoint equations* backwards in time

$$\frac{\mathrm{d}\underline{a}_u(t)}{\mathrm{d}t} = -\underline{a}_u(t)^T \frac{\partial \underline{f}_\theta\left(t, \underline{u}(t)\right)}{\partial \underline{u}}, \quad \underline{a}_u(T) = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\underline{u}(T)}$$

$$\frac{\mathrm{d}\underline{a}_\theta(t)}{\mathrm{d}t} = -\underline{a}_u(t)^T \frac{\partial \underline{f}_\theta\left(t, \underline{u}(t)\right)}{\partial \theta}, \quad \underline{a}_\theta(T) = 0$$

to obtain the gradient $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \underline{a}_\theta(0)$.

- forward solve is required to compute $\underline{u}(T)$ for initial condition $\underline{a}_u(T)$
- no need to save trajectory $\underline{u}(t)$ in memory, since neural ODE is invertible
- single backwards integration of augmented state $(\underline{u}(T), \underline{a}_u(T), \underline{a}_\theta(T))$

Optimize-then-discretize: Proof [Kid22]

Consider two points $s, t \in \mathcal{T} = [0, T]$ with $s < t$. Consider first solving the ODE from s to t, then from t to T. By the chain rule,

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\underline{u}(s)} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\underline{u}(t)} \frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}\underline{u}(s)}$$

with $\underline{a}(t)^T = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\underline{u}(t)}$. Differentiating this w.r.t. t and setting $t = s$:

$$\begin{aligned}
0 &= \frac{\mathrm{d}}{\mathrm{d}t} \left(\underline{a}(t)^T \frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}\underline{u}(s)} \right) \Bigg|_{t=s} \\
&= \frac{\mathrm{d}}{\mathrm{d}t} \left(\underline{a}(t)^T \right) \frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}\underline{u}(s)} \Bigg|_{t=s} + \underline{a}(t)^T \frac{\mathrm{d}}{\mathrm{d}t} \left(\frac{\underline{u}(t)}{\underline{u}(s)} \right) \Bigg|_{t=s} \\
&= \frac{\mathrm{d}\underline{a}}{\mathrm{d}s} + \underline{a}(t)^T \frac{\mathrm{d}}{\mathrm{d}\underline{u}(s)} \left(\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} \right) \Bigg|_{t=s} \\
&= \frac{\mathrm{d}\underline{a}}{\mathrm{d}s} + \underline{a}(t)^T \frac{\mathrm{d}}{\mathrm{d}\underline{u}(s)} \left(\underline{f}_\theta(t, \underline{u}(t)) \right) \Bigg|_{t=s} \\
&= \frac{\mathrm{d}\underline{a}}{\mathrm{d}s} + \underline{a}(s)^T \frac{\partial \underline{f}_\theta}{\partial \underline{u}} (s, \underline{u}(s)) \,.
\end{aligned}$$

The second equation is obtained analogously.
For a more rigorous derivation, refer to appendix C.3.1 in [Kid22].

Optimize-then-discretize

Method dates back to [Pon62], but was re-popularized by [CRBD18] as the continuous *adjoint sensitivity method*.

Advantages:

- memory efficient: forward computations of $\underline{u}(t)$ don't have to be stored
- can be implemented without AD framework

Disadvantages:

- high computational cost: requires solving a second backward pass of state and adjoint equations
- less accurate gradients:
  - values $\underline{u}(t)$ computed on the forward- and backward-pass might not match due to discretization errors
  - $\underline{a}_u$ and $\underline{a}_\theta$ also suffer from discretization errors

Software packages

The following software packages all implement:

- numerical solvers with fixed and adaptive step sizes
- differentiation via optimize-then-discretize and discretize-then-optimize
- GPU and CPU parallelization

Python

- JAX: diffrax
- PyTorch: torchdiffeq, torchdyn

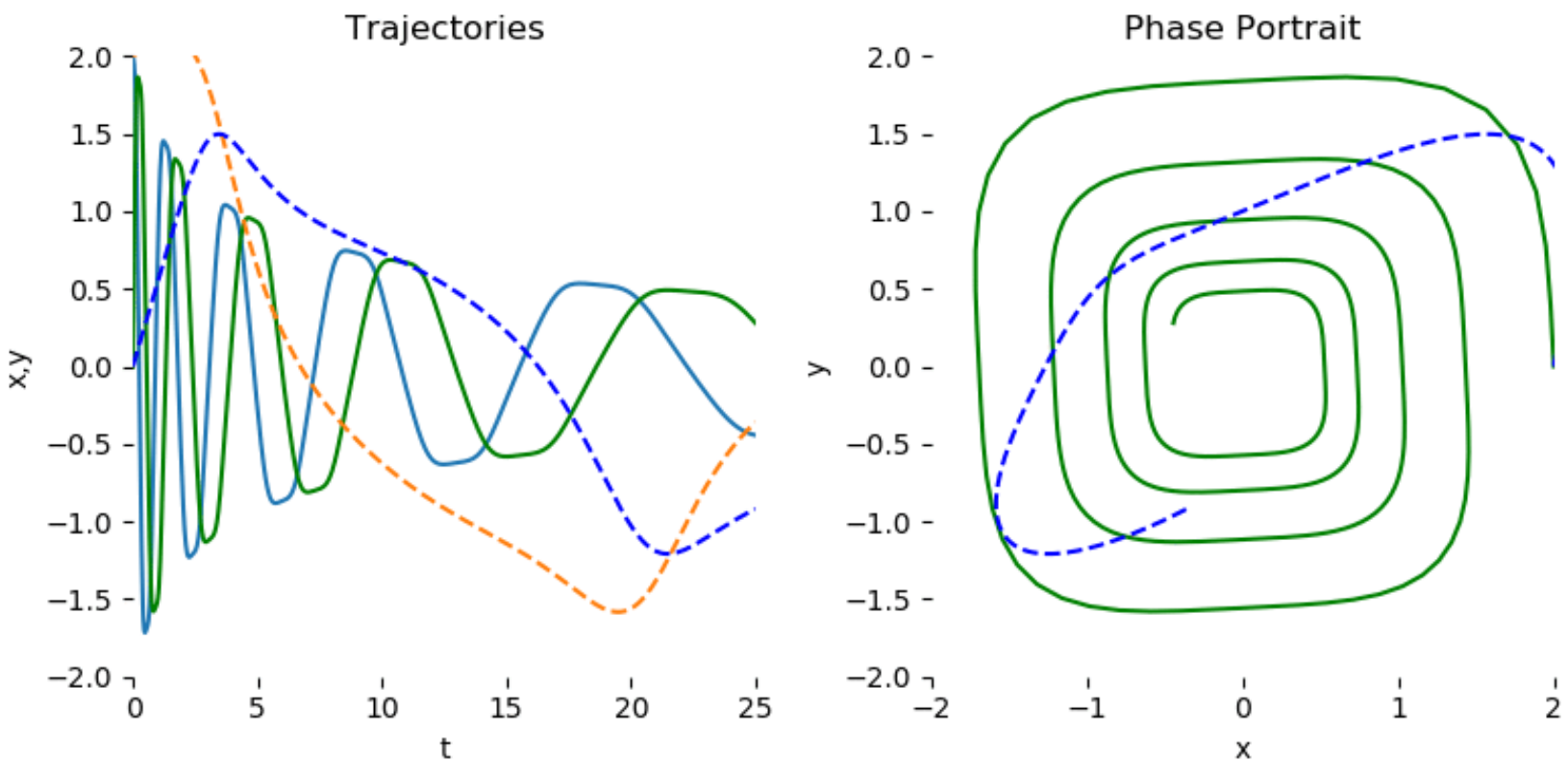
Julia: SciML ecosystem

- DifferentialEquations.jl
- DiffEqFlux.jl

Applications

Trajectory optimization

- Fit time series data
- Data can be sampled irregularly
- Works with partially observed states
- Continuously interpolates between data points



Source: [Che18]

Universal Differential Equations

So far, we have talked about Neural ODEs of the form

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}_\theta\left(t, \underline{u}\right) = \mathrm{NN}_\theta(t, \underline{u}) \quad ,$$

where the neural network NN_θ learns the entire vector field.

The neural ODE framework allows for a flexible definition of hybrid models. Structure and equations known from physical modelling can be incorporated into the model as an inductive bias.

Universal differential equations [RMM+20]

$$\frac{\mathrm{d}\underline{u}(t)}{\mathrm{d}t} = \underline{f}\left(t, \underline{u}, \mathrm{NN}_\theta(t, \underline{u})\right)$$

- generalization of Neural ODEs, physics-informed neural networks
- neural networks in or around differential equation
- Successfully used to model e.g. batteries, lab helicopters [BSF+20, RSL21]

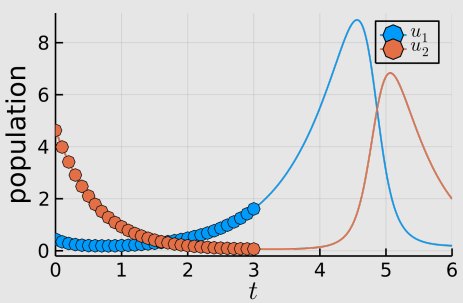
Universal Differential Equations

Example: Lotka–Volterra equations [RMM+20]

1.) Dataset

Data from dynamical system with (partially) unknown dynamics:

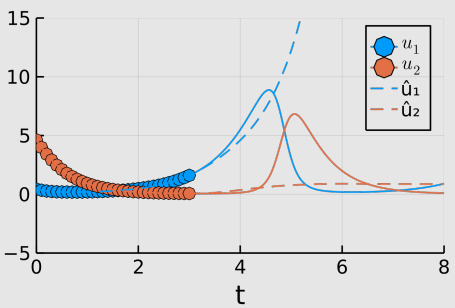
$$\dot{u}_1 = \alpha u_1 - \beta u_1 u_2$$
$$\dot{u}_2 = -\delta u_2 + \gamma u_1 u_2$$



2.) Fit UDE model

Incorporate mechanistic prior knowledge, capture unknown dynamics:

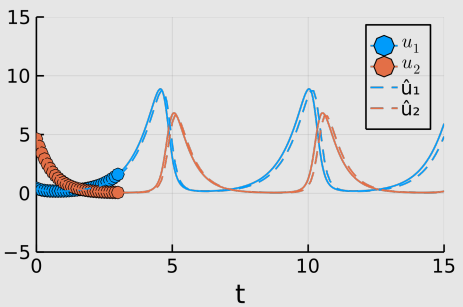
$$\dot{u}_1 = \alpha u_1 + \mathrm{NN}_1(\underline{u}; \theta)$$
$$\dot{u}_2 = -\delta u_2 + \mathrm{NN}_2(\underline{u}; \theta)$$



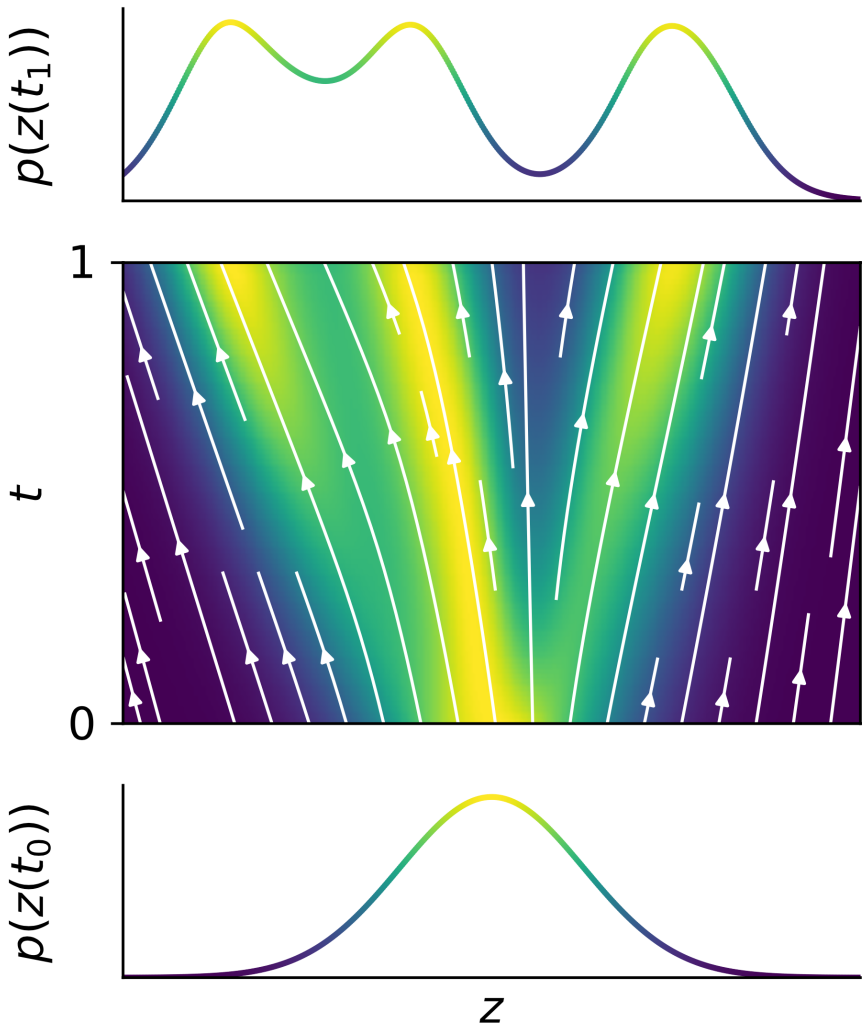
3.) Symbolic regression

Recover equations for missing terms:

$$\mathrm{NN}_1(\underline{u}; \hat{\theta}) \approx -\beta u_1 u_2$$
$$\mathrm{NN}_2(\underline{u}; \hat{\theta}) \approx \gamma u_1 u_2$$



Continuous Normalizing Flows



Source: [GCB+18]

Recapitulation: Normalizing Flows

As shown in **Lecture 3 on Generative models**, normalizing flows make use of the *change of variables* formula:

$$\log p_x(\underline{x}) = \log p_z(\underline{z}) - \log \det \left| \frac{\partial \underline{f}(\underline{z})}{\partial \underline{z}} \right|$$

where $\underline{z} \sim p_z$, $\underline{x} \sim p_x$ and $\underline{f}$ is an **invertible function** such that

$$\underline{x} = \underline{f}\left(\underline{z}\right), \quad \underline{z} = \underline{f}^{-1}\left(\underline{x}\right) \quad .$$

After selecting a distribution p_z (e.g. the standard normal distribution), a normalizing flow $\underline{f}_\theta$ can be trained by minimizing the negative log-likelihood

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{|\mathcal{D}|} \sum_{\underline{x} \in \mathcal{D}} \log p_x(\underline{x}) \quad ,$$

where $\log p_x(\underline{x})$ is computed using the change of variables formula.
This requires a **tractable log-determinant of the Jacobian**, which depends on the model $\underline{f}_\theta$.

Instantaneous change of variables

We can train a continuous normalizing flow using neural ODEs by setting the bounds of integration to $t \in [0, 1]$ and

$$\underline{u}(0) = \underline{u}_0 = \underline{x}$$
$$\underline{u}(1) = \underline{u}_1 = \underline{z}$$

such that

$$\underline{u}_1 = \underline{u}_0 + \int_0^1 \underline{f}_\theta \left(\tau, \underline{u}(\tau)\right) \mathrm{d}\tau \quad .$$

Since the density $\underline{u}_0$ is continuously transformed into $\underline{u}_1$, we can derive a continuous version of the change of variables formula called *instantaneous change of variables* formula [CRBD18]:

$$\log p\left(\underline{u}_1\right) = \log p\left(\underline{u}_0\right) - \int_0^1 \mathrm{Tr}\left(\frac{\partial \underline{f}_\theta}{\partial \underline{u}}\right) \mathrm{d}\tau$$

By augmenting the state of the ODE to keep track of this change in log-density, the negative log-likelihood can be computed to train the CNF.

Hutchinson's trace estimator

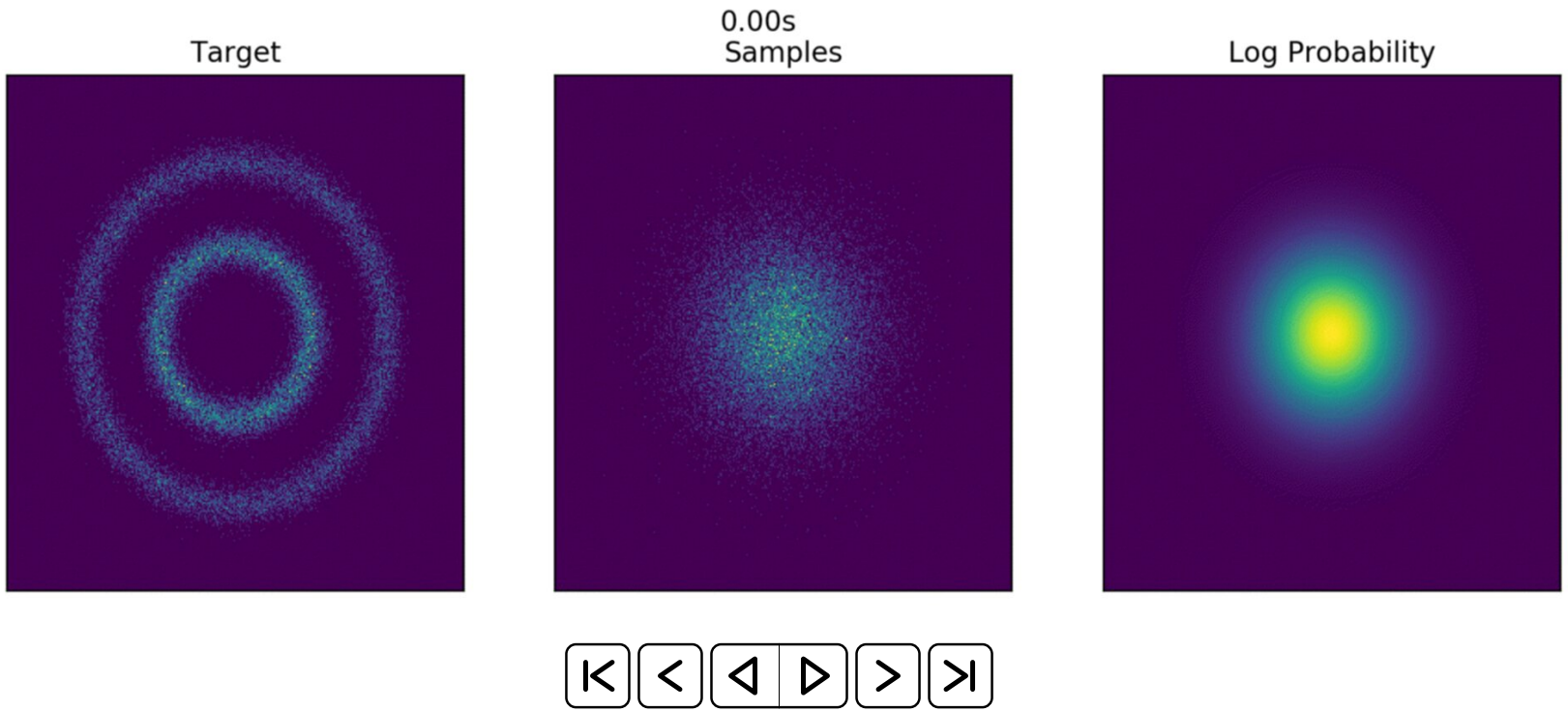
To obtain a fast and unbiased estimate of the trace of the Jacobian, we take a double product of the Jacobian with a random noise vector $\underline{\epsilon}$ [GCB+18]. This makes use of the fact that AD systems compute fast vector-Jacobian- and Jacobian-vector-products:

$$\mathrm{Tr}(A) = \mathbb{E}_{p(\underline{\epsilon})}\left[\underline{\epsilon}^T \mathbf{A} \underline{\epsilon}\right]$$

such that

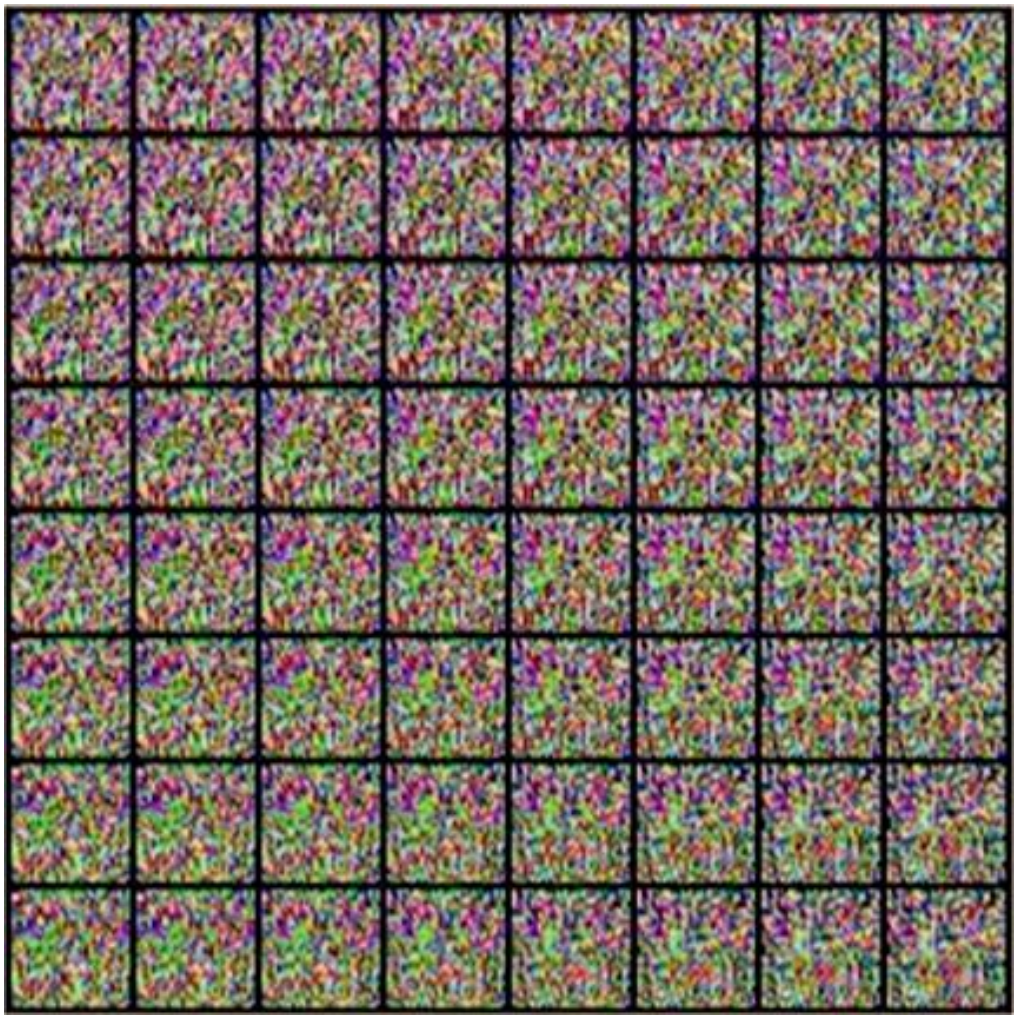
$$\begin{aligned}
\log p\left(\underline{u}_1\right) &= \log p\left(\underline{u}_0\right) - \int_0^1 \mathrm{Tr}\left(\frac{\partial \underline{f}_\theta}{\partial \underline{u}}\right) \mathrm{d}\tau \\
&= \log p\left(\underline{u}_0\right) - \int_0^1 \mathbb{E}_{p(\underline{\epsilon})}\left[\underline{\epsilon}^T \frac{\partial \underline{f}_\theta}{\partial \underline{u}} \underline{\epsilon}\right] \mathrm{d}\tau \\
&= \log p\left(\underline{u}_0\right) - \mathbb{E}_{p(\underline{\epsilon})}\left[\int_0^1 \underline{\epsilon}^T \frac{\partial \underline{f}_\theta}{\partial \underline{u}} \underline{\epsilon}\, \mathrm{d}\tau\right] \quad .
\end{aligned}$$

Applications of CNFs: Density estimation



Source: [CRBD18, Che18]

Applications of CNFs: Generative models



Source: [CRBD18]

References I

- Amos, Brandon:
  Differentiable optimization-based modeling for machine learning.
  In: *Ph. D. thesis* (2019)
- Bai, Shaojie ; Kolter, J Z. ; Koltun, Vladlen:
  Deep equilibrium models.
  In: *Advances in Neural Information Processing Systems* 32 (2019)
- Bai, Shaojie ; Koltun, Vladlen ; Kolter, J Z.:
  Multiscale deep equilibrium models.
  In: *Advances in Neural Information Processing Systems* 33 (2020), S. 5238–5250
- Bills, Alexander ; Sripad, Shashank ; Fredericks, William L. ; Guttenberg, Matthew ; Charles, Devin ; Frank, Evan ; Viswanathan, Venkatasubramanian:
  Universal battery performance and degradation model for electric aircraft.
  In: *arXiv preprint arXiv:2008.01527* (2020)
- Chen, Ricky T. Q.:
  torchdiffeq.
  https://github.com/rtqichen/torchdiffeq.
  Version: 2018

References II

- Chen, Ricky T. ; Rubanova, Yulia ; Bettencourt, Jesse ; Duvenaud, David:
  Neural ordinary differential equations.
  In: *arXiv preprint arXiv:1806.07366* (2018)
- Dupont, Emilien ; Doucet, Arnaud ; Teh, Yee W.:
  Augmented neural odes.
  In: *Advances in Neural Information Processing Systems* 32 (2019)
- Grathwohl, Will ; Chen, Ricky T. ; Bettencourt, Jesse ; Sutskever, Ilya ; Duvenaud, David:
  Ffjord: Free-form continuous dynamics for scalable reversible generative models.
  In: *arXiv preprint arXiv:1810.01367* (2018)
- Haber, Eldad ; Ruthotto, Lars:
  Stable architectures for deep neural networks.
  In: *Inverse problems* 34 (2017), Nr. 1, S. 014004
- He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian:
  Deep residual learning for image recognition.
  In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, S. 770–778
- Kidger, Patrick:
  On neural differential equations.
  In: *arXiv preprint arXiv:2202.02435* (2022)

References III

- Ma, Yingbo ; Dixit, Vaibhav ; Innes, Michael J. ; Guo, Xingjian ; Rackauckas, Chris:
  A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions.
  In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)* IEEE, 2021, S. 1–9
- Pontryagin, Lev S.:
  Mathematical theory of optimal processes.
  CRC press, 1962
- Rackauckas, Christopher ; Ma, Yingbo ; Martensen, Julius ; Warner, Collin ; Zubov, Kirill ; Supekar, Rohit ; Skinner, Dominic ; Ramadhan, Ali ; Edelman, Alan:
  Universal differential equations for scientific machine learning.
  In: *arXiv preprint arXiv:2001.04385* (2020)
- Rackauckas, Christopher ; Sharma, Roshan ; Lie, Bernt:
  Hybrid Mechanistic + Neural Model of Laboratory Helicopter.
  (2021)
- Sengupta, Biswa ; Friston, Karl J. ; Penny, William D.:
  Efficient gradient computation for dynamical models.
  In: *NeuroImage* 98 (2014), S. 521–527

References IV

- Song, Yang ; Sohl-Dickstein, Jascha ; Kingma, Diederik P. ; Kumar, Abhishek ; Ermon, Stefano ; Poole, Ben:
  Score-based generative modeling through stochastic differential equations.
  In: *arXiv preprint arXiv:2011.13456* (2020)