On Neural Differential Equations

Patrick Kidger

Mathematical Institute

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2021

Abstract

The conjoining of dynamical systems and deep learning has become a topic of great interest. In particular, *neural differential equations* (NDEs) demonstrate that neural networks and differential equation are two sides of the same coin. Traditional parameterised differential equations are a special case. Many popular neural network architectures, such as residual networks and recurrent networks, are discretisations.

NDEs are suitable for tackling generative problems, dynamical systems, and time series (particularly in physics, finance, ...) and are thus of interest to both modern machine learning and traditional mathematical modelling. NDEs offer high-capacity function approximation, strong priors on model space, the ability to handle irregular data, memory efficiency, and a wealth of available theory on both sides.

This doctoral thesis provides an in-depth survey of the field.

Topics include: neural *ordinary* differential equations (e.g. for hybrid neural/mechanistic modelling of physical systems); neural *controlled* differential equations (e.g. for learning functions of irregular time series); and neural *stochastic* differential equations (e.g. to produce generative models capable of representing complex stochastic dynamics, or sampling from complex high-dimensional distributions).

Further topics include: numerical methods for NDEs (e.g. reversible differential equations solvers, backpropagation through differential equations, Brownian reconstruction); symbolic regression for dynamical systems (e.g. via regularised evolution); and deep implicit models (e.g. deep equilibrium models, differentiable optimisation).

We anticipate this thesis will be of interest to anyone interested in the marriage of deep learning with dynamical systems, and hope it will provide a useful reference for the current state of the art.

Contents

Originality

Statement

The writing of this thesis is my original work. The material in this thesis is either (a) my original work either with or without collaborators, or (b) where relevant prior or concurrent work included for reference, so as to provide a survey of the field.

Papers

This thesis contains material from the following papers on neural differential equations (organised chronologically):

Neural Controlled Differential Equations for Irregular Time Series
Patrick Kidger, James Morrill, James Foster, Terry Lyons
Neural Information Processing Systems, 2020

"Hey, that's not an ODE": Faster ODE Adjoints via Seminorms
Patrick Kidger, Ricky T. Q. Chen, Terry Lyons
International Conference on Machine Learning, 2021

Neural Rough Differential Equations for Long Time Series
James Morrill, Cristopher Salvi, Patrick Kidger, James Foster, Terry Lyons
International Conference on Machine Learning, 2021

Neural SDEs as Infinite-Dimensional GANs
Patrick Kidger, James Foster, Xuechen Li, Harald Oberhauser, Terry Lyons
International Conference on Machine Learning, 2021

Efficient and Accurate Gradients for Neural SDEs
Patrick Kidger, James Foster, Xuechen Li, Terry Lyons
Neural Information Processing Systems, 2021

Neural Controlled Differential Equations for Online Prediction Tasks
James Morrill, Patrick Kidger, Lingyi Yang, Terry Lyons
arXiv:2106.11028, 2021

Open source software

A substantial component of my PhD has been the democratisation of neural differential equations via open-source software development. In particular I have authored or otherwise had a substantial hand in developing:

Diffrax
Ordinary, controlled, and stochastic differential equation solvers for JAX.
`https://github.com/patrick-kidger/diffrax`

torchdiffeq
Ordinary differential equation solvers for PyTorch.
`https://github.com/rtqichen/torchdiffeq`

torchcde
Controlled differential equation solvers for PyTorch.
`https://github.com/patrick-kidger/torchcde`

torchsde
Stochastic differential equation solvers for PyTorch.
`https://github.com/google-research/torchsde`

Breakdown of contributions

My personal contributions to each paper break down as follows.

For the 'Neural Controlled Differential Equations for Irregular Time Series' paper. I did the entirety of this paper. James Morrill and James Foster had concurrently worked on similar ideas and were included as authors on the paper as a courtesy.

For the '"Hey, that's not an ODE": Faster ODE Adjoints via Seminorms' paper. I had the idea, theory, wrote the library implementation, and handled the neural CDE and Hamiltonian experiments. Ricky T. Q. Chen performed the experiments for the continuous normalising flows. The written text was joint work between both of us. (And whilst of course it does not appear in the final paper, Ricky T. Q. Chen handled most of the rebuttal.)

For the 'Neural Rough Differential Equations for Long Time Series' paper. Cristopher Salvi had the idea of using the log-ODE method to reduce a neural CDE to an ODE. I spotted the practical application to long time series. James Morrill implemented it. James Foster helped with the theory. The written text was joint work between me and James Morrill.

For the 'Neural SDEs as Infinite-Dimensional GANs' paper. I had the basic idea, basic theory, and and wrote all of the experimental code. James Foster provided the necessary knowledge of SDE numerics. Xuechen Li had already started writing (and released an early version of) the 'torchsde' software library we used. Xuechen

Li and I jointly performed subsequent development of the 'torchsde' library to extend it for this paper. The more complete idea for the paper was fleshed out jointly in conversations between all three of us. The written text was joint work between all three of us. (Finally, I owe James Foster a debt of thanks: during the development of this paper, he kindly fielded endless questions from me on the topic of SDE numerics.)

For the 'Efficient and Accurate Gradients for Neural SDEs' paper. I had the idea and the theory for the Brownian Interval. I had the idea and the theory for gradient-penalty-free training of SDE-GANs. I wrote all the code for this paper. James Foster and I independently had the idea to look for an algebraically reversible SDE solver; the reversible Heun method we ended up using was due to just James Foster. Xuechen Li was included as an author as a courtesy, as the two neural SDE papers were originally intended to be published together as a single paper.

For the 'Neural Controlled Differential Equations for Online Prediction Tasks' paper. I had the idea and the abstract theory for this paper. James Morrill came up with cubic Hermite splines with backward differences, and handled the implementation. Lingyi Yang assisted with some datasets.

In every case Terry Lyons was included on each paper as my supervisor.

Previously unpublished

This thesis includes some previously unpublished material on various topics related to neural differential equations. (Usually on material that was only 'half a paper' in size.) This includes material on symbolic regression, universal approximation, parameterisations of neural differential equations, and sensitivities of differential equations.

Other

Papers My PhD work has included several other papers [Kid+19; KL20b; KML20; Mor+20; KL21], but as they cover other topics – ranging from rough path theory to universal approximation – they do not form a part of this thesis.

Software Likewise, my PhD work has included the development of several other software libraries [KL20a; Kid21d; Kid21b; Kid21c]. These software libraries are for the Julia, PyTorch and JAX ecosystems, and offer a variety of tools such as improved import systems, rich type annotations for tensors, and the elevation of parameterised functions to first-class 'PyTrees'.

Once again these are not included in this thesis.

Acknowledgements

A doctoral degree doesn't happen in a vacuum. Getting this far has meant the involvement of numerous people, all of whom I am incredibly fortunate to have in my life.

First and foremost I would like to thank my parents, Penny and Alex. I am so, so lucky to have been raised in the environment that I was, with the opportunities you gave me. You have always been my personal champions.

Mum – I know having me go to Oxford was always a dream come true for you. Finishing this doctorate means finishing the journey of a lifetime, and it's one that you started me on. Everything I know about mathematics I learnt from you.

Dad – from electronics to electromagnetism, my fondest memories of childhood are all the time we spent together on the back of an envelope. I don't doubt where my love of this subject comes from. This thesis isn't quite one of those envelopes, but I hope it comes close.

Truthfully, I have been drafting and redrafting what to say here, but what can compare to 25 years of unconditional support? I cannot put into words how blessed I feel to have you as my parents.

Thank you to my sister Eleanor, who has always been there for me. Our 4am discussions on topics from philosophy to biology were time well spent. Your kindness inspires me to be a better person. Now – go and get your own doctorate!

I love you all.

Thank you to all my friends for all the time we have spent together. There are two people who deserve to be highlighted in particular.

To Chloe: thank you. You have been a constant presence in my PhD life, from start to finish. In times of crisis you have offered to make more shopping trips on my behalf than I can count. You have been the best friend a best friend can have.

Thank you to Juliette: for friendship, food, and the south of France. (Where this document began.) Lockdown with you was unquestionably one of the best, and happiest, times of my life.

Thank you to all of my academic collaborators: Ricky T. Q. Chen, Xuechen Li, Miles Cranmer, James Morrill, James Foster, Cristopher Salvi, Adeline Fermanian, Lingyi

Yang, Patric Bonnier, and Imanol Perez Arribas.

Across late nights, failed experiments, all-too-soon deadlines, and endless redrafting of a paper or rebuttal – in a very real way, this work exists because of you.

A particular thank you must go to David Duvenaud, Ben Hambly, James Foster, and Ben Walker, who diligently proofread this manuscript for errors. Thanks to their efforts many typographical mistakes and mathematical boo-boos were squashed. (As is traditional, any errors that remain are of course mine alone.)

Last and certainly not least, thank you to my supervisor, Terry Lyons. Whenever I have needed your help, you have been generous with your time. Whenever I have needed something for my research, you have gone out of your way to help me obtain it. Your guidance over our many conversations has shaped me into the researcher I am today.

Chapter 1

Introduction

1.1 Motivation

We have two goals in writing this document. One: to satisfy the requirements of a PhD, by writing a thesis describing our original research. Two: to give an accessible survey of the new, rapidly developing, and in our opinion very exciting field of *neural differential equations*. To the best of our knowledge this is the first survey to have been written on the topic.

We hope this will prove useful to the interested reader! Along the way we shall cover a wide variety of applications, both to classical mathematical modelling, and to typical machine learning problems.

1.1.1 Getting started

Prerequisites We will assume throughout that the reader is familiar with the basics of ODEs and with the basics of modern deep learning, but we will not assume an in-depth knowledge of either. On the basis that many of our readers may come from a traditional applied mathematics background without much exposure to deep learning, then Appendix A also provides a summary of the relevant deep learning concepts we shall assume. It also provides references for learning more about deep learning.

The material on neural SDEs will assume familiarity with SDEs.

Beyond these (relatively weak) assumptions, we will introduce concepts as we need them. Various parts of the text will touch on topics such as rough path theory, or numerical methods for differential equations. In each case we assume little-to-no familiarity on the part of the reader, and where necessary provide references for learning more about them.

The next chapter (on neural ODEs) makes an effort to explicitly spell out even 'elementary' details such as the existence of solutions to ordinary differential equations,

or the use of cross entropy as a loss function. Later chapters assume increasing levels of sophistication; it is recommended to read them in sequential order.

Code The reader interested in applying these techniques is strongly encouraged to write some example code.

Each chapter contains a few numerical examples – usually on toy datasets for ease of understanding. The corresponding code is both available and well-documented: they can be found as the examples of the Diffrax software library [Kid21a], which is written for the JAX framework [Bra+18].

Indeed standard software libraries for solving and differentiating differential equations make working with NDEs essentially easy. These are discussed in Section 5.6 (including both Diffrax and other options for other frameworks). These libraries are again well-documented and contain numerous examples.

Experiments The material here focuses on presenting the theory of NDEs; correspondingly our numerical examples will tend to be on toy datasets chosen for ease of understanding. Real world (and possibly very large scale) applications of these techniques may be found in the original papers, which are referenced in the text alongside each individual topic.

1.1.2 What is a neural differential equation anyway?

A *neural differential equation* is a differential equation using a neural network to parameterise the vector field. The canonical example is a *neural ordinary differential equation* [Che+18b]:

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

Here θ represents some vector of learnt parameters, $f_\theta \colon \mathbb{R} \times \mathbb{R}^{d_1 \times \cdots \times d_k} \to \mathbb{R}^{d_1 \times \cdots \times d_k}$ is any standard neural architecture, and $y \colon [0, T] \to \mathbb{R}^{d_1 \times \cdots \times d_k}$ is the solution. For many applications f_θ will just be a simple feedforward network.

The central idea now is to use a differential equation solver as part of a learnt differentiable computation graph (the sort of computation graph ubiquitous to deep learning).

As a simple example, suppose we observe some picture $y_0 \in \mathbb{R}^{3 \times 32 \times 32}$ (RGB and 32×32 pixels), and wish to classify it as a picture of a cat or as a picture of a dog.

We proceed by taking $y(0) = y_0$ as the initial condition of the neural ODE, and evolve the ODE until some time T. An affine transformation[1] $\ell_\theta \colon \mathbb{R}^{3 \times 32 \times 32} \to \mathbb{R}^2$ is then

[1]Commonly referred to as a 'linear' transformation in deep learning, although this is not technically correct in the mathematical sense of the word. An affine transformation takes the form $x \mapsto Wx + b$ with potentially nonzero bias b; a linear transformation is one for which $b = 0$. The difference will occasionally be important to us so we endeavour to make the distinction.

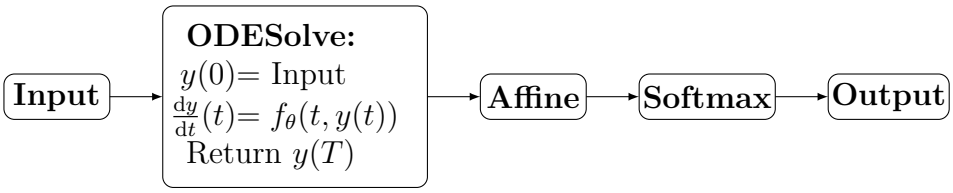


Figure 1.1: Computation graph for a simple neural ODE.

applied, followed by a softmax, so that the output may be interpreted as a length-2 tuple $(\mathbb{P}(\text{picture is of a cat}), \mathbb{P}(\text{picture is of a dog}))$.

This is summarised pictorially in Figure 1.1. In conventional mathematical notation, this computation may be denoted

$$\text{softmax}\left(\ell_\theta\left(y(0) + \int_0^T f_\theta(t, y(t))\,\mathrm{d}t\right)\right).$$

The parameters of the model are θ. The computation graph may be backpropagated through and trained via stochastic gradient descent in the usual way. We will discuss how to backpropagate through an ODE solve in Section 5.1.

In total, then: there is a neural network f_θ, embedded in a differential equation for y, embedded in a neural network (the overall computation graph).

1.1.3 A familiar example

A potentially familiar example of a 'neural' differential equation is the classic SIR model:

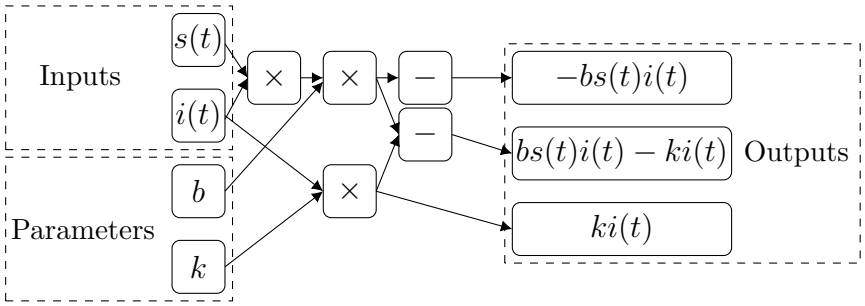
$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} s(t) \\ i(t) \\ r(t) \end{pmatrix} = \begin{pmatrix} -b\,s(t)\,i(t) \\ b\,s(t)\,i(t) - k\,i(t) \\ k\,i(t) \end{pmatrix}.$$

This is used in mathematical epidemiology to describe the spread of a disease within a population.[2] The quantity s represents the susceptible (uninfected) portion of the population, the quantity i represents the infected portion of the population, and the quantity r represents the removed (recovered or deceased) portion of the population.

The vector field is theoretically derived, with parameters b and k describing the infectivity and the (recovery + mortality) rates respectively.

The right hand side may be regarded as a particular differentiable computation graph:

[2]A rather topical choice, with this thesis having being prepared during the global Covid-19 pandemic.



The parameters may be fitted by setting up a loss between the trajectories of the model and the observed trajectories in the data, backpropagating through the model, and applying stochastic gradient descent.

This is precisely the same procedure as the more general neural ODEs we introduced earlier. At first glance, the NDE approach of 'putting a neural network in a differential equation' may seem unusual, but it is actually in line with standard practice. All that has happened is to change the parameterisation of the vector field.

1.1.4 Continuous-depth neural networks

We have just seen how neural differential equations may be approached via traditional mathematical modelling. They may also be arrived at via modern deep learning.

Recall the formulation of a residual network [He+15]:

$$y_{j+1} = y_j + f_\theta(j, y_j), \tag{1.1}$$

where $f_\theta(j, \cdot)$ is the j-th residual block. (The parameters of all blocks are concatenated together into θ.)

Now recall the neural ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

Discretising this via the explicit Euler method at times t_j uniformly separated by Δt gives

$$\frac{y(t_{j+1}) - y(t_j)}{\Delta t} \approx \frac{\mathrm{d}y}{\mathrm{d}t}(t_j) = f_\theta(t_j, y(t_j)),$$

so that

$$y(t_{j+1}) = y(t_j) + \Delta t f_\theta(t_j, y(t_j)).$$

Absorbing the Δt into the f_θ, we recover the formulation of equation (1.1).

Having made this observation – that neural ODEs are the continuous limit of residual networks – we may be prompted to start making other connections.

It transpires that the key features of a GRU [Cho+14] or an LSTM [HS97], over generic recurrent networks, are updates rules that look suspiciously like discretised differential equations (Chapter 3). StyleGAN2 [Kar+19] and (score based) diffusion

models [Son+21b] are simply discretised SDEs (Chapter 4). Coupling layers in invertible neural networks [Beh+19] turn out to be related to reversible differential equation solvers (Chapter 5). And so on.

By coincidence (or, as the idea becomes more popular, by design) many of the most effective and popular deep learning architectures resemble differential equations. Perhaps we should not be surprised: differential equations have been the dominant modelling paradigm for centuries; they are not so easily toppled.

1.1.5 An important distinction

There has been a line of work on obtaining numerical approximations to the solution y of an ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y(t))$ by representing the solution as some neural network $y = y_\theta$.

Perhaps f is known, and the model y_θ is fitted by minimising a loss function of the form

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \left\| \frac{\mathrm{d}y_\theta}{\mathrm{d}t}(t_i) - f(t_i, y_\theta(t_i)) \right\| \tag{1.2}$$

for some points $t_i \in [0, T]$. As such each solution to the differential equation is obtained by solving an optimisation problem. This has strong overtones of collocation methods or finite element methods. This is a popular line of work; see for example [LLF97a; LLF97b; HJE18; MQH18; Rai18; PSW19; RPK19; Fan+19; Zub+21] amongst many others.

This is known as a physics-informed neural network (PINN). PINNs are effective when generalised to some PDEs, in particular nonlocal or high-dimensional PDEs, for which traditional solvers are computationally expensive. (Although in most regimes traditional solvers are still the more efficient choice.) [Zub+21] provide an overview.

However, we emphasise that *this is a distinct notion to neural differential equations.* NDEs use neural networks to *specify* differential equations. Equation (1.2) uses neural networks to *obtain solutions to prespecified* differential equations. This distinction is a common point of confusion, especially as the PDE equivalent of (1.2) is sometimes referred to as a 'neural partial differential equation'.

1.2 The case for neural differential equations

1.2.1 Applications

To this author's knowledge, there are four main applications for neural differential equations:

Physical (financial, biological, ...) modelling Mechanistic theory-driven differential equation models are already ubiquitous in classical mathematical modelling. However, such theory-driven models will at some point fail to capture the details of reality. By combining existing models with deep learning (with its high-capacity function approximators), we may close the gap between theory and observation.

Time series Messy or irregular data is ubiquitous in time series. Different channels may be observed at different frequencies, data may be missing, time series may be of variable lengths, and so on. Treating discrete data in a continuous-time regime offers a way to treat irregular data on the same footing as 'regular' data.

Connections to topics such as system identification and reinforcement learning may also be made here, although they will not feature heavily in the present work.

Generative modelling Generative modelling studies how to model some target distribution ν, from which typically we only have samples. The usual framework is to pick a 'friendly' distribution μ, and then learn a map F such that (the pushforward) $F(\mu)$ approximates the target distribution ν.

It transpires that effective choices for F are derived from differential equations. For example with continuous normalising flows then μ may be a normal distribution (Section 2.2.3); in the case of a neural SDE then μ may be (the law of) a Brownian motion (Chapter 4).

Inspiration Traditional 'discrete' deep learning is widely applicable, and rightly so. We have already seen the parallels between differential equations and deep learning: a highly successful strategy for the development of deep learning models is simply to take the appropriate differential equation, and then discretise it.

1.2.2 Advantages

In summary, neural differential equations offers a best-of-both-worlds approach.

The neural network-like structure offers high-capacity function approximation and easy trainability.

The differential equation-like structure offers strong priors on model space, memory efficiency, and theoretical understanding via a well-understood and battle-tested literature.

Relative to the classical differential equation literature, neural differential equations have essentially unprecedented modelling capacity. Relative to the modern deep learning literature, neural differential equations offer a coherent theory of 'what makes a good model'.

1.3 A note on history

Practically speaking, the *topic* of neural differential equations become a *field* only a few years ago, starting with the explosion of interest following [Che+18b]; other prominent recent work also includes [E17; HR17].

However, many of the basic ideas can be found in substantially older literature, often from the 1990s. For example in [Ric+92], a neural ODE is trained to match the dynamics of a chemical reaction, using an MLP for the vector field. Meanwhile the basics of learning a controlled dynamical system are given in [CS91]. [RAK94] consider hybridising neural ODEs with traditional theory-driven mechanistic modelling, and [RK93] use implicit integrators in conjunction with neural ODEs to learn stiff dynamical systems.

This list of examples is by no means exhaustive. The above references are all short and make for easy reading, so the curious reader is encouraged to look them up.

Chapter 2

Neural Ordinary Differential Equations

2.1 Introduction

By far the most common neural differential equation is a neural ODE [Che+18b]:

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)), \tag{2.1}$$

where $y_0 \in \mathbb{R}^{d_1 \times \ldots \times d_k}$ is an any-dimensional tensor, θ represents some vector of learnt parameters, and $f_\theta \colon \mathbb{R} \times \mathbb{R}^{d_1 \times \ldots \times d_k} \to \mathbb{R}^{d_1 \times \ldots \times d_k}$ is a neural network. Typically f_θ will be some standard simple neural architecture, such as a feedforward or convolutional network.

2.1.1 Existence and uniqueness

The first question typically asked (at least by mathematicians) is about existence and uniqueness of a solution to equation (2.1). This is straightforward. Provided f_θ is Lipschitz – something which is typically true of a neural network, which is usually a composition of Lipschitz functions – then Picard's existence theorem [But16, Theorem 110C] applies:

Theorem 2.1 (Picard's Existence Theorem)**.** *Let $f \colon [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t and uniformly Lipschitz*[1] *in y. Let $y_0 \in \mathbb{R}^d$. Then there exists a unique differentiable $y \colon [0, T] \to \mathbb{R}^d$ satisfying*

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)).$$

[1]That is, it is Lipschitz in y and the Lipschitz constant is independent of t: there exists $C > 0$ such that for all t, y_1, y_2 then $\|f_\theta(t, y_1) - f_\theta(t, y_2)\| \leq C \|y_1 - y_2\|$.

2.1.2 Evaluation and training

As compared to models that are not differential equations, there are two extra concerns that must generally be kept in mind.

First, we must be able to obtain numerical solutions to the differential equation. (An analytic solution will essentially never be available.) Second, we must be able to backpropagate through the differential equation, to obtain gradients for its parameters θ.

Software for performing these tasks is now standardised (Section 5.6), so we are free to focus on the task of constructing the model architecture itself. A more in-depth look at evaluation and backpropagation is given in Chapter 5.

2.2 Applications

2.2.1 Image classification

Image classification with CNNs is nearly everybody's first introduction to deep learning. It is a natural place to start discussing neural differential equations too.

Dataset Suppose we observe some images, represented as a 3-dimensional tensor $\mathbb{R}^{3\times32\times32}$, corresponding to channels (red, green, blue), height (32 pixels), and width (32 pixels) respectively. Suppose each image has a corresponding class label in $\mathbb{R}^{10}$, corresponding to a one-hot encoding of what the image is a picture of: perhaps aeroplane, car, bird, cat, deer, dog, frog, horse, ship or lorry.

Model Let $f_\theta\colon \mathbb{R}\times\mathbb{R}^{3\times32\times32}\to\mathbb{R}^{3\times32\times32}$ be a convolutional neural network, and let $\ell_\theta\colon \mathbb{R}^{3\times32\times32}\to\mathbb{R}^{10}$ be affine.

Then we may define an image classification model as

$$\begin{aligned}&\phi\colon \mathbb{R}^{3\times32\times32}\to\mathbb{R}^{10},\\ &\phi\colon y_0\mapsto \operatorname{softmax}(\ell_\theta(y(T))),\end{aligned}$$

where $y\colon [0,T]\to\mathbb{R}^{3\times32\times32}$ solves

$$y(0)=y_0,\qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t)=f_\theta(t,y(t)).$$

Loss function By using an appropriate loss function (cross entropy) between this output and the true label, we may train this model so that its output is the probability that the input image is of each of these classes.

Explicitly: given a dataset of images $a_i \in \mathbb{R}^{3\times32\times32}$ with corresponding labels $b_i \in \mathbb{R}^{10}$, for samples $i = 1, \ldots, N$, we may minimise the cross-entropy

$$-\frac{1}{N}\sum_{i=1}^{N} b_i \cdot \log \phi(a_i)$$

by training θ, where $\cdot$ denotes a dot product and log is taken elementwise.

This example is an example only. In practice, for applications such as image classification there is usually little to be gained by using a continuous-time model. Traditional residual networks (that is, explicitly discretised neural ODEs) are simply easier to work with.

As such this example is an example only. We do not actually suggest using neural ODEs for this task, for which standard neural networks are likely to be superior.

The manifold hypothesis Neural ODEs interact elegantly with the manifold hypothesis (that the data lies on or near some low-dimensional manifold embedded in the higher-dimensional feature space; Appendix A.5). The ODE describes a flow along which to evolve the data manifold.

2.2.2 Physical modelling with inductive biases

Endowing a model with any known structure of a problem is known as giving the model an *inductive bias*. 'Soft' biases through penalty terms are one common example. 'Hard' biases through explicit architectural choices are another.

Physical problems often have known structure, and so a common theme has been to build in inductive biases by hybridising neural networks into this structure. It is this author's prediction that this will shortly become a standard technique in the toolbox of applied mathematical modelling. (If, arguably, it isn't already.)

2.2.2.1 Universal differential equations

Consider the Lotka-Volterra model, which is a well known approach for modelling the interaction between a predator species and a prey species:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t}(t) &= \alpha x(t) - \beta x(t)y(t) \in \mathbb{R},\\ \frac{\mathrm{d}y}{\mathrm{d}t}(t) &= -\gamma x(t) + \delta x(t)y(t) \in \mathbb{R}.\end{aligned} \tag{2.2}$$

Here, $x(t) \in \mathbb{R}$ and $y(t) \in \mathbb{R}$ represent the size of the population of the prey and predator species respectively, at each time $t \in [0, T]$. The right hand side is theoretically constructed, representing interactions between these species.

This theory will not usually be perfectly accurate, however. There will be some gap between the theoretical prediction and what is observed in practice. To remedy this, and letting $f_\theta, g_\theta \colon \mathbb{R}^2 \to \mathbb{R}$ be neural networks, we may instead consider the model

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t}(t) &= \alpha x(t) - \beta x(t) y(t) + f_\theta(x(t), y(t)) \in \mathbb{R},\\ \frac{\mathrm{d}y}{\mathrm{d}t}(t) &= -\gamma x(t) + \delta x(t) y(t) + g_\theta(x(t), y(t)) \in \mathbb{R},\end{aligned} \tag{2.3}$$

in which an existing theoretical model is augmented with a neural network correction term.

We broadly refer to this approach as a *universal differential equation*, a term due to [Rac+20b].[2]

Loss function and training Suppose we observe data $x_i(t_j) \in \mathbb{R}$, $y_i(t_j) \in \mathbb{R}$, where $i = 1, \ldots, N$ denote independent observations of the target process (from different initial conditions) and $j = 1, \ldots, M$ correspond to different times $t_j \in [0, T]$, with $t_1 = 0$. In practice we may only have $N = 1$, which may be sufficient provided M is large enough.

For either (2.2) or (2.3), let $x_{x_0,y_0}(t)$ denote $x(t)$ given initial condition $x(0) = x_0$ and $y(0) = y_0$. Similarly for $y_{x_0,y_0}(t)$.

Then we may fit both (2.2) and (2.3) in precisely the same way: stochastic gradient descent with respect to the loss function

$$\frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} (x_{x_i(0),y_i(0)}(t_j) - x_i(t_j))^2 + (y_{x_i(0),y_i(0)}(t_j) - y_i(t_j))^2.$$

In switching from (2.2) to (2.3), then no fundamental part of the modelling procedure has changed.

Remark 2.2. *The above presentation implicitly assumes that the locations of the observations t_j were the same for both x and y, and were the same for all training samples. This is just for simplicity of presentation and is not necessary in general.*

High capacity function approximation By switching from (2.2) to (2.3), the high-capacity function approximation provided by the neural networks f_θ, g_θ offers a way to close the gap between theory and practice. The neural network may be used to model the residual between the theoretical and the observed data.

The use of a neural network is an admission that *there is behaviour we do not understand*: but through this augmentation, we can at least model.

[2]There is little unified terminology here. Other authors have considered essentially the same idea under other names; conversely [Rac+20b] additionally consider variations and extensions to SDEs, PDEs, and so on.

These networks will frequently be very small by the standards of deep learning: [LKT16] consider a feedforward network of 10 layers each of width 10. [Rac+20b] consider feedforward networks of width 32 and a single hidden layer.

Use cases This approach becomes natural whenever one is attempting to model complex poorly understood behaviour, and for which there is sufficient data that the theoretical model clearly falls short.

Derivation of closure relations is a neat example. In this case, the differential equation features a term that lacks a precise theoretical description (representing the effects over scales smaller that the numerical solver can resolve), so the strategy becomes to approximate this term with a neural network, and learn this term from data.

Turbulence modelling is a popular example of this. In a Reynolds-averaged Navier Stokes model, [LKT16] approximate the closure relation (the Reynolds stresses) using a neural network carefully designed to satisfy certain physical invariances. See also the substantial follow-up literature: [DIX19; WWX17; Mau+19] and so on. Meanwhile as part of a climate model for the ocean, [Ram+20a] model a closure relation (for turbulent vertical heat flux) using a small MLP.

How to train your UDE Training (2.3) directly (via gradient descent) may not produce an interpretable model. The parameters $\alpha, \beta, \gamma, \delta$ may not necessarily correspond to their usual quantities, if the neural network has modelled some part of the behaviour as well.

One resolution is to fit (2.2) first, use its parameters to initialise $\alpha, \beta, \gamma, \delta$ in (2.3), and then train only the network parameters θ. This will ensure that the neural network only fits the residual between the theoretical model and the observed data.

Another option is to regularise the norm of the neural network [Yin+21], so that it is used only when necessary.

Another concern when training is that the model may become stuck in a local minimum. (Because the neural networks used with UDEs are often very small.) This may be mitigated by training on the first proportion of a time series (say the first 10%) before training on the whole time series; more generally setting some 'length schedule' that uses an increasing fraction of the time series as training progresses.

2.2.2.2 Hamiltonian neural networks

Another approach is to suppose that the observed dynamics evolve according to a Hamiltonian system; a realistic assumption for many physical systems. With respect to some known canonical coordinates $q, p \in \mathbb{R}^d$ and an unknown Hamiltonian function

$H\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, the system is assumed to evolve according to

$$\frac{\mathrm{d}q}{\mathrm{d}t}(t) = \frac{\partial H}{\partial p}(p(t), q(t)),$$
$$\frac{\mathrm{d}p}{\mathrm{d}t}(t) = -\frac{\partial H}{\partial q}(p(t), q(t)).$$

By parameterising $H = H_\theta$ as some general neural network (for example just an MLP), this system may be learnt much like a universal differential equation – in this case, the inductive bias is encoded through the use of a Hamiltonian-derived vector field, rather than explicit inclusion of known terms [GDY19].

Parameterisations of the Hamiltonian The Hamiltonian itself could be parameterised as an unstructured neural network, like an MLP. Alternatively one can go further, by parameterising the Hamiltonian according to kinetic and potential energy

$$H_\theta(q, p) = \frac{1}{2} p^\top M_\theta^{-1}(q) p + V_\theta(q),$$

where now M_θ is a learnt positive-definite mass matrix, and V_θ is a learnt potential energy [ZDC20a; ZDC20b].

Control terms Encoding this minimal amount of prior knowledge also makes available tools from classical dynamics. For example, we may suppose that the system responds to a control term β according to

$$\frac{\mathrm{d}q}{\mathrm{d}t}(t) = \frac{\partial H}{\partial p}(p(t), q(t)),$$
$$\frac{\mathrm{d}p}{\mathrm{d}t}(t) = -\frac{\partial H}{\partial q}(p(t), q(t)) + g_\theta(q)\beta(q),$$

where g_θ is some neural network. After the system has been learnt from data, then controllers may be synthesised from this description [ZDC20b].

2.2.2.3 Lagrangian neural networks

One weakness of the Hamiltonian approach is that it assumes knowledge of the canonical coordinates q, p. In general our observed data from a dynamical system may not match up against this canonical structure.

An alternative is to instead parameterise the Lagrangian. Given positions $q \in \mathbb{R}^d$ and velocities $\dot{q} = \frac{\mathrm{d}q}{\mathrm{d}t} \in \mathbb{R}^d$, a Lagrangian is parameterised as some neural network function of them both, $\mathcal{L}_\theta(q, \dot{q})$. The Euler–Lagrange equations state that a system with Lagrangian $\mathcal{L}_\theta$ evolves according to

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}_\theta}{\partial \dot{q}} = \frac{\partial \mathcal{L}_\theta}{\partial q}.$$

Rearranging, we may obtain

$$\ddot{q} = \left(\frac{\partial^2 \mathcal{L}_\theta}{\partial^2 \dot{q}}\right)^{-1} \left(\frac{\partial \mathcal{L}_\theta}{\partial q} - \frac{\partial^2 \mathcal{L}_\theta}{\partial q \partial \dot{q}} \dot{q}\right),$$

where $\frac{\partial^2 \mathcal{L}_\theta}{\partial^2 \dot{q}}$ is a Hessian and so $(\frac{\partial^2 \mathcal{L}_\theta}{\partial^2 \dot{q}})^{-1}$ denotes a matrix inverse. Once again this defines a dynamical system which may be fitted directly to data as described for universal differential equations. See [Cra+20b].

2.2.3 Continuous normalising flows

We now switch from supervised learning to unsupervised learning. Suppose we observe some distribution $\mathbb{P}$ with a density π over some state space $\mathbb{R}^{d_1 \times \cdots \times d_k}$. We wish to learn an approximation to $\mathbb{P}$.

For example we may have $\mathbb{R}^{d_1 \times \cdots \times d_k} = \mathbb{R}^{3 \times 32 \times 32}$, and $\mathbb{P}$ may denote a probability distribution over 'pictures of cats', from which we have empirical samples. By learning a generative model approximating π, we may produce synthetic pictures of cats. (An important task.)

Let $d = \prod_{m=1}^{k} d_m$ and for simplicity we replace $\mathbb{R}^{d_1 \times \cdots \times d_k}$ with $\mathbb{R}^d$.

Consider the random neural ODE defined by

$$y(0) \sim \mathcal{N}(0, I_{d \times d}), \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)) \text{ for } t \in [0, T]. \tag{2.4}$$

We seek to train this model such that the distribution of $y(1)$ (induced by the push-forward of $y(0) \sim \mathcal{N}(0, I_{d \times d})$ by $y(0) \mapsto y(1)$) is approximately $\mathbb{P}$. This is called a continuous normalising flow (CNF) [Che+18b; Gra+19]. See Figure 2.1.

2.2.3.1 Sampling

Sampling from a trained model is straightforward: sample $y(0) \sim \mathcal{N}(0, I_{d \times d})$ and then solve (2.4).

2.2.3.2 Instantaneous change of variables

We still need to train the model. We will proceed via maximum likelihood, which means that we need a tractable expression for the density of the distribution of $y(1)$.

Theorem 2.3 (Instantaneous change of variables)**.** *Recall equation* (2.4)*. Assume* $f_\theta = (f_{\theta,1}, \ldots, f_{\theta,d})$ *is Lipschitz continuous. Let*

$$p_\theta \colon [0, T] \times \mathbb{R}^d \to \mathbb{R},$$

where $p_\theta(t, \cdot)$ *is the density of* $y(t)$ *for each time* $t \in [0, T]$*. (In some works written informally as '*$p(y(t))$*'.) The subscript* θ *in* p_θ *denotes the dependence on* f_θ*.*

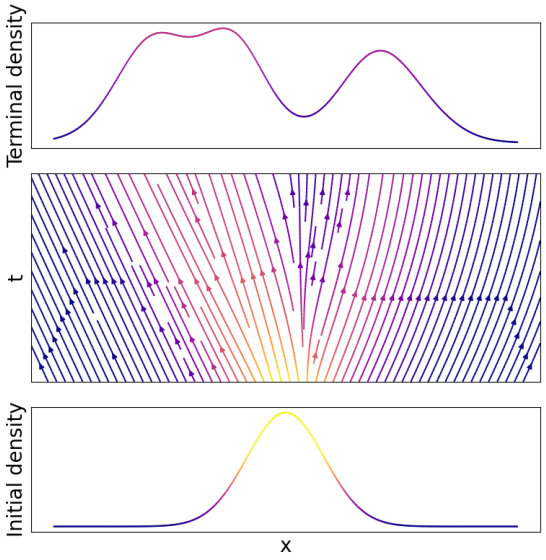


Figure 2.1: A continuous normalising flow continuously deforms one distribution into another distribution. The flow lines show how particles from the base distribution are perturbed until they approximate the target distribution.

Then p_θ evolves according to the differential equation[3]

$$\frac{\mathrm{d}}{\mathrm{d}t}\big(t \mapsto \log p_\theta(t, y(t))\big)(t) = -\sum_{k=1}^{d} \frac{\partial f_{\theta,k}}{\partial y_k}(t, y(t)), \tag{2.5}$$

where $y = (y_1, \ldots, y_d) \in \mathbb{R}^d$.

The right hand side of (2.5) is the divergence of f, or equivalently the trace of the Jacobian of f. The latter description draws the analogy to the change of variables formulas for normalising flows (Appendix A.2).

See [Che+18b, Appendix A] for a straightforward proof.

Remark 2.4. *The SDE theorist will find this expression familiar. It is the Fokker–Planck equation for deterministic dynamics, subject to a random initial condition. It has been carefully written so that the right hand side is independent of the unknown p_θ.*

Training By solving (2.5) we can train a CNF via maximum likelihood. Given any terminal condition $x \in \mathbb{R}^d$, let $y(t, x)$ denote the solution to the ODE

$$y(T, x) = x, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t, x) = f_\theta(t, y(t, x)) \text{ for } t \in [0, T] \tag{2.6}$$

which will be solved backwards in time from $t = T$ to $t = 0$.

[3]Actually, just an integral: $\log p_\theta$ does not appear on the right hand side.

Given a batch of empirical samples $y_1, \ldots, y_N \in \mathbb{R}^d$, maximum likelihood states that with respect to θ, we should minimise

$$-\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(T, y_i).$$

Substituting in (2.5), we obtain

$$-\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(T, y_i) = -\frac{1}{N}\sum_{i=1}^{N}\left[\log p_\theta(0, y(0, y_i)) - \int_0^T \sum_{k=1}^{d} \frac{\partial f_{\theta,k}}{\partial y_k}(t, y(t, y_i))\,\mathrm{d}t\right]. \tag{2.7}$$

This is now possible to evaluate.

1. Starting from some empirical sample $y_i \in \mathbb{R}^d$, we may solve equation (2.6) backwards-in-time from $t = T$ to $t = 0$.
2. As the solution progresses we obtain $y(t, x)$ for $t = T$ to $t = 0$. This is an input to the right hand side of (2.7). This integral may be solved as part of this backwards-in-time solve – just concatenate the integral together with (2.6) to form a system of differential equations.
3. Finally, evaluate $\log p_\theta(0, y(0, y_N))$ – recalling that $p_\theta(0, \cdot\,)$ is taken to be a normal distribution – and add it together with the value of the integral in order to obtain a value for (2.7).

Having evaluated (2.7), it is backpropagated and the parameters θ updated via gradient descent.[4] Note that backpropagation is a 'reverse time' procedure. In summary, and as we have already performed one reversal:

- Evaluating (2.7) involves solving from $t = T$ to $t = 0$;
- Backpropagating through (2.7) is an operation progressing from $t = 0$ to $t = T$;
- Additionally, note that sampling involves solving from $t = 0$ to $t = T$, and is only performed at inference time.

2.2.3.3 Example

As a fun example, consider a greyscale image, which we may regard as a map $f\colon [0, 1]^2 \to [0, 255]$. We may fit a continuous normalising flow to f, treating f as the unnormalised density for a probability distribution over $\mathbb{R}^2 \supseteq [0, 1]^2$. A selection of images, and some CNFs that have learnt to approximate them, are shown in Figure 2.2.

[4] And as the 'forward pass' involved a derivative, then the backward pass will compute a second derivative; this is fine.

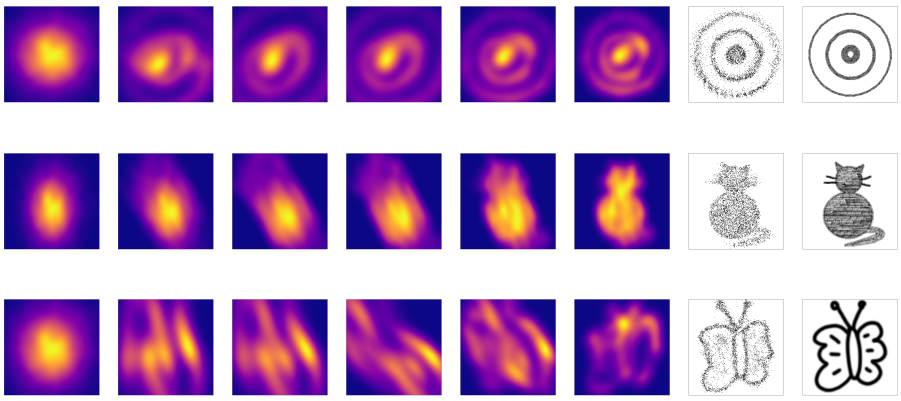


Figure 2.2: Continuous normalising flow example. **From top to bottom:** target, cat, butterfly. **From left to right:** the first six pictures show the evolution of the distribution of the CNF as it integrates from $t = 0$ to $t = T$, transforming a normal distribution into the desired distribution. The second to last picture shows samples from the learnt CNF. The final picture shows the image used to specify the desired distribution.

We see that CNFs are capable of learning relatively complex two-dimensional distributions, including those with multiple modes (such as the different concentric rings of the target), and those with fine-scale 'filaments' stretching away from the main part of the distribution (such as the whiskers or tail of the cat).

See Appendix D.1 for further details of this experiment. The code is available as an example in Diffrax [Kid21a].

CNFs are a highly flexible approach to modelling probability distributions. [Gra+19; Fin+20a] apply this approach to image generation. That is, the samples from the learnt distribution are images, rather than the whole distribution resembling an image as above. [Yan+19] represent 3D models as distributions (much like the above example representing a picture as a 2D distribution), and use this approach to generate point clouds of the model.

2.2.3.4 Efficient estimation of the trace-Jacobian

Note how (2.5), and thus (2.7), involve evaluating the expression $\sum_{k=1}^{d} \frac{\partial f_{\theta,k}}{\partial y_k}(t, y)$. This is possible simply via autodifferentiation software: evaluate the neural network $f_\theta(t, y)$, and then backpropagate.

There is one foible. Autodifferentiation calculates a product of Jacobians (Appendix A.1), which this expression is not. It may be calculated by performing $k = 1, \ldots, d$ such operations, but this implies a relatively expensive $\mathcal{O}(d^2)$ cost. Each evaluation of $f_\theta \colon \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ requires at least $\mathcal{O}(d)$ work. Each subsequent autodifferentiation operation also requires $\mathcal{O}(d)$ work; so far for a total of still only $\mathcal{O}(d)$. That we must

make $k = 1, \ldots, d$ such calls is what raises this to $\mathcal{O}(d^2)$ cost.

We can do better.

Hutchinson's trace estimator Let $A \in \mathbb{R}^{d \times d}$ be any matrix. Let ε be a random variable over $\mathbb{R}^d$ such that $\mathbb{E}[\varepsilon] = 0 \in \mathbb{R}^d$ and $\text{Cov}[\varepsilon] = I_{d \times d}$. (For example, a multivariate normal or Rademacher random variable.) Then

$$\text{tr}(A) = \mathbb{E}_\varepsilon[\varepsilon^\top A \varepsilon].$$

The Monte-Carlo approximation derived from this equation is known as Hutchinson's trace estimator [Hut89].

The trace-Jacobian That the right hand side of (2.5) is a trace-Jacobian now proves useful. We have that

$$\sum_{k=1}^{d} \frac{\partial f_{\theta,k}}{\partial y_k}(t, y) = \text{tr}\left(\frac{\partial f_\theta}{\partial y}(t, y)\right) = \mathbb{E}_\varepsilon\left[\varepsilon^\top \frac{\partial f_\theta}{\partial y}(t, y)\varepsilon\right].$$

Substituting into (2.7) we obtain

$$-\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(T, y_i) = -\frac{1}{N}\sum_{i=1}^{N}\left[\log p_\theta(0, y(0, y_i)) - \mathbb{E}_\varepsilon\left[\int_0^T \varepsilon^\top \frac{\partial f_\theta}{\partial y}(t, y(t, y_i))\varepsilon \, \mathrm{d}t\right]\right]. \tag{2.8}$$

In practice this expectation will often be approximated by a single Monte-Carlo sample, which as in (2.8) is held constant for the duration of the integration. Training already involves averaging over the batch of data $i = 1, \ldots, N$ and so further Monte-Carlo samples are often unnecessary.

And now for punchline: the integrand of (2.8) may be computed in only $\mathcal{O}(d)$ work. First $\varepsilon^\top \frac{\partial f_\theta}{\partial y}(t, y(t, y_i))$ may be computed as vector-Jacobian product (requiring only $\mathcal{O}(d)$ work), and then this is combined with the final ε via a simple dot product (also only $\mathcal{O}(d)$ work). Overall this produces an unbiased estimate of the divergence.

2.2.3.5 Comparison to normalising flows

Recall the discussion on normalising flows from Appendix A.2. In both cases, a change in log-probability densities is described in terms of the Jacobian of the transformation.

Note the difference in computational complexity. In the general normalising flow setting, the log-determinant-Jacobian costs $\mathcal{O}(d^3)$ work to evaluate (and backpropagate through). Here it has been reduced to just $\mathcal{O}(d^2)$ or $\mathcal{O}(d)$ work.

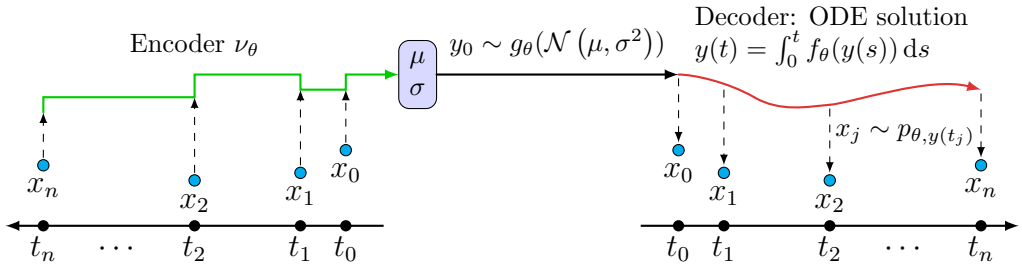


Figure 2.3: Overview of latent ODE model.

2.2.4 Latent ODEs

The previous section considered a generative model for data from some distribution without a time-varying component. For example, a static picture of a cat, rather than samples from a dynamical system evolving in time. We now consider the case that the distribution has an intrinsic time-varying component – for example, it may be a distribution over time series. Once again, we wish to model this distribution.

Consider the space of d-dimensional irregularly-sampled time series

$$\mathcal{TS}(\mathbb{R}^d) = \left\{((t_0, x_0), \ldots, (t_n, x_n)) \;\middle|\; n \in \mathbb{N}, t_j \in \mathbb{R}, x_j \in \mathbb{R}^d, t_0 = 0, t_j < t_{j+1}\right\}.$$

For ease of presentation we suppose this is fully-observed (without missing data), but the following construction extends immediately to the partially-observed case too.

We proceed by constructing a VAE. Figure 2.3 provides a summary of the construction we about to present. This is termed a latent ODE [Che+18b; RCD19].

Remark 2.5. *The use of a VAE raises the question of whether other generative approaches (GANS, ...) may be employed. The answer is yes, and indeed Chapter 4 (Neural Stochastic Differential Equations) will be almost entirely dedicated to the problem of generative time series models.*

Decoder Fix $d_l, d_m > 0$ as the dimensionality of two latent spaces. (Typically $d_l, d_m \gg d$.) Let

$$f_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}^{d_l},$$
$$g_\theta \colon \mathbb{R}^{d_m} \to \mathbb{R}^{d_l}$$

be neural networks parametrised by learnt parameters θ. Let $p_{\theta,y} \colon \mathbb{R}^{d_l} \to [0, \infty)$ be some probability density parameterised by $y \in \mathbb{R}^{d_l}$ and learnt parameters θ. (For simplicity of notation we stack all learnt parameters together into a single vector θ.)

Given $z \in \mathbb{R}^{d_m}$, let $y_0 = g_\theta(z) \in \mathbb{R}^{d_l}$ and let $y(t, y_0)$ be the solution of the neural ODE

$$y(0, y_0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t, y_0) = f_\theta(y(t, y_0)). \tag{2.9}$$

For each time t we consider $p_{\theta,y(t,y_0)}$. The full collection of $t \mapsto p_{\theta,y(t,y_0)}$ is the output of the model.

That is, given some input $z \in \mathbb{R}^{d_m}$, it is mapped into the latent space $\mathbb{R}^{d_l}$, from which the ODE y evolves. At each time the latent value parameterises a probability distribution.

This is the decoder of the VAE.

Example 2.6. *Frequently $p_{\theta,y}$ may simply be taken to be a Gaussian with fixed variance: let $\ell_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}^d$ be affine and let $p_{\theta,y}$ be the density of $\mathcal{N}(\ell_\theta(y), I_{d_l \times d_l})$. We will discuss other choices of $p_{\theta,y}$ in a moment.*

For any $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n)) \in \mathcal{TS}\big(\mathbb{R}^d\big)$, let

$$\rho_{\theta,y_0}(\mathbf{x}) = \prod_{j=0}^{n} p_{\theta,y(t_j,y_0)}(x_j),$$

which corresponds to the probability density of a full time series $\mathbf{x}$, rather than of just a single observation at a single point in time.

Encoder The encoder of the VAE is some $\nu_\theta \colon \mathcal{TS}\big(\mathbb{R}^d\big) \to \mathbb{R}^{d_m} \times (0, \infty)^{d_m}$; frequently an RNN or a neural CDE (Chapter 3).

The encoder output is the statistics of a multivariate normal distribution with diagonal covariance; we denote this by $q_{\theta,\mathbf{x}} = \mathcal{N}(\mu_{\theta,\mathbf{x}}, \mathrm{diag}(\sigma_{\theta,\mathbf{x}})^2)$ with $(\mu_{\theta,\mathbf{x}}, \sigma_{\theta,\mathbf{x}}) = \nu_\theta(\mathbf{x})$.

If the encoder is an RNN or neural CDE it will sometimes be run backwards-in-time over the input time series, so that the decoder starts where the encoder ends.

Training Given a batch or dataset of time series $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathcal{TS}\big(\mathbb{R}^d\big)$, then the end-to-end optimisation criterion is to minimise θ with respect to

$$\frac{1}{N} \sum_{i=1}^{N} \Big[\mathbb{E}_{y_0 \sim q_{\theta,\mathbf{x}_i}} \left[-\log \rho_{\theta,y_0}(\mathbf{x}_i) \right] + \mathrm{KL}\left(q_{\theta,\mathbf{x}_i} \| \mathcal{N}(0, I_{d_l \times d_l}) \right) \Big].$$

This is simply the standard VAE optimisation criterion, and provided $p_{\theta,y}$ is 'reasonable' (for example, a Gaussian), then this expression may be evaluated and backpropagated through in the usual way. The first term ensures that the decoder learns to replicate its input samples; the second term ensures that the initial distribution in the latent space matches a known distribution, which may be sampled from at inference time.

Sampling Sampling from the model is straightforward in the usual way for VAEs: sample some $z \sim \mathcal{N}(0, I_{d_m \times d_m})$, evaluate $y_0 = g_\theta(z)$, and evaluate (2.9) forward in time. $p_{\theta,y(t,y_0)}$ is the model output. If a point statistic is required (for example, just a sample from the model) then the mean of $p_{\theta,y(t,y_0)}$ may be returned.

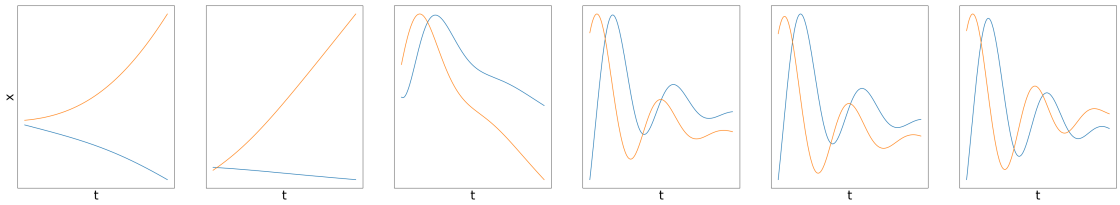


Figure 2.4: Plots of samples $y\colon [0, 12] \to \mathbb{R}^2$ drawn from the latent ODE model. The leftmost picture is a sample from the untrained model. The rightmost picture is a sample from a fully-trained model. In between are samples from partially-trained models. Quality increases as training proceeds.

Choice of distribution The choice of $p_{\theta,y}$ is dependent on what behaviour is desired when sampling during inference time. If only a point statistic is required then the choice of $p_{\theta,y}$ is essentially just a choice of loss function, and simple choices like Gaussian distributions (Example 2.6) or Laplace distributions (whose log-likelihood is the L^1 distance) are sensible.

If the full model output $p_{\theta,y(t,y_0)}$ is of interest – for example to perform uncertainty quantification – then more expressive choices of $p_{\theta,y}$ may be of interest. For example, [Den+21] consider taking it to be a normalising flow (and additionally consider replacing the latent ODE with a latent SDE; see Chapter 4).

2.2.4.1 Examples

As a simple example, consider a dataset of decaying oscillators. That is, a 2-dimensional time series consisting of (discrete observations of)

$$y(t) = \exp(At)\, y_0 \tag{2.10}$$

with $y_0, y(t) \in \mathbb{R}^2$, $A \in \mathbb{R}^{2\times 2}$, and such that the eigenvalues of A are complex with negative real component. Samples look like decaying sine and cosine waves.

We take $y_0 \sim \mathcal{N}(0, I_{2\times 2})$, and generate sample data from (2.10) at irregularly sampled timestamps over $[0, 3]$. The timestamps are not regularly spaced nor are they consistent between different batch elements.

We fit a latent ODE to this dataset. At test time, we solve the ODE over the larger interval $[0, 12]$. See Figure 2.4 for some samples generated from this model. We see that by the end of training, excellent samples are produced, even though they are over a time interval four times larger than the model was trained on.

See Appendix D.2 for precise details. The code is available as an example in Diffrax [Kid21a].

Irregular sampling The continuous-time approach handles several irregular kinds of sampling without issue: the input data is not regularly spaced, nor are different

batch elements sampled at the same times.

Meanwhile, the output is over the (continuous-time) interval $[0, 12]$, so that we are obtaining samples at all times. This is unlike the analogous RNN, which would be restricted to producing outputs only at prespecified discrete timestamps.

Extrapolation Figure 2.4 shows that the latent ODE has successfully reproduced this dataset. Moreover it exhibits good extrapolation qualities over an interval four times longer than the interval it was trained on.

Other examples Many other types of time series problem may be considered. For example [RCD19] apply a latent ODE to model the dynamics of a small (simulated) frog jumping into the air; [DFD20] consider applications to reinforcement learning; [SM21] combine latent ODEs with changepoint detection algorithms to model switching dynamical systems.

We additionally direct the reader towards Chapter 4, in which neural SDEs will also be used to model (much more general) distributions over time series.

2.2.4.2 Sequence-to-sequence models

Essentially the same construction may be used in the construction of sequence-to-sequence models, for example to perform time series forecasting. The encoder (an RNN or neural CDE, see Chapter 3) runs over the input time series; the decoder (a neural ODE or neural SDE, see Chapter 4) produces the forecasted sample.

2.2.5 Residual networks

In Section 1.1.4 we saw that residual networks are the explicit Euler discretisation of a neural ODE.

Correspondingly the theory of dynamical systems offers ways to derive variant residual networks with favourable properties.

2.2.5.1 Rotational vector fields

[HR17] consider replacing the forward pass of a residual network

$$y_{j+1} = y_j + f_\theta(y_j)$$

with

$$\begin{aligned} y_{j+1} &= y_j + \sigma(K_j z_{j+1} + b_j), \\ z_{j+1} &= z_j - \sigma(K_j^\top y_j + b_j) \end{aligned} \tag{2.11}$$

for some weights and biases K_j, b_j and some choice of activation function σ. This corresponds to a semi-implicit Euler discretisation of the neural ODE

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ z \end{pmatrix}(t) = \sigma\left(W(t)\begin{pmatrix} y(t) \\ z(t) \end{pmatrix} + b(t)\right)$$

where

$$W(t) = \begin{pmatrix} 0 & K(t) \\ -K(t)^\top & 0 \end{pmatrix}.$$

Correspondingly, the Jacobian of the right hand side is

$$\operatorname{diag}\left(\sigma'\left(W(t)\begin{pmatrix} y(t) \\ z(t) \end{pmatrix} + b(t)\right)\right) W(t)$$

Many activation functions are monotonic; if this is the case then the Jacobian is the product of a diagonal matrix with positive entries, and an antisymmetric matrix, and as such the Jacobian has pure-imaginary eigenvalues.[5]

This means that the vector field is 'purely rotational': eigenvalues with positive real part drive expansion; eigenvalues with negative real part cause contraction, but zero real part produces neither. Correspondingly, (2.11) is largely immune to vanishing/exploding gradient issues.

Remark 2.7. *The trade-off, however, is a potential reduction in expressivity. Purely rotational vector fields are volume preserving (divergence-free). Non-volume-preservation is often important for expressivity. (It is even part of the name of Real Non-Volume Preserving flows [DSB17].)*

This issue of volume preservation may be partially ameliorated by working in a higher dimensional space; see Section 2.3.3.2 later.

2.2.5.2 Momentum residual networks

[San+21] consider replacing the forward pass through a residual network with

$$\begin{aligned} v_{j+1} &= \gamma v_j + (1-\gamma) f_\theta(y_j) \\ y_{j+1} &= y_j + v_{j+1} \end{aligned} \tag{2.12}$$

for $\gamma \in (0, 1)$. ($\gamma = 0.9$ would be typical.)

Reversibility The key property of such networks is that they are *reversible*: whilst (2.12) computes (v_{j+1}, y_{j+1}) from (v_j, y_j), it also possible to reconstruct (v_j, y_j) from

[5]Proof: let D be positive diagonal with square root A. Let W be antisymmetric. Then DW is similar to AWA, which is antisymmetric and as such has pure-imaginary eigenvalues.

(v_{j+1}, y_{j+1}) via

$$\begin{aligned} y_j &= y_{j+1} - v_{j+1} \\ v_j &= \frac{1}{\gamma}(v_{j+1} - (1-\gamma)f_\theta(y_j)). \end{aligned} \tag{2.13}$$

This dramatically improves the memory efficiency of the network, at the cost of some extra computation. When backpropagating through (2.12), the intermediate values y_n need not be stored (like they would be for the corresponding residual network). Instead, (2.13) means they can be recomputed on-demand as backpropagation proceeds.

This represents a refinement of similar ideas in [Gom+17; Cha+18], and more generally the topic of *invertible neural networks* [Beh+19].

As a neural ODE Let $\varepsilon = 1/(1-\gamma)$. Then (2.12) is given by the semi-implicit Euler method, with unit step size, applied to

$$\varepsilon \frac{\mathrm{d}^2 y}{\mathrm{d}t^2}(t) + \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(y(t)). \tag{2.14}$$

Connection to reversible solvers Momentum networks are reversible because the semi-implicit Euler method is reversible. Running the solver forwards in time, then backwards in time, will recover the same numerical solution. This is sometimes described as saying that there are matching truncation errors on the forward and backward solves.

Reversible solvers come strongly recommended for use with neural differential equations for the same reason as here: they allow for backpropagation that is both time and memory efficient (Section 5.3.2). As such they are of substantial interest, and moreover in general do not require the second-order structure that (2.14) exhibits.

2.2.5.3 Alternative integration schemes

Other off-the-shelf integration schemes may be substituted for the explicit Euler method.

For example [Lu+17a] consider linear multistep methods and [Hab+19] consider IMEX methods. PolyNet [Zha+17] considers operations of the form

$$y_{j+1} = y_j + f_\theta(y_j) + f_\theta(f_\theta(y_j)),$$

which if f_θ is a linear contraction, is an approximation to the implicit Euler method

$$\begin{aligned} y_{j+1} &= y_n + f_\theta(y_{j+1}) \\ &= (I - f_\theta)^{-1}(y_j) \\ &= (I + f_\theta + f_\theta^2 + f_\theta^3 + \cdots)(y_{j+1}). \end{aligned}$$

We note that the advantages of switching to a different integration scheme are strongest when it exhibits particular additional properties, like symplecticity as in Section 2.2.5.1 or reversibility as in Section 2.2.5.2.

2.3 Choice of parameterisation

So far we have touched only lightly on the parameterisation of the vector field f_θ. (Although we have discussed some mathematically-inspired parameterisations, such as Hamiltonian-based parameterisations in Section 2.2.2.2.)

Should f_θ be a feedforward network, convolutional network, residual network, ...? Should it use batch normalisation? What kinds of activation functions are appropriate? And so on.

Good architectural choices and good choices of optimiser are often crucial for success. However (even with the following guidelines) it is not always clear what good choices are. Frequently this is still just a matter of hyperparameter optimisation – or perhaps 'try it and see what works'.

2.3.1 Neural architectures

Nearly every work uses either a feedforward or convolutional neural network for the vector field f_θ. Feedforward networks are straightforward: simply concatenate t and $y(t)$ together as inputs. These are what are typically used when the data is anything other than an image.

If the data y_0 has the (channel, height, width) structure of an image, then a suitable vector field may be obtained by using convolutional layers. Recall that the input and output of $f_\theta(t, \cdot)$ must be the same size. This typically means either using padding, or combining convolutional layers with transposed convolutional layers. Time t is often appended to $y(t)$ as an additional channel.

Remark 2.8. *Other parameterisations are occasionally used. For example [Pol+19; Cra+20a; Den+19; Cha+21] consider graph neural networks, which can for example encode equivariance with respect to permutations of the input points. (Such as may be exhibited in many physical systems; for example the positions of n equally-sized masses evolving under gravity.)*

Much of the following discussion carries through to this setting, although we will not discuss graph-structured networks and graph-structured data in detail here.

2.3.1.1 Activation functions

The theory of backpropagating through ODEs does technically ask that the vector field (and thus the activation function) be continuously differentiable (Section 5.1),

which ReLUs are not.

As such continuously differentiable activation functions like SiLU [HG16; EUD17; RZL17], softplus, or tanh are typically used.[6]

Despite this theoretical point, ReLU activations are still often used successfully in practice.

2.3.1.2 Normalisation

Normalisation schemes, such as batch normalisation and layer normalisation [IS15; BKH16], are typically not used, at least within the vector field f_θ. For batch normalisation, this is because the same neural network f_θ is evaluated at $y(t)$ for different t, and each might have different statistical properties. This is the same problem that occurs when using batch normalisation in recurrent neural networks [BKH16; Coo+17].

Meanwhile layer normalisation lacks a satisfying explanation for its lack of efficacy, but at least for CNFs it has been reported that this typically breaks training [Che20].

2.3.1.3 Initialisation

Initialising the neural vector fields close to zero often improves training, it being easier to perturb a nearly constant $t \mapsto y(t)$ than random initial dynamics. For most neural architectures this may be accomplished by choosing the initial parameters θ close to zero.

2.3.2 Non-autonomy

We have deliberately chosen to include t as an input to the vector field f_θ. A residual network has different layers at different depths. Analogously, neural ODE models usually exhibit higher modelling capacity by allowing f_θ to depend on the 'continuous depth' parameter t. Such differential equations are referred to as being *non-autonomous*.

This can be handled simply by concatenating t and $y(t)$ together as inputs to f_θ. A far more expressive choice is to additionally explicitly encode certain time dependencies.

[6] [Cra21a] cooked up, and reports being fond of, the 'squareplus' activation $x \mapsto \frac{1}{2}x + \frac{1}{2}\sqrt{x^2 + 4}$.

2.3.2.1 Depth discretisation: stacking

One straightforward and effective approach is to parameterise f_θ in piecewise fashion as several different networks, selected based on the value of t. For example,

$$f_\theta(t, y) = \begin{cases} f_{\theta_1,1}(t, y) & t \in [t_0, t_1] \\ \quad \vdots & \\ f_{\theta_n,n}(t, y) & t \in [t_{n-1}, t_n] \end{cases}$$

where $\theta = (\theta_1, \ldots, \theta_n)$, and each θ_j is itself some vector of parameters.

In principle each $f_{\cdot,1}, \ldots, f_{\cdot,n}$ could represent different architectures. Often $f_{\cdot,j}$ will all be the same neural architecture, and differ only in which parameter vector θ_i they depend upon.

Two options must be considered when using this architecture in practice, with a numerical differential equation solver: whether to use a single call to an ODE solver over $[t_0, t_n]$, or whether to solve over each $[t_i, t_{i+1}]$ region separately, and call an ODE solver n times. Both options are valid but both introduce details that one should be aware of; we defer this numerical discussion to Section 5.3.3.

2.3.2.2 Spectral discretisation

Let $\psi_j \colon [0, T] \to \mathbb{R}$ be some family of (smooth) functions parameterised by $j \in \{1, \ldots, n\}$. Take the parameter vector θ to be such that $\theta = (\theta_1, \ldots, \theta_n)$ with $\theta_j \in \mathbb{R}^{d_\theta}$ for some $d_\theta \in \mathbb{N}$. Now define

$$\alpha_\theta(t) = \sum_{j=1}^{n} \theta_j \psi_j(t).$$

Then another choice of non-autonomy is given by

$$f_\theta(t, y(t)) = \widetilde{f}_{\alpha_\theta(t)}(t, y(t)),$$

where $\widetilde{f}$ is some fixed neural network architecture which at time t uses parameters $\alpha_\theta(t) \in \mathbb{R}^{d_\theta}$.

The choice of ψ_j is up to us. Ideally they should be quite different to each other, for the greatest possible expressivity of the model. For example they could be chosen as Chebyshev polynomials, or as a truncated Fourier basis of sines and cosines (which is what motivates the terminology 'spectral discretisation' [Mas+20]).

2.3.2.3 Hypernetworks

Another choice is to let the parameters of the neural ODE be themselves parameterised as the solution of a neural ODE.

That is, let $\alpha\colon [0, T] \to \mathbb{R}^{d_\alpha}$ be the solution of the neural ODE

$$\alpha(0) = \alpha_\theta \qquad \frac{\mathrm{d}\alpha}{\mathrm{d}t}(t) = g_\theta(t, \alpha(t)),$$

with learnt parameters θ, vector field $g_\theta\colon \mathbb{R} \times \mathbb{R}^{d_\alpha} \to \mathbb{R}^{d_\alpha}$, and learnt initial condition α_θ.

We then let the hidden state y of our 'original' neural ODE evolve according to

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = \widetilde{f}_{\alpha(t)}(t, y(t)),$$

where $\widetilde{f}$ is some fixed neural network architecture which at time t uses parameters $\alpha(t) \in \mathbb{R}^{d_\alpha}$.

In practice these two differential equations may be concatenated and solved simultaneously as a system. Overall this may be seen as just a neural ODE as originally formulated, with a particular beneficial structure to its vector field. See [Zha+19; Cho+20].

2.3.2.4 Variant layers

Other high-performing time-dependent layers may be dreamt up. For example (and inspired by [Gra+19]) the example CNF seen in Section 2.2.3.3 uses an MLP whose affine layers are replaced with layers of the form

$$(x, t) \mapsto (Ax + b) * \sigma(ct + d) + et$$

where $x \in \mathbb{R}^{d_1}$, $t \in \mathbb{R}$, $A \in \mathbb{R}^{d_2 \times d_1}$, $b, c, d, e \in \mathbb{R}^{d_2}$, σ denotes the sigmoid function, and $*$ denotes elementwise multiplication.

The dependency on the time t is coming in at each layer of the MLP, rather than being concatenated with $y(t)$ as just another input.

This is reminiscent of gating procedures in GRUs and LSTMs.

2.3.2.5 Enforcing autonomy

One exception to the above procedure sometimes occurs when using neural ODEs equations for time series problems, such as with a latent ODE (Section 2.2.4). In this case, we may sometime suppose that the underlying dynamics are not time-dependent, and would instead prefer to remove t as an input. (The same will often also be true of the upcoming neural CDEs and neural SDEs in Chapters 3 and 4.)

2.3.3 Augmentation

For a moment let us focus on performing image classification with neural ODEs (Section 2.2.1); a problem chosen for its simplicity. In Section 2.2.1, the input to the

model was the same size as the hidden state: both the input picture and hidden state were of shape $\mathbb{R}^{3\times32\times32}$. In general however this is neither necessary nor desirable.

'Augmentation' refers to the practice of inserting an affine map between input and initial value, to increase the dimension of the hidden state. That is, given some input $x \in \mathbb{R}^d$, the initial value of the ODE is taken to be $y(0) = g_\theta(x)$ for some learnt $g_\theta \colon \mathbb{R}^d \to \mathbb{R}^{d_l}$ with $d_l > d$, rather than simply $y(0) = x$. We have

$$y(0) = g_\theta(x), \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

Standard choices of g_θ are either zero augmentation: $g_\theta(x) = [x, 0]$, learnt augmentation: $g_\theta(x) = [x, \widetilde{g}_\theta(x)]$ for some learnt $\widetilde{g}_\theta$, or just an affine map: $g_\theta(x)$ is learnt and affine. The choice is usually unimportant; the increase in dimensionality is the main point. In each case, the output of the model is still obtained by applying some affine map $\ell_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}^{d_o}$ to $y(T)$, with $d_o \in \mathbb{N}$ the desired output dimensionality.

This improves model performance dramatically. The reason is that the continuous flow of an ODE is incapable of modifying the topology of its input – so staying in the same space means that topological properties of the input manifold (in the sense of the manifold hypothesis; Appendix A.5) are necessarily preserved. This is a statement we will make precise in Section 2.4, by describing the universal approximation properties of neural ODEs.

Returning now to the general setting (beyond just image classification), we have already seen an example of augmentation: the latent ODE (Section 2.2.4) evolved in some higher-dimensional space $\mathbb{R}^{d_l}$, and used an affine map to $\mathbb{R}^d$ to obtain the output.

(Conversely, note that CNFs cannot use augmentation: as with all normalising flows, it is a requirement of the construction that every operation be bijective.)

Remark 2.9. *Lifting into a higher-dimensional space may be regarded as a relaxation of the Markov property. For $s < t$ then the output $\ell_\theta(y(s))$ does not completely determine $\ell_\theta(y(t))$. In contrast $y(s)$ does determine $y(t)$. (Whether y is the output of an unaugmented neural ODE or the latent value of an augmented neural ODE.)*

The Markov setting can be very beneficial if the problem is known to exhibit this structure, in particular when modelling physical systems. If the data is densely sampled then it can then be possible to avoid the ODE solve entirely: estimate $\frac{\mathrm{d}y}{\mathrm{d}t}$ with finite differences and do direct supervised regression of $\frac{\mathrm{d}y}{\mathrm{d}t}$ against (t, y). See [RRS21; RPK18] for variations on this idea. The Markov setting is also the one used for symbolic regression (Section 6.1).

In general however the Markov setting is a restrictive assumption usually worth avoiding. The Markov/non-Markov distinction is an important one to watch out for in the NDE literature, as many works have implicitly restricted to the Markov setting without discussion.

2.3.3.1 Second-order-augmentation

[Nor+20] introduce an interesting variant on this: they take $d_l = 2d$ and structure the vector field so that the extra dimensions correspond to velocities. For example using a learnt augmentation,

$$y(0) = \begin{bmatrix} x \\ g_\theta(x) \end{bmatrix}, \qquad y(t) = \begin{bmatrix} s(t) \\ v(t) \end{bmatrix}, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = \begin{bmatrix} v(t) \\ f_\theta(t, s(t), v(t)) \end{bmatrix}.$$

This may also be written as a second-order neural ODE $\frac{\mathrm{d}^2 s}{\mathrm{d}t^2}(t) = f_\theta\left(t, s(t), \frac{\mathrm{d}s}{\mathrm{d}t}(t)\right)$.

This is a choice that makes particular sense if using neural ODEs to model an oscillatory dynamical system.

2.3.3.2 Augmenting rotational vector fields

Recall Remark 2.7. Augmentation is one way to ameliorate the lack of expressivity of rotational vector fields.

Example 2.10. *Let $a = -1, b = 0, c = \frac{1}{2} \in \mathbb{R}$, and suppose we wish to classify $\{a, c\}$ versus $\{b\}$, by constructing a neural ODE followed by an affine layer. We will prove in Section 2.4.1 that this is actually impossible with an unaugmented neural ODE; there does not exist a flow whose terminal values linearly separate $\{a, c\}$ from $\{b\}$.*

However, projecting these into $a = (-1, 0), b = (0, 0), c = (\frac{1}{2}, 0) \in \mathbb{R}^2$, then the (volume-preserving) flow

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t) = y$$
$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = -x$$

will linearly separate $\{a, c\}$ from $\{b\}$ after any arbitrarily small amount of time.

2.4 Approximation properties

We now examine the universal approximation properties of neural ODEs, as maps from their initial value to their terminal value. See Appendix A.3 for an introduction to the topic of universal approximation.

2.4.1 'Unaugmented' neural ODEs are not universal approximators

Consider the map $y(0) \mapsto y(T)$, where $y\colon [0, T] \to \mathbb{R}^d$ solves some neural ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

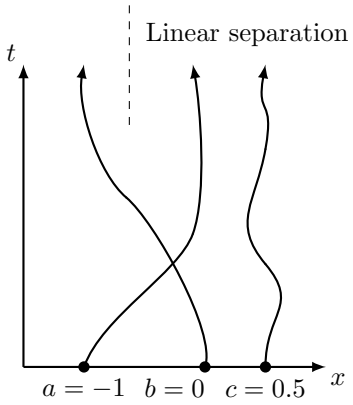


Figure 2.5: ODE flows need to cross to linearly separate $\{a, c\}$ from $\{b\}$.

What functions can this approximate?

Unfortunately, the answer is 'not many'. More precisely, the continuous evolution of the ODE ensures that any topological property of its input must be preserved.

Let $a = -1, b = 0, c = \frac{1}{2} \in \mathbb{R}$, and suppose we wish to classify $\{a, c\}$ versus $\{b\}$ by constructing a neural ODE followed by an affine layer. That is, the flow of the ODE should linearly separate $\{a, c\}$ from $\{b\}$.

This is impossible: we are asking that either $a < b$ and $c < b$, or that $a > b$ and $c > b$. Correspondingly, either the trajectories for a and b must cross, or the trajectories for b and c must cross. See Figure 2.5.

This is a contradiction, as ODE flows never cross.

Remark 2.11. *This is not isolated to $d = 1$. Higher dimensional counterexamples may be considered by considering analogous 'nested shells', in which $\left\{y \in \mathbb{R}^d \,\middle|\, \|y\| < r_1\right\}$ and $\left\{y \in \mathbb{R}^d \,\middle|\, r_2 < \|y\| < r_3\right\}$ are classified from each other, with $0 < r_1 < r_2 < r_3$ [DDT19].*

2.4.2 'Augmented' Neural ODEs are universal approximators, even if their vector fields are not universal approximators

Fortunately, this is an issue easily remedied, through augmentation as introduced in Section 2.3.3.

2.4.2.1 When the vector field is a universal approximator

We first consider the case that the vector fields are universal approximators.

Theorem 2.12. *Fix $d, d_l, d_o \in \mathbb{N}$ with $d_l \geq d + d_o$. For $f \in \mathrm{Lip}(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l})$,*

$\ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l})$, $\ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o})$, let $\phi_{f,\ell_1,\ell_2} \colon \mathbb{R}^d \to \mathbb{R}^{d_o}$ denote the map $x \mapsto z$ with

$$y(0) = \ell_1(x), \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)) \quad \text{for } t \in [0, T], \qquad z = \ell_2(y(T)).$$

Then

$$\left\{\phi_{f,\ell_1,\ell_2} \;\middle|\; f \in \mathrm{Lip}(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l}), \ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l}), \ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o})\right\}$$

is a universal approximator for $C(\mathbb{R}^d; \mathbb{R}^{d_o})$.

(For simplicity this theorem has assumed that the vector field may be drawn from $\mathrm{Lip}(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, not just some dense subset of it.)

See [Zha+20, Theorem 7] for a short-and-sweet proof of Theorem 2.12.

2.4.2.2 When the vector field is not a universal approximator

Perhaps surprisingly, the condition that the vector field must be a universal approximator is not a necessary condition.

Theorem 2.13. *Fix $d, d_o \in \mathbb{N}$. For $d_l \in \mathbb{N}$, $f \in C(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, $\ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l})$, $\ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o})$, let $\phi_{p,f,\ell_1,\ell_2} \colon \mathbb{R}^d \to \mathbb{R}^{d_o}$ be the map $x \mapsto z$ with*

$$y(0) = \ell_1(x), \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)) \quad \text{for } t \in [0, T], \qquad z = \ell_2(y(T))$$

for those f for which the solution is unique.[7]

For each $d_l \in \mathbb{N}$ there exists an $f_{d_l} \in C(\mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, for which the above equation has a unique solution, such that

$$\left\{\phi_{d_l,f_{d_l},\ell^1,\ell^2} \;\middle|\; d_l \in \mathbb{N}, \ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l}), \ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o})\right\}$$

is a universal approximator for $C(\mathbb{R}^d; \mathbb{R}^{d_o})$.

See Appendix C.1 for the proof.

2.4.2.3 Comparison

If the vector field is a universal approximator, then the width of the latent space d_l is fixed, and complexity is obtained through the vector field. In contrast, if the vector field is not a universal approximator, then the latent dimensionality d_l is allowed to become arbitrarily large, and complexity is instead obtained through the affine maps.

[7]The Peano existence theorem implies existence as f is continuous; but as f is not necessarily Lipschitz then the stronger Picard existence theorem, which gives uniqueness, does not apply.

Remark 2.14. *These have direct analogues in the theory of universal approximation for neural networks.*

The case for which the vector field is not a universal approximator is directly analogous to the classical universal approximation theorem, which states that sufficiently wide feedforward neural networks may be used to approximate arbitrary continuous functions [Pin99].

The case for which the vector field is a universal approximator is directly analogous to the 'deep and narrow' universal approximation theorem, which states that sufficiently deep feedforward networks, of bounded width, may be used to approximate arbitrary continuous functions [Lu+17b; HS17; KL20b; Par+21].

2.5 Comments

Neural ODEs were originally considered (to the best of this author's knowledge) in early works from the 1990s, such as [CS91; Ric+92; RK93; RAK94]. A recent revival of the neural-network-as-dynamical-system was started with works such as [E17; HR17], and popularised (in particular in continuous time) by [Che+18b].

Indeed [Che+18b] introduced continuous-time neural ODEs for image classification (Section 2.2.1), continuous normalising flows (Section 2.2.3), and latent ODEs (Section 2.2.4). The latter two were expanded on in [Gra+19; RCD19].

Applications of neural ODEs to physical problems span multiple literatures; we can give at most a small selection of examples. Examples from machine learning include [GDY19; Rac+20b; BBS21], whilst examples from engineering include [LKT16; LKB18; Por+19; Ji+21]. Other examples include physics [XZW21], climate science [Ram+20a; MN20; Hwa+21], epidemiology [Wan+21], neuroscience [Kim+21b], pharmacodynamics [Kim+21a] and so on.

Connections between neural ODEs and their discrete-time counterparts include [HR17; DDT19; San+21; Sch+21] amongst others.

Extensions of neural ODEs to handle discontinuities, such as the velocity of a bouncing ball, include [ZDC21; CAN21a; Pol+21].

[MN20; Lou+20; FF20] generalise CNFs to manifolds, and for example then use CNFs to perform density estimation over distributions on a sphere. [Roz+21] offer a variation suitable for low-dimensional manifolds, that elides the ODE solve.

[Fin+20a; Fin+20b; Onk+21; Roz+21] amongst others discuss connections between CNFs and optimal transport, to select good parameterisations and regularisations for the vector field.

Good parameterisations for the vector field are often to be found by examining the code attached to any given neural ODE paper. Works discussing this topic explicitly include [HR17; DDT19; Zha+19; Cho+20; Mas+20; Nor+20].

[DDT19] note the lack of universal approximation for 'unaugmented' neural ODEs. [Zha+20] demonstrate universal approximation with 'augmented' neural ODEs provided the vector field is a universal approximator. More subtle universal approximation results may also be found in the literature [LLS19; Tes+20]. The material on universal approximation when the vector is *not* a universal approximator (Theorem 2.13) is new here.

A few other review articles combining ordinary dynamical systems and deep learning have recently been published, which the reader may find complements this chapter. For example [Li20a] focus on interpreting deep learning via control theory, [BNK20] focus on applications to fluid mechanics, and [Thu+21] place great emphasis on performing experiments. Most such works place a strong focus on specifically hybrid neural/mechanistic modelling with neural ODEs, which is our Section 2.2.2.

Chapter 3

Neural Controlled Differential Equations

3.1 Introduction

Neural ODEs were the continuous-time limit of residual networks. We will now introduce *neural controlled differential equations* as the continuous-time limit of *recurrent* neural networks. The following chapter will be of particular interest for those studying RNNs or time series; also to those studying rough path theory, control theory, or reinforcement learning.

Controlled differential equations have until recently been relatively esoteric, so we do not assume familiarity with them on the part of the reader. The forthcoming section will form a 'mini-chapter' offering a self-contained summary of the key ideas, applications, and *raison d'être* for CDEs and neural CDEs.

Recall the equations for a neural ODE:

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f_\theta(s, y(s))\,\mathrm{d}s. \tag{3.1}$$

An extra time-like dimension is introduced and then integrated over. The presence of this extra (artificial) dimension motivates us to consider whether this model can be extended to data already exhibiting sequential structure, such as time series.

Given some ordered data $(y_0, \ldots, y_n)$, the goal is to extend the $y(0) = y_0$ condition to one resembling '$y(0) = y_0, \ldots, y(n) = y_n$', to align the introduced time-like dimension with the natural ordering of the data. The key difficulty is that the solution of an ODE is determined by the initial condition at $y(0)$, so there is no direct mechanism for incorporating data that arrives later.

Fortunately, it turns out that the resolution of this issue – how to incorporate incoming information into a differential equation – is already a well-studied problem in mathematics, via *controlled differential equations*.

Much of this chapter is due to [Kid+20a].

3.1.1 Controlled differential equations

Let $T > 0$ and let $d_x, d_y \in \mathbb{N}$. Let $x\colon [0,T] \to \mathbb{R}^{d_x}$ be a continuous function of bounded variation. Let $f\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be Lipschitz continuous. Let $y_0 \in \mathbb{R}^{d_y}$.

A continuous path $y\colon [0,T] \to \mathbb{R}^{d_y}$ is said to solve a *controlled differential equation*, controlled or driven by x, if

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s) \quad \text{for } t \in (0,T]. \tag{3.2}$$

Here '$\mathrm{d}x(s)$' denotes a Riemann–Stieltjes integral, and '$f(y(s))\,\mathrm{d}x(s)$' refers to a matrix-vector multiplication.

Bounded variation and Riemann–Stieltjes integration Beyond the ODE case of the last chapter, then CDEs depend on two new concepts: bounded variation paths, and Riemann–Stieltjes integration.

Suppose x is differentiable and has bounded derivative – a relatively weak assumption. Then x will be of bounded variation, and the Riemann–Stieltjes integral may be reduced to an ordinary integral

$$\int_0^t f(y(s))\,\mathrm{d}x(s) = \int_0^t f(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)\,\mathrm{d}s. \tag{3.3}$$

As such whilst we will continue to treat the general case, the reader unfamiliar with these concepts should feel free to mentally substitute the above treatment throughout.

Remark 3.1. *Equation* (3.3) *is essentially about reducing a CDE to an ODE. Correspondingly, the term 'vector field' may be used to refer to either $f(y(s))$ or $f(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)$.*

CDEs are operators A controlled differential equation should be interpreted as a function from path-space to path-space. The input is a path x. The output is a path y. By choosing f carefully, we may use a CDE to compute specific functions of its control.

Example 3.2 (Value and integral of control)**.** *Let $f\colon \mathbb{R}^2 \to \mathbb{R}^{2\times 2}$ be defined by*

$$f(y) = \begin{bmatrix} 0 & 1 \\ y_1 & 0 \end{bmatrix}$$

(where $y \in \mathbb{R}^2$ is decomposed into $y = [y_1, y_2]$).

Given any control $x\colon [0,T] \to \mathbb{R}$, let $y\colon [0,T] \to \mathbb{R}^2$ be the solution of the CDE driven by $t \mapsto (t, x(t))$, with vector field f, with initial condition $y(0) = [x(0), 0] \in \mathbb{R}^2$. Then $y(t)$ will compute the value, and the first integral, of x.

For example, consider the input signal $x(t) = \sin(t) \in \mathbb{R}$.

Then

$$\begin{aligned} y(t) &= y(0) + \int_0^t f(y(s)) \,\mathrm{d} \begin{bmatrix} s \\ x(s) \end{bmatrix} \\ &= \int_0^t \begin{bmatrix} 0 & 1 \\ y_1(s) & 0 \end{bmatrix} \begin{bmatrix} 1 \\ \cos(s) \end{bmatrix} \mathrm{d}s \\ &= \int_0^t \begin{bmatrix} \cos(s) \\ y_1(s) \end{bmatrix} \mathrm{d}s. \end{aligned}$$

Solving the first component, we see that

$$y_1(t) = \int_0^t \cos(s) \,\mathrm{d}s = \sin(t)$$

and so

$$y_2(t) = \int_0^t y_1(s) \,\mathrm{d}s = \int_0^t \sin(s) \,\mathrm{d}s.$$

As advertised: $y(t)$ *computes both the value* $\sin(t)$ *of the input signal, and its first integral* $\int_0^t \sin(s) \,\mathrm{d}s$.

Moreover there was nothing special about the choice of $\sin(t)$*, and this CDE will compute the value and first integral of any input signal.*

We will make the equivalence more precise later on, but the connection to RNNs should be intuitive: much like CDEs, they compute some function of their time-varying input.

Existence and uniqueness The Picard existence theorem (Theorem 2.1) may be adapted to this setting.

Theorem 3.3 (Picard existence theorem, [LCL04, Theorem 1.3] or [FV10, Theorem 3.8])**.** *Let* $f \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ *be Lipschitz. Let* $x \colon [0, T] \to \mathbb{R}^{d_x}$ *be of bounded variation. Let* $y_0 \in \mathbb{R}^{d_y}$*. Then there exists a unique continuous* $y \colon [0, T] \to \mathbb{R}^{d_y}$ *satisfying*

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f(y(s)) \,\mathrm{d}x(s) \quad \text{for } t \in (0, T].$$

Remark 3.4. *The differential equation for a CDE is (by convention) autonomous, in the sense that* f *is independent of the time* t*. If really desired then* t *may be included by adding it to the state: replace* x *with* $[t, x]$ *and* f *with* $\begin{bmatrix} 1 & 0 \\ 0 & f \end{bmatrix}$*. This implies we have replaced* y *with* $[t, y]$*.*

Remark 3.5. *We might wonder about also using right hand sides of the form '*$f_\theta(y(s), x(s))$*'. Whilst there is nothing fundamentally wrong with this alternate approach, it is less*

theoretically neat. When using '$f_\theta(y(s), x(s))$' it is not possible to have $x \mapsto y$ be the identity function (see Section 3.3.2 later), whilst the '$f_\theta(y(s))\,\mathrm{d}x(s)$' form has connections to integration against Brownian motion, as with stochastic differential equations.

3.1.2 Neural vector fields

Suppose we observe some data in the form of a (continuous and bounded variation) path $x\colon [0, T] \to \mathbb{R}^{d_x}$. This is often a little unrealistic as usually we observe discrete samples, for example in a time series. We shall fix this in a moment, when we consider applications.

Let $f_\theta\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be any (Lipschitz) neural network depending on parameters θ. The value $d_y \in \mathbb{N}$ is a hyperparameter describing the size of the hidden state. Let $\zeta_\theta\colon \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$ be any neural network depending on the parameters θ. Both f_θ and ζ_θ will often just be parameterised as MLPs.

We define a *neural controlled differential equation* [Kid+20a] as the solution of the CDE

$$y(0) = \zeta_\theta(x(0)), \qquad y(t) = y(0) + \int_0^t f_\theta(y(s))\,\mathrm{d}x(s) \quad \text{for } t \in (0, T]. \tag{3.4}$$

The quantity y is hidden state, modified in response to observations x. This is directly analogous to an RNN. This hidden state reflects an evolving belief about the system, updated continuously as observations x are made.

Let $d_o \in \mathbb{N}$ be the desired output dimensionality of the model, and let $\ell_\theta\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_o}$ be a learnt affine map. Then the output of the model can be $\ell_\theta(y(t))$ if a time-evolving output is desired, or $\ell_\theta(y(T))$ if it is not, for example when performing whole-time-series classification. Once again, this parallels the construction of an RNN, for which a learnt affine readout is typically used to map from hidden state to output.

The resemblance between equations (3.1) and (3.4) is clear. The essential difference is that equation (3.4) is driven by the data process x, whilst equation (3.1) is driven only by the identity function $\mathbb{R} \to \mathbb{R}$. In this way, the neural CDE is naturally adapting to incoming data, as changes in x change the local dynamics of the system.

3.1.3 Solving CDEs

As with neural ODEs, we expect to numerically discretise the CDE so as to obtain an approximate solution.

A CDE may be discretised in two different ways. One option is to treat the '$\mathrm{d}x(s)$' analogous to time inside a numerical differential equation solver, so that for example the explicit Euler method becomes

$$y_{j+1} = y_j + f_\theta(y_j)(x(t_{j+1}) - x(t_j)).$$

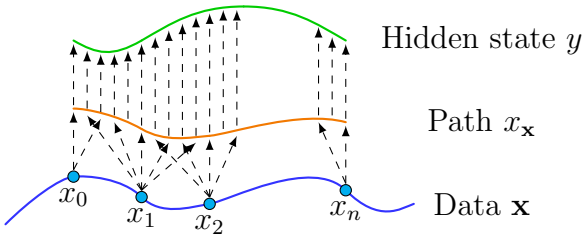


Figure 3.1: The hidden state of the neural CDE evolves continuously, driven by observational data.

In practice however most software libraries do not support this (with the notable exception of Diffrax [Kid21a]).

Provided x is differentiable – in practice it often will be – then the CDE may also be reduced to an ODE. Let

$$g_{\theta,x}(y, s) = f_\theta(y)\frac{\mathrm{d}x}{\mathrm{d}s}(s), \tag{3.5}$$

so that for $t \in (0, T]$,

$$\begin{aligned} y(t) &= y(0) + \int_0^t f_\theta(y(s))\,\mathrm{d}x(s) \\ &= y(0) + \int_0^t f_\theta(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)\,\mathrm{d}s \\ &= y(0) + \int_0^t g_{\theta,x}(y(s), s)\,\mathrm{d}s. \end{aligned} \tag{3.6}$$

It is now possible to solve and train the neural CDE using the same techniques as for neural ODEs, and in particular using the same software. See Section 5.6 for more discussion on software for neural differential equations.

3.1.4 Application to regular time series

Let us now consider a concrete application to 'regular' time series. That is to say, the observations are at regularly-spaced points, these points are the same for each batch element, and there is no missing data. (Extension to irregular time series will be considered in Section 3.2.1.)

Let each time series be some sequence $\mathbf{x} = (x_0, \ldots, x_n)$ with each $x_j \in \mathbb{R}^{d_x - 1}$. Let $x_{\mathbf{x}} \colon [0, n] \to \mathbb{R}^{d_x}$ be some interpolation such that $x_{\mathbf{x}}(j) = (j, x_j)$. For example, $x_{\mathbf{x}}$ could be a cubic spline. Then $x_{\mathbf{x}}$ may be used to drive a neural CDE. See Figure 3.1.

Remark 3.6. *$x_{\mathbf{x}}$ is sometimes interpreted as an approximation to some underlying process that $\mathbf{x}$ has been sampled from. This is true, but not really relevant. Rather, $x_{\mathbf{x}}$ is just a continuous-time representation of the input data. If we had used $-x_{\mathbf{x}}$ instead then this would have represented the information contained in $\mathbf{x}$ just as well, despite being neither an interpolation nor an approximation.*

We discuss choices of interpolation scheme in more detail in Section 3.5.

3.1.4.1 Spiral classification

As a toy example, we construct a two-dimensional dataset consisting of time series of the (x, y)-position of spirals, and train a neural CDE to perform binary classification of clockwise against anticlockwise. We consider data both with and without corruption by additive Gaussian noise.

The hidden state y of the CDE evolves in $\mathbb{R}^{d_l}$ with $d_l = 8$, and the prediction of the model at time t is given by $\sigma(\ell_\theta(y(t))) \in (0, 1)$, where $\ell_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}$ is a learnt affine readout and σ is the sigmoid function. The model is trained with binary cross entropy on $\sigma(\ell_\theta(y(T)))$.

The final output of the model is given by $\sigma(\ell_\theta(y(T)))$, but we may examine the evolving $t \mapsto \sigma(\ell_\theta(y(t)))$ for interest. See Figure 3.2. The prediction updates as the input sequence is fed into the model, converging towards a steady state of the correct classification. (On this simple problem the model achieves perfect accuracy.)

Precise experimental details may be found in Appendix D.3. The code is available as an example in Diffrax [Kid21a].

Remark 3.7. *The presence of noise does not necessitate any changes to this approach. If really desired the data could for example be smoothed with a filter, but in principle this is not necessary. The interpolation is just a continuous-time representation of the noisy data, which the model consumes as input.*

3.1.4.2 The inclusion of time

There is only one foible with this construction, which is that $x_{\mathbf{x}}(j) = (j, x_j)$ and not simply $x_{\mathbf{x}}(j) = x_j$. This one detail is important for expressivity of the model.

Example 3.8. *Suppose the function computed by the neural CDE should be the length of the input time series. If $x_{\mathbf{x}}(j) = (j, x_j)$ then this is straightforward. Take $d_y = 1$ so that $y(t) \in \mathbb{R}$, and let $f_\theta(y) = [1, 0, \ldots, 0] \in \mathbb{R}^{d_x}$ be constant. Let the initial value network $\zeta_\theta(x) = 0$ for all inputs. Then*

$$\begin{aligned} y(n) &= y(0) + \int_0^n f_\theta(y(s)) \, \mathrm{d}x(s) \\ &= \int_0^n \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 1 \\ * \\ \vdots \\ * \end{bmatrix} \mathrm{d}s \\ &= \int_0^n \mathrm{d}s \\ &= n, \end{aligned}$$

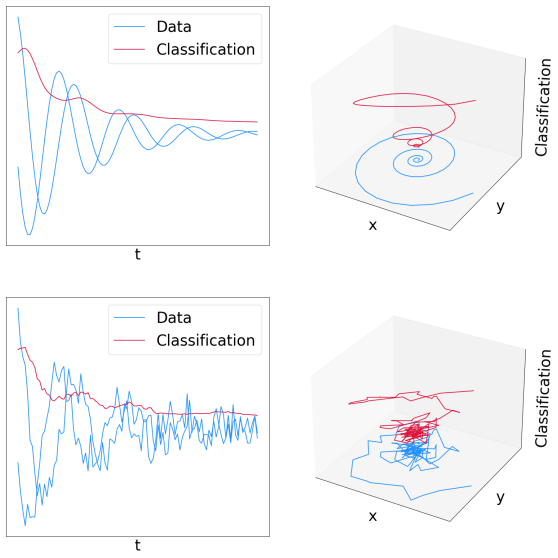


Figure 3.2: Data, and evolving prediction, of the neural CDE. **Top:** Data without noise. **Bottom:** Data is corrupted with Gaussian noise prior to training. **Left:** The (x, y) data, and the prediction of the neural CDE, are shown evolving over time. The prediction converges towards a steady state (of zero; a classification of a clockwise spiral) as sufficient input data becomes available. **Right:** The (x, y) position is shown at the bottom of the figure. Above it is shown the current prediction of the neural CDE.

where the '$*\cdots*$' refers to whatever the derivative of the interpolation of $\mathbf{x}$ *is.*

If this extra 'time' variable is missed out, and simply $x_{\mathbf{x}}(j) = x_j$*, then computing the length is impossible. For example suppose* $x_j = 0$ *for all* j*, and correspondingly any reasonable interpolation scheme will have* $x_{\mathbf{x}}(t) = 0$ *for all* t*. Then* $\frac{\mathrm{d}x}{\mathrm{d}t}(t) = 0$ *as well, and* $y(t) = y(0)$ *regardless of the choice of* f_θ*. And so* $y(t)$ *cannot calculate the length of* $\mathbf{x}$ *for this particular choice of* $\mathbf{x}$*.*

(Note how the Example 3.2, earlier, also included time as an additional channel.)

3.1.5 Discussion

Neural CDEs offer several advantages, both conceptually and practically.

3.1.5.1 Universal approximation

Provided this formulation is followed carefully – and this extra time-like variable is included – then the neural CDE will be a universal approximator.

(Informal) Theorem 3.9. *An affine map on the terminal value of a neural CDE is a universal approximator from* {sequences in $\mathbb{R}^{d_x}$} *to* $\mathbb{R}$.

We will discuss this further in Section 3.3.1, and provide a formal statement and proof in Appendix C.2.1.

3.1.5.2 Continuous-time updates

Neural CDEs update their hidden state in continuous time. In many contexts this is far more natural than the discrete-time updates typical of an RNN.

Irregular data Suppose the data arrives at irregular times. (We'll discuss this use case in much more detail in the next section.) If the times are close together then the hidden state of an RNN or neural CDE will often need only a small update. If the times are far apart then the belief about the system may need a very large update.

An RNN, however, devotes equal processing power to both of these use cases. (A single update step.) This may be inefficient if the times were close together, and insufficient if they were far apart.

In contrast a neural CDE updates continuously. The amount of computational work scales with the gap between observations, and this is likely close to the 'natural timescale' at which we should update our belief about the system.[1]

Decoupled data and computation Put precisely, continuous-time updates decouple data and computation; the latter is no longer tied to the former. This is particularly true if solving a neural CDE with an adaptive step size numerical ODE/CDE solver. Such a solver automatically detects the complexity of the dynamics and takes appropriately-sized numerical steps.

Special cases of neural CDEs In light of this, there have now been several proposals in which the hidden state of an RNN is updated in continuous time between observations; popular examples are GRU-D or ODE-RNNs [Che+18a; RCD19; De +19]. These are special cases or discretisations of neural CDEs. (Exercise for the reader!)

3.1.5.3 Memory efficient backpropagation

Given an RNN, for which evaluating and backpropagating a single step consumes H memory, then backpropagating an RNN evaluated on a time series of length T will consume $\mathcal{O}(HT)$ memory.

[1]Could we instead repeatedly give the last piece of input data to an RNN, whilst waiting a long time for another observation? Yes, and in doing so have just reinvented a particular discretisation of a neural CDE.

In contrast, neural CDEs can reduce this to only $\mathcal{O}(H + T)$ memory. This consists of $\mathcal{O}(H)$ to backpropagate through each step individually, and $\mathcal{O}(T)$ to hold the underlying data x in memory. That each step can be backpropagated through individually is due to the use of 'optimise-then-discretise' backpropagation. We will discuss this style of backpropagation alongside our other numerical discussions, in Section 5.2.

3.1.6 Summary

The goal of this section, Section 3.1, has been to summarise the main ideas behind CDEs and neural CDEs. Now that these are in place, we will be ready to move on to some more serious applications of neural CDEs.

We conclude this section with some thoughts on the connections of CDEs to other fields of study.

3.1.6.1 Rough path theory

The theory of CDEs may be extended to highly irregular driving paths x, which are not even of bounded variation. This is known as *rough path theory*, and correspondingly such CDEs become rebranded as *rough differential equations* (RDEs).

There is broadly speaking a hierarchy from ODEs to CDEs to RDEs to SDEs: CDEs introduce the notion of control; RDEs additionally consider when the control is rough; SDEs additionally consider when the control is stochastic (usually Brownian motion).[2]

We will use rough path theory in a few contexts: when applying neural CDEs to long time series (Section 3.2.3), in the proof of universal approximation for neural CDEs (Section 3.3.1), and in a later chapter to construct the 'optimise-then-discretise' equation for neural SDEs (Section 5.2.3).

These all tend towards the theoretical end of things, and we emphasise that a familiarity is neither expected nor required to read this thesis, or to work with the techniques discussed.

Remark 3.10. *For those with the right background (graduate-level analysis), then rough path theory gives an excellent framework for understanding neural differential equations. It offers a pathwise theory, and a general framework through which ODEs/CDEs/SDEs may all be unified; for example Diffrax [Kid21a] uses its principles to construct a unified system of numerical differential equation solvers. The first few pages of [Hod+20] give a brief introduction to the essential ideas of rough path theory, [LCL04] is a typical introductory text, and [FV10] is the canonical textbook.*

[2]For example this hierarchy is demonstrated by numerical SDE solvers, which typically operate by drawing a sample of the Brownian motion and then solving the SDE pathwise.

3.1.6.2 Control theory

Despite their similar names, and treatment of similar problems, controlled differential equations and control theory are typically treated as separate fields.

The difference is to some extent philosophical. In control theory, the system f is typically specified[3], and the task is to find a control x producing the desired response y. Meanwhile with (neural) CDEs, this is flipped around: the control x is typically specified, and we shall attempt to find a system f that produces a desired response y.

This is not a distinction we particularly wish to enforce, though – there is still substantial overlap.

3.2 Applications

Neural CDEs have a number of applications, usually to time series. We will see applications to difficult time series (such as irregular or long time series), and will later briefly touch on connections to reinforcement learning. In addition we have previously remarked that RNNs and neural CDEs are linked, and we will also make this connection explicit.

3.2.1 Irregular time series

Suppose we observe some irregular time series of the form $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n))$, with each $t_j \in \mathbb{R}$ the timestamp of the observation

$$x_j = (x_{j,1}, \ldots, x_{j,d_x - 1}) \in (\mathbb{R} \cup \{*\})^{d_x - 1}.$$

Here $*$ denotes the possibility of missing data, and $t_0 < \cdots < t_n$. The length n is not assumed to be consistent between different time series.

Let $T > 0$ and $0 = s_0 < s_1 < \cdots < s_n = T$. Let $x_{\mathbf{x}} \colon [0, T] \to \mathbb{R}^{d_x}$ be some interpolation such that $x_{\mathbf{x}}(s_j) = (t_j, x_j)$ (with the equality being defined up to those elements of x_j which are not missing). For example, we could take $s_j = t_j$ (or $s_j = j$ as in the previous section) and $x_{\mathbf{x}}$ to be a cubic spline with knots at $s_0, \ldots, s_n$.

Then $x_{\mathbf{x}}$ may be used to drive a neural CDE [Kid+20a]; see Figure 3.3.

Remark 3.11. *We stress that this interpolation is not imputing missing data. It is simply constructing a continuous-time representation of the input data. We will discuss how to appropriately handle missing data in a moment.*

[3]Perhaps incompletely via observations, necessitating the additional step of performing system identification.

The choice of interpolation scheme, including the choice of s_j, is one we will defer until Section 3.5. A few different choices may be made, depending on the type of problem.

3.2.1.1 Missingness as a channel

It has been observed that the frequency of observations may carry information [Che+18a]. For example, doctors may take more frequent measurements of patients they believe to be at greater risk. Some previous work has for example sought to incorporate this information by learning an intensity function [SM19; RCD19; Che+18b].

A simple (non-learnt) procedure is just to concatenate the index j as an additional channel. That is, construct the path $x_{\mathbf{x}}\colon [0, T] \to \mathbb{R}^{d_x+1}$ such that $x_{\mathbf{x}}(s_j) = (t_j, x_j, j)$ instead of just (t_j, x_j). The extra channel of $x_{\mathbf{x}}$ then gives the cumulative number of observations over time.

As the derivative of $x_{\mathbf{x}}$ is what is then used when evaluating the neural CDE model, as in equation (3.6), then it is the current observational rate that then determines the vector field.

3.2.1.2 Partially observed data

When some data is missing, then the frequency of observations in each individual channel may carry information. The previous procedure may now be straightforwardly extended, by having a separate observational channel for each original channel.

Explicitly, take $x_{\mathbf{x}}(s_j) = (t_j, x_j, c_j(\mathbf{x}))$, where $c_j(\mathbf{x}) = (c_{j,1}, \ldots, c_{j,d_x}) \in \mathbb{R}^{d_x}$, where $c_{j,k} = \sum_{m=0}^{j} \mathbb{1}_{x_{m,k} \neq *}$ counts the number of observations in the kth channel by time t_j. This means that $x_{\mathbf{x}}\colon [0, T] \to \mathbb{R}^{2d_x-1}$.

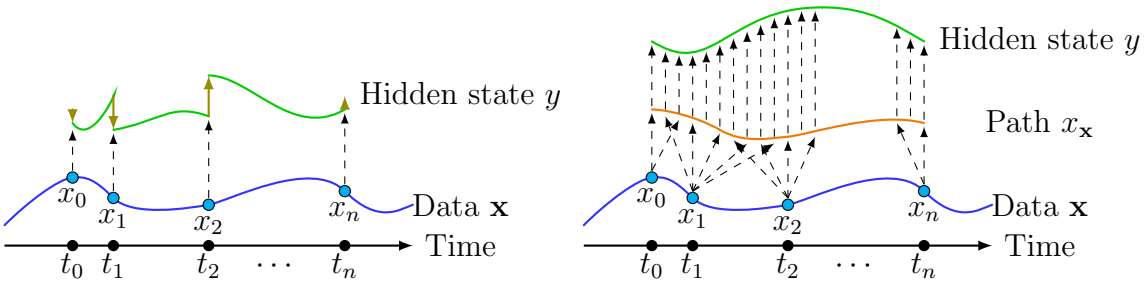


Figure 3.3: Some data process is observed at times $t_0, \ldots, t_n$ to give observations $x_0, \ldots, x_n$. It is otherwise unobserved. **Left:** RNNs modify the hidden state at each observation; common variants [Che+18a; RCD19; De +19] continuously evolve the hidden state between observations. **Right:** In contrast, the hidden state of the neural CDE model has continuous dependence on the observed data.

Adding observational masks is standard practice when working with informatively missing data [Che+18a]; this is the appropriate continuous-time analogy.

3.2.1.3 Batching irregular data and choice of s_j

In the context of CDEs, the data consists of multiple $x_{\mathbf{x}}\colon [s_0, s_n] \to \mathbb{R}^{d_x}$ that need to be batched together. In principle each interval $[s_0, s_n]$ may be different for each batch element, for example if we chose $s_j = t_j$ and the data is irregularly sampled.

Batchable differential equation solvers Some (very few) differential equation software libraries allow batching over different regions of integration. In this case the problem is straightforward: simply use the capabilities of the library. For example this is the case with Diffrax [Kid21a].

Other differential equation solvers Most differential equation software libraries do not intrinsically support batching over different regions of integration. For example this is the case with `torchdiffeq` and `torchcde` [Che18; Kid20].

Fortunately, the structure of neural CDEs mean this is not a serious hurdle, as we may choose specifically $s_j = j$. This ensures that each region of integration begins at the same value, namely zero, and we need merely integrate forwards in time for sufficiently long that every batch element has been integrated. (See also Section 3.5.2 for more discussion on the choice of s_j.)

As an additional benefit, the fact that all s_j are the same for each batch element can be used to simplify the storage of batches of multiple control paths $x_{\mathbf{x}}\colon [0, T] \to \mathbb{R}^{d_x}$. (Instead of juggling different collections of intervals $[s_j, sj + 1]$ for each batch element.)

Remark 3.12. *This was actually a mistake we made in [Kid+20a] – the above procedure was not done, in favour of an alternate (storage-inefficient) scheme that involved taking the union over the times t_j needed to handle each batch of data.*

Remark 3.13. *One minor quirk in this case arises when using an adaptive step size solver. A subtle dependency between batch elements is introduced, as the step size will be determined by the behaviour across the whole (batched) system. This is usually not a major issue, and this is simply tolerated. (This quirk is not unique to neural CDEs, and is true whenever a batch of neural differential equations are solved with an unbatched adaptive differential equation solver.)*

3.2.1.4 Example

We now refer back to the regularly-spaced example of Section 3.1.4. Had the observations been irregularly spaced, they could have been handled in the manner just described. Morally speaking neural CDEs make little difference between regular and

irregular time series; once a continuous path $x_{\mathbf{x}}$ is obtained then both are treated in exactly the same way.

3.2.2 RNNs are discretised neural CDEs

We will now make the connection between RNNs and CDEs explicit; see also [Kid+20a].

3.2.2.1 CDEs as RNNs

Consider the CDE

$$y(t) = y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s) \quad \text{for } t \in (0, T].$$

Discretising this with Euler's method produces either

$$y(t_{j+1}) = y(t_j) + f(y(t_j))(x(t_{j+1}) - x(t_j))$$

or

$$y(t_{j+1}) = y(t_j) + f(y(t_j))\frac{\mathrm{d}x}{\mathrm{d}t}(t_j)(t_{j+1} - t_j)$$

depending on whether the CDE is converted into an ODE first.

In either case, this is an RNN-like structure: suppose $f = f_\theta$ is some neural network and x is some input data.

3.2.2.2 RNNs as CDEs

Conversely consider an RNN of the form

$$y_{j+1} = h_\theta(y_j, x_j).$$

This is an explicit Euler discretisation with unit timestep of

$$y(t) = y(0) + \int_0^t h_\theta(y(s), x(s)) - y(s)\,\mathrm{d}s. \tag{3.7}$$

Equations of this form, in which the integrand is some function of $y(s)$ and $x(s)$, are special cases of neural CDEs. (We'll discuss this in Section 3.3.2.)

3.2.2.3 RNN variants

The '$-\,y(s)$' term in (3.7) feels a little out of place. It would not appear for RNNs of the form

$$y_{j+1} = y_j + h_\theta(y_j, x_j).$$

RNNs of this form resemble residual networks, and indeed this parameterisation clearly provides a better differential-equation-like structure.

Example 3.14. *A GRU is of the above form. Recall that a GRU is defined by*

$$\begin{aligned} i_j &= \text{sigmoid}(W_1 x_j + W_2 h_j + b_1), \\ r_j &= \text{sigmoid}(W_3 x_j + W_4 h_j + b_2), \\ n_j &= \tanh(W_5 x_j + b_3 + r_j * (W_6 h_j + b_4)), \\ h_{j+1} &= n_j + i_j * (h_j - n_j), \end{aligned}$$

for input time series x_j evolving hidden state h_j, and suitably shaped weight matrices $W_1, W_2, W_3, W_4, W_5, W_6$ and bias vectors b_1, b_2, b_3, b_4. Here $*$ denotes elementwise multiplication.

This is an explicit Euler discretisation of

$$\begin{aligned} i(t) &= \text{sigmoid}(W_1 x(t) + W_2 h(t) + b_1) \\ r(t) &= \text{sigmoid}(W_3 x(t) + W_4 h(t) + b_2), \\ n(t) &= \tanh(W_5 x(t) + b_3 + r(t) * (W_6 h(t) + b_4)), \\ \frac{\mathrm{d}h}{\mathrm{d}t}(t) &= (1 - i(t)) * (n(t) - h(t)). \end{aligned}$$

Remark 3.15. *Note the $-h(t)$ that appears on the right hand side of the continuous-time GRU. This corresponds to exponential decay of the hidden state of the GRU, just as with the differential equation $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = -y(t)$ for exponential decay [Pol21]. This explains the classic fact that GRUs/LSTMs struggle to learn long-term time dependencies.*

3.2.3 Long time series and rough differential equations

Neural CDEs, as with RNNs, begin to break down for very long time series. Loss/accuracy worsens, and training time becomes prohibitive due to the sheer number of operations required to evaluate a single pass of the model.

We will now see how this may be remedied. The key idea is to take very large integration steps – much larger than the sampling rate of the data – whilst incorporating sub-step information through additional terms in the numerical solver, through what is known as the *log-ODE method*.

A CDE treated in this way is termed a *rough differential equation*, in the sense of rough path theory. Correspondingly we refer to this approach as *neural rough differential equations*. (Or less snappily, 'the log-ODE method applied to neural CDEs'.) This was introduced in [Mor+21b].

Remark 3.16. *For the reader familiar with numerical SDEs, this inclusion of 'sub-step information' is directly analogous to the difference between the Euler–Maruyama method and Milstein's method [KP92]. For the reader familiar with the Magnus expansion [Bla+09], then the log-ODE method is a generalisation to nonlinear differential equations.*

More so than the rest of this chapter, this will rely on advanced theoretical tools from rough path theory. As such this material is deferred to Appendix B to avoid breaking the flow.

3.2.4 Training neural SDEs

We will see in Chapter 4 that SDEs, as generative models, may be trained as GANs. However samples from SDEs are continuous-time paths, which necessitate a discriminator that admits a continuous-time path as an input – such as a neural CDE.

The main ideas for this were introduced in [Kid+21b; Kid+21a], and this will be discussed alongside neural SDEs in Chapter 4.

3.3 Theoretical properties

3.3.1 Universal approximation

In the CDE literature, it is a well-known theorem that they represent general functions on streams. We think [Per18, Theorem 4.2], see also [Kid+19, Proposition A.6], give the clearest statement of this result. This may be applied to show that neural CDEs are universal approximators, which we summarise in the following informal statement.

(Informal) Theorem 3.9. *An affine map on the terminal value of a neural CDE is a universal approximator from* {sequences in $\mathbb{R}^{d_x}$} *to* $\mathbb{R}$.

The essential idea is that a suitably large CDE can compute a truncated basis for the space of continuous functions of its input. The final affine map may then take some affine combination of these, and in doing so approximate any continuous function.

Theorem C.25 in Appendix C.2.1 gives a formal statement and a proof, which generalises the original presentation in [Kid+20a, Appendix B].

This property is not necessary for good empirical performance (GRUs frequently achieve good performance without being universal approximators [WGY18]), but it is reassuring to know that it may be accomplished in principle.

3.3.2 Comparison to alternative ODE models

If unfamiliar with CDEs, then it may seem natural to replace $f_\theta(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)$ with some $h_\theta(y(s), x(s))$ that is directly applied to, and potentially nonlinear in, $x(s)$. Indeed, special cases of this have been suggested before, in particular to derive the 'GRU-ODE' analogous to a GRU [Cho+14; De +19; JSP19] (Example 3.14).

However, it turns out that something is lost by doing so, which we summarise in the following statement.

(Informal) Theorem 3.17. *Any equation of the form* $y(t) = y(0) + \int_0^t h_\theta(y(s), x(s))\,\mathrm{d}s$ *may be represented exactly by a neural CDE of the form* $y(t) = y(0) + \int_0^t f_\theta(y(s))\,\mathrm{d}x(s)$. *However the converse statement is not true.*

The essential idea is that a neural CDE can easily represent the identity function between paths, whilst the alternative is incapable of doing so. Theorem C.27 in Appendix C.2.2 provides the formal statement and proof, which originally appeared in [Kid+20a, Appendix C].

(This does not preclude using the $h_\theta(y(s), x(s))$ form if this happens to work on any given problem, of course.)

3.3.3 Invariances

CDEs exhibit two possible invariances. In general these invariances are often undesirable and are removed as follows.

3.3.3.1 Translation invariance and initial value networks

The integral defining the evolution of a neural CDE depends upon the control x only through its derivative $\frac{\mathrm{d}x}{\mathrm{d}t}$. If this were the only way that x was input to the model, then y would be invariant to translations of x.

It is for this reason that the initial hidden state $y(0)$ depends on $x(0)$ through the initial value network ζ_θ, so as to ensure sensitivity to translations. Other alternatives may also be admitted: a channel whose first derivative includes translation-sensitive information could be appended, for example by replacing x with $\widetilde{x}$ where $\widetilde{x}(t) = (x(t), tx(0))$.

3.3.3.2 Reparameterisation invariance

CDEs exhibit a *reparameterisation invariance* property.[4]

Proposition 3.18. *Let* $\psi\colon [0, S] \to [0, T]$ *be differentiable, increasing, and such that* $\psi(0) = 0$ *and* $\psi(S) = T$. *Let* y *solve a CDE driven by a path* x. *Then* $y \circ \psi$ *solves the same CDE driven by* $x \circ \psi$, *and in particular their terminal values are the same:* $(y \circ \psi)(S) = y(T)$.

This means that a CDE is blind to the speed at which x is traversed. Typically, the speed at which input data arrives is important (for example consider data indicating that a patient's health is declining over years – or over minutes), so this means that the speed at which events occur must be explicitly encoded as a channel in x. Indeed, this is precisely what is done in Section 3.1.4.2 by including time as a channel.

[4]In fact they also exhibit a *tree-like invariance* property [HL10], which is a slight generalisation.

See Appendix C.2.3 for a proof, which is straightforward change of variables.

3.4 Choice of parameterisation

So far we have discussed 'the mathematics'. Now we must discuss 'the engineering'. We must still choose optimisers, learning rates, model architectures and so on. As is often the case with deep learning, these choices can make or break the efficacy of the model.

3.4.1 Neural architectures and gating procedures

The initial value network ζ_θ is typically parameterised as an MLP.

The vector field f_θ is typically parameterised as $f_\theta = \tanh \circ \text{MLP}_\theta$, where tanh is applied elementwise, and MLP_θ denotes an MLP with weights and biases θ and as in Section 2.3 uses a continuously differentiable activation function.

If the tanh were removed then the MLP could produce arbitrarily large and unconstrained outputs. In contrast the inclusion of a squashing function such as tanh *constrains the rate of change of the hidden state.* As f_θ is iteratively evaluated multiple times over the differential equation, then large outputs from f_θ can easily result in the model exploding, with large and untrainable losses.

This is precisely analogous to RNNs, where one of the key features of GRUs and LSTMs are gating procedures which control the rate of change of the hidden state. (Their other key feature is a differential equation like structure.)

Slightly more complex variations on the same theme may be considered. For example let $\phi_\theta, \psi_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be neural networks; for efficiency often the same MLP just with different final affine layers. Then we may define $f_\theta(y) = \sigma(\phi_\theta(y)) * \tanh(\psi_\theta(y))$, where $*$ denotes an elementwise product and σ is the sigmoid function applied elementwise.

3.4.2 State-control-vector field interactions

If the vector field $f_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ is a feedforward neural network, with final hidden layer of size $d_h \in \mathbb{N}$, then the number of scalars for the final affine transformation is of size $\mathcal{O}(d_h d_y d_x)$, which can easily be very large.

As such this layer is often the greatest computational bottleneck of a neural CDE. But it is possible that not every three-way interaction between the hidden state y (of size d_y), control x (of size d_x), and penultimate layer in the vector field (the layer of size d_h) actually needs to be modelled.

Anecdotally, making the layer sparse (down to a density of about 1%) often still pro-

duces good results, whilst rank-one representations of the final matrix $f_\theta(y)$, as an outer product of transformations $\mathbb{R}^{d_y} \to \mathbb{R}^{d_y}$ and $\mathbb{R}^{d_y} \to \mathbb{R}^{d_x}$ seem to produce bad results. One can easily imagine many other kinds of reduced-parameter parameterisations, and this is a topic that merits further investigation.

3.4.3 Multi-layer neural CDEs

Let $y_1 \colon [0, T] \to \mathbb{R}^{d_{y_1}}$ solve a neural CDE driven by $x \colon [0, T] \to \mathbb{R}^{d_x}$, with system $f_{\theta,1} \colon \mathbb{R}^{d_{y_1}} \to \mathbb{R}^{d_{y_1} \times d_x}$:

$$y_1(t) = y_1(0) + \int_0^t f_{\theta,1}(y_1(s)) \, \mathrm{d}x(s) \quad \text{for } t \in (0, T].$$

We may now repeat this procedure: let $y_2 \colon [0, T] \to \mathbb{R}^{d_{y_2}}$ solve a neural CDE driven by y_1, with system $f_{\theta,2} \colon \mathbb{R}^{d_{y_2}} \to \mathbb{R}^{d_{y_2} \times d_{y_1}}$:

$$y_2(t) = y_2(0) + \int_0^t f_{\theta,2}(y_2(s)) \, \mathrm{d}y_1(s) \quad \text{for } t \in (0, T].$$

This idea of stacking may of course be repeated arbitrarily many times.

The joint system may be solved together as a single CDE driven by x. For example in the two-layer case, we obtain

$$\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} + \int_0^t \begin{bmatrix} f_{\theta,1}(y_1(s)) \\ f_{\theta,2}(y_2(s)) f_{\theta,1}(y_1(s)) \end{bmatrix} \mathrm{d}x(s) \quad \text{for } t \in (0, T].$$

The main disadvantage of this approach is that the output dimension of $f_{\theta,2}$ is large ($d_{y_1} \times d_{y_2}$), and therefore computationally expensive.

This offers a sensible way to increase the model capacity of a neural CDE; see for example [Jhi+21]. This is precisely analogous to multi-layer RNNs, in which the hidden state of one RNN is used as the input to another.

3.5 Interpolation schemes

Suppose we observe some (potentially irregular) time series, each of the form $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n))$, as in Section 3.1.4 or Section 3.2.1. Each $t_j \in \mathbb{R}$ is the timestamp of observation

$$x_j = (x_{j,1}, \ldots, x_{j,d_x - 1}) \in (\mathbb{R} \cup \{*\})^{d_x - 1},$$

where $*$ denotes the possibility of missing data, $t_0 < \cdots < t_n$. The length n is not assumed to be consistent between different time series.

We are interested in picking $T > 0$, $0 = s_0 < \cdots < s_n = T$, and constructing interpolations $x_{\mathbf{x}} \colon [0, T] \to \mathbb{R}^{d_x}$ such that $x_{\mathbf{x}}(s_j) = (t_j, x_j)$.

Note that we could also include $c_j(\mathbf{x})$ channels, which in general are needed when handling missing data; see Sections 3.2.1.1 and 3.2.1.2. These are handled in precisely the same way as the x_j channels (just construct interpolations $x_{\mathbf{x}} \colon [0, T] \to \mathbb{R}^{2d_x - 1}$ such that $x_{\mathbf{x}}(s_j) = (t_j, x_j, c_j(\mathbf{x}))$) so for simplicity of notation we leave them out here.

Remark 3.19. *That neural CDEs need interpolation schemes sometimes attracts skepticism. Are we imputing missing data? Are we constructing a continuous-time approximation to some underlying data process? Why, morally speaking, should we need to construct an interpolation scheme when the actual data we have observed is discrete?*

The answers are: 'no', 'yes (but it's not important)', and 'to process the data at its natural timescale', respectively. Each of these points has been discussed earlier in the text, in Remark 3.11, Remark 3.6, and Section 3.1.5.2 respectively.

We begin with some theoretical conditions that we would like ideal interpolation schemes to satisfy, and then present some sensible choices. The best choice of interpolation scheme will depend on the problem at hand.

Much of the following section is drawn from [Mor+21a].

3.5.1 Theoretical conditions

There are two main theoretical conditions, namely *measurability* and *smoothness*.

3.5.1.1 Measurability

Any given time series problem needs outputs at particular times. For example we may wish to only have an output after having observed an entire time series. Alternatively we may wish to produce a continuously-evolving output, updated as more data arrives over time.

We formalise this distinction in terms of *measurability*,[5] and will require an interpolation scheme that supports the desired behaviour.

We describe problems, and interpolation schemes, as exhibiting one of three different kinds of measurability. Figure 3.4 provides a visual summary.

Continuously measurable We say that an interpolation scheme is *continuously measurable* if $x_{\mathbf{x}}(s)$ depends only on those (t_j, x_j) with $s_j \leq s$. (Recalling that $x_{\mathbf{x}}(s_j) = (t_j, x_j)$.) That is to say, only observations in the past or present may be used to define the interpolation scheme.

[5] In [Mor+21a] the terminology of 'online' is used instead.

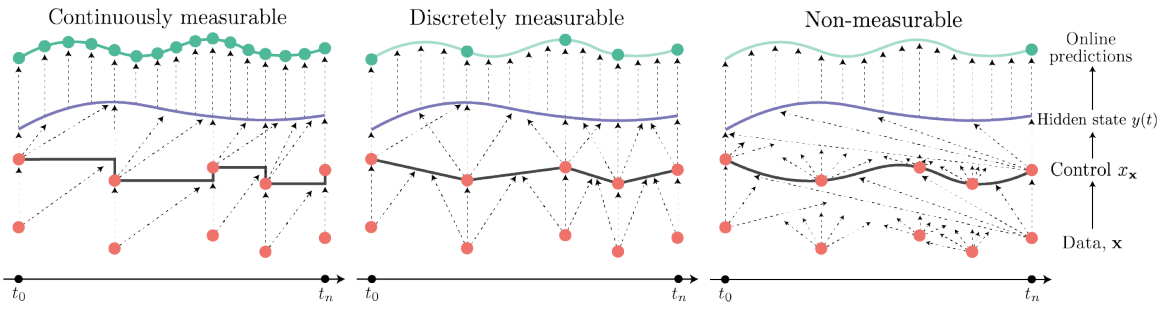


Figure 3.4: Summary of measurability definitions. Arrows indicate what data can influence. A green dot indicates that a measurable prediction can be made at that point. **Left:** a continuously measurable model in which no information is passed backward in time, resulting in a measurable solution at all points in time. **Middle:** a discretely measurable model in which information can be passed backwards-in-time, but no further than the preceding observation. **Right:** a non-measurable scheme in which information is passed backward in time further than the preceding observation.

This is probably the most intuitively natural definition of measurability, but it is relatively difficult to construct interpolation schemes satisfying it. (We will present only a single such scheme.) Practically speaking many use cases will find a weaker notion of measurability acceptable.

This kind of measurability is needed if data is arriving over time at inference time (sometimes described as the problem being *online*), and either:

- model predictions are needed between observations;
- model predictions are needed prior to the final observation in a time series, and there is missing data.

Discretely measurable We say that an interpolation scheme is *discretely measurable* if $x_{\mathbf{x}}(s)$ depends only on those (t_j, x_j) with $s_{j-1} \leq s$. That is to say, we may look up to one observation into the future when defining the interpolation scheme.

For example, this is the case with linear interpolation. $s \mapsto (1-s)a + sb$ with $s \in [0, 1]$ depends on b for all s, even though b will only be attained at $s = 1$.

This kind of measurability is needed if data is arriving over time at inference time (sometimes described as the problem being *online*), and model predictions are only needed at observations, and there is no missing data. Simply integrate up to some s_j, wait until (t_{j+1}, x_{j+1}) are observed, and then interpolate and integrate over $[s_j, s_{j+1}]$.

Non-measurable Finally we say that a scheme is not measurable if $x_{\mathbf{x}}(s)$ may depend only on any and all (t_j, x_j). For example, this is the case with natural cubic splines.

Such schemes are appropriate if, at inference time, the whole time series will be available prior to evaluating the model.

3.5.1.2 Smoothness

The second desirable theoretical property for an interpolation scheme is smoothness.

We will not define smoothness in a mathematically rigorous way. Rather, we point out that 'smoother' interpolation schemes will result in easier-to-integrate dynamics, which will improve the computational efficiency of the model.

The main dichotomy here is whether the dynamics are globally smooth (for example a cubic spline) or piecewise smooth (for example linear interpolation). The former is preferable but the latter can be tolerated.

Remark 3.20. *See Section 5.3.3 for how to correctly integrate piecewise smooth dynamics when using adaptive step size solvers.*

See also Section 5.4.1.4, which (mainly in the neural ODE setting) seeks to regularise higher-order derivatives in order to promote easy-to-integrate dynamics.

3.5.2 Choice of interpolation points

The choice of $T > 0$ and interpolation points s_j (such that $x_{\mathbf{x}}(s_j) = (t_j, x_j)$) is really about determining the desired behaviour of the numerical solver.

In continuous time, this choice is arbitrary, by the reparameterisation property of Section 3.3.3.2. The value of the integral is invariant to the choices of T and s_j.

Practically speaking, this does not completely carry through to the numerical discretisation. For example consider using $s_j = j$ and a fixed-step numerical solver with unit step size. Then a single numerical step is made between each observation, a fixed number of vector field evaluations would be made, and the neural CDE reduces to an RNN.

Overall, the spacing between s_j corresponds roughly to the amount of computational work that should be done between those observations. (Either precisely, when using a fixed solver, or approximately, when using an adaptive solver.) As per Section 3.1.5.2, a desirable choice is for the amount of computational work to scale with the 'natural timescale' at which the data varies. For many datasets this means a reasonable choice is $s_j = t_j$.

3.5.3 Particular interpolation schemes

There are a few main interpolation schemes of interest. In every case, each channel is interpolated separately. If there is missing data then it is interpolated over; for

example if $x_{j,1}$ and $x_{j+2,1}$ are observed but $x_{j+1,1}$ is missing then we apply the following procedures over $[s_j, s_{j+2}]$ rather than $[s_j, s_{j+1}]$.

3.5.3.1 Hermite cubic splines with backward differences

The first choice of interest are *Hermite cubic splines with backward differences*. Each interval $[s_j, s_{j+1})$ is treated independently, and the interpolation over this interval is chosen to satisfy

$$\begin{aligned} x_{\mathbf{x}}(s_j) &= (t_j, x_j), \\ x_{\mathbf{x}}(s_{j+1}) &= (t_{j+1}, x_{j+1}), \\ \frac{\mathrm{d}x_{\mathbf{x}}}{\mathrm{d}t}(s_j) &= \left(\frac{t_j - t_{j-1}}{s_j - s_{j-1}}, \frac{x_j - x_{j-1}}{s_j - s_{j-1}}\right), \\ \frac{\mathrm{d}x_{\mathbf{x}}}{\mathrm{d}t}(s_{j+1}) &= \left(\frac{t_{j+1} - t_j}{s_{j+1} - s_j}, \frac{x_{j+1} - x_j}{s_{j+1} - s_j}\right). \end{aligned}$$

Measurability This scheme is discretely measurable.

Smoothness Such splines are by construction continuously differentiable, so adaptive step size solvers will find the resulting dynamics easy to integrate.

Choice of s_j Typically $s_j = t_j$ is a reasonable choice.

3.5.3.2 Linear interpolation

One simple option is just linear interpolation:

$$x_{\mathbf{x}}(s) = (t_j, x_j) + (t_{j+1} - t_j, x_{j+1} - x_j)\frac{s - s_j}{s_{j+1} - s_j} \qquad \text{for } s \in [s_j, s_{j+1}).$$

(Mapping over each entry of the (t, x) tuple.)

Measurability This scheme is discretely measurable.

Smoothness This scheme is only piecewise continuously differentiable. If reducing the neural CDE to an ODE by taking $\int f_\theta(y(s))\,\mathrm{d}x(s) = \int f_\theta(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)\,\mathrm{d}s$ as in equation (3.6), then the vector field $f_\theta(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)$ will be piecewise constant.

This makes linear interpolation a poor choice if using an adaptive step size numerical solver, which will struggle with these jumps. However if using a fixed step size solver it can be just as good as Hermite cubic splines with backward differences, whilst being slightly cheaper to compute.

Choice of s_j Typically $s_j = t_j$ is a reasonable choice.

3.5.3.3 Rectilinear interpolation

Define $\bar{x}_j = (\bar{x}_{j,1}, \ldots, \bar{x}_{j,d_x-1}) \in \mathbb{R}^{d_x-1}$ as being the fill-forward[6] of x_j.

Now additionally select points $r_j \in [0, T]$, for $j \in \{1, \ldots, n\}$, such that

$$s_0 < r_1 < s_1 < r_2 < \cdots < s_{n-1} < r_n < s_n.$$

Rectilinear interpolation is defined by taking $x_{\mathbf{x}}(s_j) = (t_j, \bar{x}_j)$, $x_{\mathbf{x}}(r_j) = (t_{j+1}, \bar{x}_j)$, and linearly interpolating between $s_0, r_1, s_1, r_2, \ldots, r_n, s_n$.

Measurability This now satisfies the measurability condition we require. At inference time we can wait at each s_j for the next data point to arrive (regardless of whether it is only partially observed / has missing data), interpolate over $[s_j, s_{j+1}]$ in the manner above, and then solve the CDE over $[s_j, s_{j+1}]$.

Smoothness Rectilinear interpolation is only piecewise differentiable. As with linear interpolation, when reduced to an ODE then the vector field has jumps. Fixed step size solvers are one resolution to this problem. Alternatively an adaptive step size numerical solver can be made aware of these jumps so as to treat them in the appropriate way, see Section 5.3.3.

Choice of s_j and r_j Typically $s_j = t_j + j$ and $r_j = t_j + j - 1$ is a reasonable choice.

3.5.3.4 Natural cubic splines

One simple approach is to use natural cubic splines. (Indeed this was used in the original neural CDE paper [Kid+20a].)

Measurability Unfortunately, natural cubic splines are not measurable. This limits their applicability.

Smoothness However, natural cubic splines are at least very smooth.

Choice of s_j Typically $s_j = t_j$ is a reasonable choice.

[6]Explicitly: let $\bar{x}_{j,k} = x_{\bar{j}(j,k),k}$, where $\bar{j}(j,k) = \max\{m \leq j \mid x_{m,k} \neq *\}$, recalling that $*$ denotes missing data. If this set is empty then define $\bar{x}_{j,k} = 0$.

Model performance For reasons unknown, neural CDEs that use natural cubic splines produce slightly worse results than those using other interpolation schemes [Mor+21a].

3.5.3.5 Overall

Most problems will find either Hermite cubic splines with backward differences or rectilinear interpolation to be of most interest. Use Hermite cubic splines with backward differences if possible, due to their smoothness and relatively good measurability. If their measurability properties are insufficient, then use rectilinear interpolation.

3.6 Comments

CDEs are a classic piece of mathematics, emerging essentially as a meaningful special case of the more general rough differential equations introduced by [Lyo98]. The first few pages of [Hod+20] give an excellent brief introduction to the essential ideas. [LCL04] is our recommended introductory text. [FV10] is the canonical reference text.

(As a fascinating historical note, the essential ideas behind CDEs may actually be traced back to Newton [New36, Prob. 1, Prob. 2]. Newton considers evolving systems in multiple variables, with position (fluent) and derivative (fluxion). Given a relation for either fluent or fluxion, then the other is then solved for. Obtaining the relation of fluents from a relation of fluxions is precisely what we would term 'solving a CDE'. We thank Terry Lyons for this observation.)

Various texts use different pieces of terminology to refer to the concept of a CDE. Much of the rough path literature uses the terms CDE and RDE interchangeably. Meanwhile [FV10] prefer to use CDE and ODE interchangeably. Some texts use the term 'controlled ordinary differential equations'.

Specifically *neural* CDEs were first introduced in [Kid+20a], where they were applied to both regular and irregular time series, and most of the relevant theoretical properties discussed. The follow-up work [Mor+21a] investigated the choice of interpolation scheme, and promoted the use of the alternative ones just presented ([Kid+20a] used only natural cubic splines). The discussion here is an extension of the one introduced there, and is partly new here. Rectilinear interpolation is due to [LLN13]; the pieces of the interpolation for which the time channel is constant are sometimes referred to as 'virtual time'.

Many of the computational concerns discussed (batching, smoothness and so on) arose from experimentation using both [Kid20] and [Kid21a]. The discussion on batching (Section 3.2.1.3) is new here, and is relevant beyond just CDEs – much of the literature has assumed that a differential equation solver must operate over the same time interval for each batch element, and had to work around this limitation

(see for example the 'time-varying CNF' of [CAN21b]).

We recall the link between CDEs and control theory (Section 3.1.6.2). Meanwhile control theory has well-known links to reinforcement learning (RL). RL applications either explicitly including or of essentially similar character to neural CDEs therefore include [ARF20; Kil+20; Lut+21] amongst others.

The neural RDE formulation applying neural CDEs to long time series was introduced in [Mor+21b]. The same rough path theoretic ideas also appear in [Fer+21], to frame RNNs as a kernel method.

Applications to training neural SDEs (really the focus of our next chapter) were introduced in [Kid+21b; Kid+21a].

Much of the discussion on good architectural choices, gating, sparsity and so on, is new here.

Chapter 4

Neural Stochastic Differential Equations

4.1 Introduction

4.1.1 Stochastic differential equations

Stochastic differential equations have seen widespread use for modelling real-world random phenomena, such as particle systems [CKW12; Pav14; LS16], financial markets [BS73; CIR85; BM01], population dynamics [Ara03; SP03] and genetics [Hui07]. They are a natural extension of ordinary differential equations (ODEs) for modelling systems that evolve in continuous time subject to uncertainty.

The dynamics of an SDE consist of a deterministic term and a stochastic term:

$$\mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t), \tag{4.1}$$

where

$$\mu\colon [0,T] \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y},$$
$$\sigma\colon [0,T] \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_w}$$

are suitably regular functions, $w\colon [0,T] \to \mathbb{R}^{d_w}$ is a d_w-dimensional Brownian motion, and $y\colon [0,T] \to \mathbb{R}^{d_y}$ is the resulting d_y-dimensional continuous stochastic process.

The strong solution y is guaranteed to exist and be unique given mild conditions: that μ, σ are Lipschitz, and that $\mathbb{E}\left[y(0)^2\right] < \infty$.

We refer the reader to [RY13] for a rigorous account of stochastic integration.

Itô versus Stratonovich The notation "$\circ$" in the noise refers to the SDE being understood in the sense of Stratonovich integration. This is as an alternative to the standard notion of Itô integration.

The reader unfamiliar with Stratonovich integration should generally feel free to ignore this subtlety. Stratonovich SDEs will sometimes be slightly more efficient to backpropagate through (Remark 5.12, later). However, any Itô SDE may be converted to a Stratonovich SDE, and vice versa, so as we will shortly introduce learnt (neural) vector fields then modelling-wise the choice is arbitrary.

Theoretical construction of SDEs SDEs have typically been constructed theoretically, and are usually relatively simple.

One frequent and straightforward technique is to fix a constant matrix σ, and add "$\sigma \circ \mathrm{d}w(t)$" to a pre-existing ODE model.[1]

As another example, the Black–Scholes equation, widely used to model asset prices in financial markets, has only two scalar parameters: a fixed drift and a fixed diffusion [BS73].

Calibrating SDEs Once an SDE model has been chosen, then model parameters must be calibrated[2] from real-world data.

Since SDEs produce random sample paths, the parameters are typically chosen so that the average behaviour of the SDE matches some statistic(s). A classical approach to calibrating SDEs to observed data y_{true} is to pick some prespecified functions of interest $F_1, \ldots, F_N$, and then ask that $\mathbb{E}_y\left[F_i(y)\right] \approx \mathbb{E}_{y_{\text{true}}}\left[F_i(y_{\text{true}})\right]$ for all i. For example this may be done by optimising

$$\min_{\theta} \max_{i=1,\ldots,N} \left| \mathbb{E}_y\left[F_i(y)\right] - \mathbb{E}_{y_{\text{true}}}\left[F_i(y_{\text{true}})\right] \right| \tag{4.2}$$

where the model y depends implicitly on parameters θ.

This ensures that the model and the data behave the same with respect to the functions F_i. The functions F_i are known as either 'witness functions' or 'payoff functions' depending on the field [Li+17; CKT20]. If the SDE is simple enough – for example the analytically tractable Black–Scholes model – then equation (4.2) can often be computed explicitly [BS73].

4.1.2 Generative and recurrent structure

SDEs feature inherent randomness. In modern machine learning parlance SDEs are generative models.

[1] In passing we remark that Itô and Stratonovich are identical in this case as the noise is additive so the corresponding Itô–Stratonovich correction term is zero. We could equally well have written "$\sigma\, \mathrm{d}w(t)$".

[2] Fit, trained.

Comparison to random RNNs As usual, a numerically discretised neural (stochastic) differential equation has a correspondence in the deep learning literature. As with neural CDEs, the appropriate analogy is an RNN. In this case its input is random noise – Brownian motion – and its output is a generated sample.

Consider the autonomous one-dimensional Itô SDE

$$\mathrm{d}y(t) = \mu(y(t))\,\mathrm{d}t + \sigma(y(t))\,\mathrm{d}w(t),$$

with $y(t), \mu(y(t)), \sigma(y(t)), w(t) \in \mathbb{R}$. Then its numerical Euler–Maruyama discretisation is

$$y_{j+1} = y_j + \mu(y_j)\Delta t + \sigma(y_j)\Delta w_j,$$

where Δt is some fixed time step and $\Delta w_j \sim \mathcal{N}(0, \Delta t)$. This numerical discretisation is clearly just an RNN of a particular form.

Generative time series models Each sample y from an SDE

$$\mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t),$$

is a continuous-time path $y\colon [0, T] \to \mathbb{R}^{d_y}$. As such, we may treat neural SDEs as generative time series models.

(Generative) time series models are of classical interest, with forecasting models such as Holt–Winters [Hol57; Win60], ARMA [HR82], ARCH [Eng82], GARCH [Bol86] and so on.

It has also attracted much recent interest with, besides neural SDEs, the development of ODE-based models like latent ODEs (Section 2.2.4)[3]; discrete-time models like Time Series GAN [YJS19]; non-ODE continuous-time models like CTFPs [Den+20; Den+21] and Copula Processes [WG10].

See Figure 4.1 for an abstract summary of many of the essential ideas: that SDEs are generative time series models, how SDEs may classically be calibrated, and one of the ways in which we will later generalise this approach to neural networks, via *SDE-GANs* (Section 4.3.1).

'Static' generative models We may also consider just the terminal value $y(T)$ of an SDE

$$\mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t).$$

This is a sample drawn from some distribution over $\mathbb{R}^{d_y}$. As such we may also treat neural SDEs as 'static' generative models – that is to say, not over a time series.

This immediately draws natural connections to a variety of topics. This is the same basic set-up as a continuous normalising flow (Section 2.2.3), except that the randomness is injected via a Brownian motion w rather than a random initial condition $y(0)$.

[3]And related ideas such as ODE^{2}VAE [YHL19] or Neural ODE Processes [Nor+21]

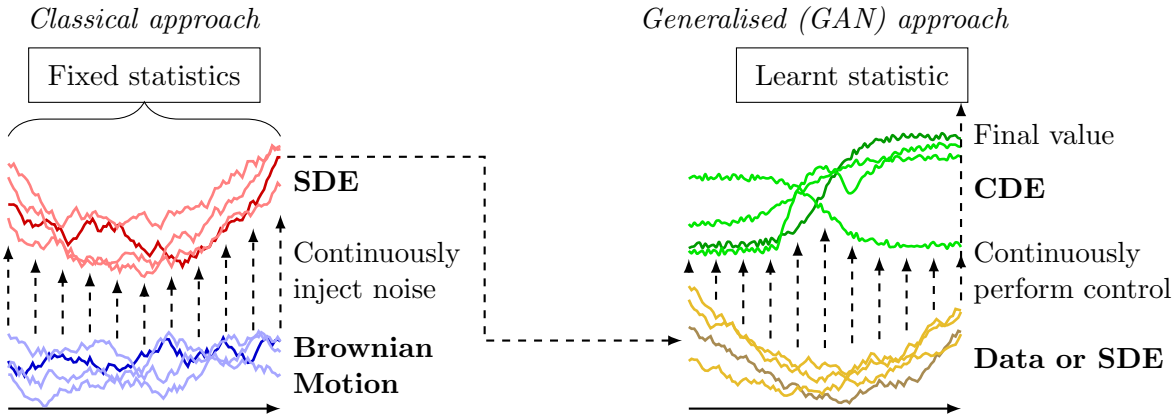


Figure 4.1: Brownian motion is continuously injected as noise into an SDE to generate time series. The classical approach fits the SDE to prespecified statistics. One (important) way of handling neural SDEs is to generalise from prespecified statistics to a learnt statistic, namely the discriminator of a (Wasserstein) GANs.

It is also the same starting point used in score-based generative modelling, in which a neural drift and fixed additive diffusion is used, with the initial-to-terminal map calculating a transition between two distributions [Son+21b; Bor+21].

We shall focus mainly on the time-series case discussed in the previous heading. At time of writing, the connections between neural SDEs as presented here, and CNFs and score-based modelling, are largely unexplored.

Comparison to neural CDEs We have now described both neural CDEs and neural SDEs as 'continuous time RNNs'. It is worth being precise about the distinction.

(Neural) CDEs model *functions of time series*, or equivalently *functions of paths*. The path is an input and the output is, for example, a classification result determining whether the input path is a clockwise or anticlockwise spiral.

(Neural) SDEs model *distributions on time series*, or equivalently *distributions on paths*. Rather than modelling some function of the path, it is is the paths themselves that are being modelled.

In this respect the terminology of differential equations is slightly more precise than the terminology of neural networks, which uses 'RNN' to describe both concepts.

4.2 Construction

The following constructions are primarily from [Kid+21b].

Let $T > 0$ be a fixed time horizon and consider a path-valued random variable

$x_{\text{true}}\colon [0,T] \to \mathbb{R}^{d_x}$, with $d_x \in \mathbb{N}$ the dimensionality of the data. x_{true} is what we wish to model, and is the random variable we assume we have observed samples from. For example, this may correspond to the evolution of stock prices over time.

(Typically we actually observe x_{true} only at some discretised time stamps; not over a full continuous-time path. For ease of presentation we neglect this detail for now and will return to it later.)

Let $w\colon [0,T] \to \mathbb{R}^{d_w}$ be a d_w-dimensional Brownian motion, and let $v \sim \mathcal{N}(0, I_{d_v \times d_v})$ be drawn from a d_v-dimensional standard multivariate normal. The values $d_w, d_v \in \mathbb{N}$ are hyperparameters describing the size of the noise. Let

$$\begin{aligned} \zeta_\theta &\colon \mathbb{R}^{d_v} \to \mathbb{R}^{d_y}, \\ \mu_\theta &\colon [0,T] \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y}, \\ \sigma_\theta &\colon [0,T] \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_w}, \\ \alpha_\theta &\in \mathbb{R}^{d_x \times d_y}, \\ \beta_\theta &\in \mathbb{R}^{d_x}, \end{aligned} \tag{4.3}$$

where ζ_θ, μ_θ and σ_θ are neural networks. Collectively ζ_θ, μ_θ, σ_θ, α_θ and β_θ are parameterised by θ. The dimension d_y is a hyperparameter describing the size of the hidden state.

Then a *neural stochastic differential equation* is a model of the form

$$\begin{aligned} y(0) &= \zeta_\theta(v), \\ \mathrm{d}y(t) &= \mu_\theta(t, y(t))\,\mathrm{d}t + \sigma_\theta(t, y(t)) \circ \mathrm{d}w(t), \\ x(t) &= \alpha_\theta y(t) + \beta_\theta, \end{aligned} \tag{4.4}$$

for $t \in [0,T]$, with $y\colon [0,T] \to \mathbb{R}^{d_y}$ the (strong) solution to the SDE.

The objective will be to train θ so that the distribution of the model x is approximately equal to the distribution of the data x_{true}. (For some notion of 'approximate'.)

Architecture Equation (4.4) has a certain minimum amount of structure. First, the solution y represents hidden state. If it were the output, then future evolution would satisfy a Markov property which need not be true in general. This is the reason for the additional readout operation to x.

Second, there must be an additional source of noise for the initial condition, passed through a nonlinear ζ_θ, as $x(0) = \alpha_\theta \zeta_\theta(v) + \beta_\theta$ does not depend on the Brownian noise w. This will be a learnt approximation to the initial condition of the SDE.

ζ_θ, μ_θ, and σ_θ may be taken to be any standard network architectures, such as feed-forward networks.

RNNs as discretised SDEs This minimal amount of structure parallels that of RNNs. The solution y corresponds to the hidden state of an RNN.

Sampling Given a trained model, we sample from it by sampling some initial noise v and some Brownian motion w, and then solving equation (4.4) with a numerical SDE solver.

Comparison to the Fokker–Planck equation The distribution of an SDE, as learnt by a neural SDE, contains more information than the distribution obtained by learning a corresponding Fokker–Planck equation. The solution to a Fokker–Planck equation gives (the time evolution of) the probability density of a solution *at fixed times*. It does not encode information about the time evolution of individual sample paths. This is exemplified by stationary processes, whose sample paths may be nonconstant but whose distribution does not change over time.

4.3 Training criteria

Equation (4.4) produces a random variable $x\colon [0, T] \to \mathbb{R}^{d_x}$ implicitly depending on parameters θ. This model must still be fit to data. This may be done by optimising a distance between the probability distributions (laws) for x and x_{true}.

There are two main options: fitting a Wasserstein distance, or fitting a KL divergence. These correspond to *SDE-GANs* and *latent SDEs* respectively.

4.3.1 SDE-GANs

Let $\mathbb{P}_x$ denote the law of the model x. Likewise let $\mathbb{P}_{x_{\text{true}}}$ denote the (empirical) law of the data x_{true}. Let $W(\mathbb{P}_x, \mathbb{P}_{x_{\text{true}}})$ denote the 1-Wasserstein distance between them. We may train the model by optimising

$$\min_\theta W(\mathbb{P}_x, \mathbb{P}_{x_{\text{true}}}),$$

where $\mathbb{P}_x$ depends implicitly on the learnt parameters θ.

We will do so in the usual way for Wasserstein GANs, by constructing a discriminator and training adversarially [ACB17].

Each sample from the generator is a continuous path $x\colon [0, T] \to \mathbb{R}^{d_x}$; these are infinite dimensional and the discriminator must accept such paths as inputs. Fortunately there is a natural choice: parameterise the discriminator as a neural CDE, as in Chapter 3.

This approach is due to [Kid+21b].

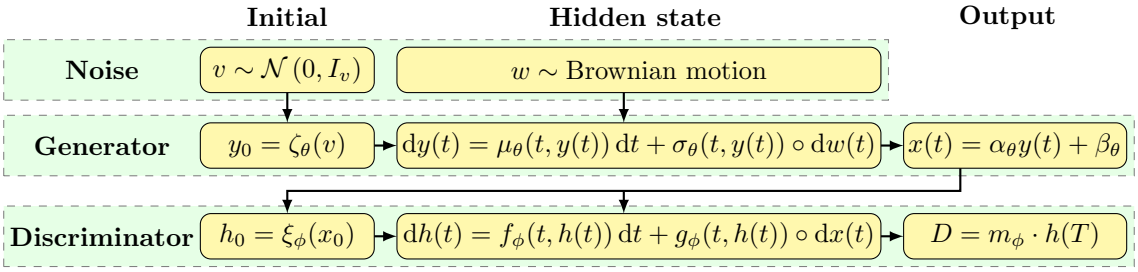


Figure 4.2: Summary of equations for an SDE-GAN.

Architecture Let

$$\begin{aligned}\xi_\phi &\colon \mathbb{R}^{d_x} \to \mathbb{R}^{d_h},\\ f_\phi &\colon [0,T] \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h},\\ g_\phi &\colon [0,T] \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h \times d_x},\\ m_\phi &\in \mathbb{R}^{d_h},\end{aligned}$$

where ξ_ϕ, f_ϕ and g_ϕ are (Lipschitz) neural networks. Collectively they are parameterised by ϕ. The value $d_h \in \mathbb{N}$ is a hyperparameter describing the size of the hidden state.

Recalling that x is the generated sample, we take the discriminator to be a CDE

$$\begin{aligned}h(0) &= \xi_\phi(x(0)),\\ \mathrm{d}h(t) &= f_\phi(t, h(t))\,\mathrm{d}t + g_\phi(t, h(t)) \circ \mathrm{d}x(t),\\ D &= m_\phi \cdot h(T),\end{aligned} \tag{4.5}$$

for $t \in [0,T]$, with $h \colon [0,T] \to \mathbb{R}^{d_h}$ the (strong) solution to this CDE, and where $\cdot$ denotes the dot product.

The solution to the CDE exists given mild conditions, namely Lipschitz f_ϕ and g_ϕ; simply concatenate (4.4) and (4.5) together and treat the joint system as an SDE.

The value $D \in \mathbb{R}$, which is a function of the terminal hidden state $h(T)$, is the discriminator's score for real versus fake; correspondingly we define the overall action of the discriminator via $F_\phi(x) = D$. This is a deterministic function of the generated sample x.

Summary of equations See Figure 4.2 for a summary of equations, combining together both generator and discriminator.

Training loss The training loss is the usual one for Wasserstein GANs [Goo+14; ACB17], namely optimisation with respect to

$$\min_\theta \max_\phi \left(\mathbb{E}_x \left[F_\phi(x) \right] - \mathbb{E}_{x_{\text{true}}} \left[F_\phi(x_{\text{true}}) \right] \right).$$

Training is performed via stochastic gradient descent techniques as usual.

This generalises the classical approach to calibration seen in equation (4.2). Instead of optimising over some fixed collection of payoff functions $\{F_i\}_{i=1}^N$, we optimise over some infinite collection of discriminators $\{F_\phi\}_\phi$.

Remark 4.1. *In Chapter 3, we emphasised the need to include time as a channel in the control of a neural CDE. This corresponds to the inclusion of a 'drift' term in equation* (4.5)*. Equivalently we could replace x with $t \mapsto (t, x(t))$ and use the same notation as Chapter 3.*

4.3.1.1 Lipschitz regularisation

Wasserstein GANs need a Lipschitz discriminator. A variety of methods have been proposed in the GAN literature, such as weight clipping [ACB17], gradient penalty [Gul+17], or spectral normalisation [Miy+18]. The recurrent nature of the SDE setting means that a little care is needed to employ these successfully – see Section 4.4.3.

4.3.1.2 Discretised observations

Observations of x_{true} are typically a discrete time series, rather than a true continuous-time path. This is not a serious hurdle. Simply evaluate (4.5) on an interpolation x_{true} of the observed data. The effect of this is as follows.

Dense data regime Suppose we observe samples from x_{true} 'densely' – that is, with little gap between successive values in time. (Of approximately no more than the step size of the numerical solver.) Then interpolation produces a distribution in path space; the one desired to be modelled. Simple linear interpolation will be sufficient, but due to the dense sampling of the data this is a choice that is largely unimportant.

Technically speaking, as (linear) interpolation will produce a path of bounded variation, then (4.5) will be defined as a Riemann–Stieltjes integral.

Sparse data regime Now suppose data is not observed densely, and may even have substantial time gaps between observations. In this case, we fall back to the neural CDE approach: sample the generated paths at some collection of time points, and interpolate both the generated sample *and* the true data. (Before passing them to the discriminator defined as a Riemann–Stieltjes integral in both cases.)

This is the familiar setting for applying neural CDE to time series, as set up in Chapter 3. The interpolation scheme has simply become part of the discriminator, and no modelling or discriminatory power is lost.

4.3.1.3 Single SDE solve

If working in the dense data regime, then (4.4) and (4.5) may be concatenated together into a single SDE solve. This is of relevance if training using optimise-then-discretise, or with a reversible solver. Both of these are topics we will discuss in Chapter 5; the reader unfamiliar with these concepts should feel free to skip this heading for now.

The state is the combined $[y, h]$, the initial condition is the combined

$$[\zeta_\theta(v),\ \xi_\phi(\alpha_\theta\zeta_\theta(v) + \beta_\theta)],$$

the drift is the combined

$$[\mu_\theta(t, y(t)),\ f_\phi(t, h(t)) + g_\phi(t, h(t))\alpha_\theta\mu_\theta(t, y(t))],$$

and the diffusion is the combined

$$[\sigma_\theta(t, y(t)),\ g_\phi(t, h(t))\alpha_\theta\sigma_\theta(t, y(t))].$$

Then $h(T)$ is extracted from the final hidden state, and m_ϕ applied, to produce the discriminator's score for that sample.

Training in this way improves memory efficiency, as the SDE solution $y\colon [0, T] \to \mathbb{R}^{d_y}$ and the output $x\colon [0, T] \to \mathbb{R}^{d_x}$ are not recorded during training. The asymptotics improve from $\mathcal{O}(H + T)$, as in Section 3.1.5.3, to just $\mathcal{O}(H)$, where H is the memory cost of evaluating and backpropagating the vector fields once.

4.3.2 Latent SDEs

We will now consider training not with respect to the Wasserstein distance, but with respect to the KL divergence. This approach is due to [Li+20a].

Let

$$\begin{aligned}\xi_\phi &\colon \mathbb{R}^{d_x} \to \mathbb{R}^{d_y} \times (0, \infty)^{d_y},\\ \nu_\phi &\colon [0, T] \times \mathbb{R}^{d_y} \times \{[0, T] \to \mathbb{R}^{d_x}\} \to \mathbb{R}^{d_y},\end{aligned} \tag{4.6}$$

be Lipschitz neural networks parameterised by ϕ. The notation $\{[0, T] \to \mathbb{R}^{d_x}\}$ denotes the space of all functions $[0, T] \to \mathbb{R}^{d_x}$.

Remark 4.2. *We do not discuss the regularity of the functions in $\{[0, T] \to \mathbb{R}^{d_x}\}$, as this input to ν_θ will actually be a sample of x_{true}, and in practice we will have discrete observations.*

ν_ϕ is commonly parameterised as $\nu_\phi(t, y, x) = \nu_{\phi,1}(t, y, \nu_{\phi,2}(x|_{[t,T]}))$, where $\nu_{\phi,1}$ is an MLP and $\nu_{\phi,2}$ is either a reverse-time RNN/NCDE or the evaluation function $\nu_{\phi,2}(x|_{[t,T]}) = x(t)$.

Let $(m, s) = \xi_\phi(x_{\text{true}}(0))$, let $\widehat{v} \sim \mathcal{N}(m, \operatorname{diag}(s)^2)$, and let

$$\widehat{y}(0) = \zeta_\theta(\widehat{v}), \qquad \mathrm{d}\widehat{y}(t) = \nu_\phi(t, \widehat{y}(t), x_{\text{true}})\,\mathrm{d}t + \sigma_\theta(t, \widehat{y}(t)) \circ \mathrm{d}w(t), \qquad \widehat{x}(t) = \alpha_\theta \widehat{y}(t) + \beta_\theta.$$

Note that w is the same Brownian motion as used in (4.4). Similarly $\alpha_\theta \in \mathbb{R}^{d_x \times d_y}$, $\beta_\theta \in \mathbb{R}^{d_x}$ and σ_θ are the same objects defined in (4.3).

In doing so, we have constructed another SDE using the same diffusion as the main generative model, but with a different initial condition and drift. There is a standard formula for the KL divergence between two SDEs with the same diffusion, which in this case is given by

$$\mathrm{KL}\left(\widehat{y} \| y\right) = \mathbb{E}_w \int_0^T \frac{1}{2} \left\| (\sigma_\theta(t, \widehat{y}(t)))^{-1} (\mu_\theta(t, \widehat{y}(t)) - \nu_\phi(t, \widehat{y}(t), x_{\text{true}})) \right\|_2^2 \mathrm{d}t, \tag{4.7}$$

where $(\sigma_\theta(t, \widehat{y}(t)))^{-1}$ is the Moore–Penrose pseudoinverse of $\sigma_\theta(t, \widehat{y}(t))$. Note that although y does not appear explicitly on the right hand side, y defines (and is defined by) the μ_θ and σ_θ which do appear.

Remark 4.3. *Equation* (4.7) *may be identified as an integral over the KL divergence between two Gaussians.*

This opens up a possible training procedure. This 'auxiliary' SDE, which depends on samples of the observed data x_{true}, may be used to autoencode the data. Once the data is represented as an SDE, we may remove the dependence on x_{true} by minimising a KL divergence between our original generative model and the auxiliary model.

Explicitly, this corresponds to training according to

$$\min_{\theta,\phi} \mathbb{E}_{x_{\text{true}}} \left[(\widehat{x}(0) - x_{\text{true}}(0))^2 + \mathrm{KL}\left(\widehat{v} \| v\right) + \mathbb{E}_w \int_0^T (\widehat{x}(t) - x_{\text{true}}(t))^2 \,\mathrm{d}t + \mathrm{KL}\left(\widehat{y} \| y\right) \right].$$

Remark 4.4. *Note that this training procedure only involves solving the auxiliary SDE, never the original SDE. The main generative model is trained without ever being evaluated.*

As a variational autoencoder [Li+20a] interpret this procedure as a variational autoencoder, with a learnt prior, whose latent space is an entire stochastic process. (And indeed the above formula may be derived as an evidence lower-bound.) For this reason [Li+20a] refer to the auxiliary SDE as a posterior SDE.

Interpreted in this way, the first two terms are a VAE for generating $x(0)$, with latent v. Meanwhile the third term and fourth term are a VAE for generating x, by autoencoding x_{true} to $\widehat{x}$, and then fitting y to $\widehat{y}$.

Single SDE solve Equation (4.7) is an integral, and so may be estimated by concatenating it alongside the SDE solve.

Alternate probability densities The first and third terms of (4.7) are the L^2 loss, which corresponds to maximising the log-likelihood of $x_{\text{true}}(t)$ with respect to a fixed-variance Gaussian whose mean is $\widehat{x}(t)$:

$$\mathcal{N}\left(\widehat{x}(t), \frac{1}{\sqrt{2}} I_{d_y \times d_y}\right).$$

However other probability densities are also admissible. As such the above presentation is chosen for simplicity, and compatibility with the presentation of the generative model in Section 4.2. The affine map corresponding to $\alpha_\theta, \beta_\theta$ is being used to produce the mean of a fixed-variance Gaussian, but it may be replaced by any other procedure for producing the parameters of some probability distribution, and the log-likelihood optimised as normal.

4.3.3 Comparisons and combinations

The difference between SDE-GANs and latent SDEs is essentially the standard GAN/VAE split. SDE-GANs are more finicky to train, but exhibit substantially higher modelling capacity. Conversely, latent SDEs are easy to train, but often produce worse final models; in particular it is a common feature of latent SDEs that their diffusion will be too small.

It is possible to combine both latent SDEs and SDE-GANs together. (And indeed GAN/VAE hybrids have been proposed in the main deep learning literature too [Lar+15; Bou+17; Ros+17].) This is a way to offset the weakness of each approach with the strengths of the other. An example of this is given in Section 4.5, applied to modelling a Lorenz system.

4.4 Choice of parameterisation

As usual with deep learning, the theoretical construction is only half of the work needed to produce a workable model, and the 'engineering details' – of finding good hyperparameters, optimisers, and so on – still remain.

At time of writing, finding good choices is still largely an open problem for neural SDEs. Much inspiration can likely be drawn from the mainstream generative modelling literature, which has spent the past few years investigating this topic in depth: see for example negative momentum [Gid+19], complex momentum [Lor+21], stochastic weight averaging (Cesàro means) [Izm+18; Yaz+19], progressive growing [Kar+18], Lipschitz regularisation [Gul+17; Miy+18], architectural choices [Luc+18; Kar+19] and so on.

4.4.1 Choice of optimiser

4.4.1.1 SDE-GANs

SDE-GANs can be relatively unstable to train.

Adadelta Empirically, Adadelta [Zei12], or the similar RMSprop, seems to outperform either SGD or Adam when training SDE-GANs. In part this is because Adadelta lacks momentum; a lack of momentum is beneficial as the optimisation criterion for a GAN is a moving target.

Adam with $\beta_1 = 0$, where β_1 is its momentum hyperparameter, also seems to be outperformed by Adadelta [Wal21].

Learning rate The initial networks ζ_θ and ξ_ϕ often work best with a larger learning rate than is used for the rest of the model. (For example a factor of 10 would be typical.) This helps to offset the fact that the initial distribution (of $x(0)$) often gets relatively weak supervision compared to the time-varying component (of $t \mapsto x(t)$).

Stochastic weight averaging Using the Cesàro mean of both the generator and discriminator weights, averaged over training, can improve performance in the final model [Izm+18; Yaz+19]. This averages out the oscillatory training behaviour for the min-max objective used in GAN training.

4.4.1.2 Latent SDEs

Latent SDEs are relatively easy to train. Given their VAE-like structure, standard optimisers like Adam [KB15] work without difficulty.

Once again it is still usually worth increasing the learning rate for ζ_θ and ξ_ϕ.

4.4.2 Choice of architecture

4.4.2.1 Generator

Recall that μ_θ and σ_θ were the drift and diffusion of the SDE, defined in (4.3).

μ_θ and σ_θ are typically taken to be MLPs. Numerical SDE solvers will usually demand that the vector fields be sufficiently smooth (for example, bounded with continuous bounded first and second derivatives), so the activation function is often taken to be smooth, like softplus or SiLU.

Final nonlinearities It is common to add a final tanh nonlinearity to μ_θ and σ_θ. This is for the same reason as neural CDEs: to prevent an unconstrained rate of change in the hidden state and the model potentially exploding (especially at initialisation). If this constrains the rate of change too strongly, then this may be managed by parameterising μ_θ and σ_θ as

$$(t, y) \mapsto \gamma \tanh(\text{MLP}_\theta(t, y)),$$

where $\gamma \in \mathbb{R}$ is a learnt scalar (part of θ).

Initialisation As with ODEs (Section 2.3.1.3), training dynamics may be improved by initialising μ_θ and σ_θ close to zero.

Choice of driving noise The construction of this chapter has taken the driving noise w to be a Brownian motion. This choice is not necessary; for example fractional Brownian motion or Lévy processes could also be used, together with or instead of the Brownian motion w.

A choice of particular interest are counting processes $t \mapsto N(t)$ (for example the cumulative sum of a Poisson process) so that the resulting SDE is a jump process

$$\mathrm{d}y(t) = \mu_\theta(t, y(t))\,\mathrm{d}t + \sigma_\theta(t, y(t)) \circ \mathrm{d}w(t) + \lambda(t, y(t-))\,\mathrm{d}N(t),$$

where the notation '$t-$' is used to emphasise that the vector field depends upon the value immediately prior to the jump.

The optimisation criteria can get slightly more involved in these cases: whilst the SDE-GAN approach translates over without any changes, at time of writing the latent SDE approach has not yet been explored. See also [JB19] who develop a direct likelihood-based approach to optimise diffusionless drift/jump processes of the form

$$\mathrm{d}y(t) = \mu_\theta(t, y(t))\,\mathrm{d}t + \lambda(t, y(t-))\,\mathrm{d}N(t),$$

Diffusions for latent SDEs When training a latent SDE, then the KL divergence of equation (4.7), used in the latent SDE, multiplies by the (pseudo)inverse of σ_θ. This is expensive to compute for general matrices.

One effective simplification is to take $d_w = d_y$ and parameterise the diffusion σ_θ as a diagonal matrix. This is cheap to compute the inverse of: take the reciprocal of each diagonal element.

For numerical stability it is additionally often desirable to then bound these diagonal elements away from zero: use $z \mapsto \text{sigmoid}(z) + 10^{-4}$ as a final nonlinearity for σ_θ, or alternatively clamp any values in the range $[-10^{-6}, 10^{-6}]$ to the edges of that range.

Approximation properties Provided μ_θ and σ_θ are drawn from suitable (universal approximating) classes of functions, then it is clear that (4.4) is more than capable of approximating any Markov SDE, by the universal approximation theorem for neural networks [Pin99; KL20b] and standard approximation results for SDEs.

What is less clear is its ability to model non-Markov SDEs. Certainly this is possible to some extent, due to the explicit use of hidden state. (Indeed this is the reason hidden state is introduced in the first place.) At time of writing a formal result has not been derived.

4.4.2.2 Discriminator

When training an SDE-GAN, then additional networks ξ_ϕ, f_ϕ, g_ϕ are introduced. These should be parameterised in accordance with neural CDEs (Section 3.4).

The initial distribution, learnt by ζ_θ, can often be improved by providing it additional supervision during training. Redefine $D = m_\phi \cdot h(0) + m_\phi \cdot h(T)$ or $D = \kappa_\phi(x(0)) + m_\phi \cdot h(T)$ instead of just $D = m_\phi \cdot h(T)$ in equation (4.5), where κ_ϕ is some neural network.

As with any Wasserstein GAN, the discriminator should be Lipschitz. This is the focus of our next section.

4.4.3 Lipschitz regularisation

This section is specific to SDE-GANs. SDE-GANs, as with any Wasserstein GAN, need a Lipschitz discriminator.

A variety of methods for enforcing Lipschitzness have been proposed in the general GAN literature, such as weight clipping [ACB17], gradient penalty [Gul+17], or spectral normalisation [Miy+18]. However a little care must be taken when applying these to the discriminator of an SDE-GAN.

Much of the following discussion originated in [Kid+21a].

4.4.3.1 Exponential Lipschitz constant

Given vector fields with Lipschitz constant λ, then the recurrent structure of the discriminator means that the Lipschitz constant of the overall discriminator will be $\mathcal{O}(\lambda^T)$. This is a key consideration in performing Lipschtiz regularisation, and unfortunately, the aforementioned techniques cannot simply be applied 'off the shelf'.

Lipschitz constant one The first option will be to somehow ensure that the vector fields f_ϕ and g_ϕ of the discriminator are not only Lipschitz, but have Lipschitz constant

at most one. Ensuring $\lambda \approx 1$ with $\lambda \leq 1$ will enforce that the overall discriminator is Lipschitz, with a Lipschitz constant of approximately one, as well.

This will be the approach we take in Section 4.4.3.2.

Hard constraint The exponential size of $\mathcal{O}(\lambda^T)$ means that λ only slightly greater than one is still insufficient for stable training. This is why we specify '$\lambda \approx 1$ with $\lambda \leq 1$' and not merely '$\lambda \approx 1$'. Moreover, it rules out enforcing $\lambda \approx 1$ via soft constraints like spectral normalisation.

Whole-discriminator regularisation The second option is to regularise the Lipschitz constant of whole discriminator, without regard for its recurrent structure. This will be the approach we take in Section 4.4.3.3.

4.4.3.2 Careful clipping

Let us (within this subsection) now assume that our discriminator vector fields f_ϕ, g_ϕ are MLPs. This is also a common choice made in practice.

Careful clipping Consider each linear operation from $\mathbb{R}^a \to \mathbb{R}^b$ as a matrix in $A \in \mathbb{R}^{a \times b}$. After each gradient update, clip its entries to the region $[-1/b, 1/b]$. Given $z \in \mathbb{R}^a$ then this enforces $\|Az\|_\infty \leq \|z\|_\infty$.

LipSwish activation function Next we must pick an activation function with Lipschitz constant at most one. It should additionally be at least twice continuously differentiable to ensure convergence of a numerical SDE solver. In particular this rules out the ReLU.

There remain several admissible choices. We tend to use the LipSwish activation function introduced by [Che+19], defined as $\rho(z) = 0.909\, z\sigma(z)$, where σ denotes the sigmoid function. This has Lipschitz constant one (due to the carefully-chosen 0.909 scaling factor), and is smooth. Moreover the SiLU activation function from which it is derived has been reported as an empirically strong choice [HG16; EUD17; RZL17].

Overall The overall vector fields f_ϕ, g_ϕ of the discriminator consist of linear operations (which are constrained by clipping), adding biases (an operation with Lipschitz constant one), and activation functions (taken to be LipSwish). Thus the Lipschitz constant of the overall vector field is at most one, as desired.

4.4.3.3 Gradient penalty

Another option is to directly regularise the Lipschitz constant of the entire discriminator, via gradient penalty. Add

$$\mathbb{E}_{\widehat{x}}\left[\left(\left\|\frac{\partial F_\phi}{\partial x}(\widehat{x})\right\| - 1\right)^2\right] \tag{4.8}$$

as a regularisation term to the training loss, where $\widehat{x}$ is sampled according to $\widehat{x} = \alpha x + (1-\alpha)x_{\text{true}}$ with $x \sim \mathbb{P}_x$ and $x_{\text{true}} \sim \mathbb{P}_{x_{\text{true}}}$ and $\alpha \sim \text{Uniform}[0, 1]$.

This approach works, which is more than can be said for other naïve approaches. However, compared to the careful clipping of Section 4.4.3.2, this approach mostly comes with disadvantages.

Disadvantages Because (4.8) involves calculating a gradient, then optimising it involves calculating a second derivative – a 'double backward'.

This is of relevance if training using optimise-then-discretise, which is a topic we will discuss in Chapter 5. (The reader unfamiliar with this concept should feel free to skip this heading for now.)

If training proceeds using optimise-then-discretise, then as a single backward constructs an 'adjoint SDE', a double backward constructs an 'adjoint-of-adjoint SDE'. This starts to imply substantial errors in the numerical discretisation, and this can be sufficient to degrade or destroy training.

Another negative is the additional computational cost implied by computing, and autodifferentiating, (4.8). This can easily result in a training procedure that takes about 50% longer than the careful clipping approach.

Remark 4.5. *We sidestep questions of how the derivative in* (4.8) *is defined – given that $\widehat{x}$ is path valued – by defining it with respect to the numerically discretised solution of $\widehat{x}$. In practice gradient penalty is not the preferred option, due to the disadvantages already discussed, so this is not an issue we will seek to tackle formally.*

4.5 Examples

Brownian motion As a simplest-possible first example, consider a dataset of samples of (univariate) Brownian motion, with initial condition Uniform$[-1, 1]$. Each element of the dataset is a time series of observations along a single Brownian sample path. We train a small SDE-GAN to match the distribution of the initial condition and the distribution of the time-evolving samples; see Figure 4.3.

This example may seem almost trivially simple, and yet it highlights a class of time series that would be almost impossible to learn with a latent ODE (Section 2.2.4). A Brownian motion represents pure diffusion, whilst a latent ODE is pure drift.

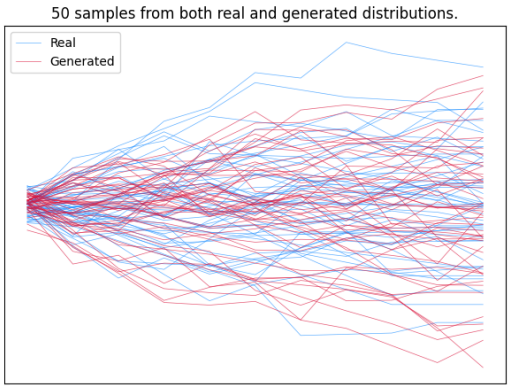


Figure 4.3: Coarsely-spaced (t, y) samples of a Brownian motion, and an SDE-GAN trained to match its distribution.

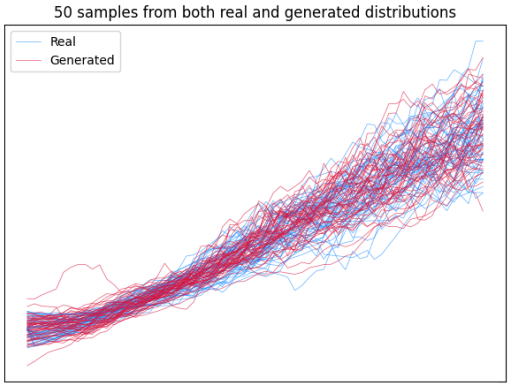


Figure 4.4: Finely-spaced (t, y) samples of a time-dependent Ornstein–Uhlenbeck process, and an SDE-GAN trained to match its distribution.

Time-dependent Ornstein–Uhlenbeck process Next we consider training an SDE-GAN to recover the distribution of

$$y(0) \sim \text{Uniform}[-1, 1], \qquad \mathrm{d}y(t) = (at - by(t))\,\mathrm{d}t + ct\,\mathrm{d}w(t), \qquad \text{for } t \in [0, 63].$$

We take in particular $a = 0.02$, $b = 0.1$, $c = 0.013$.

This example introduces explicit time dependency; in particular a time-dependent diffusion.

That this has both nontrivial drift and diffusion makes it an example of a process that is easy to learn via SDEs, but would be difficult to learn with models such as a latent ODE (which is pure drift; Section 2.2.4) or a CTFP (which is almost-pure diffusion; [Den+20]).

See Figure 4.4.

Damped harmonic oscillator Next we consider a dataset of samples from a two-dimensional damped harmonic oscillator.

$$y_1(0), y_2(0) \sim \text{Uniform}[-1, 1], \qquad \mathrm{d}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} -0.01 & 0.13 \\ -0.1 & -0.01 \end{bmatrix} \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} \mathrm{d}t,$$

for $t \in [0, 100]$.

This example is multidimensional, pure-drift, and solved over a long time interval.

In this case we train a latent SDE to recover the distribution. See Figure 4.5, which shows a single sample, in the (y_1, y_2)-plane, from both the true and generated dataset. (So that time evolves as the trajectory spirals inwards.)

It is by coincidence that the generated sample begins so close to, and partway along, the true sample. (They are not necessarily meant to overlap.) The generated sample

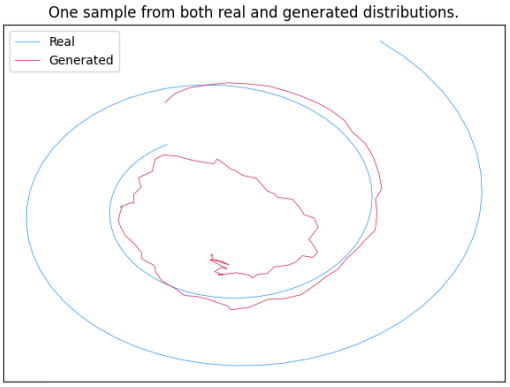


Figure 4.5: A single (y_1, y_2)-plane sample of a damped harmonic oscillator, and a latent SDE trained to match its distribution.

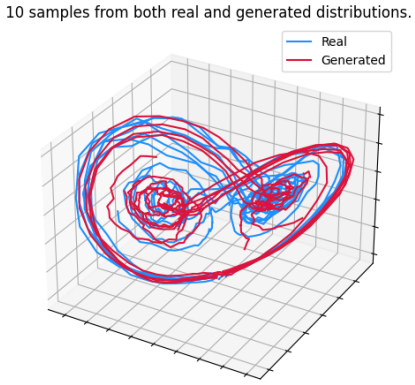


Figure 4.6: Evolving (y_1, y_2, y_3) samples from a Lorenz attractor, and a 'latent SDE-GAN' trained to match its distribution

does an excellent job at matching the drift, even extrapolating past the end of the true sample. The only issue is that the diffusion is still too high – indeed the true diffusion is zero – demonstrating that some additional training may still be required. Nonetheless this demonstrates how neural SDEs subsume neural ODEs as a special case, practically as well as theoretically.

Lorenz attractor We consider a dataset of samples from the Lorenz attractor

$$\begin{aligned} y &\sim \mathcal{N}\left(0, I_{3\times 3}\right), \\ \mathrm{d}y_1(t) &= a_1(y_2(t) - y_1(t))\,\mathrm{d}t + b_1 y_1(t)\mathrm{d}w(t), \\ \mathrm{d}y_2(t) &= (a_2 y_1(t) - y_1(t)y_3(t))\,\mathrm{d}t + b_2 y_2(t)\mathrm{d}w(t), \\ \mathrm{d}y_3(t) &= (y_1(t)y_2(t) - a_3 y_3(t))\,\mathrm{d}t + b_3 y_3(t)\mathrm{d}w(t), \end{aligned}$$

for $t \in [0, 2]$. We take specifically $a_1 = 10$, $a_2 = 28$, $a_3 = \frac{8}{3}$, $b_1 = 0.1$, $b_2 = 0.28$, $b_3 = 0.3$.

This example is multidimensional, chaotic, and has state-dependent diffusion.[4]

We train a combined latent SDE / SDE-GAN on this dataset. They are combined simply by interchanging separate training steps: one as a latent SDE, followed by one as an SDE-GAN. See Figure 4.6. The model has correctly learnt the distribution of this chaotic multidimensional time series.

Further details See Appendix D.4 for precise details on the experiments considered here. The code is available as an example in Diffrax [Kid21a].

[4]In passing, note that this is an Itô SDE. As discussed in Section 4.1.1, it is no issue that we are about to learn it with a Stratonovich neural SDE.

Irregular sampling Both the Brownian motion and the Ornstein–Uhlenbeck example were irregularly sampled with missing data. The process was observed at each integer (in the time domain) with only 70% probability, and unobserved otherwise.

The continuous-time approach discussed in this chapter means that this irregularity requires no special treatment. Moreover the output of each model evolves in continuous time and may be observed at any location.

Other examples Other (real-world) time series problems may be considered.

[Li+20a] give an example training latent SDEs to perform short-term forecasting on a 50-dimensional motion capture dataset.

[Kid+21b] consider a dataset of 14.6 million observations of Google/Alphabet stock prices, and train an SDE-GAN to replicate the evolution of the midpoint and spread as it evolves over a minute.

[Kid+21a] train both latent SDEs and SDE-GANs, and give an example modelling the air quality over Beijing.

4.6 Comments

Several authors have independently introduced notions of neural SDEs.

[Li+20a; Kid+21b] were the main works to derive the material presented here, whilst our presentation is derived from the follow-up [Kid+21a].

We have focused on using the Wasserstein distance or KL divergence to match model against data. In principle the classical calibration approach, using fixed statistics, may be employed in conjunction with neural vector fields, and this is now essentially the formulation of an MMD (Appendix A.5). Some care should be taken as to the choice of feature map. For example some authors have used only the mean and variance of the marginal distributions at each time t, and this fails to distinguish $t \mapsto w(t)$ from $t \mapsto \sqrt{t}w(1)$. A good choice of feature map is the signature transform [KL21]; for example this is done in [Kid+21b, Section 4] and [Kid+19, Section 4.1]. (There also exists a corresponding signature kernel [Sal+20; CLX21].)

[Bri+20; Gie+20] consider variations on the formulation given here, but optimise a distance only between finite-dimensional marginal distributions, rather than optimising the continuous-time model. [CKT20] consider another variation on this formulation, by adding known structure to the discriminator, corresponding to prespecified payoff functions of interest. [CRW21] consider specifically Markov neural SDEs and optimise via maximum likelihood (more-or-less equivalent to optimising the KL criterion considered here) as part of a larger framework for market models; indeed many of the above references target financial applications.

Meanwhile [TR19a; TR19b] obtain neural SDEs as a continuous limit of deep latent

Gaussian models, and largely focus on the theoretical construction.

The connections between score-based generative modelling and neural SDEs as presented here has not yet been explored in detail. We recommend the first few pages of [Bor+21] for an introduction to score-based generative modelling. [MRO20; ZC21] emphasise connections to continuous normalising flows. [Shi+21] give an application to molecular conformation, [Ho+21; DN21] give large-scale applications to image generation, and [Men+21] give an application to image editing. [HLC21; Kin+21; Son+21a] give variational/likelihood-based perspectives.

The mainstream deep learning literature frequently uses stochasticity as a regulariser. A neural SDE may likewise be treated as a regularised neural ODE or CDE [Liu+19; OVV20; Hod+20]. This will also be discussed as part of our numerical treatment of differential equations in Section 5.4.1.3.

[KSZ20] give one application of neural SDEs not discussed here, by using the stochasticity as part of procedure to distinguish between epistemic and aleatoric uncertainty.

Chapter 5

Numerical Solutions of Neural Differential Equations

5.1 Backpropagation through ODES

Training a neural differential equation usually means backpropagating through the differential equation solve. There are actually several ways to do this.

For clarity of exposition we begin by studying ODEs only, and will return to backpropagation through CDEs and SDEs in the next section.

We shall see three main ways of differentiating through an ODE.

- Discretise-then-optimise – memory inefficient, but accurate and fast;
- Optimise-then-discretise – memory efficient, but approximate and a little slow;
- Reversible ODE solvers – memory efficient and accurate, but a little slow.

Generally speaking discretise-then-optimise is the preferred approach. If this is not possible, typically due to memory constraints, then reversible ODE solvers are the next best option. Finally, if this is not suitable then optimise-then-discretise methods may be used, but these are typically the least-favoured approach.

All of these choices may typically be found in major differential equation software libraries (Section 5.6), so that the choice of backpropagation is usually an easy thing to change.

5.1.1 Discretise-then-optimise

The first option is simply to backpropagate through the internal operations of the differential equation solver.

A differential equation solver internally performs the usual arithmetic operations of addition, multiplication, and so on, each of which is differentiable. Given that a solve operation is a composition of differentiable operations, it is also differentiable.

This is known as 'discretise-then-optimise'. The derivatives are computed with respect to the discretised version of the differential equation that the solver computed, and not with respect to the idealised continuous-time equation.

5.1.1.1 Advantages

Accuracy of gradients The computed gradients will be accurate for the discrete model that is actually being used. This is in contrast to some of the techniques we shall see later, which compute only approximate gradients.

Speed This is often the quickest way to backpropagate. One reason for this is that the full computation graph is known prior to performing the backpropagation, and so the underlying autodifferentiation library may better exploit parallelism.

Ease of implementation The implementation of discretise-then-optimise is generally straightforward: provided the differential equation solver is written in an autodifferentiable framework (such as PyTorch or JAX), then gradients may automatically be computed in the usual way for these frameworks.

5.1.1.2 Disadvantages

Memory inefficiency This approach is memory-inefficient, as every internal operation of the solver must be recorded. If the memory cost of recording the operations of a single differential equation step is H, and recalling that T is the time horizon, then this approach consumes $\mathcal{O}(HT)$ memory.

This is in contrast to the techniques we shall see later, which reduce this to only $\mathcal{O}(H)$.

Remark 5.1. *In some sense it's a little unfair to state that discretise-then-optimise is memory-inefficient. It's simply performing backpropagation as normal, as with any other neural network model, and we do not usually refer to those as memory-inefficient. It is simply that the other options we see later can reduce memory costs to essentially negligible amounts.*

Difficulty of implementation In contrast to the 'ease of implementation' just discussed – if the differential equation solver is provided without having been written in an autodifferentiable framework, then this approach is essentially impossible to implement.

5.1.1.3 Checkpointing

It is possible to finesse the problem of memory inefficiency through checkpointing. That is, record the value of the forward pass at certain points during the solve, and use these to reconstruct values during the backward pass. This is a general technique in deep learning [Gri92]. [GKB19] discuss this in the specific context of neural ODEs.

5.1.2 Optimise-then-discretise

We now move on to the optimise-then-discretise approach. This instead works by differentiating the idealised continuous-time model. Doing so produces a backwards-in-time differential equation, which is then solved numerically. (References include [Pon+62; Hag00; SH05; Che+18b] but this technique is widespread.)

Theorem 5.2. *Let $y_0 \in \mathbb{R}^d$ and $\theta \in \mathbb{R}^m$. Let $f_\theta \colon [0,T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t, uniformly Lipschitz in y, and continuously differentiable in y. Let $y \colon [0,T] \to \mathbb{R}^d$ be the unique solution to*

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

Let $L = L(y(T))$ be some (for simplicity scalar) function of the terminal value $y(T)$. Then $\frac{\mathrm{d}L}{\mathrm{d}y(t)} = a_y(t)$ and $\frac{\mathrm{d}L}{\mathrm{d}\theta} = a_\theta(0)$, where $a_y \colon [0,T] \to \mathbb{R}^d$ and $a_\theta \colon [0,T] \to \mathbb{R}^m$ solve the system of differential equations

$$\begin{aligned} a_y(T) &= \frac{\mathrm{d}L}{\mathrm{d}y(T)}, & \frac{\mathrm{d}a_y}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial y}(t, y(t)), \\ a_\theta(T) &= 0, & \frac{\mathrm{d}a_\theta}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial \theta}(t, y(t)). \end{aligned} \tag{5.1}$$

Remark 5.3. *The vector-matrix products in equation (5.1) are, more specifically, vector-Jacobian products. These can be computed efficiently via autodifferentiation; see Appendix A.1.*

Remark 5.4. *Note that the $a_y(t)$ in the second equation (rather than $a_\theta(t)$) is not a typographical error. The vector fields are independent of a_θ. This may be exploited to speed up backpropagation through neural ODEs; this is a topic we shall return to in Section 5.4.2.1.*

Equations (5.1) are known as *the continuous adjoint equations*.

These give a way to backpropagate through an ODE solve. Consider for example Figure 1.1. The gradient $\frac{\mathrm{d}L}{\mathrm{d}y(T)}$ is calculated by backpropagating through the softmax and affine layer in the usual way. Then the system of equation (5.1) is solved backwards in time from $t = T$ to $t = 0$, to compute the parameter gradients $\frac{\mathrm{d}L}{\mathrm{d}\theta} = a_\theta(0)$.[1]

[1] And had there also been any preceding operations, then backpropagation could then continue as usual from the computed $\frac{\mathrm{d}L}{\mathrm{d}\theta} = a_\theta(0)$, $\frac{\mathrm{d}L}{\mathrm{d}y_0} = a_y(0)$.

Note that equation (5.1) requires knowing the solution y as an input. The usual approach is to find this by augmenting equations (5.1) with the original neural ODE solved backwards-in-time, starting from the numerical approximation to $y(T)$ computed on the forward pass. In integral notation, this is

$$y(t) = y(T) + \int_T^t f_\theta(s, y(s))\,\mathrm{d}s. \tag{5.2}$$

This is known as the 'continuous adjoint method', or as the 'optimise-then-discretise' approach. The derivatives are calculated with respect to the idealised continuous-time model, and then the adjoint equations of (5.1) must themselves then be discretised.

Remark 5.5. *The continuous adjoint method is also commonly referred to as simply 'the adjoint method', especially in the modern neural differential equation literature.*

This is an unfortunate ambiguity of terminology. Across several prominent works, 'the adjoint method' has been used to refer to both the the discretise-then-optimise approach [GG06], and the optimise-then-discretise approach [Che+18b]. Moreover it has sometimes been used to refer to something else entirely: [HNW08, Definition 8.2] use it to refer to the inverse map of a numerical integration step, so that for example an implicit Euler step is the 'adjoint' of an explicit Euler step.[2]

In this text we strive to be unambiguous by always making clear which adjoint method we are referring to, and would strongly discourage simply writing 'the adjoint method' without further qualification.

5.1.2.1 Proof: 'continuous-time backpropagation'

Proving Theorem 5.2 is straightforward, and it is informative to compare this to backpropagation.

Consider two points $s, t \in [0, T]$ with $s < t$, and consider solving the ODE from s to t, and then from t to the terminal time T. Then by the chain rule,

$$\frac{\mathrm{d}L}{\mathrm{d}y(s)} = \frac{\mathrm{d}L}{\mathrm{d}y(t)}\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}.$$

For notation's sake let $a(t)^\top = \frac{\mathrm{d}L}{\mathrm{d}y(t)}$. Now the left hand side is independent of t, so

[2] This relationship between numerical integration steps is something we will actually need later. We refer to it as *analytic reversibility*, see Section 5.3.2.1.

differentiate with respect to t and set $t = s$:

$$\begin{aligned}
0 &= \frac{\mathrm{d}}{\mathrm{d}t}\left(a(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right)\Big|_{t=s} \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\left(a(t)^\top\right)\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\Big|_{t=s} + a(t)^\top \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right)\Big|_{t=s} \\
&= \frac{\mathrm{d}a}{\mathrm{d}s}(s) + a(t)^\top \frac{\mathrm{d}}{\mathrm{d}y(s)}\left(\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right)\Big|_{t=s} \\
&= \frac{\mathrm{d}a}{\mathrm{d}s}(s) + a(t)^\top \frac{\mathrm{d}}{\mathrm{d}y(s)}\left(f_\theta(t, y(t))\right)\Big|_{t=s} \\
&= \frac{\mathrm{d}a}{\mathrm{d}s}(s) + a(s)^\top \frac{\partial f_\theta}{\partial y}(s, y(s)).
\end{aligned}$$

This is now precisely the first adjoint equation of equation (5.1). The second adjoint equation can be derived by replacing y with the $[y, \theta]$ and f_θ with $[f., 0]$ (that is to say treating the parameters θ as additional state, subject to zero vector field), and applying the same argument as before.

In this way we see that the adjoint equations are essentially 'continuous time back-propagation'.[3]

5.1.2.2 Advantages

Memory efficiency The continuous adjoint method has one clear advantage: memory efficiency. The forward computations for y need not be stored, as y is recomputed on the backward pass. So whilst differentiating through the internal operations has memory cost $\mathcal{O}(HT)$, the continuous adjoint method has a memory cost of only $\mathcal{O}(H)$, independent of the time horizon T.

Ease of implementation Another advantage is a practical one: the differential equation solver need not be written using autodifferentiation software, as required for the discretise-then-optimise approach. The differential equation solver can instead be treated as a black box; for example the solver may have been written for some other purpose as part of some other software package.

5.1.2.3 Disadvantages

Computational cost The continuous adjoint method incurs additional computations necessary to recalculate $y(t)$ on the backward pass; this implies a slightly slower and more computationally expensive procedure.

[3]The mathematically precise reader may still be skeptical: we have not justified the change of limits, nor that the solution to the adjoint differential equation actually exists. See Appendix C.3.1 for these technical points.

Truncation errors The second disadvantage is numerical discretisation error: there will be a difference in the value computed for $y(t)$ on the forward pass (computed starting from the initial condition $y(0)$), and the value computed for $y(t)$ on the backward pass (computed starting from the numerical approximation to the terminal condition $y(T)$ obtained on the forward pass).

Furthermore and in addition to the recomputation of $y(t)$, the continuous adjoint equations for $a_y(t)$ and $a_\theta(t)$ must themselves be solved numerically, and in doing so incur some additional numerical error.

The result is that gradients calculated via the continuous adjoint method will not be as accurate as those computed by backpropagating through the solver. (Which are the gold standard, corresponding to the model actually used.) This means that training may be slower, final model performance may be impacted, and in the worst case training may fail altogether. [GKB19] give a description of the possible failure modes of the continuous adjoint method, and [OR20] perform a thorough empirical investigation comparing optimise-then-discretise against discretise-then-optimise, in favour the of the latter.

Example 5.6. *Consider solving the system* $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = \lambda y(t)$ *with a numerical ODE solver, where* $y(t) \in \mathbb{R}$*,* $y(0) = y_0$*, and suppose* $\lambda < 0$*. Most differential equation solvers – those with a nontrivial region of stability [HW02, Definition 2.1] – will handle this without trouble, as errors will decay exponentially. However when this is instead solved backwards-in-time from* $y(T)$*, as in equation* (5.2)*, then small errors are instead magnified exponentially. Moreover if* $\lambda > 0$ *then the same problem arises simply by interchanging the forward and backward passes in the above discussion.*

However ... Despite these dire warnings, continuous adjoint methods often (but not always) still work in practice, without needing any special care. The continuous adjoint method's suitability for any given problem is typically determined empirically – 'Does training seem to be working?' – and difficulties are frequently not a concern for practitioners.

5.1.2.4 Interpolated adjoints

It is possible to finesse the problem of numerical errors in the continuous adjoint method.

Record the values of $y(t)$ at specific locations on the forward pass (but not the internal operations of the solver used to obtain these $y(t)$). Interpolate these recorded values to form an approximation to $y(t)$ for all t. Then solve the continuous adjoint equations on the backward pass *without* additionally recomputing $y(t)$, by instead using the interpolated approximation at whatever values of t are required during the backward solve.

Interpolated adjoints are actually the default backpropagation method used in [SH05] and [Rac+20b], for example. [Kim+21a] report finding that optimise-then-discretise

adjoints failed on a stiff differential equation, but that both interpolated adjoints and discretise-then-optimise succeeded.

Stability and stiffness The use of interpolated adjoints implies that the differential equations solved – for $y(t)$ forward in time, and $a_y(t)$ backward in time – now exhibit very similar behaviour, in particular with respect to stability and stiffness.

The local behaviour of a differential equation is understood through the eigenvalues of the Jacobian of the vector field[4] [But16, Section 112], [HNW08, Section I.13, Equation (13.2)].

For $y(t)$, this is $\frac{\partial f_\theta}{\partial y}(t, y(t))$. Meanwhile for $a_y(t)$ and $a_\theta(t)$, this is

$$\frac{\partial}{\partial a_y}\left(-a_y(t) \cdot \frac{\partial f_\theta}{\partial y}(t, y(t))\right) = -\frac{\partial f_\theta}{\partial y}(t, y(t)),$$
$$\frac{\partial}{\partial a_\theta}\left(-a_y(t) \cdot \frac{\partial f_\theta}{\partial \theta}(t, y(t))\right) = 0.$$

The (local) behaviour of $a_\theta(t)$ is trivial as the vector field has zero Jacobian. Meanwhile recalling that the equation for $y(t)$ is solved forward-in-time and the equation for $a_y(t)$ is solved backward in time, we see that their Jacobians are identical.

Remark 5.7. *Morally speaking this is expected: differentiation obtains a local linear approximation – that is to say a Jacobian – to a function. In this case the function is the overall act of solving an ODE. Meanwhile, the present analysis on local behaviour of a system is about constructing a local linear approximation – a Jacobian – to the vector field.*

Note that this discussion is not true of standard optimise-then-discretise, for which the equation for y is solved in both the forward and backward directions, which exhibit opposite behaviour to each other.

Memory efficiency Recording the values of $y(t)$ incurs a small memory cost, but not as much as recording every internal operation of the solver as in the discretise-then-optimise approach.

Choice of interpolation There are several sensible ways to record and interpolate $y(t)$. [SH05] use cubic Hermite interpolation whilst [Dau+20] use barycentric Lagrange interpolation, in both cases recording $y(t_1), \ldots, y(t_n)$ for some prespecified values $t_1, \ldots, t_n$.

[4]An eigenvalue with positive real part describes a mode of a system that is 'locally expansive': points diverge from each other exponentially fast. Likewise negative real part describes a mode of a system that is 'locally contractive': points draw closer together exponentially fast. The imaginary part of an eigenvalue corresponds to the 'local rotation' of a system – see also Section 2.2.5.1.

5.1.2.5 Checkpointing

Another way to finesse the problem of numerical errors is to use checkpointing as in Section 5.1.1.3. Each time we hit a checkpoint recorded on the forward pass, then we effectively reset any accumulated truncation error in y acquired on the backward pass. The trade-off being that this increases the memory usage from $\mathcal{O}(H)$ to $\mathcal{O}(H + C)$, where C is the number of checkpoints. (Although in practice this is often a relatively modest amount.)

5.1.3 Reversible ODE solvers

Reversible ODE solvers offer a best-of-both-worlds approach compared to discretise-then-optimise and optimise-then-discretise. Reversible solvers offer both memory efficiency and accuracy of the computed gradients. Like optimise-then-discretise, they do require a small amount of extra computational work, to recompute the forward solution during backpropagation.

This is a topic still in its infancy. At present, two general reversible ODE solvers are known: the reversible Heun method, and the asynchronous leapfrog method (ALF). In addition, if the differential equation has the right structure, then symplectic solvers are typically also reversible.

The main drawback of reversible ODE solvers is that, at time of writing, all such solvers are low-order and exhibit poor stability properties. This is often not a problem if using pure-neural-network vector fields, but if the differential equation has known structure as in Section 2.2.2 then in principle this may result in a poor-quality solution.

We defer a full discussion of reversible solvers to sit alongside our discussion of other numerical solvers, in Section 5.3.2.

5.1.4 Forward sensitivity

Whilst not technically *back*propagation, for completeness we mention that it is also possible to compute forward sensitivities of an ODE. (Or indeed a CDE or SDE.)

Once again this can be done either in discretise-then-optimise fashion or in optimise-then-discretise fashion. Discretise-then-optimise is accomplished by computing the forward sensitivity of the solver's computation graph.

Optimise-then-discretise is accomplished by simply differentiating equation (5.3) with respect to y_0 and θ, through which we obtain the following theorem.

Theorem 5.8. *Let $y_0 \in \mathbb{R}^d$, $\theta \in \mathbb{R}^m$, $f_\theta \colon [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t, uniformly Lipschitz in y, and continuously differentiable in y. Let $y \colon [0, T] \to \mathbb{R}^d$ be the unique solution to*

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)). \tag{5.3}$$

Then $\frac{\mathrm{d}y(t)}{\mathrm{d}y_0} = J_y(t)$ *and* $\frac{\mathrm{d}y(t)}{\mathrm{d}\theta} = J_\theta(t)$*, where* $J_y\colon [0,T] \to \mathbb{R}^{d\times d}$ *and* $J_\theta\colon [0,T] \to \mathbb{R}^{d\times m}$ *solve the system of differential equations*

$$\begin{aligned} J_y(0) &= I_{d\times d}, & \frac{\mathrm{d}J_y}{\mathrm{d}t}(t) &= \frac{\partial f_\theta}{\partial y}(t, y(t))J_y(t), \\ J_\theta(0) &= 0, & \frac{\mathrm{d}J_\theta}{\mathrm{d}t}(t) &= \frac{\partial f_\theta}{\partial y}(t, y(t))J_\theta(t) + \frac{\partial f_\theta}{\partial \theta}(t, y(t)). \end{aligned}$$

The right hand side consists of Jacobian-vector products, which can be computed efficiently via autodifferentiation.

As usual, forward sensitivity is typically less efficient than reverse-mode autodifferentiation when considering machine learning problems with many parameters, so this approach is infrequently used. [Rac+20a] report finding it useful (only) on problems with very few (<100) parameters.

5.2 Backpropagation through CDEs and SDEs

We now study backpropagation through differential equations more generally.

5.2.1 Discretise-then-optimise

This is exactly the same as in the ODE case (Section 5.1.1) – simply differentiate through the internal operations of the controlled/stochastic differential equation solvers, typically by using solvers written in an autodifferentiable framework.

5.2.2 Optimise-then-discretise for CDEs

There are two approaches to constructing continuous adjoint methods for CDEs. One is to reduce the CDE to an ODE as in Chapter 3, and then applying the continuous adjoint method for ODEs. For example this is what is done in the `torchcde` library [Kid20].

Alternatively a backwards-in-time CDE may be constructed, and then numerically solved in whatever manner is desired, by reduction to an ODE or otherwise. The corresponding theorem is as follows.

Theorem 5.9. *Let* $f\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y\times d_x}$ *be both Lipschitz and continuously differentiable. Let* $x\colon [0,T] \to \mathbb{R}^{d_x}$ *be continuous and of bounded variation. Let* $L\colon \mathbb{R}^{d_y} \to \mathbb{R}$ *be differentiable (and scalar just for simplicity). Let* $y_0 \in \mathbb{R}^{d_y}$ *and let* $y\colon [0,T] \to \mathbb{R}^{d_y}$ *solve*

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s). \tag{5.4}$$

Then the adjoint process $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)}$ *satisfies the backwards-in-time linear CDE*

$$a(t) = a(T) + \int_T^t -a(s)^\top \frac{\partial f}{\partial y}(y(s))\,\mathrm{d}x(s), \tag{5.5}$$

starting from the terminal condition $a(T) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(T)}$*, and where the right hand side denotes a vector-Jacobian product.*

For simplicity we have avoided explicitly encoding the dependence on the parameterisation θ. This case may be recovered by replacing y, y_0, f_θ with $[y, \theta]$, $[y_0, \theta]$, and $f(y, \theta) = [f_\theta(y), 0]$.

After having solved equation (5.5) backward-in-time, $a(0) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(0)}$ are the desired gradients. As with the ODE case, y need not be recorded on the forward pass and may instead by recomputed on the backward pass, by stacking (5.4) and (5.5) together and solving as a joint system backwards-in-time.

See Appendix C.3.2 for a proof.

5.2.3 Optimise-then-discretise for SDEs

(Informal) Theorem 5.10. *Let* $\mu\colon \mathbb{R} \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y}$ *and* $\sigma\colon \mathbb{R} \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_w}$ *be sufficiently regular. Let* $L\colon \mathbb{R}^{d_y} \to \mathbb{R}$ *be differentiable (and scalar just for simplicity). Let* $y_0 \in \mathbb{R}^{d_y}$ *and let* $y\colon [0, T] \to \mathbb{R}^{d_y}$ *solve the Stratonovich SDE*

$$y(0) = y_0, \qquad \mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t). \tag{5.6}$$

Then the adjoint process $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)} \in \mathbb{R}^{d_y}$ *is a (strong) solution to the backwards-in-time linear Stratonovich SDE*

$$\mathrm{d}a_{k_1}(t) = -a_{k_2}(t)\frac{\partial \mu_{k_2}}{\partial y_{k_1}}(t, y(t))\,\mathrm{d}t - a_{k_2}(t)\frac{\partial \sigma_{k_2,k_3}}{\partial y_{k_1}}(t, y(t)) \circ \mathrm{d}w_{k_3}(t), \tag{5.7}$$

using Einstein notation over the indices k_1, k_2, k_3*, starting from the terminal condition* $a(T) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(T)}$*.*

In particular w *is the same Brownian motion as used in the forward pass.*

As in the previous subsection, we have avoided explicitly encoding the dependence on the parameterisation θ, the desired gradients are the computed value $a(0)$, and (5.6)–(5.7) may be stacked together to recover $y(t)$ during the backpropagation. The right hand side of (5.7) consists of vector-Jacobian products, which may be calculated using autodifferentiation.

Note that the nondifferentiability of Brownian sample paths is unrelated to being able to compute derivatives $\frac{\mathrm{d}L(y(T))}{\mathrm{d}y(0)}$. (It just means that derivatives like $\frac{\mathrm{d}y}{\mathrm{d}t}$ do not exist.)

See Appendix C.3.3 for a precise statement and a proof of Theorem 5.10 via rough path theory.

Rough path theory Theorem 5.10 is stated informally, because putting a precise meaning on the solution of (5.7) is a little outside the usual framework for SDEs. In particular (5.7) fails to exhibit measurability with respect to the natural filtration of w.

Rough path theory provides an elegant (and intuitive) solution, by allowing solutions to (5.6) and (5.7) to be defined *pathwise*. We simply fix a single sample of w, and evaluate the forward pass via (5.6) and the backward pass via (5.7) – in every respect just like the ODE case.

Remark 5.11. *When backpropagating through an SDE solve via discretise-then-optimise, then a Brownian motion is sampled on the forward pass of the numerical solver, its random samples are fixed as part of the computation graph, and then this computation graph is backpropagated. (Indeed just like any neural generative model; the noise sampled on the forward pass is the same noise on the backward pass.) As such 'discretise-then-optimise' is somehow intrinsically also pathwise.*

Remark 5.12. *Note the use of Stratonovich integration. This is naturally 'time reversible', unlike Itô integration. If* (5.6) *was an Itô SDE then the equivalent of* (5.7) *is substantially more thorny to work with: it would be derived by applying the Itô-Stratonovich correction term to convert* (5.6) *into a Stratonovich integral, applying Theorem 5.10, and then applying the Stratonovich-Itô correction term to* (5.7)*.*

In a practical implementation then this double-correction implies substantial computational overhead, so it is preferable to use Stratonovich SDEs instead of Itô SDEs when training via optimise-then-discretise methods.

5.2.4 Reversible differential equation solvers

CDEs may be reduced to ODEs as discussed in Chapter 3, and correspondingly any reversible ODE solver may be applied. Meanwhile SDEs have a single known reversible solver, namely the reversible Heun method. See Section 5.3.2.

5.3 Numerical solvers

5.3.1 Off-the-shelf numerical solvers

Neural networks represent unstructured vector fields. This means that many of the more specialised differential equation solvers (developed for any particular equation) do not apply, and we must rely on 'general' solvers.

There is a rich literature of such numerical differential equation solvers. We will largely focus on explicit Runge–Kutta solvers, in particular for ODEs and CDEs, which are a popular family of numerical solvers. Other reasonable choices exist – for

example linear multistep methods – but it is not our purpose to restate the numerical differential equation literature.

5.3.1.1 General principles

There are some principles, specific to neural differential equations over differential equations in general, that help guide the choice of numerical solver.

Implicit solvers Implicit solvers, for example the implicit Euler method $y_{j+1} = y_j + \Delta t f(t_{j+1}, y_{j+1})$, are rarely used.

They are computationally expensive: implicit solvers solve a linear or nonlinear system at every step, often through a fixed point iteration. Neural differential equations are a regime in which the vector field evaluations are expensive, and many vector field evaluations are already being made (over a batch, and over the course of training). Reducing computational cost is of substantial interest.

A major use-case for implicit solvers is solving stiff differential equations.[5] However stiffness is often not a problem for neural differential equations – if stiffness and an explicit solver produce a poor solution, then the loss between model and data may be large. As this is the criteria we explicitly train to avoid (to achieve a small loss), then the issue is avoided.

Remark 5.13. *The above description is typical for 'machine learning neural differential equations' such as CNFs or neural CDEs – Section 2.2.3 and Chapter 3 respectively – but it is not a universal rule. For example if the vector field incorporates known structure (Section 2.2.2), or the data has multiple different timescales, then stiffness may be unavoidable and an implicit solver may become a reasonable choice. See for example [Kim+21a].*

Adaptive versus fixed step solvers Both fixed step size and adaptive step size solvers are often reasonable choices for neural differential equations.

Given a time horizon $T > 0$, then fixed step size solvers choose some step locations $0 = t_0 < t_1 < \cdots < t_n = T$ in advance, usually with $\Delta t = t_{j+1} - t_j$ independent of j.

Adaptive step solvers vary the size of the next step $t_{j+1} - t_j$, so that the (local) error made during the solve is approximately equal to some tolerance. For example embedded Runge–Kutta methods are of this type. This implies a variable computational cost, typically increasing over the course of training as model complexity increases [Che+18b, Figure 3(d)], [Fin+20a, Figure 3(c)].[6]

[5]Somewhat tautologically, as stiff differential equations are broadly categorised as 'equations for which explicit solvers fail'.

[6]Loosely speaking, neural networks tend to increase in complexity over the course of training [Kal+19], [JGH18, Section 5]. This manifests as the training and validation losses following the classic bias-variance curves during training.

Example 5.14. *Consider solving a neural CDE with densely-sampled and slowly-varying time series data as input. The slow variation of the input data means that processing every piece of it – as we may do with an RNN – is likely overkill. The adaptivity of a solver may automatically detect the slow timescale at which the differential equation is driven, and produce integration steps of the appropriate size, larger than the discretisation of the data.*

Moreover, RNN training often breaks down as the length of a time series increases. If this length has been achieved by sampling the same signal more and more densely, then it is a bit perverse that this extra information should cause our model to fail to train. Philosophically speaking, it is reassuring to be able to overcome this issue using adaptive solvers.

Baked-in discretisations If only a single solver (in particular a low-order solver or a solver with a fixed step size) is used during model training, then this choice of discretisation may become an intrinsic part of the model. The neural vector fields will have been trained to work best at this discretisation, and may fail with other discretisations [Ott+21; Que+21].

For many applications this need not be a problem. In the introduction to this thesis (Section 1.2), 'inspiration for a discretised model' was described as a good use case for neural differential equations, and baking in the numerical discretisation is simply a subtle instance of this.

Step size and error tolerance Step size (for fixed step size solvers) or error tolerance (for adaptive step size solvers) will often be very large compared to that seen in the usual numerical differential equation literature. For example when using a neural SDE to generate a time series sampled at some points $t_0 < \cdots < t_n$, then we may elect to take a single numerical step over each interval $[t_j, t_{j+1}]$, even when $t_{j+1} - t_j$ is relatively large.

Once again this may be thought of as treating the continuous model as an ideal, and then deliberately fitting a discretised model. Large step sizes are often motivated by a desire to reduce computational cost, and thus training time.

Example 5.15. *In this 'large step size regime', do note that taking smaller steps may produce slightly more expressive models. When looking to increase or decrease the modelling capacity of a neural differential equation, both step sizes and vector field complexity are options that may be adjusted.*

As an example, consider fitting a neural SDE as in Chapter 4. If taking a single numerical step using the Euler–Maruyama method, then the conditional distribution of $y(t_{j+1})|y(t_j)$ will only be Gaussian, which is relatively inexpressive.

Order of solver Low-order solvers are often reasonable choices, especially when explicitly baking in the discretisation in the large step size regime. For example

Euler's method is first-order convergent; midpoint or Heun's method are second-order convergent.

If aiming to fit the idealised continuous model (for example when training via optimise-then-discretise), then higher-order solvers such as Dormand–Prince are often preferred. (At least when they are available, that is to say for ODEs and CDEs but not SDEs.)

5.3.1.2 ODEs and CDEs

Bringing the above points together: when solving ODEs, or CDEs reduced to ODEs, then standard low-order solvers are the explicit Euler method (first order), the midpoint method (second order), or Heun's method (second order). Standard higher-order[7] methods are RK4 (fourth order), Dormand–Prince (fifth order), or Tsit5 (fifth order).

[Rac21a] offer an extensive comparison of explicit Runge–Kutta methods.

5.3.1.3 SDEs

Standard choices of numerical solver for neural SDEs include the Euler–Maruyama method or Heun's method, for Itô or Stratonovich SDEs respectively.[8] If the problem has commutative noise[9] then Milstein's method may be applied for both Itô and Stratonovich SDEs.

5.3.2 Reversible solvers

We indicated in Sections 5.1 and 5.2 that besides discretise-then-optimise and optimise-then-discretise, there is a third option, namely reversible solvers.

Consider a differential equation solver iteratively computing $(y_j, \alpha_j) \mapsto (y_{j+1}, \alpha_{j+1})$, where $\{y_j\}_{j=0}^n$ is the numerical approximation to the solution of some differential equation – whether it be an ODE, CDE, or SDE – and α_j denotes any extra state that the differential equation solver wishes to keep around.

By definition, (y_{j+1}, α_{j+1}) may be computed from (y_j, α_j). We say that a solver is *reversible* if (y_j, α_j) may be computed from (y_{j+1}, α_{j+1}). (Note that we have not yet made precise what is meant by 'computed'.)

[7]Relatively speaking.

[8]Broadly speaking SDEs solvers are distinguished by whether they converge to the Itô or Stratonovich solution.

[9]This is a condition that is satisfied in several common special cases: if the Brownian motion is scalar valued; if the diffusion matrix is independent of the SDE solution; if the diffusion matrix is diagonal.

Algorithm 1: Backward pass through a reversible solver.
t denotes time, y denotes the numerical solution, α denotes any additional state the solver keeps around, and Δt denotes a step size. x denotes a possible control input, which may be unused if the equation is an ODE and may be a Brownian motion if the equation is an SDE.

Input: $t_{j+1}, y_{j+1}, \alpha_{j+1}, \Delta t, x, \frac{\partial L(y_n)}{\partial y_{j+1}}, \frac{\partial L(y_n)}{\partial \alpha_{j+1}}$

$y_j, \alpha_j = \text{Reverse}(t_{j+1}, y_{j+1}, \alpha_{j+1}, \Delta t, x)$

$y_{j+1}, \alpha_{j+1} = \text{Forward}(t_j, y_j, \alpha_j, \Delta t, x)$

$\frac{\partial L(y_n)}{\partial (y_j, \alpha_j)} = \frac{\partial L(y_n)}{\partial (y_{j+1}, \alpha_{j+1})} \cdot \frac{\partial (y_{j+1}, \alpha_{j+1})}{\partial (y_j, \alpha_j)}$ # vector-Jacobian product

Output: $t_j, y_j, \alpha_j, \frac{\partial L(y_n)}{\partial y_j}, \frac{\partial L(y_n)}{\partial \alpha_j}$

Backpropagation with reversible solvers Backpropagation through a reversible solver is shown in Algorithm 1. The 'local' forward is needed to construct a computational graph, through which the vector-Jacobian product is calculated.

Computational cost Assuming that a forward and reverse step cost the same, the total computational cost of evaluating and backpropagating through a step of a reversible solver is three forward operations and one backward operation: one forward operation on the forward pass, two forward operations on the backward pass, and a single backward operation on the backward pass. As a combined forward/backward operation costs at most four forward operations [GW08, Equation (4.21)], then the overall computational cost is approximately that of six forward operations.

This cost should be contrasted with discretise-then-optimise and optimise-then-discretise. Discretise-then-optimise involves simply a forward operation on the forward pass, and a backward operation on the backward pass, for an overall computational cost of approximately four forward operations. Optimise-then-discretise involves a forward operation on the forward pass, a single forward operation on the backward pass, and a single backward operation on the backward pass, for an overall computational cost of approximately five forward operations.

Remark 5.16. *It is sometimes possible to elide the local forward in Algorithm 1. Given suitable structure in the solver, it may be possible to reuse the computational graph of the reverse step, and in doing so save the cost of a single forward operation. See for example the asynchronous leapfrog method coming up in Section 5.3.2.3.*

Precise gradients As the same numerical solution $\{y_j\}_{j=0}^n$ is recovered on both the forward and backward passes, the computed gradients are precisely the discretise-then-optimise gradients of the numerical discretisation of the forward pass.

As it is the discretised model that is fit to data, discretise-then-optimise represents the gold standard – the 'true' gradients of the model – so this property is desirable.

5.3.2.1 Analytic and algebraic reversibility

We now make precise what it means to 'compute' (y_n, α_n) from (y_{n+1}, α_{n+1}).

Analytic reversibility In some sense, essentially every solver is reversible. For example, the explicit Euler method

$$y_{j+1} = y_j + f(y_j)\Delta t$$

may be reversed via the (backwards-in-time) implicit Euler method

$$y_j = y_{j+1} - f(y_j)\Delta t,$$

for those Δt small enough that the contraction mapping theorem ensures this non-linear equation has a unique solution.

Unfortunately this requires solving a fixed-point iteration, and computing y_j from y_{j+1} is both approximate and computationally expensive. We refer to reversibility of this type as *analytic reversibility*.

Example 5.17. *For example [Beh+19] consider residual networks as the explicit Euler discretisation of a neural ODE, and exactly as above, use the implicit Euler method to invert the operation of each layer during backpropagation.*

Algebraic reversibility Substantially more preferable is what we shall refer to as *algebraic reversibility*. These are solvers for which the solution of (y_j, α_j) may be written as a closed-form expression with respect to (y_{j+1}, α_{j+1}). As such they are much more computationally reasonable.

We additionally refer to an algebraically reversible solver as *symmetric* if the reverse computation is of the same form as the forward step, simply performed backward-in-time. (If '$(y_{j+1}, \alpha_{j+1}) = \text{step}(y_j, \alpha_j, \Delta t)$' implies '$(y_j, \alpha_j) = \text{step}(y_{j+1}, \alpha_{j+1}, -\Delta t)$'.)

5.3.2.2 Reversible Heun method

The reversible Heun method, introduced in [Kid+21a], is a symmetric algebraically reversible ODE, CDE, or SDE solver. We will consider solving the SDE

$$y(0) = y_0, \quad \mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t),$$

over $[0, T]$, for which we will obtain the numerical solution $\{y_j\}_{j=0}^n$. If solving an ODE simply set $\sigma = 0$. If solving a CDE then either reduce it to an ODE, or *mutatis mutandis* replace w with some control x.

To the best of this authors' knowledge, and at time of writing, this is the first and only general (non-symplectic) algebraically reversible SDE solver.

Algorithm 2: Forward pass for the reversible Heun method.

Input: $t_j, y_j, \widehat{y}_j, \mu_j, \sigma_j, \Delta t, w$

$$
\begin{aligned}
t_{j+1} &= t_j + \Delta t \\
\Delta w_j &= w(t_{j+1}) - w(t_j) \\
\widehat{y}_{j+1} &= 2y_j - \widehat{y}_j + \mu_j \Delta t + \sigma_j \Delta w_j \\
\mu_{j+1} &= \mu(t_{j+1}, \widehat{y}_{j+1}) \\
\sigma_{j+1} &= \sigma(t_{j+1}, \widehat{y}_{j+1}) \\
y_{j+1} &= y_j + \frac{1}{2}(\mu_j + \mu_{j+1})\Delta t \\
&\quad + \frac{1}{2}(\sigma_j + \sigma_{j+1})\Delta w_j
\end{aligned}
$$

Output: $t_{j+1}, y_{j+1}, \widehat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1}$

Algorithm 3: Reverse pass for the reversible Heun method.

Input: $t_{j+1}, y_{j+1}, \widehat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1}, \Delta t, w$

$$
\begin{aligned}
t_j &= t_{j+1} - \Delta t \\
\Delta w_j &= w(t_{j+1}) - w(t_j) \\
\widehat{y}_j &= 2y_{j+1} - \widehat{y}_{j+1} - \mu_{j+1}\Delta t - \sigma_{j+1}\Delta w_j \\
\mu_j &= \mu(t_j, \widehat{y}_j) \\
\sigma_j &= \sigma(t_j, \widehat{y}_j) \\
y_j &= y_{j+1} - \frac{1}{2}(\mu_{j+1} + \mu_j)\Delta t \\
&\quad - \frac{1}{2}(\sigma_{j+1} + \sigma_j)\Delta w_j
\end{aligned}
$$

Output: $t_j, y_j, \widehat{y}_j, \mu_j, \sigma_j$

Initialisation The solver tracks several extra pieces of state, $\widehat{y}_j$, μ_j, σ_j, which are initialised at $\widehat{y}_0 = y_0$, $\mu_0 = \mu(0, y_0)$, $\sigma_0 = \sigma(0, y_0)$, and which have solution-like, drift-like and diffusion-like interpretations respectively. We additionally take w to be a single sample of Brownian motion, which must be the same on both the forward and backward passes.

Stepping The forward iteration then proceeds by iterating Algorithm 2. Note the similarity to Heun's method.

Convergence Convergence results are as follows.

Theorem 5.18. *The reversible Heun method, when applied to ODEs, is a second-order method.*

Theorem 5.19. *The reversible Heun method, when applied to SDEs, exhibits strong convergence of order $\frac{1}{2}$. If the noise is additive then this increases to order 1.*

A proof of the ODE case is given in Appendix C.4. A proof of the SDE case is given in [Kid+21a, Appendix D].

Stability One drawback of the reversible Heun method is its unimpressive stability properties.

Theorem 5.20. *The region of stability for the reversible Heun method (for ODEs) is the complex interval $[-i, i]$.*

A proof is given in Appendix C.4.

Adaptive step sizing The step size Δt may either be fixed in advance (to make this a fixed step size solver) or it may be adapted over the course of the integration.

When solving ODEs, then the reversible Heun method may be treated in the same way as embedded Runge–Kutta method by returning the error estimate $y_{\text{error}} = (\mu_{j+1} - \mu_j)\Delta t/2$. This may now be used to adapt step sizing in the usual way (Section 5.4.2).

Reversibility For completeness, the reverse pass through the reversible Heun method is given in Algorithm 3. Note the similarity to the reversible Heun method, as the reversible Heun method is not just algebraically reversible but also symmetric.

Use cases If training via discretise-then-optimise is not an option, then the reversible Heun is an excellent choice of solver for any of neural ODEs, CDEs, or SDEs. If using pure-neural-network vector fields then its low order and lack of stability need not always be a concern, especially if the discretisation is baked-in as in Section 5.3.1.1.

It is only not recommended if solving neural differential equations with built-in structure as in Section 2.2.2.1, for which the low order and lack of stability may be concerns.

Computational efficiency Unlike the traditional Heun method, the reversible Heun method makes only a single evaluation per step. This can mean that it is more computationally efficient.

5.3.2.3 Asynchronous leapfrog method

The asynchronous leapfrog method is a symmetric algebraically reversible ODE/CDE (but not SDE) solver, introduced in [Mut13] and popularised by [Zhu+21]. We will consider solving the ODE

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)),$$

over $[0, T]$, for which we will obtain the numerical solution $\{y_j\}_{j=0}^n$.

Initialisation The solver tracks a single extra piece of extra state v_j in addition to the numerical solution y_j. This extra state has a velocity-like interpretation and is initialised as $v_0 = f(y_0)$.

Stepping The stepping procedure is then given by iterating Algorithm 4. Note the similarity to the midpoint method.

Algorithm 4: Forward pass through the asynchronous leapfrog method.

Input: $t_j, y_j, v_j, \Delta t$

$$\begin{aligned}\widehat{t}_j &= t_j + \Delta t/2\\ \widehat{y}_j &= y_j + v_j \Delta t/2\\ \widehat{v}_j &= f(\widehat{t}_j, \widehat{y}_j)\end{aligned}$$

$$\begin{aligned}t_{j+1} &= t_j + \Delta t\\ y_{j+1} &= y_j + \widehat{v}_j \Delta t\\ v_{j+1} &= 2\widehat{v}_j - v_j\end{aligned}$$

Output: $t_{j+1}, y_{j+1}, v_{j+1}$

Convergence The following convergence result may be shown.

Theorem 5.21. *The asynchronous leapfrog method is a second-order method. Specifically, the local truncation error in y is $\mathcal{O}(\Delta t^3)$, whilst the local truncation error in v is $\mathcal{O}(\Delta t^2)$.*

See [Zhu+21, Theorem 3.1].

Stability As with the reversible Heun method, one drawback of the asynchronous leapfrog method are its unimpressive stability properties.

Theorem 5.22. *The region of stability for the asynchronous leapfrog method is the complex interval $[-i, i]$.*

A proof is given in [Zhu+21, Appendix A.4].

Adaptive step sizing If adaptively setting the step size, then the error estimate $y_{\text{error}} = (v_{j+1} - v_j)/2$ may be used (in the same way as an embedded Runge–Kutta method).

Efficient reverse pass The general reversibility algorithm given in Algorithm 1 involves both Reverse and Forward operations. For the asynchronous leapfrog method, the extra Forward operation may be elided. This is accomplished by instead differentiating the Reverse pass, and appropriately adjusting the surrounding calculation.

This is because the quantity $f(\widehat{t}_j, \widehat{y}_j)$ is computed during both Reverse and Forward. As f is generally both complicated and user-supplied, this is the piece we are most interested in autodifferentiating. We may calculate by hand the appropriate derivatives for the surrounding structure of the algorithm.

Algorithm 5: Efficient backward pass through the asynchronous leapfrog method.

Input: $t_{j+1}, y_{j+1}, \mu_{j+1}, \Delta t, \frac{\partial L(y_n)}{\partial y_{j+1}}, \frac{\partial L(y_n)}{\partial \mu_{j+1}}$

$$\begin{aligned}\widehat{t}_j &= t_{j+1} - \Delta t/2\\ \widehat{y}_j &= y_{j+1} - \mu_{j+1}\Delta t/2\\ \widehat{\mu}_j &= \mu(\widehat{t}_j, \widehat{y}_j)\end{aligned}$$

$$\begin{aligned}t_j &= t_{j+1} - \Delta t\\ y_j &= y_{j+1} - \widehat{\mu}_j \Delta t\\ \mu_j &= 2\widehat{\mu}_j - \mu_{j+1}\end{aligned}$$

$$\begin{aligned}\frac{\partial L(y_n)}{\partial y_j} &= \left(2\frac{\partial L(y_n)}{\partial \mu_{j+1}} + \frac{\partial L(y_n)}{\partial z_{j+1}}\Delta t\right)^\top \frac{\partial \widehat{\mu}_j}{\partial \widehat{z}_j} + \frac{\partial L(y_n)}{\partial z_{j+1}}\\ \frac{\partial L(y_n)}{\partial \mu_j} &= \frac{1}{2}\frac{\partial L(y_n)}{\partial z_j}\Delta t - \frac{\partial L(y_n)}{\partial \mu_{j+1}}\end{aligned}$$

Output: $t_j, y_j, \mu_j, \frac{\partial L(y_n)}{\partial y_j}, \frac{\partial L(Z_T)}{\partial \mu_j}$

In doing so, we obtain Algorithm 5.

Use cases The asynchronous leapfrog method is useful in essentially the same cases as the reversible Heun method, except that it applies only for ODEs.

Remark 5.23. *The asynchronous leapfrog method does not seem to extend to SDEs. There is a clearly analogous procedure, tracking a diffusion-like quantity in addition to a drift-like quantity. However it does not seem obviously possible to demonstrate theoretical convergence of this solver, and neural SDEs trained using it perform relatively poorly empirically.*

5.3.2.4 Symplectic solvers

Many pre-existing symplectic solvers are already algebraically reversible. There are a great many symplectic solvers; we highlight only a few interesting ones here.

Semi-implicit Euler method Given a pair of differential equations

$$\begin{aligned} y(0) &= y_0, & \frac{\mathrm{d}y}{\mathrm{d}t}(t) &= f(t, v(t)), \\ v(0) &= v_0, & \frac{\mathrm{d}v}{\mathrm{d}t}(t) &= g(t, y(t)), \end{aligned}$$

the semi-implicit Euler method is defined by

$$\begin{aligned} y_{j+1} &= y_j + f(t_j, v_{j+1})\Delta t, \\ v_{j+1} &= v_j + g(t_j, y_j)\Delta t. \end{aligned}$$

A common special case is $f(t, v) = v$ so that v is the velocity of y, and y is understood as the solution of a second-order system.

This is notable for its popularity in deep learning papers; it is the solver used with the rotational vector fields and momentum residual networks of Section 2.2.5.

Leapfrog/midpoint Consider the integrator

$$\begin{aligned} y_{j+1} &= y_{j-1} + f(t_j, y_j)\Delta t, \\ y_{j+2} &= y_j + f(t_{j+1}, y_{j+1})\Delta t \end{aligned}$$

for solving

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)),$$

over $[0, T]$, where $\{y_j\}_{j=0}^n$ is the numerical solution.

We refer to this as the 'leapfrog/midpoint integrator' in accordance with the title of [Sha09], but other texts will call it simply 'leapfrog' (in ambiguity with the integrator for second order systems of the same name), or the 'explicit midpoint method' (in ambiguity with the Runge–Kutta method of the same name).

This is both algebraically reversible and symmetric, and is applicable to general first-order systems.

Remark 5.24. *As a linear multi-step method it does not admit an immediate way to adapt step sizes during integration, nor does it have very good stability properties [Sha09]. Fixing the first problem produces the asynchronous leapfrog integrator of Section 5.3.2.3 [Mut13].*

5.3.3 Solving vector fields with jumps

One common scenario is that the vector field of a neural ODE has a piecewise structure with respect to time. That is we are solving

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)),$$

where

$$f_\theta(t, y) = \begin{cases} f_{\theta,j}(t, y) & t \in [t_0, t_1] \\ \vdots & \\ f_{\theta,n}(t, y) & t \in [t_{n-1}, t_n] \end{cases}.$$

For example this occurs when solving a stack of neural ODEs as in Section 2.3.2.1, or when solving a neural CDE, reduced to an ODE, using linear or rectilinear interpolation (Section 3.5).

In this case we have two options: make n separate calls to an ODE solver, over each $[t_j, t_{j+1}]$, or to make a single call to an ODE solver, over the whole $[t_0, t_n]$.

Both options are fine, but in some cases each requires a small amount of caution.

Separate calls If making n separate calls to an ODE solver, and training the neural ODE either via optimise-then-discretise or via reversible ODE solvers (Section 5.3.2), then practically speaking the memory cost will be n times larger than if we had made a single ODE solve: each ODE solve will store $y(t_{j+1})$ at the end of its solve, for the sake of the later backpropagation through the ODE solve.

This may be desirable – essentially implementing checkpointing as in Section 5.1.2.5. Alternatively it may be undesirable due to the increased memory cost.

Single call If making a single call to the ODE solver, and using an adaptive step size ODE solver, then the solver should be informed about the location of the jumps. Otherwise, the error control in the step size controller will detect a large error every time a threshold is crossed, slow down to resolve it, and then speed back up again.

It is substantially more efficient to simply step directly to the discontinuity; failing to do so can result in an order-of-magnitude slow-down. The software libraries we recommend in Section 5.6 support this as an option.

5.3.4 Hypersolvers

Known versus unknown structure When motivating the use of standard off-the-shelf solvers like Euler or Dormand–Prince, we wrote:

Neural networks represent unstructured vector fields. This means that many of the more specialised differential equation solvers (developed for any particular equation) do not apply, and we must rely on 'general' solvers.

This was, in fact, a white lie.

Neural differential equations *do* exhibit structure – they exhibit whatever the structure of the problem being modelled is. The problem is simply that this structure is specified by a black-box neural vector field, and is not understood.

A running theme throughout machine learning, and thus also this work, has been that we may substitute theoretical understanding with data – and that given sufficient data, we may close the gap between a theoretical model and the behaviour observed in practice. We may apply the same principle here.

Learnt error corrections For $q \in \mathbb{N}$, consider some q-th order ODE solver[10] with update rule ψ. That is to say, for some time points $\{t_j\}_{j=0}^n$ (for simplicity with constant step size $\Delta t = t_{j+1} - t_j$), the numerical solution $y_j \approx y(t_j)$ is obtained by iterating

$$y_{j+1} = y_j + \psi(t_j, y_j)\Delta t.$$

For example $\psi(t, y) = f_\theta(t, y)$ for Euler's method.

As a q-th order solver, the local truncation error is of order $q + 1$:

$$\|y(t_{j+1}) - y(t_j) - \psi(t_j, y(t_j))\Delta t\| = \mathcal{O}(\Delta t^{q+1}).$$

A *hypersolver* [Pol+20] is now defined by a learnt correction

$$y_{j+1} = y_j + \psi(t_j, y_j)\Delta t + g_\omega(t_j, y_j, \Delta t)\Delta t^{q+1} \tag{5.8}$$

with $g_\omega \colon \mathbb{R} \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ some neural network depending on learnt parameters ω.

Example 5.25. *For example, recall Heun's method*

$$\begin{aligned}\widehat{y}_{j+1} &= y_j + f_\theta(t_j, y_j)\Delta t \\ y_{j+1} &= y_j + \frac{1}{2}(f_\theta(t_j, y_j) + f_\theta(t_{j+1}, \widehat{y}_{j+1}))\Delta t.\end{aligned}$$

Then HyperHeun is defined by

$$\begin{aligned}\widehat{y}_{j+1} &= y_j + f_\theta(t_j, y_j)\Delta t \\ y_{j+1} &= y_j + \frac{1}{2}(f_\theta(t_j, y_j) + f_\theta(t_{j+1}, \widehat{y}_{j+1}))\Delta t + g_\omega(t_j, y_j, \Delta t)\Delta t^3.\end{aligned}$$

(This implicitly features a $\mathcal{O}(\Delta t^2)$ term due to the $\widehat{y}_{j+1}$.)

Training Training is performed by assuming access to the true solution of the neural ODE. In practice this may be approximately obtained by using a traditional numerical solver with high order, small step sizes, or tight error tolerances.

Let

$$R(t, y_{\text{prev}}, y_{\text{next}}) = \frac{1}{\Delta t^{q+1}}(y_{\text{next}} - y_{\text{prev}} - \psi(t, y_{\text{prev}})\Delta t).$$

Then a hypersolver may then trained by minimising either

$$\frac{1}{n}\sum_{j=0}^{n-1} \|R(t_j, y(t_j), y(t_{j+1})) - g_\omega(t_j, y(t_j), \Delta t)\| \tag{5.9}$$

[10]We will treat only ODEs; the extensions to CDEs and SDEs are natural but so far unexplored.

or

$$\frac{1}{n}\sum_{j=0}^{n-1}\left\|R(t_j, y_j, y(t_{j+1})) - g_\omega(t_j, y_j, \Delta t)\right\|, \tag{5.10}$$

where $\{y_j\}_{j=1}^n$ is the numerical solution obtained by iterating (5.8). (The former is analogous to training an RNN using teacher forcing; the latter to training an RNN without it.)

Applications The primary interest in hypersolvers is to obtain solutions that are both fast and accurate. As speed is of interest, then typically the base update rule ψ is very simple (Euler or Heun), whilst g_ω may only be a single-layer MLP. This is often sufficient to obtain excellent results. For example [Pol+20] report striking results in which 80 Dormand–Prince steps may be replaced by only 2 HyperHeun steps, without degrading accuracy (in that example, on a continuous normalising flow).

However, hypersolvers are generally only useful for speeding up inference, not training. The neural ODE changes during training, and so (5.9)–(5.10) become a moving target.

There is related work on training neural networks as differential equation solvers [QWX19]. This is a related but distinct notion to training a neural network as the solution of the differential equation itself [LLF97b; LLF97a; HJE18; MQH18; Rai18; PSW19; RPK19; Fan+19].

5.4 Tips and tricks

5.4.1 Regularisation

5.4.1.1 Weight decay

Adding weight decay to the parameters of a neural vector field may help to improve model performance, just as in traditional deep learning.

For many neural networks, the scale of its output is roughly proportional to the scale of its weights. That is to say, $\|f_\theta\| \,/\, \|\theta\|$ may be approximately constant over different values of θ. As such an additional implication of weight decay is that the vector field may be closer to zero, and thus numerically easier to integrate.

5.4.1.2 Temporal regularisation

For applications of neural differential equations to 'non time series' problems, one form of regularisation is to select the region of integration randomly. For example when training a continuous normalising flow, then instead of taking a fixed a region of integration $[\tau, T]$, the endpoints τ, T may be sampled from some distribution;

perhaps $\tau = 0$ remains fixed whilst $T \sim \text{Uniform}[0.9, 1.1]$. This is computationally cheap whilst encouraging models which are robust to small perturbations [Gho+20].

5.4.1.3 Additive noise

Mainstream deep learning often uses stochasticity as a regulariser, such as dropout. Correspondingly, and for neural ODEs and neural CDEs specifically, then including some small additive noise after each step (so that the model becomes an SDE) is another computationally cheap option that encourages more robust models [OVV20; Cra21b].

In this case the added noise should be fixed. If it is learnt then the training process will shrink it to zero and no regularisation will be applied.

5.4.1.4 Regularising higher-order derivatives

Consider the usual set-up for a neural ODE in which y solves $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t))$.

Let $q \in \mathbb{N}$ and consider some q-th order numerical ODE solver. Over any given numerical step $[t_j, t_{j+1}]$, such solvers operate by locally approximating the solution y by some q-th order polynomial. Correspondingly the error made over some step is determined by the $q+1$-th total derivative $\frac{\mathrm{d}^{q+1}}{\mathrm{d}t^{q+1}}(y(t)) = \frac{\mathrm{d}^q}{\mathrm{d}t^q}(f_\theta(t, y(t)))$.

We may seek to minimise numerical errors, and promote easy-to-integrate dynamics, by regularising

$$\int_0^T \left\| \frac{\mathrm{d}^q}{\mathrm{d}t^q}\left(f_\theta(t, y(t))\right) \right\|_2^2 \mathrm{d}t. \tag{5.11}$$

This was investigated in [Kel+21].

(As this is only a regularisation term – this will be made small, not precisely zero – then we may also consider regularising lower-order derivatives instead.)

Taylor-mode autodifferentiation In principle evaluating (5.11) may be done via autodifferentiation, although a little care is needed to get the correct derivatives: each derivative on the right hand side involves taking a derivative of y with respect to t, so that $\frac{\mathrm{d}y}{\mathrm{d}t} = f_\theta$ will start to appear multiple times [But16, Section 310].

However doing so via naïve autodifferentiation will be unnecessarily expensive. A Jacobian-vector product typically costs 2.5 times the cost of the corresponding forward evaluation, so nesting K such evaluations will result in a $2.5^K = \mathcal{O}(\exp(K))$ cost. That this is a higher-order derivative implies certain structure that may be exploited to reduce the cost to only $\mathcal{O}(K^2)$; see [Kel+21, Appendix A] or [GW08, Chapter 13].

Continuous normalising flows A special case arises – see [Fin+20a] – when regularising low-order derivatives of continuous normalising flows (Section 2.2.3). When evaluating (5.11) with $q = 1$, then

$$\left\| \frac{\mathrm{d}}{\mathrm{d}t}(f_\theta(t, y(t))) \right\|_2^2 = \left\| \frac{\partial f_\theta}{\partial t}(t, y(t)) + \frac{\partial f_\theta}{\partial y}(t, y(t)) f_\theta(t, y(t)) \right\|_2^2$$

and so we may accomplish similar goals by regularising

$$\int_0^T \left\| \frac{\partial f_\theta}{\partial y}(t, y(t)) \right\|_2^2 \mathrm{d}t \qquad \text{and} \qquad \int_0^T \|f_\theta(t, y(t))\|_2^2 \, \mathrm{d}t.$$

Because this is a CNF, we are already computing derivatives of f. This means that the Jacobian (left) expression may be computed very cheaply, without additional calls to autodifferentiation. (This is also the reason that the $\frac{\partial f_\theta}{\partial t}$ term is neglected, as computing that would require an additional autodifferentiation operation.)

If using exact Jacobian computations then the Jacobian expression may be evaluated directly using the already-computed Jacobian. If using Hutchinson's trace estimator, then letting $A = \frac{\partial f_\theta}{\partial y}(t, y(t))$ the following expression applies:

$$\|A\|_2^2 = \operatorname{tr}\left(AA^\top\right) = \mathbb{E}_{\varepsilon \sim \mathcal{N}(0, I_{d\times d})} \, \varepsilon A A^\top \varepsilon = \mathbb{E}_{\varepsilon \sim \mathcal{N}(0, I_{d\times d})} \, \|\varepsilon A\|_2^2 .$$

5.4.2 Exploiting the structure of adaptive step size controllers

For simplicity we will now focus on explicit embedded Runge-Kutta methods as a class of numerical ODE solvers. As per Section 5.3.1.2 these include many of the typical solvers used for neural differential equations, like Heun's method or Dormand–Prince.

Remark 5.26. *The discussions of this section will often actually apply to other solvers and differential equation types too. For example both the reversible Heun method and asynchronous leapfrog method (Section 5.3.2) are very similar to Runge–Kutta methods.*

Such solvers may be decomposed into two main components: an update rule (defined by a Butcher tableau [HNW08]), and a step size controller for updating the step size. (Which may simply be to use a constant step size.)

The update rule is typically the better-advertised component of a solver. Here we will instead focus on how the step size controller may be used or modified to our advantage.

We begin with a brief exposition of how step sizes are adjusted; see [Rac21b], [HNW08, Section II.4], [But16, Section 271] for reference.

Set-up We begin with the usual setup. Let $y_0 \in \mathbb{R}^d$, $\theta \in \mathbb{R}^m$. Let $f_\theta \colon [0,T] \times \mathbb{R}^d \to \mathbb{R}^d$ be uniformly Lipschitz and continuously differentiable, and let $y \colon [0,T] \to \mathbb{R}^d$ solve

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)). \tag{5.12}$$

Let $y_j \approx y(t_j)$ be some numerical approximation to the solution of (5.12). Over a step size $t_{j+1} - t_j$, then a numerical ODE solver may propose some $y_{j+1}^{\text{candidate}} \approx y(t_{j+1})$, along with a local error estimate $y_{j+1}^{\text{error}} \in \mathbb{R}^d$ of the numerical error made in each channel during that step.

Scale and error ratios Given some prespecified absolute tolerance ATOL (for example 10^{-6}) and relative tolerance RTOL (for example 10^{-3}) and (semi)norm $\|\cdot\| : \mathbb{R}^d \to [0,\infty)$ (for example $\|y\| = \sqrt{\frac{1}{d}\sum_{k=1}^d y_k^2}$ the RMS norm), then an estimate of the *scale* of the equation is given by

$$\text{SCALE} = \text{ATOL} + \text{RTOL}\, \max(y_j, y_{j+1}^{\text{candidate}}) \in \mathbb{R}^d$$

with an elementwise maximum. The *error ratio* r is then computed as

$$r = \left\| \frac{y_{j+1}^{\text{error}}}{\text{SCALE}} \right\| \in \mathbb{R}$$

with an elementwise division.

Note the dependence on the choice of norm $\|\cdot\|$. In particular this determines the relative importance of each channel.

Accepting/rejecting steps If $r \leq 1$ then the error is deemed acceptable, the step is accepted and $y_{j+1} = y_{j+1}^{\text{candidate}}$ is taken. If $r > 1$ then the error is deemed too large, the step is rejected and the procedure is repeated with a smaller step size.

Step size changes Regardless of whether the step is accepted or rejected, then the next step size ($t_{j+2} - t_{j+1}$ or $t_{j+1} - t_j$ if the step was accepted or rejected respectively) is selected based on the size of SCALE.

For example the step size may be updated by a multiplicative factor

$$\max\left(\min\left(\frac{\text{SAFETY}}{\text{SCALE}^{1/\text{ORDER}}}, \text{IFACTOR}\right), \text{DFACTOR}\right) \tag{5.13}$$

where ORDER refers to the order of the solver (2 for Heun, 5 for Dormand–Prince and so on), and SAFETY, IFACTOR, DFACTOR are hyperparameters. Typical values would be SAFETY $= 0.9$, IFACTOR $= 10$, DFACTOR $= 0.2$.

This is the 'textbook' step size controller, which is memoryless (each multiplicative factor is dependent only on the previous step). Other step size controllers, often with memory, may also be considered [Rac21b], [But16, Section 271].

This will be all the necessary background material we need on step size controllers.

5.4.2.1 Not-an-ODE and adjoint seminorms

Consider specifically when training neural ODEs via optimise-then-discretise, in which a backward-in-time adjoint ODE is constructed. The particular structure of the continuous adjoint equations actually means that the usual choice of norm for computing the error ratio, such as the RMS norm, is unnecessarily stringent: steps are unnecessarily rejected, and step sizes are too small [KCL21].

By replacing it with a more appropriate (semi)norm, then on the backward pass:

1. Fewer steps are rejected overall;
2. Fewer steps are accepted overall;
3. Fewer steps are rejected, as a proportion of the overall number of steps.

That fewer steps are both accepted and rejected corresponds to generally larger step sizes being used. Moreover, this occurs without adversely impacting model performance.

Continuous adjoint equations For convenience we begin by recalling the set-up for backpropagating via optimise-then-discretise.

Let $L = L(y(T))$ be some (for simplicity scalar) function of the terminal value $y(T)$, so that the continuous adjoint equations (Theorem 5.2) correspond to $a_y \colon [0, T] \to \mathbb{R}^d$ and $a_\theta \colon [0, T] \to \mathbb{R}^m$ solving

$$\begin{aligned} a_y(T) &= \frac{\mathrm{d}L}{\mathrm{d}y(T)}, & \frac{\mathrm{d}a_y}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial y}(t, y(t)), \\ a_\theta(T) &= 0, & \frac{\mathrm{d}a_\theta}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial \theta}(t, y(t)), \end{aligned} \tag{5.14}$$

which are solved backward-in-time from a terminal condition.

Integral, not an ODE The continuous adjoint equations exhibit certain structure: their vector fields are independent of a_θ, and correspondingly the second equation in (5.14) is merely an integral: not an ODE. (See also Remark 5.4.)

As such, whilst it is convenient to evaluate the a_θ component of (5.14) as part of the backward-in-time ODE solve, the ODE solver makes the false assumption that small errors in a_θ may propagate to create larger errors later.

Adjoint seminorms When numerically solving (5.14) backward-in-time, the easy solution is to pick a choice of $\|\cdot\|$ that scales down the influence of the a_θ channels. A simple such choice is to take $\|\cdot\|$ as a seminorm, such as $\|(y, a_y, a_\theta)\| = \sqrt{\frac{1}{2d}\sum_{k=1}^{d}(y_k^2 + a_{y,k}^2)}$ the RMS norm over the y and a_y components, and independent

of the a_θ component. (Recall that the y component is often solved backward-in-time alongside (5.14).)

Does this reduce the accuracy of the parameter gradients? One obvious concern is that we are ultimately interested in the parameter gradients $a_\theta(0)$, in order to train a model. In this respect, this approach seems counter-intuitive. Empirically this does not appear to negatively affecting training, however – we explain this by noting that as the y and a_y channels truly are ODEs, they are likely to be the dominant source of error overall.

Results This can dramatically reduce the computational cost of training. [KCL21] reduce the cost of the backward pass through neural CDEs and Hamiltonian neural networks (Chapter 3, Section 2.2.2) by 40%–62%, and (less dramatically) through CNFs (Section 2.2.3) by 5%.

Quadrature Other methods for evaluating a_θ may also be admitted – for example, whilst it is less convenient than simply using an already-existing ODE solver, a_θ could also be evaluated using a quadrature rule [Hin+21, Section 2.5].

5.4.2.2 Non-backpropagation through adaptive step size controllers

Consider backpropagation via discretise-then-optimise. Technically speaking, we should expect to backpropagate through the entire computational graph, including through updates to step sizes, and through rejected steps.

Even rejected steps will in principle have a small effect on the backpropagated gradients. Every step (accepted or rejected) is used as an input to SCALE, which is used to compute the multiplicative factor by which a step size is updated (equation (5.13)), which determines the timestep values $\{t_j\}_{j=1}^n$, which in general may be used as an input to the neural vector field f_θ.

In practice this is not always desirable. Backpropagating through rejected steps implies additional computational work [Zhu+20b], and anecdotally we have observed that backpropagating through equation (5.13) will sometimes introduce gradient pathologies that hinder training.

For this reason it is very common not to backpropagate through step size selection – when differentiating the computational graph we treat the result of equation (5.13) as a constant.[11] Neither the `torchdiffeq` nor Diffrax software libraries backpropagate through step size selection, for example [Che18; Kid21a].

[11] In PyTorch this means applying `detach` to the output of equation (5.13); in JAX or TensorFlow this means applying `stop_gradient`.

5.4.2.3 Regularising error estimates

[Pal+21] seek to encourage easy-to-integrate dynamics by adding

$$\sum_j y_j^{\text{error}} \left|(t_{j+1} - t_j)\right|$$

as a regularisation term when solving a neural ODE. ([Pal+21] also consider a variant of this, by regularising a term used for detecting stiffness of the differential equation.)

This is computationally almost free, as all y_j^{error} will already have been computed.

This technique relies on optimising the neural ODE via discretise-then-optimise (or with a bit of work, a reversible solver). If using optimise-then-discretise then the computational graph for computing y_j^{error} is not saved for later backpropagation.

5.5 Numerical simulation of Brownian motion

Numerically solving an SDE requires sampling a Brownian motion $w\colon [0, T] \to \mathbb{R}^{d_w}$.

Brownian bridges Mathematically, sampling Brownian motion is straightforward. A fixed-step numerical solver may simply sample independent Gaussian random variables during its time stepping. An adaptive solver (which may reject steps) may use Lévy's Brownian bridge formula [RY13] to generate the appropriate correlations: for any $s < t < u$,

$$w(t)|(w(s), w(u)) \sim \mathcal{N}\left(w(s) + \frac{t-s}{u-s}(w(u) - w(s)), \frac{(u-t)(t-s)}{u-s} I_{d_w \times d_w}\right) \quad (5.15)$$

and this quantity is (conditionally) independent of $w(v)$ for $v < s$ or $v > u$.

Brownian reconstruction However, there are computational difficulties. The main one is that during backpropagation, the same Brownian sample as the forward pass must be used, and if using optimise-then-discretise (Section 5.2.3) may potentially be queried at locations other than were sampled on the forward pass.

In addition, we need to efficiently track the value of the Brownian motion at each end of any interval we will later need to condition on. (To apply the Brownian bridge formula, for example when rejecting steps.)

Brownian sampling We will now see three approaches to handling this: the Brownian Path, the Virtual Brownian Tree, and the Brownian Interval.[12] The Brownian Path and Virtual Brownian Tree are included as 'warm-ups' for pedagogical purposes; in practice the Brownian Interval will usually be the go-to choice.

[12]These choices of terminology are not completely standard; we adopt the names used in the `torchsde` library [Li20b].

5.5.1 Brownian Path

One approach is simply to store every sample, and apply equation (5.15) when appropriate. There are some questions about the optimal data structure to store these values in, for efficient querying later – in practice the tree-like structure we will later introduce for the Brownian Interval is often a good choice – but otherwise there is little to discuss here.

This approach is simple and usually gets the job done. During an SDE solve, querying takes $\mathcal{O}(1)$ time (assuming a suitable data structure, see the Brownian Interval later). The main downside is the consumption of $\mathcal{O}(d_w T)$ memory.

5.5.2 Virtual Brownian Tree

The memory cost of the previous approach can sometimes be large enough to be a concern. This is especially true when taking many small steps to solve the SDE, or when using the continuous adjoint method or reversible SDE solvers (Sections 5.2.3 and 5.3.2) for which the Brownian motion samples represent a higher proportion of the overall memory usage.

As such, [Li+20a], motivated by [GL97], introduce the 'Virtual Brownian Tree'.

Splittable PRNGs The first key ingredient is 'splittable' pseudo-random number generator (PRNG) seeds [Sal+11; CP13].

Given an m-bit random seed $\rho \in \{0,1\}^m$, splitting is an operation that produces some n new m-bit random seeds $\rho_1, \ldots, \rho_n \in \{0,1\}^m$, as a deterministic function of ρ, for which $\rho, \rho_1, \ldots, \rho_n$ produce statistically independent streams of random numbers when used as the seed for a PRNG.

Given any rooted tree, we can associate a random seed with every node in the tree in the following way.

Let $(V, E, *)$ be a rooted tree, where V denotes some vertex set,

$$E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$$

denotes some edge set (connected and without cycles), and $* \in V$ denotes the root. For any $x \in V$, let $\Gamma(x) = \{y \in V \mid \{x, y\} \in E\}$ denote the set of vertices adjacent to x.

Let $\rho \in \{0,1\}^m$ be an m-bit seed, which we associate with the root $*$. Split ρ into $\rho_1, \ldots, \rho_{|\Gamma(*)|}$ random seeds and pair each one with a corresponding element $v_i \in \Gamma(*)$. Recursively split each ρ_i and pair the resulting seeds with the elements of $\Gamma(v_i)$, and so on, recursing this procedure throughout the tree.

By fixing a rooted tree $(V, E, *)$ and a root seed ρ, we may deterministically create a PRNG at every node in the tree. Provided we remember only the tree structure and

the root seed s, we can later rematerialise every PRNG sequence, for every node of the tree, without holding the samples in memory.

Example 5.27. *A function call graph is an example of such a rooted tree. We begin by calling a function. This in turn may call other functions, which in turn may call other functions, and so on – which we consider a tree, rather than a DAG, by treating multiple calls to the same function separately. Splittable PRNGs may be used to deterministically generate pseudorandomness at any point in this call graph, for example when writing pure functions. This is actually the procedure used throughout the JAX software library [Bra+18] whenever generating random samples is required.*

Generating Brownian samples Let $\varepsilon > 0$ be some fixed (small) tolerance. Consider the collection of dyadic points $V = \left\{ Tj2^{-k} \;\middle|\; j, k \in \mathbb{N}, T2^{-k+1} > \varepsilon \right\}$. These form a tree-like structure: each $Tj2^{-k}$ has $T\left\lceil j/2 \right\rceil 2^{-k+1}$ as its parent.

By recording only some root-level seed $\sigma \in \{0,1\}^m$, and associating seeds with the elements of this tree as in the previous heading, then a Brownian sample $w(v)$ is completely determined for all $v \in V$: see Algorithm 6.

For $x \in [0,T]$ let $[x]_V$ denote the member of V closest to x. To approximately sample a Brownian increment $w(t) - w(s)$ (when an SDE solver steps from s to t), we first discretise s and t to $[s]_V$ and $[t]_V$, sample $w([s]_V)$ and $w([t]_V)$ as in Algorithm 6, and then return $w([t]_V) - w([s]_V)$. The main downsides are that this takes $\mathcal{O}(\log(1/\varepsilon))$ time, and produces only approximate samples. However, it has the advantage that this requires only $\mathcal{O}(1)$ memory.

Whilst sampling Brownian motion is easy, the key point of this construction is how it additionally allows for *reconstructing* the same Brownian motion sample, without holding the individual samples in memory.

5.5.3 Brownian Interval

We are now ready to present the Brownian Interval, which improves upon the Brownian Tree with exact sampling and $\mathcal{O}(1)$ query times.

5.5.3.1 Overview

Sampling intervals Let $w(s,t)$ denote $w(t) - w(s) \in \mathbb{R}^w$.

We begin by shifting from a point-evaluation approach, in which each query to the Brownian object produces some $w(t)$, to an interval-evaluation approach, in which each query to the Brownian object generates some $w(s,t)$.

Rewriting the Brownian bridge equation (5.15) gives

$$w(s,t)|w(s,u) \sim \mathcal{N}\left(\frac{t-s}{u-s}w(s,u), \frac{(u-t)(t-s)}{u-s}I_{d_w \times d_w}\right). \tag{5.16}$$

Algorithm 6: Sampling the Virtual Brownian Tree. `split_seed` denotes splitting a seed into two, and `bridge` denotes the Brownian bridge of equation (5.15).

Input: Time horizon $T > 0$, sample time $\tau \in [0, T]$, seed ρ, error tolerance $\varepsilon > 0$
Result: Approximation to $w(\tau)$

$\rho, \widehat{\rho} = \texttt{split\_seed}(\rho)$
$w_T \sim \mathcal{N}(0, TI_{d_w \times d_w})$ sampled with seed $\widehat{\rho}$
$s = 0$
$t = T/2$
$u = T$
$w_s = 0$
$w_t \sim \texttt{bridge}(0, T/2, T, 0, w_T)$ sampled with seed ρ
$w_u = W_T$

while $|\tau - t| > \varepsilon$ **do**
  $\rho_1, \rho_2 = \texttt{split\_seed}(\rho)$
  if $\tau > t$ **then**
    $s = t$
    $w_s = w_t$
    $\rho = \rho_1$
  else
    $u = t$
    $w_u = w_t$
    $\rho = \rho_2$
  $t = (s + u)/2$
  $w_t \sim \texttt{bridge}(s, t, u, w_s, w_u)$ sampled with seed ρ
return w_t

The complement $w(t, u)|w(s, u)$ is calculated as $w(t, u)|w(s, u) = w(s, u) - w(s, t)|w(s, u)$.

Binary tree of (interval, seed) pairs Similar to the binary tree of (point, seed) pairs used in the Brownian Tree, we will now have a binary tree of (interval, seed) pairs. Each parent interval will be the disjoint union of its child intervals.

The tree starts as a stump consisting of the global interval $[0, T]$ and an m-bit random seed ρ. New leaf nodes are created as queries over intervals are made. For example, making a first query at $[s, t] \subseteq [0, T]$ (an operation that will return $w(s, t)$) produces the binary tree shown in Figure 5.1a; making a subsequent query at $[u, v]$ with $u < s < v < t$ produces Figure 5.1b. Using a splittable PRNG as in Section 5.5.2, each child node has a random seed deterministically produced from the seed of its parent. Unlike the Virtual Brownian Tree, which has a fixed (dyadic) tree construction, the tree used in the Brownian interval is query-dependent.

Figure 5.1: Binary tree of intervals. Only the intervals, without the corresponding seeds, are shown.

The tree thus completely encodes the conditional statistics of a Brownian motion, conditional on all previous queries: $w(s,t), w(t,u)$ are completely specified by s, t, u, $w(s,u)$, equation (5.16), and the random seed associated with $[s,u]$.

Generating Brownian samples In principle we may now calculate $w(s,t)$ for any $s < t$. The query over $[s,t]$ adds extra nodes to the tree (if necessary; $[s,t]$ may have been queried before), so that the conditional statistics of the query, with respect to all previous queries, are captured. As in Figure 5.1b, this may decompose $[s,t]$ into some disjoint union of subintervals. We then calculate $w(s,t)$ by applying equation (5.16) to each subinterval.

This calculation does require the Brownian increment $w(s,u)$ over the parent interval $[s,u]$. In principle this is calculated recursively in the same way, working our way up the tree. (As with the Virtual Brownian Tree.) However, this may be improved by adding a least recently used (LRU) cache to the computed increments $w(s,t)$.

Queries are exact because the tree aligns with the query points. Queries are fast because of the LRU cache: in SDE solvers, subsequent queries are likely to be close to (and thus conditional on) previous queries. The average-case (modal) time complexity is thus $\mathcal{O}(1)$. Even in the event of cache misses all the way up the tree, the worst-case time complexity will only be $\mathcal{O}(\log(1/h))$ in the average step size h of the SDE solver. The (GPU) memory cost is essentially the size of the LRU cache, which is constant and thus $\mathcal{O}(1)$.

The trade-off here is that we must store the tree structure itself, which grows each time a query is made. For an SDE solve on $[0,T]$ then this will consume $\mathcal{O}(T)$ CPU memory. In practice however this is unlikely to be a limitation: GPU memory is usually the limiting factor, in comparison to which CPU memory is essentially infinite.

5.5.3.2 Algorithmic definitions and further discussion

Precise algorithmic definitions and (substantial) further discussion on the Brownian Interval is deferred to Appendix C.5 to avoid breaking the flow of the presentation.

Remark 5.28. *To what extent may either the point-based approach of the Virtual Brownian Tree, or the interval-based approach of the Brownian Interval, be interchanged?*

An interval-based approach to the Virtual Brownian Tree is possible, but would likely be inefficient. For small approximation tolerance ε and step sizes possibly much larger than ε, then a query for some interval $[s, t]$ would require logarithmically many dyadic intervals (each of length $Tj2^{-k}$ for some j, k), to construct $[[s]_V, [t]_V]$. This is as opposed to just making two point-based queries. (The Brownian Interval largely avoids this issue with its query-dependent trees.)

A point-based approach to the Brownian Interval is possible (despite the name), but the interval-based approach has several upsides.

- *Elegance. We directly query for the sample $w(s, t)$ actually used in the SDE solver.*
- *Efficiency. Making only a single query for $w(s, t)$ as opposed to making two queries for $w(s)$ and $w(t)$.*
- *Lévy area approximation. Some numerical SDE solvers sample additional randomness beyond just the point evaluations $w(t)$: typically these samples are of higher-order integrals $\int_s^t w_{k_1}(u)\,\mathrm{d}w_{k_2}(u)$, computed over intervals $[s, t]$. For example stochastic Runge–Kutta methods may require space-time Lévy area samples, and the log-ODE method for SDEs*[13] *uses full Lévy area samples. As they are defined over intervals $[s, t]$, these quantities are intrinsically interval-based values, requiring an interval-based Brownian motion construction. The details of this are a topic beyond our scope here; James Foster's doctoral thesis [Fos20] introduce the requisite formulas analogous to* (5.16)*, and the necessary extensions to the Brownian Interval are implemented in* `torchsde` [Li20b].

5.6 Software

Software packages for the numerical solving and training of neural differential equations are now relatively standardised. They handle most of the details described over the course of this chapter, and correspondingly the user is free to focus on the modelling details that have been the focus on the other chapters in this thesis.

At time of writing, there are a selection of options.

- In the JAX ecosystem [Bra+18] there is Diffrax.
  - `https://github.com/patrick-kidger/diffrax`

[13] This is the same log-ODE method as seen in Appendix B. The additional Lévy area terms used here correspond to the logsignature terms used there.

- In the PyTorch ecosystem [Pas+19] there is the `torchdiffeq`, `torchcde`, and `torchsde` family of libraries. (And additionally `torchdyn` as a higher-level wrapper providing some common models.)
  - `https://github.com/rtqichen/torchdiffeq`
  - `https://github.com/patrick-kidger/torchcde`
  - `https://github.com/google-research/torchsde`
  - `https://github.com/DiffEqML/torchdyn`
- In the Julia [Bez+17] ecosystem there is DifferentialEquations.jl.
  - `https://github.com/SciML/DifferentialEquations.jl`

Every package we recommend is open source, offers a stable API, is relatively feature-complete, and comes with comprehensive documentation and examples – including code examples for many of the techniques discussed in this thesis.

Whilst exact functionality differs slightly by package, one can expect most of:

1. Explicit and implicit solvers;
2. Fixed and adaptive step size solvers;
3. Differentiation via both optimise-then-discretise and discretise-then-optimise;
4. Reversible differential equation solvers;
5. Event handling;
6. Callbacks;
7. Handling of jumps in the vector field;
8. For neural CDEs: all interpolation schemes discussed here;
9. For neural RDEs: logsignature pre-processing as in Appendix B;
10. Solving of both Itô and Stratonovich SDEs;
11. Solving SDEs with varying noise types (scalar, additive, diagonal, general);
12. Brownian Interval simulation as in Section 5.5;
13. Levy area approximation;
14. Gradient checkpointing;
15. Distributed computing;
16. CPU parallelism;

17. GPU support.

Any choice amongst these libraries is a reasonable one.

Remark 5.29. *If the reader is free to choose, then we would recommend Diffrax. It is the newest of these libraries, and is quite exciting on a technical level, as it solves ODEs, CDEs, and SDEs in a unified way by internally lowering all of them to rough differential equations. In addition if working with irregular time series, then it is the only one amongst these libraries to offer the ability to batch over different regions of integration. We must admit to some bias – Diffrax is the author's own project, created whilst writing this thesis.*

5.7 Comments

The choice of discretise-then-optimise versus optimise-then-discretise backpropagation is a classical one in the context of differential equations. [GKB19] is the canonical reference on this topic in the context of neural ODEs. See also [Rac+20a; OR20] for related comparisons.

Reversible differential equation solvers, as applied to backpropagation, are quite a new topic. [Mut13; Zhu+21] introduce a reversible ODE solver, whilst [Kid+21a] introduce the first reversible SDE solver.

The broader comparison of discretise-then-optimise against optimise-then-discretise against reversible differential equation solvers is new here. (And mistakes on this topic are frequent in the literature; we have come across several erroneous statements about favouring optimise-then-discretise over discretise-then-optimise, in contexts where the opposite is true.)

The proof of optimise-then-discretise for ODEs (relegated to Appendix C.3.1), and its sketchproof (Section 5.1.2.1), are new here. To the best of our knowledge the existing literature has relied on only more complicated proofs.

The proofs of optimise-then-discretise for CDEs and SDEs (relegated to Appendices C.3.2 and C.3.3) are new here. Once again to the best of our knowledge, the existing literature has relied on (substantially) more complicated proofs. A proof of optimise-then-discretise for SDEs appeared in [Li+20a]. A proof of optimise-then-discretise for CDEs (along with a rough path theory proof of optimise-then-discretise for SDEs) first appeared in [Kid+20b], although this was never published.

The discussion on the choice of numerical solver is part of the folklore of neural differential equations; our presentation here is based on our own anecdotal experience and our conversations with others.

Baked-in discretisations (both that they occur and that they are acceptable) are again part of the folklore, although they have been explicitly studied in [Ott+21; Que+21].

The terminology of analytic and algebraic reversibility is new here. Some existing texts do refer to just 'reversible solvers', usually in the context of symplectic solvers.

The more efficient backward step for the asynchronous leapfrog method (Algorithm 5) is new here. ([Zhu+21] used the more general, less efficient, Algorithm 1). It is actually also possible to construct a more efficient backward step through the reversible Heun method as well, so as to elide the local forward operation. It is however more finicky to do so – the local backward needs to occur on the reverse pass of the *previous* step – so for simplicity we omit this here.

On the stability of the asynchronous leapfrog method: [Zhu+21] do additionally introduce a 'damped asynchronous leapfrog method' with nontrivial region of stability. In practice the region of stability remains very small, and one of the main advantages of reversible solvers is the ability to use them with very large step sizes,[14] so the benefit of this is not clear. We note that [Zhu+21] mangle terminology slightly by referring to a 'region of A-stability' when merely 'region of stability' or 'region of absolute stability' would be correct. ('A-stability' is a property of the region of stability itself.)

The 'not-an-ODE'/'adjoint seminorm' trick for improving backpropagation speed through neural ODEs may also be applied to forward sensitivities [Hin+21, Section 5.5].

The comparison of different software libraries is new here. In fact, the Diffrax software library was written by the author for the express purpose of writing this thesis (or perhaps to procrastinate from writing this thesis). Realistically this has been a fast-moving space, and we would not be surprised if the section on software rapidly becomes outdated.

[14]Whilst still getting both memory efficiency and accurate gradients; discretise-then-optimise giving only the latter and with large step sizes optimise-then-discretise giving only the former.

Chapter 6

Miscellanea

6.1 Symbolic regression

6.1.1 Introduction to symbolic regression

Deep learning, including neural differential equations, typically produces 'black-box' models. Once the model has been trained, it is a relatively opaque neural network whose mode of operation is essentially mysterious. It may be a good model, but a good model is not always the end goal. Scientific progress may be predicated upon understanding the model as well.

It is often desirable to obtain symbolic expressions – an imprecise term which we use here to refer to some relatively shallow tree of primitive operations, for example $x \times ((y - 4.2) + z)$. These primitive operations typically include addition, multiplication, exponentiation and so on.

Symbolic regression is the process of deriving such expressions from data in an automated way. One difficulty is the lack of differentiability of the space of such expressions. Whilst any constant in the expression (such as the 4.2 above) may be optimised differentiably, the space between expressions is usually traversed via genetic algorithms. Another difficulty is the size of this space: there are $\frac{(2n)!}{(n+1)!n!}$ binary trees with n vertices, and so as a rough approximation we may expect there to be a similar number of possible expressions to consider. This is a big number.

For these reasons, symbolic regression is a difficult task that often works best only on simple problems; past a certain point the complexity grows too large and the problem becomes intractable.

6.1.2 Symbolic regression for dynamical systems

Example 6.1. *Suppose we observe paired samples of both $y(t)$ and $\frac{\mathrm{d}y}{\mathrm{d}t}(t)$, assumed to satisfy an equation of the form*

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(y(t)).$$

Then SINDy [BPK16] seeks a symbolic expression for f by selecting some features f_i in advance, parameterising $f(y) = \sum_{i=1}^{N} \theta_i f_i(y)$, and directly regressing $\frac{\mathrm{d}y}{\mathrm{d}t}(t)$ against $\{f_i(y(t))\}_{i=1}^{N}$. A sparsity penalty such as L^1-regularisation is applied to θ so that only a few terms are selected in the final expression.

This procedure is simply standard LASSO, and the dynamical character of the problem is essentially irrelevant. SINDy is arguably the dominant technique for symbolic regression with dynamical systems; some example extensions and applications include [Rud+17; AF20; Kah+19; Kap+21; Cha+19a].

However, SINDy has made two strong assumptions: (a) that paired observations of both y and $\frac{\mathrm{d}y}{\mathrm{d}t}(t)$ are available, and (b) that f is a shallow tree of expressions – just a linear combination of preselected features.

We will now see how NDEs offer ways to remove both of the assumptions made in Example 6.1.

Removing assumption (a): no paired observations Suppose we observe samples $y(t)$ assumed to come from some dynamical system

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(y(t)), \tag{6.1}$$

which for simplicity we assume is an autonomous ODE. (Although this is not necessary – the same ideas apply equally well to non-autonomous dynamical systems, and to non-ODEs such as CDEs and SDEs.)

Given this data, we learn some $f = f_\theta$ as a neural network as described in the rest of this thesis. For example, minimising some empirical loss between the data and a numerical solution of the initial value problem, and optimising f_θ via backpropagation.

Remark 6.2. *Note that unlike SINDy, we have not assumed access to paired observations of both y and $\frac{\mathrm{d}y}{\mathrm{d}t}(t)$.*

SINDy sometimes works around this by approximating $\frac{\mathrm{d}y}{\mathrm{d}t}(t)$ using finite differences. However this requires densely-packed observations, whilst the above procedure applies even when observations of $y(t)$ are sparse.

Removing assumption (b): deep symbolic expressions Subsequently, we perform symbolic regression across the learnt f_θ. That is, for each observed sample y we evaluate $f_\theta(y)$, and symbolically regress $f_\theta(y)$ against y.

The symbolic regression itself may be performed in any number of ways. A reasonable choice for most tasks is regularised evolution [Rea+19], which is capable of learning complex trees of expressions, and traverses the space between them via genetic algorithms. Open source software libraries exist to perform this task – at time of writing we recommend the PySR and SymbolicRegression.jl libraries [Cra20] for Python and Julia respectively.

More advanced techniques include [WC13; Gui+20; ML16; Li+19]. For example *deep symbolic regression* introduces learnt neural network optimisers to tackle the task of searching through symbolic expressions.

(And if we really wanted, we could just take the simple approach of applying regularised linear regression against preselected features as in Example 6.1 – any symbolic regression technique will do.)

The result of this symbolic regression is our final result.

Remark 6.3. *Note the Markov assumption that is being made in equation* (6.1)*: the vector field f depends entirely on the observations y, so that there is no dependence on the past. In contrast observe that our typical set-up for NDEs has been to have the dynamical system operate in some latent space (that is, y is hidden state), and then linearly project this space down to the data space. See for example Remark 2.9, or the use of readout maps with neural CDEs and SDEs (Chapters 3 and 4).*

Extending symbolic regression to the non-Markov setting is nontrivial. The essential difficulty is that symbolic regression in a latent space is not obviously meaningful. Supposing that the latent space is some $\mathbb{R}^{d_y} \ni y(t)$, and letting $\mathcal{A} \subseteq \{\mathbb{R}^{d_y} \to \mathbb{R}^{d_y}\}$ be some collection of automorphisms of this space, then for any $\phi \in \mathcal{A}$, both f and $\phi^{-1} \circ f \circ \phi$ define essentially the same dynamics. That is, we may only identify f up to conjugacy by elements of $\mathcal{A}$.

We note that [Cha+19b] do consider symbolic regression in a latent space. The above problem is dealt with implicitly in an ad-hoc manner, by (a) using only simple symbolic regression techniques (LASSO as with SINDy) to constrain the complexity of the vector field; (b) relying on Lipschitz embeddings/decodings from the latent space, again to constrain complexity; (c) manually selecting the 'best' element of the conjugacy class $\{\phi^{-1} \circ f \circ \phi \mid \phi \in \mathcal{A}\}$ after training has completed. As such 'latent symbolic regression' has seen some success, but is in many respects still an open problem.

6.1.3 Example

Let $T > 0$ and consider the nonlinear oscillator

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x \\ y \end{bmatrix}(t) = \begin{bmatrix} \frac{y(t)}{1+y(t)} \\ \frac{-x(t)}{1+x(t)} \end{bmatrix} \quad \text{for } t \in [0, T], \tag{6.2}$$

with $x(0), y(0) \sim \text{Uniform}[-0.6, 1]$. Samples from this equation resemble warped and deformed sines and cosines.

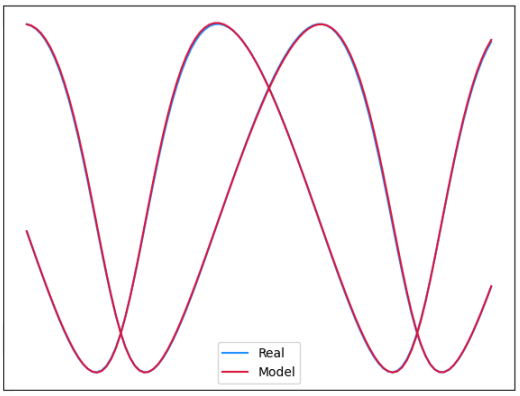


Figure 6.1: Sample, and trained reconstruction, of the nonlinear oscillator (6.2). Both dimensions (x, y) of the oscillator are shown plotted against time t. Model and data align almost perfectly.

.

We aim to learn the symbolic form of this differential equation from data. Note the form of the vector fields, which would be very difficult to learn using SINDy.[1]

Data We fix some points $t_j \in [0, T]$, with $t_0 = 0$. For initial conditions $x(0), y(0) \sim \text{Uniform}[-0.6, 1]$ we assume access to observations of the corresponding $x(t_j), y(t_j)$. For simplicity we take the samples to be noiseless.

Neural regression We train a neural ODE via the L^2 loss as described above or as in Chapter 2, to reconstruct $x(t_j), y(t_j)$ given $x(0), y(0)$. That is, we consider the model

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x \\ y \end{bmatrix} = f_\theta(x(t), y(t))$$

where $f_\theta \colon \mathbb{R}^2 \to \mathbb{R}^2$ is a neural network, and for each initial condition $(x(0), y(0))$ the above equation is solved as an initial value problem using a numerical ODE solver.

The result of training this neural ODE is shown in Figure 6.1. The model has perfectly learnt the structure of the problem.

Symbolic regression We now have access to a function $f_\theta \colon \mathbb{R}^2 \to \mathbb{R}^2$ which we may treat in isolation. The dynamic structure of the problem has been removed.

Symbolically regressing $f_\theta(x(t_j), y(t_j))$ against $(x(t_j), y(t_j))$ via regularised evolution produces the expression

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} \frac{y}{1.01+y} \\ \frac{-1.04x}{1.03+x} \end{bmatrix}. \tag{6.3}$$

[1] Requiring for example the additional knowledge that the vector field is in fact a rational function [Man+16].

Neural-symbolic regression Finally, we treat (6.3) as the vector field of a 'neural' differential equation and perform another round of gradient-based optimisation against the original dataset to optimise the constants in the symbolic expressions.

Rounding each constant to the nearest multiple of 0.01 then produces the desired vector field

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} \frac{y}{1+y} \\ \frac{-x}{1+x} \end{bmatrix}. \tag{6.4}$$

Further details Further details may be found in Appendix D.5. The code is available as an example in Diffrax [Kid21a].

6.2 Limitations of neural differential equations

We have spent most of this thesis discussing the numerous advantages and applications of neural differential equations. It is only fair we dedicate some space to their limitations.

6.2.1 Data requirements

Neural differential equations have one major difference to classical differential equations. Using a neural network as the vector field (or as a component of the vector field, as with UDEs), results in greatly increased model expressivity, and so correspondingly more data is needed to train the model.

As such data requirements are typically comparable to neural network based approaches. A few hundred samples represents an optimistic lower bound on the amount of data required. The toy example problems considered in this thesis use a few thousand samples. Some examples ([Kid+21b, Section 4.2]) use millions of samples.

This is a limitation compared to classical differential equations – in return for which we receive more expressive models – but compared to neural networks this is of course quite normal.

6.2.2 Speed

In particular when using higher-order differential equation solvers, which make multiple vector field evaluations, then neural differential equations can be somewhat slow to evaluate or train.

This can be mitigated by using cheaper, lower-order, solvers.[2] For example a low-

[2]Which need not affect model efficacy. Model efficacy, and accuracy at solving the idealised differential equation, are two different things.

order reversible solver (Section 5.3.2) can be used to obtain accurate gradients despite its low order.

Additionally, neural differential equations have a trick not available to standard neural networks: the choice of solver can be varied. For example a cheap low-order solver can be used for the bulk of training, and a more expensive higher-order solver used for fine-tuning and inference.

6.2.3 Other discretised architectures

There are successful neural network architectures not so easily explained by being discretised neural differential equations. For example neither U-Net [RFB15] nor Transformers [Vas+17] admit obvious descriptions of this type, although Transformers do have a continuous theory of their own [Ram+20b].

Whilst neural differential equations are one (very successful) paradigm for constructing discrete architectures, it is apparent they are not the only one.

6.3 Beyond neural differential equations: deep implicit layers

Neural differential equations are part of a larger family of models, known as *deep implicit models* or *deep implicit layers*.

Most layers (operations) used in machine learning models are 'explicit': given an input x they return an output y, as in

$$y = f_\theta(x),$$

where f_θ denotes some function depending on trainable parameters θ.

In contrast, implicit layers take the form

$$\text{Find } y \text{ such that } f_\theta(x, y) = 0. \tag{6.5}$$

That is to say, the output is specified implicitly as one satisfying a certain condition.

This immediately opens up a host of questions – such as uniqueness – that we will not attempt to address in detail here. We aim only to give a high-level flavour of some of this broader family of models.

By its very nature, equation (6.5) cannot usually be solved explicitly or in closed form. As such the common thread running through such models is the use of a numerical scheme to find an approximate solution to equation (6.5).

The word 'implicit' thus takes on a dual meaning: not only is the solution y specified implicitly, but the computational steps to locate it need not be explicitly specified either.

Backpropagation through such models is possible – as in Section 5.1, there are both discretise-then-optimise and optimise-then-discretise approaches available. One option is to backpropagate through the operations of some numerical solver for (6.5). Another option is to apply the implicit function theorem, and then implicitly differentiate (6.5) itself. Variations on this are discussed in [GW08, Chapter 15] and [BKK19; Blo+21; Fun+21].

A recent line of work has begun to suggest that implicit models may consistently outperform explicit models [Flo+21; Lu+21; Fun+21].

6.3.1 Neural differential equations as implicit layers

A neural ODE is an implicit model: it is specified as

$$\text{Find } y \text{ such that } y(0) = y_0 \text{ and } \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)),. \tag{6.6}$$

That the computational steps for computing it are left implicit is the very reason Chapter 5 exists.

6.3.2 Deep equilibrium models

'Deep Equilibrium Models' (DEQs) [BKK19] are essentially another term for implicit modelling in general, although the term is often used to refer to the case in which f_θ takes the form of some 'large' neural network architecture, such as a Transformer [Vas+17], in which (6.5) is solved via fixed-point iterations.

As before, given an input x and a neural network f_θ, the output y is simply defined as

$$\text{Find } y \text{ such that } f_\theta(x, y) = 0. \tag{6.7}$$

For example if x and y are sequences then taking f_θ to be a Transformer is a reasonable choice.

[BKK19] consider applications to time series whilst [BKK20] consider applications to computer vision. Monotone Operator Equilibrium Models (monDEQ) impose additional structure to ensure that 6.7 has a unique solution [WK20]. Stability may be improved via regularisation [BKK21].

6.3.3 Multiple shooting: DEQs meet NODEs

Let $d \in \mathbb{N}$ and let $f \in \mathrm{Lip}(\mathbb{R} \times \mathbb{R}^d; \mathbb{R}^d)$. For $u < v$ and $x \in \mathbb{R}^d$, let y denote the solution to

$$y(0) = x, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)) \text{ for } t \in [u, v]$$

and then let $\phi_\theta(x, u, v) = y(v)$ denote the map from initial condition to terminal condition.

Given an input $x \in \mathbb{R}^d$, a time horizon $T > 0$, and time points $0 = t_0 < \cdots < t_n = T$, then *multiple shooting* reframes the solution of an ODE as the solution to the implicit problem:

$$\text{Find } b = (b_0, \ldots, b_n) \in \mathbb{R}^d \times \cdots \times \mathbb{R}^d \text{ such that } b_0 = x \text{ and } b_{j+1} = \phi_\theta(b_j, t_j, t_{j+1}).$$

In many ways this is an implicit problem like any other – for example, we may aim to solve it via a fixed-point iteration, perhaps via Newton methods. Each step of this fixed point iteration itself involves solving multiple ODEs, to evaluate each $\phi_\theta(b_j, t_j, t_{j+1})$.

The key advantage of this approach is that the solution to the multiple ODEs over each interval $[t_i, t_{i+1}]$ may be performed in parallel. Provided that only few steps of the fixed-point solver are required, then this can reduce the overall computation time. (Even though it might not necessarily reduce the overall computational work, due to parallelism.) This is generally the case provided a good initial guess for b can be obtained.

Example 6.4. *For example this may be used during training. Fix a single batch of data, and train a neural ODE for several steps (of parameter optimisation, updating θ) on the same batch of data.*

The first such step should obtain a solution via other numerical methods, for example as discussed in Chapter 5. This provides an initial value for each b_j. As the learnt parameters θ evolve only slightly over the course of each training step, these b_j can be used to provide a good initial guess for subsequent parameter optimisation steps, for which the neural ODE is solved using multiple shooting.

This may be used to train on the same batch of data for a few steps, before sampling a fresh batch.

Example 6.5. *Another example arises when using a neural ODE to model a fully-observed dynamical system. As it is a fully-observed dynamical system we may suppose it is Markov, and attempt to model the observations directly. (So that our neural ODE is 'unaugmented' and does not evolve in a latent space as in Section 2.3.3.)*

Suppose further that each training sample consists of multiple observations $y(s_j)$ for $s_j \in [0, T]$. Then during training we may choose $t_j = s_{m_j}$ for some choice of m_j, and take $b_j = y(t_j) = y(s_{m_j})$.

Example 6.6. *A final example, this time during inference, is a classical use-case for multiple shooting: when using an ODE to provide forecasts into the future, continuously updated as new information arrives. Each forecast involves solving an ODE from the current time to some future time (for example, the time now plus ten minutes). As new data arrives we update our forecast, and the solution of the old forecast may be used to initialise each b_j.*

See [Mas+21] for more details.

6.3.4 Differentiable optimisation

A final kind of implicit model is the solution to optimisation problems.

To be clear, whilst 'optimisation' often refers to the training of parameters, we are here referring to a model or layer whose operation is defined as the solution to an optimisation problem. We might express this as an implicit layer as

$$\begin{aligned} &\min_y f_\theta(x, y) \\ \text{such that} \quad &y \in C_\theta(x) \end{aligned}$$

where θ denotes trainable parameters, x denotes an input to the model, and $C_\theta(x)$ is some constraint set.

Equivalently, and once again setting aside concerns such as uniqueness,

$$y = \arg\min_{\widetilde{y} \in C_\theta(x)} f_\theta(x, \widetilde{y}).$$

Brandon Amos' doctoral thesis [Amo19] gives more details on differentiable optimisation, building on [AXK17; AK17; Agr+19].

Example 6.7. *For example, we might consider the optimisation problem*

$$\begin{aligned} &\min_{y \in \mathbb{R}^d} \|x - y\| \\ \text{such that} \quad &\theta_1 \cdot y \leq \phi_1, \ldots, \theta_n \cdot y \leq \phi_n \end{aligned} \tag{6.8}$$

where each $\theta_i \in \mathbb{R}^d$ and each $\phi_i \in \mathbb{R}$. This projects x onto some convex polytope, defined as the intersection of n half-spaces. In this case, θ_i and ϕ_i are trainable parameters. Given some paired observations x, y, denoting points x and their projections y onto some unknown polytope, then by optimising (6.8) *we may learn an approximation to this unknown polytope.*

6.4 Comments

The material on symbolic regression is joint work with Miles Cranmer, and is new here.

The discussion on limitations of neural differential equations is a standard part of the folklore.

The notion of deep implicit layers is a recent one in deep learning, largely popularised by [KDJ20].

Chapter 7

Conclusion

7.1 Future directions

Having discussed the story so far – what future directions do we anticipate?

Boutique versus 'off the shelf' Most applications of neural differential equations are still 'boutique', rather than 'off the shelf'. The model is tailored – with largely unstructured vector fields retrained from scratch – for each individual use case.

This is unlike traditional differential equations, for which we have numerous well-studied models, and in each case may expect to find a wealth of literature discussing their long-term behaviour, bifurcation properties and so on. There already exists an analogous literature in modern deep learning, studying the behaviour of models such as GPT-3 [Bro+20], CLIP [Rad+21] and so on.

In time the same development may take place for neural differential equations.

Neural ODEs Thousands of papers are written every year applying non-neural ODEs to topics across science, finance, economics, ..., and so on. Correspondingly, one significant opportunity is to apply neural ODEs to many of the tasks to which only non-neural ODEs have so far been applied.

Neural CDEs and SDEs Neural CDEs and neural SDEs are much newer. Work remains to be done on the practical machine learning details: finding expressive choices of vector field, and determining how to train these models efficiently. As with neural ODEs, another future direction is their application to practical topics, or how to hybridise them with their non-neural equivalents.

In addition, CDEs and neural CDEs have natural connections to control theory, and from that to reinforcement learning; these connections are largely unexplored.

Connections between neural SDEs, score matching diffusions, continuous normalising flows, optimal transport and Schrödinger bridges are still in their infancy.

Numerical methods Numerical methods for neural differential equations are the single largest chapter in this thesis, and with good reason. Numerical differential equation solvers are an old topic, but recent developments such as reversible solvers and hypersolvers offer opportunities yet to be exploited. For example it would be desirable to have higher order reversible ODE solvers, or to be able to apply hypersolvers during training.

Symbolic regression Symbolic regression – both the underlying techniques and their application to dynamical systems – is still more alchemy than science. That is, it is more a matter of 'seeing what sticks' than of applying guiding principles.

For dynamical systems, only SINDy and its variants are well-established. The development and application of more advanced techniques such as regularised evolution and deep symbolic regression remains almost entirely wide open.

Neural PDEs One topic, made conspicuous by its absence from this thesis, is the possibility of neural partial differential equations.

There have been a selection of ideas in this space. For example a convolutional network is roughly equivalent to the discretisation of a parabolic PDE. [Li+20b; Li+20c; Li+21] consider the 'Fourier Neural Operator', which is probably the most well-developed current theory for something approaching neural PDEs. [SLG21] present some initial thoughts on neural stochastic partial differential equations. This list of references is far from exhaustive.

In practice many of the ideas in this space have yet to converge. (Perhaps unsurprisingly: there are a great many types of PDE to consider, after all.) This represents a major open direction for the field of neural differential equations.

7.2 Thank you

Finally, it remains to thank the reader for their attention. We hope we have adequately conveyed some amount of insight (and our own enthusiasm) for this new, rapidly developing, and in our opinion highly exciting field of neural differential equations.

Neural differential equations sit at the intersection of arguably the two most successful modelling paradigms ever invented. In doing so, they demonstrate that these 'two' paradigms are perhaps much closer to *one* paradigm than at first glance we might imagine.

Appendix A

Review of Deep Learning

We expect that a reasonable proportion of our audience may come from a traditional mathematical modelling background, and may not be familiar with deep learning.

Whilst all sorts of concepts will be important at various points in the presentation, we will assume familiarity with the following elementary concepts throughout:

- Common neural architectures (i.e. differentiable computation graphs):
  - Feedforward networks;
  - Convolutional networks;
  - Recurrent networks, GRUs, LSTMs;
  - Residual networks;
  - Activation functions: ReLU, tanh, sigmoid, softplus;
  - Batch normalisation;
- Optimisation:
  - Maximum likelihood;
  - Stochastic gradient descent;
  - Batching;
  - Backpropagation;
  - Backpropagation through time for RNNS;
  - Weight regularisation;
  - Dropout;
- Supervised learning:
  - L^2 loss;
  - Softmax;

  - Cross-entropy;
  - Binary cross-entropy;
- Unsupervised learning:
  - (Wasserstein) generative adversarial networks;
  - Variational autoencoders;
  - Wasserstein distance;
  - KL divergence;

If the reader is indeed unfamiliar with deep learning, then we recommend either [Gér17] or [SAV20] for the necessary introductions to much of the above list. (At least at time of writing. Their discussion on the details of individual software frameworks may soon become out-of-date.)

The above list tends towards practical concerns. This thesis additionally assumes familiarity with a few other concepts, slightly more academic in nature. As these appear less frequently in introductory texts, then for readability's sake we provide an introduction to them now. The emphasis will be on brevity over completeness.

A.1 Autodifferentiation

Let $f_1, \ldots, f_n$ be some collection of functions whose derivatives we know how to compute. (Referred to as 'differentiable primitives'.)

Then for any composition of these functions $x \mapsto f(x) = f_{i_m}(\cdots(f_{i_1}(x))\cdots)$, with $i_1, \ldots, i_m \in \{1, \ldots, n\}$, we also know how to compute the derivative of f via the chain rule:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{i_{m-1}}} \cdots \frac{\mathrm{d}f_{i_2}}{\mathrm{d}f_{i_1}} \frac{\mathrm{d}f_{i_1}}{\mathrm{d}x}. \tag{A.1}$$

More generally one may consider any (topologically sorted) directed acyclic graph of compositions; we focus on the easy-to-present case.

Autodifferentiation frameworks offer an automated way to compute (A.1), by providing differentiable primitives such as matrix multiplies, sines, cosines, ReLUs, and so on.

It remains to consider how best to evaluate (A.1). There are two main approaches.

Forward-mode Forward-mode autodifferentiation, also known as forward sensitivity, proceeds by recursively computing

$$\frac{\mathrm{d}f_{i_q}}{\mathrm{d}x} = \frac{\mathrm{d}f_{i_q}}{\mathrm{d}f_{i_{q-1}}} \frac{\mathrm{d}f_{i_{q-1}}}{\mathrm{d}x} \tag{A.2}$$

for $q = 2, \ldots, m$.

Reverse-mode Reverse-mode autodifferentiation, also known as backpropagation or reverse sensitivity, proceeds by recursively computing

$$\frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{q-1}} = \frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{i_q}} \frac{\mathrm{d}f_{i_q}}{\mathrm{d}f_{q-1}} \tag{A.3}$$

for $q = m-1, \ldots, 1$, and for convenience denoting $x = f_0$.

Efficiency The main difference is computational efficiency. Suppose x is a vector and f_{i_m} outputs a scalar. Suppose all intermediate layers are vectors. (This is the common case for neural networks, with a vector of parameters as input and a scalar loss as output.) Then each evaluation of (A.2) is a matrix-matrix product, whilst each evaluation of (A.3) is only a vector-matrix product; this is substantially cheaper to compute. It is for this reason that backpropagation, not forward-mode autodifferentiation, is typically used to train neural networks.

We may more precisely characterise the above statement as follows. Let d_{input} be the dimensionality of x and let d_{output} be the dimensionality of the output of f_{i_m}. Then under a reasonable model of computation, the cost of computing both f and $\frac{\mathrm{d}f}{\mathrm{d}x}$ via forward-mode autodifferentiation may be upper bounded by $2.5\, d_{\text{input}}$ times the cost of evaluating just f [GW08, Equation (4.17)]. Computing both f and $\frac{\mathrm{d}f}{\mathrm{d}x}$ via reverse-mode autodifferentiation may be upper bounded by $4\, d_{\text{output}}$ times the cost of evaluating just the function [GW08, Equation (4.21)]. In each case we refer to 'computing both f and $\frac{\mathrm{d}f}{\mathrm{d}x}$' as computing $\frac{\mathrm{d}f}{\mathrm{d}x}$ typically relies on computing f first.

Jacobian-vector and vector-Jacobian products Consider again the case that f_{i_m} outputs a scalar, so that reverse-mode autodifferentiation computes a sequence of vector-matrix products. As the matrix is a Jacobian this is referred as a a vector-Jacobian product ('vjp'). Likewise if x is a scalar then forward-mode autodifferentiation computes a sequence of Jacobian-vector products ('jvp').

For these reasons, 'jvp' and 'vjp' are sometimes used as synonyms for forward- and reverse-mode autodifferentiation, even when x or f_{i_m} are not necessarily scalar.

Comparison Note that (A.2) may be evaluated *during* the 'forward' evaluation of $f(x) = f_{i_m}(\cdots(f_{i_1}(x))\cdots)$: just compute each $\frac{\mathrm{d}f_{i_{q-1}}}{\mathrm{d}f_1} \mapsto \frac{\mathrm{d}f_{i_q}}{\mathrm{d}f_1}$ alongside each $f_{i_{q-1}}(\cdots(f_{i_1}(x))\cdots) \mapsto f_{i_q}(\cdots(f_{i_1}(x))\cdots)$.

In contrast(A.3) must be evaluated *after* the 'forward' evaluation – we cannot evaluate $\frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_q}$, which is evaluated at $f_{i_{q-1}}(\cdots(f_{i_1}(x))\cdots)$, until $f_{i_{q-1}}(\cdots(f_{i_1}(x))\cdots)$ has been computed. As such all $f_{i_q}(\cdots(f_{i_1}(x))\cdots)$ must first be evaluated and then held in memory. The amount of space available in memory can become a concern.

The canonical reference text on autodifferentiation is [GW08].

A.2 Normalising flows

Fix $d \in \mathbb{N}$ and let $f\colon \mathbb{R}^d \to \mathbb{R}^d$ be bijective and sufficiently smooth.

Let X be some random variable taking values in $\mathbb{R}^d$, with density $p_X\colon \mathbb{R}^d \to [0,\infty)$. Let $Y = f(X)$. Then the change of variables formula gives that the density $p_Y\colon \mathbb{R}^d \to [0,\infty)$ of Y is

$$p_Y(y) = p_X(f^{-1}(y)) \left| \det \frac{\mathrm{d}f}{\mathrm{d}x}(f^{-1}(y)) \right|^{-1}.$$

Now let X be a multivariate normal, let Y correspond to (the empirical samples of) the data, and let $f = f_\theta$ to be some flexible neural network model. Consider training f_θ by maximum likelihood, by directly optimising

$$\max_\theta \mathbb{E}_{y\sim Y} \log p_Y(y) = \max_\theta \mathbb{E}_{y\sim Y} \left[\log p_X(f_\theta^{-1}(y)) - \log \left| \det \frac{\partial f}{\partial x}(f^{-1}(y)) \right| \right].$$

After training we obtain a generative model capable of producing approximate samples of Y: simply sample X then evaluate $f(X)$.

The snag is that training is computationally expensive: in general evaluating the (log-determinant-)Jacobian costs $\mathcal{O}(d^3)$. Correspondingly much of the literature has focused on finding neural architectures f_θ that (a) exhibit structure that may be exploited to cheapen the cost of the Jacobian computation, whilst (b) still being expressive enough to produce good models.

Normalising flows were introduced in [RM15]. We recommend [Kos18] for further introduction.

A.3 Universal approximation

'Universal approximation' is what a mathematician would call 'density'. That is, given some normed vector space[1] V and some set $W \subseteq V$, then W is said to exhibit universal approximation with respect to V if for all $\varepsilon > 0$ and $v \in V$, there exists $w \in W$ such that $\|v - w\| < \varepsilon$.

Let $K \subseteq \mathbb{R}^{d_x}$ be compact. It is often desirable to demonstrate that some set of neural networks $\mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$ exhibit universal approximation with respect to (typically) $C(K;\mathbb{R}^{d_y})$. As long as we have enough data, take a large enough network, and train for long enough – known as the 'infinite data limit' – then in principle we may hope to obtain an arbitrarily good approximation to the target function.

Most famously, the set of feedforward networks of arbitrary width is a universal approximator for the set of continuous functions. ('The' universal approximation theorem.)

[1] Or a topological space in general.

Definition A.1. *Let $\rho\colon \mathbb{R} \to \mathbb{R}$ be any continuous function. Then let $\mathcal{N}_d^\rho$ denote the set of feedforward neural networks with activation function ρ, with d neurons in the input layer, one neuron in the output layer, and a single hidden layer with an arbitrary number of neurons.*

Theorem A.2 (Universal Approximation Theorem [Pin99])**.** *Let $K \subseteq \mathbb{R}^d$ be compact. Then $\mathcal{N}_d^\rho$ is dense in $C(K)$ if and only if ρ is nonpolynomial.*

Many introductory texts repeat weaker versions of this theorem, apparently unaware that this simpler stronger version exists.[2]

Other variations on this theorem can also be found. Most notably, the set of feedforward networks of arbitrary *depth* is also a universal approximator for the set of continuous functions.

Definition A.3. *Let $\rho\colon \mathbb{R} \to \mathbb{R}$ and $d_x, d_y, d_w \in \mathbb{N}$. Then let $\mathcal{NN}_{d_x,d_y,d_w}^\rho$ denote the set of feedforward neural networks with d_x neurons in the input layer, d_y neurons in the output layer, and an arbitrary number of hidden layers of width d_w with activation function ρ.*

Theorem A.4 (Deep-and-Narrow Universal Approximation [KL20b])**.** *Let $\rho\colon \mathbb{R} \to \mathbb{R}$ be any nonaffine continuous function, which is continuously differentiable at at least one point, with nonzero derivative at that point. Let $K \subseteq \mathbb{R}^{d_x}$ be compact. Then $\mathcal{NN}_{d_x,d_y,d_x+d_y+2}^\rho$ is dense in $C(K; \mathbb{R}^{d_y})$.*

A.4 Irregular time series

Time series are often 'messy' or 'irregular'. Consider the space of d-dimensional irregularly-sampled time series

$$\left\{((t_0, x_0), \ldots, (t_n, x_n)) \;\middle|\; n \in \mathbb{N}, t_j \in \mathbb{R}, x_j \in (\mathbb{R} \cup \{*\})^d, t_j < t_{j+1}\right\},$$

where $*$ denotes the possibility of missing data.

A few practical things must be considered. (See also [Che+18a] for more discussion.)

Irregular sampling The choice of points t_j may not be the same for different time series in the dataset. The values of t_j may be informative, and generally we should concatenate (t_j, x_j) together before passing them to a model. (Sometimes the increments $(t_j - t_{j-1}, x_j)$ are used instead.)

Variable length The length n may not be the same for different time series in the dataset. This can affect how easy it is to batch different time series together.

[2] And in fact slightly stronger (but slightly more complex) versions than the one we have stated here also exist; see [Les+93].

Missing data Each observation $x_j \in (\mathbb{R} \cup \{*\})^d$ may have missing data.

Some texts suggest 'imputing' missing data: that is, filling in any missing data in some sensible way prior to applying a model. Despite its popularity this is frequently the wrong thing to do: that the data was missing is information that has been lost. In general, whether the data was missing may itself be informative.

It is better to fix a vector space V and an injective map $(\mathbb{R} \cup \{*\})^d \to V$, and then apply this map to every x_j prior to applying the model. That this is injective means no information is lost. That it maps into a vector space means that the result is in a form the model can use.

A frequent choice is $V = \mathbb{R}^{2d}$ and $(z_1, \ldots, z_d) \mapsto (\phi(z_1), \ldots, \phi(z_d), \psi(z_1), \ldots, \psi(z_d))$, where

$$\begin{aligned}
&\phi\colon \mathbb{R} \cup \{*\} \to \mathbb{R},\\
&\phi\colon z \mapsto \begin{cases} 0 & \text{if } z = *,\\ z & \text{if } z \in \mathbb{R}, \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\psi\colon \mathbb{R} \cup \{*\} \to \mathbb{R},\\
&\psi\colon z \mapsto \begin{cases} 0 & \text{if } z = *,\\ 1 & \text{if } z \in \mathbb{R}. \end{cases}
\end{aligned}$$

The map ψ is sometimes referred to as a 'mask', 'observational intensity', or similar.

A.5 Miscellanea

Maximum mean discrepancy The maximum mean discrepancy (MMD) is a (pseudo)distance between probability distributions. Let $\mathcal{X}$ be some set and let $\phi\colon \mathcal{X} \to \mathbb{R}^d$ be fixed. Let $\mathbb{P}$, $\mathbb{Q}$ be two probability distributions over $\mathcal{X}$. Then the MMD between $\mathbb{P}$ and $\mathbb{Q}$ is defined to be

$$d(\mathbb{P}, \mathbb{Q}) = \|\mathbb{E}_{x \sim \mathbb{P}}[\phi(x)] - \mathbb{E}_{x \sim \mathbb{Q}}[\phi(x)]\| \tag{A.4}$$

for any fixed choice of norm $\|\cdot\|$ on $\mathbb{R}^d$.

Like the KL divergence or the Wasserstein distance, this is a popular optimisation criterion when fitting generative models.

This may be extended from a pseudodistance (in which $d(\mathbb{P}, \mathbb{Q}) = 0$ need not imply $\mathbb{P} = \mathbb{Q}$) to a true distance by replacing $\mathbb{R}^d$ with some infinite-dimensional Hilbert space.

A biased estimate of equation (A.4) may be obtained by estimating each expectation with N Monte-Carlo samples. This takes $\mathcal{O}(N)$ work to evaluate. An unbiased

estimate may be obtained by taking $\|\cdot\| = \|\cdot\|_2$, and squaring and expanding (A.4). This produces nested expectations that take $\mathcal{O}(N^2)$ work to evaluate via Monte-Carlo.

MMDs were introduced in [Gre+13]. The KID criterion for evaluating quality of generative image models is an example of an MMD [Bińkowski+18].

The manifold hypothesis Consider the dataset of all possible pictures of cats. (The 'underlying' dataset from which in practice we observe some finite collection of samples.) For example each image may be a point in $\mathbb{R}^{3\times32\times32}$, corresponding to (red, green, blue) channels and 32×32 pixels.

This is a very high-dimensional space, and it is clear that the majority of this space is of all kinds of pictures, other than of cats. Indeed most points in this space will resemble random noise. Our dataset covers only some small region of the overall space.

'The manifold hypothesis' is the informal statement that most datasets tend to behave in this way: that if you were to zoom out and squint at them, they would look a bit like a low-dimensional manifold embedded in this higher-dimensional space.

We could not find a good reference introducing the manifold hypothesis; it appears to simply be part of the folklore.

SiLU activation function The SiLU activation function (also known as 'swish') is defined as $x \mapsto x\sigma(x)$, where σ is the sigmoid activation function. It is a popular activation function sometimes held to produce slightly better results than traditional options such as the ReLU [HG16; EUD17; RZL17].

Appendix B

Neural Rough Differential Equations

This appendix follows from the introduction given in Section 3.2.3; the material is from [Mor+21b]. Here, we will apply neural CDEs to long time series, which are a regime in which both neural CDEs and RNNs tend to break down.

The key idea will be to solve a CDE by taking very large integration steps – much larger than the sampling rate of the data – whilst incorporating sub-step information through additional terms in the numerical solver, through what is known as the *log-ODE method*.

A CDE treated in this way is termed a *rough differential equation*, in the sense of rough path theory. Correspondingly we refer to this approach as either a *neural rough differential equation*, or simply 'the log-ODE method applied to neural CDEs'.

B.1 Background

We begin with some necessary background on rough path theory.

B.1.1 Signatures and logsignatures

B.1.1.1 Signatures

Let $x = (x_1, \ldots, x_{d_x})\colon [a, b] \to \mathbb{R}^{d_x}$ be continuous and of bounded variation. Weaker conditions may also be admitted, but this will suffice for our purposes here.

Define the iterated Riemann–Stieltjes integrals

$$S_{a,b}^{k_1,\ldots,k_m}(x) = \int_{a<t_1<\cdots<t_m<b} \cdots \int \mathrm{d}x_{k_1}(t_1) \cdots \mathrm{d}x_{k_m}(t_m) \in \mathbb{R}, \tag{B.1}$$

and, up to some maximal index $M \in \mathbb{N}$, put all such integrals together into a single object:

$$\mathrm{sig}_{a,b}^{M}(x) = \left(1, \{S_{a,b}^{k}(x)\}_{k=1}^{d}, \{S_{a,b}^{k_1,k_2}(x)\}_{k_1,k_2=1}^{d}, \ldots, \{S_{a,b}^{k_1,\ldots,k_M}(x)\}_{k_1,\ldots,k_M=1}^{d}\right). \quad \text{(B.2)}$$

By convention $1 \in \mathbb{R}$ is also included at the start.

Then $\mathrm{sig}_{a,b}^{M}(x)$ is known as the *depth-M signature transform of x*. (Or similar variations on this theme, like 'M-step signature of x'.)

B.1.1.2 Signatures as Taylor expansions

Signatures are interesting because they appear in the Taylor expansion of a controlled differential equation. Let y solve a CDE with vector field f, driven by x. Then in Einstein notation over indices k_1, k_2, k_3, k_4,

$$\begin{aligned}
y_{k_1}(t) &= y_{k_1}(a) + \int_a^t f_{k_1,k_2}(y(s))\,\mathrm{d}x_{k_2}(s) \\
&= y_{k_1}(a) + \int_a^t \Big(f_{k_1,k_2}(y(a)) \\
&\qquad + \frac{\partial f_{k_1,k_2}}{\partial y_{k_3}}(y(a))(y_{k_3}(s) - y_{k_3}(a)) + \mathcal{O}((t-a)^2)\Big)\mathrm{d}x_{k_2}(s) \\
&= y_{k_1}(a) + f_{k_1,k_2}(y(a)) \int_a^t \mathrm{d}x_{k_2}(s) \\
&\qquad + \frac{\partial f_{k_1,k_2}}{\partial y_{k_3}}(y(a)) \int_a^t (y_{k_3}(s) - y_{k_3}(a))\,\mathrm{d}x_{k_2}(s) + \mathcal{O}((t-a)^3) \\
&= y_{k_1}(a) + f_{k_1,k_2}(y(a)) \int_a^t \mathrm{d}x_{k_2}(s) \\
&\qquad + \frac{\partial f_{k_1,k_2}}{\partial y_{k_3}}(y(a)) \int_a^t \int_a^s f_{k_3,k_4}(y(s))\,\mathrm{d}x_{k_4}(u)\,\mathrm{d}x_{k_2}(s) + \mathcal{O}((t-a)^3) \\
&= y_{k_1}(a) + f_{k_1,k_2}(y(a)) \int_a^t \mathrm{d}x_{k_2}(s) \\
&\qquad + \frac{\partial f_{k_1,k_2}}{\partial y_{k_3}}(y(a)) f_{k_3,k_4}(y(a)) \int_a^t \int_a^s \mathrm{d}x_{k_4}(u)\,\mathrm{d}x_{k_2}(s) + \mathcal{O}((t-a)^3) \\
&= y_{k_1}(a) + f_{k_1,k_2}(y(a)) S_{a,t}^{k_2}(x) + \frac{\partial f_{k_1,k_2}}{\partial y_{k_3}}(y(a)) f_{k_3,k_4}(y(a)) S_{a,t}^{k_4,k_2}(x) + \mathcal{O}((t-a)^3).
\end{aligned} \quad \text{(B.3)}$$

The right hand side is an affine combination of terms in the signature. If higher order terms had been taken in the Taylor expansions of f, then higher orders in the in the signature would have appeared on the right hand side.

This property means that the signature may be used to produce a good approximation to the solution of the CDE. Intuitively, over small time scales, the signature extracts the information 'most important' to solving the CDE.

Remark B.1. *Independent of neural CDEs, there has also been a line of work investigating the use of the signature transform as a feature extractor for time series [CK16; Kid+19; Mor+20].*

B.1.1.3 Logsignatures

The signature has some redundancy. For example a little algebra shows that $S^{1,2}_{a,b}(x) + S^{2,1}_{a,b}(x) = S^1_{a,b}(x)S^2_{a,b}(x)$, so that we already know any one of these quantities given the other three.

Definition B.2 (Lyndon word)**.** *Let $q \in \mathbb{N}$ and consider some set $\mathcal{A} = \{a_1, \ldots, a_q\}$, which we refer to as an alphabet. A* word *in this alphabet is any finite-length sequence of elements of $\mathcal{A}$, for example $a_2a_3a_1a_4$. A* Lyndon word *is any word which occurs lexicographically strictly earlier than any word obtained by cyclically rotating its elements. For example, $a_2a_2a_3a_4$ is a Lyndon word as it occurs strictly earlier than $a_2a_3a_4a_2$ or $a_3a_4a_2a_2$ or $a_4a_2a_2a_3$, whilst a_1a_1 is not a Lyndon word as it does not occur strictly earlier than a_1a_1 (which is a rotation).*

The logsignature transform is obtained by computing the signature transform, and throwing out redundant terms to produce some minimal collection. This choice of minimal collection is nonunique. One computationally efficient choice is to retain precisely those terms $S^{k_1,\ldots,k_m}_{a,b}(x)$ for which $k_1 \cdots k_m$ is a Lyndon word over the alphabet $\{1, \ldots, d_x\}$. This is introduced in [KL21] (and is confusingly a distinct notion from the 'Lyndon basis', which is one of the other nonunique choices).

By fixing such a procedure – via Lyndon words or otherwise – we obtain the *depth-M logsignature transform of x*, denoted $\mathrm{logsig}^M_{a,b}(x) \in \mathbb{R}^{\beta(d_x, M)}$, with

$$\beta(d, M) = \sum_{k=1}^{M} \frac{1}{k} \sum_{j|k} \mu\left(\frac{k}{j}\right) d^j$$

where μ is the Möbius function.

See [Rei17; Rei18] for further introduction to the logsignature transform.

Geometric interpretation The first two $m = 1, 2$ levels of the logsignature have geometric interpretations. The depth 1 terms are simply the increments of the path. The depth 2 is the signed (Lévy) area between the path and the chord joining its endpoints; equivalently this corresponds to a notion of order. See Figure B.1.

Higher terms in the logsignature correspond to 'repeated areas' and are not so easily visualised.

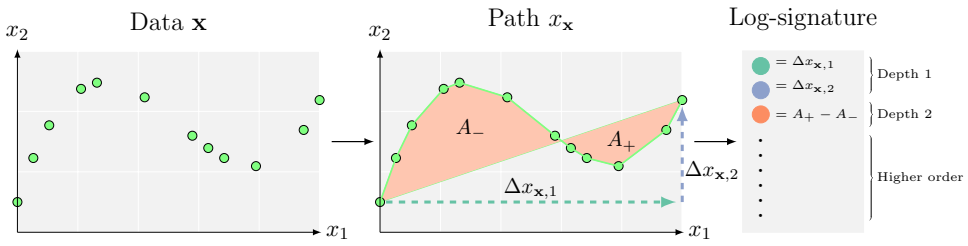


Figure B.1: Geometric intuition for the first two levels of the logsignature for a 2-dimensional path. The depth-1 terms correspond to the change in each of the coordinates over the interval. The depth-2 term corresponds to the *Lévy area* of the path, this being the signed area between the curve and the chord joining its start and endpoints.

Order interpretation Consider just the region A_- in Figure B.1. Progressing from left to right, the green curve shown makes a large change in x_2, followed by a change in x_1, and correspondingly the initial part of this curve incurs a negative signed area A_-. If the order of these changes had been reversed then a positive signed area would have been accumulated instead. Likewise, applying the same procedure to clockwise and anticlockwise spirals would have produced areas of different signs.

As such (log)signatures – including the higher order terms in (log)signatures – encode information by capturing a notion of *order of events*.

Now recall that the universal approximation theorem for CDEs (Theorem 3.9) is proven by reducing CDEs to signatures, whilst Section 3.2.2 reduces RNNs to CDEs. This brings us full circle, as RNNs are a model predicated upon the assumption that the order of inputs matter.

This appears to be the fundamental difference between RNNs and Transformers [Vas+17]. RNNs are predicated on assuming order is important to the data; Transformers are predicated on assuming that order is (mostly) unimportant. This is reflected in their typical use cases. RNNs are often preferred for time series, whilst Transformers are often preferred in natural language processing.

See also [TBO21], who make concrete some of these (relatively abstract) notions about order.

B.1.2 The log-ODE method

Let $f\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be Lipschitz. Let $x\colon [0,T] \to \mathbb{R}^{d_x}$ be of bounded variation. Let $y_0 \in \mathbb{R}^{d_y}$. Consider $y\colon [0,T] \to \mathbb{R}^{d_y}$ satisfying the CDE

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s) \quad \text{for } t \in (0,T].$$

Then the log-ODE method states that for all $M \in \mathbb{N}$ there exists some $\widehat{f}_M\colon \mathbb{R}^{d_y} \to$

$\mathbb{R}^{d_y \times \beta(d_x, M)}$ such that $\widehat{y}(T) \to y(T)$ as $M \to \infty$, where $\widehat{y}$ solves the ODE

$$\widehat{y}(0) = y_0, \qquad \widehat{y}(t) = \widehat{y}(0) + \int_0^t \widehat{f}_m(\widehat{y}(s)) \frac{\operatorname{logsig}^M_{0,T}(x)}{T} \, \mathrm{d}s. \tag{B.4}$$

The right hand side denotes a matrix-vector product between $\widehat{f}_M(\widehat{y}(s)) \in \mathbb{R}^{d_y \times \beta(d_x, M)}$ and $\operatorname{logsig}^M_{0,T}(x) \in \mathbb{R}^{\beta(d_x, M)}$.

The exact form of $\widehat{f}_M$ is actually known, but is expensive to compute. For this reason we will soon circumvent the need for this computation.

Example B.3. *Consider when $M = 1$. Here in fact $\widehat{f}_1 = f$. Suppose T is small. Then equations* (B.1) *and* (B.2) *imply*

$$\frac{\operatorname{logsig}^1_{0,T}(x)}{T} = \frac{x(T) - x(0)}{T} \approx \frac{\mathrm{d}x}{\mathrm{d}t}(0).$$

As such (B.4) *is an approximation to equation* (3.3)*. If x is piecewise linear then this approximation is exact, and the depth-1 log-ODE method is identical to equation* (3.3)*.*

B.2 Neural vector fields

We begin with the usual set-up for neural CDEs applied to potentially irregular time series, as in Sections 3.2.1 and 3.5.

We assume observations of a time series $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n))$ with $t_j \in \mathbb{R}$ the timestamp for the observation $x_j \in (\mathbb{R} \cup \{*\})^{d_x - 1}$, and $*$ denotes the possibility of missing data, and $t_0 < \cdots < t_n$.

Let $\mathbf{x} \mapsto x_{\mathbf{x}}$ be an interpolation scheme. We additionally require that each $x_{\mathbf{x}}$ be a continuous piecewise linear function. (So that either linear interpolation or rectilinear interpolation, see Section 3.5, would suffice.)

Let $f_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be any (Lipschitz) neural network depending on parameters θ. The value $d_y \in \mathbb{N}$ is a hyperparameter describing the size of the hidden state. Let $\zeta_\theta \colon \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$ be any neural network depending on the parameters θ.

A neural controlled differential equation was defined as the solution of the CDE

$$y(t) = y(0) + \int_0^t f_\theta(y(s)) \, \mathrm{d}x(s) \quad \text{for } t \in (0, T],$$

where $y(0) = \zeta_\theta(x(0))$.

This was (typically) solved by reducing the CDE to an ODE, as in equations (3.5) and (3.6). We reproduce equations (3.5) and (3.6) here: let

$$g_{\theta,x}(y, s) = f_\theta(y) \frac{\mathrm{d}x}{\mathrm{d}s}(s), \tag{3.5 revisited}$$

so that for $t \in (0, T]$,

$$\begin{aligned} y(t) &= y(0) + \int_0^t f_\theta(y(s))\,\mathrm{d}x(s) \\ &= y(0) + \int_0^t f_\theta(y(s))\frac{\mathrm{d}x}{\mathrm{d}s}(s)\,\mathrm{d}s \\ &= y(0) + \int_0^t g_{\theta,x}(y(s), s)\,\mathrm{d}s. \end{aligned} \tag{3.6 revisited}$$

B.2.1 Applying the log-ODE method

Pick points r_j such that $t_0 = r_0 < r_1 < \cdots < r_m = t_n$. In principle these can be variably spaced but in practice we will typically space them equally far apart. The number of points m should be chosen much smaller than n, that is to say $m \ll n$. The number and spacing of r_j is a hyperparameter.

We also pick a logsignature depth hyperparameter $M \geq 1$.

We now replace (3.5) with the piecewise

$$g_{\theta,x}(y, s) = f_\theta(y)\frac{\operatorname{logsig}^M_{r_j, r_{j+1}}(x)}{r_{j+1} - r_j} \quad \text{for } s \in [r_j, r_{j+1}), \tag{B.5}$$

where $f_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times \beta(d_x, M)}$ is some neural network, $\operatorname{logsig}^M_{r_j, r_{j+1}}(x) \in \mathbb{R}^{\beta(d_x, M)}$, and the right hand side denotes a matrix-vector product.

Equation (3.6) remains unchanged:

$$y(t) = y(0) + \int_0^t g_{\theta,x}(y(s), s)\,\mathrm{d}s.$$

This is an alternate method by which a CDE may be reduced to an ODE. This may now be solved as a (neural) ODE using standard ODE solvers.

An overview of this process is shown in Figure B.2.

Remark B.4. *These two approaches are intuitively similar. The quotient*

$$\frac{\operatorname{logsig}^M_{r_j, r_{j+1}}(x)}{r_{j+1} - r_j}$$

is roughly equivalent to the difference quotient

$$\frac{x_{\mathbf{x}}(r_{j+1}) - x_{\mathbf{x}}(r_j)}{r_{j+1} - r_j} \approx \frac{\mathrm{d}x_{\mathbf{x}}}{\mathrm{d}t}(r_j).$$

Remark B.5. *A continuous piecewise linear interpolation $x_{\mathbf{x}}$ is chosen as these are the only paths for which efficient algorithms for computing the (log)signature are known [KL21].*

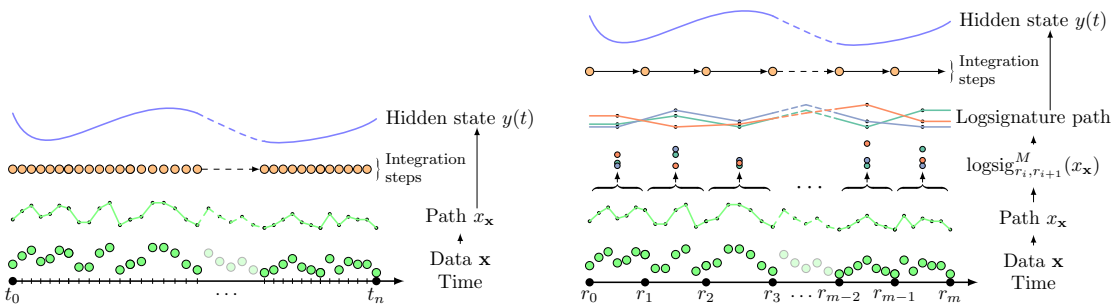


Figure B.2: An overview of the log-ODE method. **Left:** The CDE approach as introduced in Chapter 3. The path x is quickly varying, so that many integration steps may be needed to resolve it. **Right:** The log-ODE method with integration steps larger than the discretisation of the data. The path of logsignatures is more slowly varying (in a higher dimensional space), and needs fewer integration steps to resolve.

B.2.2 Discussion

Length/channel trade-off The sequence of logsignatures is now of length $m \ll n$. As such, it is much more slowly varying over the interval $[t_0, t_n]$ than the original data, which was of length n. Correspondingly the differential equation (B.5) is better-behaved than the original (3.6), and so larger integration steps may be used in the numerical solver. This is the source of the speed-ups of this method; we observe typical speed-ups by a factor of about ten.

Ease of implementation Note that (B.5) is of precisely the same form as (3.6), with the driving path taken to be piecewise linear in logsignature space.

Correspondingly the log-ODE method may be implemented by preprocessing the data into logsignatures, calculated over each window $[r_j, r_{j+1})$, interpolating the sequence of logsignatures into a piecewise linear path, and then solving a neural CDE as normal.

Every step in this procedure already exists as a software library, see [Kid20].

The log-ODE method as a binning procedure The interpretation of the previous heading draws an important connection to machine learning: the logsignature may be treated as a carefully-selected binning method, to reduce the amount of data considered whilst retaining the information most important for solving a CDE.

Modelling the vector field We avoided modelling some $f_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ and then computing some '$\widehat{f_{\theta M}}$', and instead modelled an $f_\theta \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times \beta(d_x, N)}$ directly. Doing so avoids the computational expensive of computing some '$\widehat{f_{\theta M}}$'.

Depth and step hyperparameters To solve a neural RDE accurately via the log-ODE method, we should be prepared to take the depth M suitably large, or the intervals $r_{j+1} - r_j$ suitably small. In practice accomplishing this would require that these are taken infeasibly large or small, respectively. Instead, we treat these as hyperparameters. This makes the use of the log-ODE method a modelling choice rather than an implementation detail. This is a baked-in discretisation as in Section 5.3.1.1.

Increasing step size will lead to faster (but less informative) training by reducing the number of operations. Increasing depth will lead to slower (but more informative) training, as more information about each local interval is used in each update.

B.2.3 Efficacy on long time series

In principle the log-ODE method may be applied when applying neural CDEs in any context. However it is particularly helpful when applied to long time series.

Improved gradients It is a classical fact that it is relatively difficult to train RNNs (and, as they are of essentially the same character, neural CDEs) directly on long time series. During training RNNs may suffer from vanishing/exploding gradients, reducing overall model performance. See also Remark 3.15, which discusses the exponential decay of hidden state over time.

Reducing the length of the time series, as with the log-ODE method, is a simple and effective way to combat this.

Computational efficiency The sheer number of operations required to process a long time series implies a long computation time.

This is aggravated by the fact that the inherently serial nature of a neural CDE or RNN, working its way along the time series, is almost impossible to parallelise. (Section 6.3.3 notwithstanding.) In contrast a large number of channels in the time series is much less of an issue, due to the availability of parallelism.

Once again, reducing the length of the time series helps combat this. That this is performed via preprocessing is particularly beneficial for training: the computation of logsignatures need only be done once prior to training.

Memory efficiency Long time series consume substantial memory in order to perform backpropagation-through-time. As discussed in Section 3.1.5.3, CDEs offer an attractive way to handle this through the availability of continuous adjoint methods, which consume only $\mathcal{O}(H + n)$ memory in the network size H and the input length n. The log-ODE method further improves upon this by reducing the memory cost to $\mathcal{O}(H + m)$ with $m \ll n$.

B.2.4 Limitations

Number of hyperparameters Two new hyperparameters – truncation depth and step size – with substantial effects on training time and memory usage must now also be tuned.

Number of input channels The log-ODE method is most feasible with few input channels, as the number of log-signature channels $\beta(d, M)$ grows exponentially in d. For larger d then the available parallelism may become saturated.

B.3 Examples

Datasets We apply neural RDEs to three real-world datasets from the TSR archive [Tan+20], coming originally from the Beth Israel Deaconess Medical Centre (BIDMC).

The goal is to predict a person's respiratory rate (RR), heart rate (HR), or oxygen saturation (SpO2) at the end of the sample, having observed photoplethysmography (PPG) and electrocardiogram (ECG) data over the length of the sample. The data is regularly sampled at 125Hz and each series has length 4 000. There are 3 channels (including time). Performance is evaluated using the L^2 loss.

Every problem was chosen for its long length. The lengths are sufficiently long that optimise-then-discretise backpropagation (Section 5.2) was needed simply to avoid running out of memory at any reasonable batch size.[1]

Models The logsignature depth M is varied over $\{2, 3\}$. ($M = 1$ is identical to the standard neural CDE as per Example B.3.) Likewise the number of observations within each interval $[r_j, r_{j+1}]$ were varied over $\{8, 128, 512\}$, which we refer to as the step size. In practice both depth and step size should be chosen as hyperparameters.

Two baseline models are also included. The first is a neural CDE; as the model we are extending then comparisons to this are our primary concern. A baseline against the ODE-RNN of [RCD19] is also included. For the neural CDE model, increased step sizes correspond to naïve subsampling of the data. For the ODE-RNN model, the time dimension is instead folded into the feature dimension, so that at each step the ODE-RNN model sees several adjacent time points; this is an alternate technique for dealing with long time series.

Results The results are shown in table B.1.

We find that the depth-3 neural RDE is the top performer for every task at every step size, reducing test loss by 30–59% compared to the corresponding neural CDE.

[1] Reversible solvers should/could also have been employed.

Model	Step size	L^2			Time (Hrs)			Memory (Mb)
		RR	HR	SpO_2	RR	HR	SpO_2	
ODE-RNN (folded)	1	–	13.06 ± 0.0	–	–	10.5	–	3653.0
	8	2.47 ± 0.35	13.06 ± 0.00	3.3 ± 0.00	1.5	1.2	0.9	917.2
	128	1.62 ± 0.07	13.06 ± 0.00	3.3 ± 0.00	0.2	0.1	0.1	81.9
	512	1.66 ± 0.06	6.75 ± 0.9	1.98 ± 0.31	0.0	0.1	0.1	40.4
NCDE	1	2.79 ± 0.04	9.82 ± 0.34	2.83 ± 0.27	23.8	22.1	28.1	56.5
	8	2.80 ± 0.06	10.72 ± 0.24	3.43 ± 0.17	3.0	2.6	4.8	14.3
	128	2.64 ± 0.18	11.98 ± 0.37	2.86 ± 0.04	0.2	0.2	0.3	8.7
	512	2.53 ± 0.03	12.22 ± 0.11	2.98 ± 0.04	0.1	0.0	0.1	8.4
NRDE (depth 2)	8	2.63 ± 0.12	8.63 ± 0.24	2.88 ± 0.15	2.1	3.4	3.3	21.8
	128	1.86 ± 0.03	6.77 ± 0.42	1.95 ± 0.18	0.3	0.4	0.7	10.9
	512	1.81 ± 0.02	5.05 ± 0.23	2.17 ± 0.18	0.1	0.2	0.4	10.3
NRDE (depth 3)	8	**2.42 ± 0.19**	**7.67 ± 0.40**	**2.55 ± 0.13**	2.9	3.2	3.1	43.3
	128	**1.51 ± 0.08**	**2.97 ± 0.45***	**1.37 ± 0.22**	0.5	1.7	1.7	17.3
	512	**1.49 ± 0.08***	**3.46 ± 0.13**	**1.29 ± 0.15***	0.3	0.4	0.4	15.4

Table B.1: Experiments on the three BIDMC datasets. Mean ± standard deviation of test set L^2 loss, measured over three repeats, over each of three different vital signs prediction tasks (RR, HR, SpO_2). Also reported are memory usage and training time. Only mean times are shown for space. '–' denotes that the model could not be run within GPU memory. Bold denotes the best model score for a given step size, and * denotes that the score was the best achieved over all models and step sizes.

Moreover, it does so with roughly an order of magnitude less training time. The ODE-RNN baseline produces poor results whilst requiring significantly more memory.

We attribute the improved test loss to the neural RDE model being better able to learn long-term dependencies due to the reduced sequence length: the performance of the rough models actually improves as the step size is increased.

Details of hyperparameter selection, optimisers, normalisation, and so on can be found in Appendix D.6. Additional results can be found in [Mor+21b, Appendix D].

B.4 Comments

The log-ODE method is a classic approach in CDEs and rough path theory; see for example [Lyo04, Section 7]. A great many standard numerical SDE solvers – such as the Euler–Maruyama method or Heun's method – are obtained as ODE solvers applied to the depth-1 log-ODE method. Higher-order numerical SDE solvers – such as Milstein's method – are almost (but not exactly) equivalent to applying an ODE solver to the depth-2 log-ODE method.

An excellent brief introduction to signatures and logsignatures are provided by [Rei17; Rei18]. An efficient computational implementation of signatures and logsignatures is provided by [KL21].

Neural rough differential equations were introduced in [Mor+21b], which is also where their application to long time series was considered. In particular see the appendices of [Mor+21b] (and references therein) for further mathematical details beyond those presented here, such as the convergence of the log-ODE method.

Appendix C

Proofs and Algorithms

C.1 Augmented neural ODEs are universal approximators even when their vector fields are not universal approximators

Recall Theorem 2.13.

Theorem 2.13. *Fix $d, d_o \in \mathbb{N}$. For $d_l \in \mathbb{N}$, $f \in C(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, $\ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l})$, $\ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o})$, let $\phi_{p,f,\ell_1,\ell_2} \colon \mathbb{R}^d \to \mathbb{R}^{d_o}$ be the map $x \mapsto z$ with*

$$y(0) = \ell_1(x), \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)) \quad \text{for } t \in [0, T], \qquad z = \ell_2(y(T))$$

for those f for which the solution is unique.[1]

For each $d_l \in \mathbb{N}$ there exists an $f_{d_l} \in C(\mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, for which the above equation has a unique solution, such that

$$\left\{ \phi_{d_l, f_{d_l}, \ell^1, \ell^2} \;\middle|\; d_l \in \mathbb{N}, \ell_1 \in L_b(\mathbb{R}^d; \mathbb{R}^{d_l}), \ell_2 \in L_b(\mathbb{R}^{d_l}; \mathbb{R}^{d_o}) \right\}$$

is a universal approximator for $C(\mathbb{R}^d; \mathbb{R}^{d_o})$.

Proof. Given some $x \in \mathbb{R}^d$, consider the system of ODEs $y_0, \ldots, y_M$ solving

$$\begin{aligned} y_0(0) = x \in \mathbb{R}^d \quad & \frac{\mathrm{d}y_0}{\mathrm{d}t}(t) = 0, \\ y_1(0) = 0 \in \mathbb{R}^{d \times d} \quad & \frac{\mathrm{d}y_1}{\mathrm{d}t}(t) = y_0(t) \otimes y_0(t), \\ y_2(0) = 0 \in \mathbb{R}^{d \times d \times d} \quad & \frac{\mathrm{d}y_2}{\mathrm{d}t}(t) = y_1(t) \otimes y_0(t), \end{aligned}$$

[1] The Peano existence theorem implies existence as f is continuous; but as f is not necessarily Lipschitz then the stronger Picard existence theorem, which gives uniqueness, does not apply.

$$\begin{aligned} y_3(0) = 0 \in \mathbb{R}^{d\times d\times d\times d} \quad & \frac{\mathrm{d}y_3}{\mathrm{d}t}(t) = y_2(t) \otimes y_0(t), \\ & \cdots \\ y_M(0) = 0 \in \mathbb{R}^{d\times\cdots\times d} \quad & \frac{\mathrm{d}y_M}{\mathrm{d}t}(t) = y_{M-1}(t) \otimes y_0(t). \end{aligned} \tag{C.1}$$

The solution may be written down immediately:

$$\begin{aligned} y_0(t) &= x, \\ y_1(t) &= \int_0^t x \otimes x \,\mathrm{d}s = tx \otimes x, \\ y_2(t) &= \int_0^t sx \otimes x \,\mathrm{d}s = \frac{1}{2}t^2 x^{\otimes 3}, \\ y_3(t) &= \int_0^t \frac{1}{2}s^2 x^{\otimes 3} \otimes x \,\mathrm{d}s = \frac{1}{6}t^3 x^{\otimes 4}, \\ &\cdots \\ y_M(t) &= \frac{1}{M!}t^M x^{\otimes(M+1)}. \end{aligned}$$

Evaluating at $t = 1$ we obtain the collection of all (scaled) monomials in $x \in \mathbb{R}^d$ up to degree $M + 1$, namely

$$\left(x, x^{\otimes x}, \frac{1}{2!}x^{\otimes 3}, \frac{1}{3!}x^{\otimes 4}, \ldots, \frac{1}{M!}x^{\otimes(M+1)}\right).$$

The Stone–Weierstrass theorem states that polynomials are dense in the space of continuous functions $C(K; \mathbb{R}^{d_o})$. Thus for any target $F \in C(K; \mathbb{R}^{d_o})$ and $\varepsilon > 0$, there exists some $M \in \mathbb{N}$ large enough, and some affine map combining these monomials to form a polynomial P, such that

$$\|F - P\|_{\infty,K} < \infty.$$

The result is now proved. For each $d_l = \sum_{k=1}^{M+1} d^k$ let f_{d_l} be the vector field specified in equation (C.1). Let each ℓ_1 be the affine map augmenting x with sufficient zeros for the initial condition. Let ℓ_2 be the affine map transforming the monomials to form any given polynomial. □

C.1.1 Comments

It is perhaps a little questionable whether the construction shown here is truly a 'neural ODE'. The only learnt parameters are in the final affine ℓ_2. More subtly, the equation of (C.1) are questionably ODEs: the vector field for each y_k does not depend on y_k (only y_{k-1} and y_0), and is thus 'only' an integral.

On the other hand, this is still essentially the same argument for universal approximation as for wide neural networks ([Pin99]) or a Fourier series – that is, a linear combination of enough terms – so perhaps we should not complain.

This result is actually a special case of the universal approximation theorem for CDEs (Appendix C.2.1). Given the input $x \in \mathbb{R}^d$, define the continuous path $z\colon [0, T] \to \mathbb{R}^d$ by $z(t) = tx$. Then the proof here is just a simplification and particular application of that result.

C.2 Theoretical properties of neural CDEs

C.2.1 Neural CDEs are universal approximators

We begin with universal approximation of CDEs with respect to continuous paths X. We then show how to extend this to universal approximation with respect to time series, in a generic way independent of the choice of interpolation, by requiring that the interpolation satisfy certain conditions.

C.2.1.1 Universal approximation with respect to paths

Definition C.1. *Let $T > 0$ and let $d \in \mathbb{N}$. Let*

$$\mathcal{V}^1([0, T]; \mathbb{R}^d) = C([0, T]; \mathbb{R}^d) \cap \mathrm{BV}([0, T]; \mathbb{R}^d)$$

represent the space of continuous functions of bounded variation. Equip this space with the norm

$$x \mapsto \|x\|_{\mathcal{V}} = \|x\|_\infty + |X|_{\mathrm{BV}}\,.$$

Remark C.2. *This is a somewhat unusual norm to use, as bounded variation seminorms are more closely aligned with L^1 norms than L^∞ norms.*

Definition C.3. *Let $\mathcal{V}^1_0([0, T]; \mathbb{R}^d) = \left\{x \in \mathcal{V}^1([0, T]; \mathbb{R}^d) \;\middle|\; x(0) = 0\right\}$.*

Definition C.4. *For $f \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times d_x})$, $\zeta \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y})$, $x \in \mathcal{V}^1([0, T]; \mathbb{R}^{d_x})$, let $y_{f,\zeta,x}\colon [0, T] \to \mathbb{R}^{d_y}$ denote the unique solution to the CDE*

$$y_{f,\zeta,x}(t) = y_{f,\zeta,x}(0) + \int_0^t f(y_{f,\zeta,x}(s))\,\mathrm{d}x(s) \quad \text{for } t \in (0, T],$$

with $y_{f,\zeta,x}(0) = \zeta(x(0))$.

Definition C.5. *For any $d, M \in \mathbb{N}$, let $\kappa(d, M) = \sum_{i=0}^M d^i$.*

Definition C.6 (Signature transform)**.** *Let $x \in \mathcal{V}^1([0, T]; \mathbb{R}^d)$. Define the iterated Riemann–Stieltjes integrals*

$$S^{k_1,\ldots,k_m}_{a,b}(x) = \int\limits_{a<t_1<\cdots<t_m<b} \cdots \int \mathrm{d}x_{k_1}(t_1) \cdots \mathrm{d}x_{k_m}(t_m) \in \mathbb{R}.$$

Let $M \in \mathbb{N}$. Put all such integrals, up to maximal index M, together into a single object:

$$\mathrm{sig}_{a,b}^{M}(x) = \left(1, \{S_{a,b}^{k}(x)\}_{k=1}^{d}, \{S_{a,b}^{k_1,k_2}(x)\}_{k_1,k_2=1}^{d}, \ldots, \{S_{a,b}^{k_1,\ldots,k_M}(x)\}_{k_1,\ldots,k_M=1}^{d}\right). \quad \text{(C.2)}$$

By convention $1 \in \mathbb{R}$ is also included at the start. Then $\mathrm{sig}_{a,b}^{M}(x)$ is known as the depth-M signature transform of x.

It is immediate from the definition that each term in the signature satisfies

$$S_{a,t}^{k_1,\ldots,k_m}(x) = \int_0^t S_{a,s}^{k_1,\ldots,k_{m-1}}(x)\,\mathrm{d}x_{k_m}(s).$$

By stacking all such equations together it is clear that there exists some

$$f \in L(\mathbb{R}^{\kappa(d,M)}; \mathbb{R}^{\kappa(d,M)\times d_x})$$

such that $\mathrm{sig}_{a,\cdot}^{M}(x)$ satisfies the CDE

$$\mathrm{sig}_{a,t}^{M}(x) = (1, 0, \ldots, 0) + \int_a^t f(\mathrm{sig}_{a,t}^{M}(x))\,\mathrm{d}x(t).$$

Definition C.7. *Let $K \subseteq \mathcal{V}^1([0,T];\mathbb{R}^d)$. We say that K has uniqueness of signatures if for all $x, \widehat{x} \in K$ with $x(0) = \widehat{x}(0)$, there exists $M \in \mathbb{N}$ such that $\mathrm{sig}_{0,T}^{M}(x) \neq \mathrm{sig}_{0,T}^{M}(\widehat{x})$.*

Practically speaking uniqueness of signatures is most easily obtained through the following lemma.

Lemma C.8. *For any $K \subseteq \mathcal{V}^1([0,T];\mathbb{R}^{d-1})$, then*

$$K' = \left\{(t \mapsto (t, x(t))) \in \mathcal{V}^1([0,T];\mathbb{R}^d) \;\middle|\; x \in K\right\}$$

has uniqueness of signatures.

Proof. Without loss of generality assume $d = 2$, as we will treat each channel of K separately.

Fix $x \in K$ with corresponding element $x' \in K'$. The (arbitrary depth) signature of x' over $[0,T]$ contains all terms of the form

$$\int_0^T \int_{0<t_1<\cdots<t_m<t} \cdots \int \mathrm{d}t_1 \cdots \mathrm{d}t_m \mathrm{d}x(t) = \int_0^T \frac{1}{m!} t^m\,\mathrm{d}x(t). \quad \text{(C.3)}$$

Fix $g \in C([0,T])$. Let p_n be some sequence of polynomials for which $p_n \to g$ uniformly over $[0,T]$, which exist by the Weierstrass Approximation Theorem. Then $\int_0^T p_n(t)\,\mathrm{d}x(t) \to \int_0^T g(t)\,\mathrm{d}x(t)$ [FV10, Proposition 2.8]. By (C.3) all $\int_0^T p_n(t)\,\mathrm{d}x(t)$ are determined by the signature of x', and so for all $g \in C([0,T])$ we have that $\int_0^T g(t)\,\mathrm{d}x(t)$ is determined by the signature of x'.

Fix $\varepsilon > 0$ and $s \in [0, T - \varepsilon]$ and consider specifically $g_{s,\varepsilon} \in C([0,T])$ defined by

$$g_{s,\varepsilon}(t) = \begin{cases} 1 & t \in [0, s), \\ \frac{1}{\varepsilon}(s + \varepsilon - t) & t \in [s, s + \varepsilon), \\ 0 & t \in [s + \varepsilon, T]. \end{cases}$$

Then $\int_0^T g_{s,\varepsilon}(t)\,\mathrm{d}x(t) = x(s) - x(0) + \mathcal{O}(\varepsilon)$. Letting $\varepsilon \to 0$ we have that every increment $x(s) - x(0)$ is determined by the signature of x'. □

Remark C.9. *This fact is the fundamental reason that time is included as a channel in Section 3.1.4.2.*

Remark C.10. *[HL10] give a precise characterisation of this property, which is that all $x, \widehat{x}$ must lie in different equivalence classes with respect to 'tree-like equivalence'.*

With these definitions out of the way, we are ready to state the famous universal nonlinearity property of the signature transform. We think [Per18, Theorem 4.2] gives the most straightforward proof of this result. This essentially states that the signature gives a basis for the space of functions on compact path space.

Theorem C.11 (Universal nonlinearity)**.** *Let $T > 0$ and let $d_x, d_o \in \mathbb{N}$. Let $K \subseteq \mathcal{V}_0^1([0,T];\mathbb{R}^{d_x})$ be compact and have uniqueness of signatures.*

Then

$$\bigcup_{M \in \mathbb{N}} \left\{ x \mapsto \ell(\mathrm{sig}_{0,T}^M(x)) \;\middle|\; \ell \in L(\mathbb{R}^{\kappa(d,M)}; \mathbb{R}^{d_o}) \right\}$$

is dense in $C(K;\mathbb{R}^{d_o})$.

With the universal nonlinearity property, we can now prove universal approximation of CDEs with respect to controlling paths x.

Theorem C.12 (Universal approximation with CDEs)**.** *Let $T > 0$ and let $d_x, d_o \in \mathbb{N}$. Let $K \subseteq \mathcal{V}^1([0,T];\mathbb{R}^{d_x})$ be compact and have uniqueness of signatures.*

Then

$$\bigcup_{d_y \in \mathbb{N}} \left\{ x \mapsto \ell(z_{f,\zeta,x}(T)) \;\middle|\; f \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times d_x}), \zeta \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y}), \ell \in L(\mathbb{R}^{d_y}; \mathbb{R}^{d_o}) \right\}$$

is dense in $C(K;\mathbb{R}^{d_o})$, where $z_{f,\zeta,x}$ is as defined in Definition C.4.

Proof. We begin by prepending a straight line segment to every element of K. For every $x \in K$, define $x^* \colon [-1,T] \to \mathbb{R}^{d_x}$ by

$$x^*(t) = \begin{cases} (t+1)x(0) & t \in [-1, 0), \\ x(t) & t \in [0, T]. \end{cases}$$

Then $K^* = \{x^* \mid x \in K\} \subseteq \mathcal{V}_0^1([-1,T];\mathbb{R}^{d_x})$ is compact. By Theorem C.11,

$$\bigcup_{M\in\mathbb{N}} \left\{x^* \mapsto \ell(\mathrm{sig}^m_{-1,T}(x^*)) \;\middle|\; \ell \in L(\mathbb{R}^{\kappa(d,M)};\mathbb{R}^{d_o})\right\}$$

is dense in $C(K^*;\mathbb{R}^{d_o})$.

So let $\alpha \in C(K;\mathbb{R}^{d_o})$ and $\varepsilon > 0$. The map $x \mapsto x^*$ is a homeomorphism, so we may find $\beta \in C(K^*;\mathbb{R}^{d_o})$ such that $\beta(x^*) = \alpha(x)$ for all $x \in K$. We have just established there exists some $M \in \mathbb{N}$ and $\ell \in L(\mathbb{R}^{\kappa(d,M)};\mathbb{R}^{d_o})$ such that γ defined by $\gamma\colon x^* \mapsto \ell(\mathrm{sig}^M_{-1,T}(x^*))$ is ε-close to β.

By Definition C.6 there exists $f \in \mathrm{Lip}(\mathbb{R}^{\kappa(d,M)};\mathbb{R}^{\kappa(d,M)\times d_x})$ so that $\mathrm{sig}^M_{-1,t}(x^*)$ is the unique solution of the CDE

$$\mathrm{sig}^M_{-1,t}(x^*) = \mathrm{sig}^M_{-1,-1}(x^*) + \int_{-1}^{t} f(\mathrm{sig}^M_{-1,t}(x^*))\,\mathrm{d}x^*(s) \quad \text{for } t \in (-1,T],$$

with $\mathrm{sig}^M_{-1,-1}(x^*) = (1,0,\ldots,0)$.

Now let $\zeta \in C(\mathbb{R}^{d_x};\mathbb{R}^{d_y})$ be defined by $\zeta(x(0)) = \mathrm{sig}^M_{-1,0}(x^*)$, which we note is well defined because for $t \in [-1,0]$ the value of $\mathrm{sig}^M_{-1,t}(x^*)$ only depends on $x(0)$. Then (by uniqueness of solution) we have that $\mathrm{sig}^M_{-1,t}(x^*) = z_{f,\zeta,x^*}(t)$ for $t \in [0,T]$.

For all $x \in K$,

$$\ell(z_{f,\zeta,x^*}(T)) = \ell(\mathrm{sig}^M_{-1,T}(x^*)) = \gamma(x^*)$$

is ε-close to $\beta(x^*) = \alpha(x)$. Thus density has been established. □

Remark C.13. *For the reader familiar with rough path theory, the above proof is essentially just premultiplying the signature of x by the signature of the straight line increment from 0 to $x(0)$ so as to remove translational invariance.*

C.2.1.2 Universal approximation with respect to time series

Of course, the input to a neural CDE will often not be a continuous path. Very often it will instead be a discretised time series, which we interpolate. We need to extend our universal approximation result to this case. Our approach here will be agnostic to the choice of interpolation, and will instead impose conditions that the interpolation scheme must satisfy in order to provide universal approximation.

Definition C.14 (Space of time series)**.** *Let $d \in \mathbb{N}$. We define the set of irregularly sampled time series over $\mathbb{R}^d$ as*

$$\mathcal{TS}(\mathbb{R}^d) = \left\{((t_0,x_0),\ldots,(t_n,x_n)) \;\middle|\; n \in \mathbb{N}, t_j \in \mathbb{R}, x_j \in (\mathbb{R}\cup\{*\})^d, t_0 = 0, t_j < t_{j+1}\right\}.$$

where $*$ denotes the possibility of missing data.

Definition C.15. *For each* $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n)) \in \mathcal{TS}(\mathbb{R}^d)$*, let* $x_j = (x_{j,1}, \ldots, x_{j,d}) \in \mathbb{R}^d$ *and for notational convenience let* $x_{j,0} = t_j$*. Then we define* $\omega(\mathbf{x})$ *by*

$$\omega(\mathbf{x}) = \max\left\{n, \max_{j=0,\ldots,n}\max_{k=0,\ldots,d}|x_{j,k}|, \max_{j=0,\ldots,n-1}\max_{k=0,\ldots,d}\frac{x_{j+1,k} - x_{j,k}}{t_{j+1} - t_j}, \max_{j=0,\ldots,n-1}\max_{k=0,\ldots,d}\frac{x_{j+1,k} - x_{j,k}}{(t_{j+1} - t_j)^2}\right\}.$$

For simplicity the above definition ignores the presence of missing data. If necessary replace each $*$ *with a* 0 *to make the above well-defined.*

Definition C.16. *For all* $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n)) \in \mathcal{TS}(\mathbb{R}^d)$*, decompose* $x_j = (x_{j,1}, \ldots, x_{j,d}) \in \mathbb{R}^d$*, and then define* $c_j(\mathbf{x}) = (c_{j,1}(\mathbf{x}), \ldots, c_{j,d}(\mathbf{x})) \in \mathbb{R}^d$*, where* $c_{j,k}(\mathbf{x}) = \sum_{m=0}^{j} \mathbb{1}_{x_{m,k} \neq *}$ *counts the number of observations in the* k*th channel by time* t_j*.*

Definition C.17 (Interpolation)**.** *Let* $T > 0$*. Let* $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$*. We define an* interpolation *as a map*

$$\begin{aligned}\mathcal{X} &\to \mathcal{V}^1([0, T]; \mathbb{R}^{2d+1}),\\ \mathbf{x} &\mapsto x_{\mathbf{x}},\end{aligned}$$

together with a collection of $0 = s_0 < \cdots < s_n = T$*, such that*

$$x_{\mathbf{x}}(s_j) = (t_j, x_j, c_j(\mathbf{x})) \in \mathbb{R}^{2d+1} \tag{C.4}$$

for all $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n)) \in \mathcal{X}$ *and* $j \in \{0, \ldots, n\}$*. (And any missing values* $*$ *are ignored for the purposes of determining equality in equation* (C.4)*.) The values of* s_j *may depend upon* $\mathbf{x}$*.*

Remark C.18. *If the full dataset of time series* $\mathbf{x}$ *has no missing values then we may need only a single* c_j *channel to capture the rate of observations. If every time series is additionally regularly sampled then these channels may be omitted altogether, as not carrying any information.*

Definition C.19 (Bounded interpolation)**.** *Let* $T > 0$*. Let* $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$*. Consider the interpolation*

$$\begin{aligned}\mathcal{X} &\to \mathcal{V}^1([0, T]; \mathbb{R}^{2d+1})\\ \mathbf{x} &\mapsto x_{\mathbf{x}}.\end{aligned}$$

We call this a bounded interpolation *if there exists* $C > 0$ *so that for all* $\mathbf{x} \in \mathcal{X}$*,*

$$\|x_{\mathbf{x}}\|_\infty + \left\|\frac{\mathrm{d}x_{\mathbf{x}}}{\mathrm{d}t}\right\|_\infty + \left|\frac{\mathrm{d}x_{\mathbf{x}}}{\mathrm{d}t}\right|_{\mathrm{BV}} < C\omega(\mathbf{x}).$$

Remark C.20. *It is really for ease of this definition that we restrict an interpolation to being defined on only some* $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$*. If an interpolation was defined on all of* $\mathcal{TS}(\mathbb{R}^d)$*, and we wished to define a bounded interpolation, then the codomain would need to be all of* $\cup_{T>0}\mathcal{V}^1([0, T]; \mathbb{R}^{2d+1})$*.*

(Otherwise what must happen as the length of a time series increases? The points s_j must be packed closer and closer together, and correspondingly the derivative of the interpolation may tend towards infinity, violating boundedness. Given that we would often like to take $s_j = t_j$ in practice, then the 'natural' resolution is to allow the resulting interpolation to be defined over any $[0, T]$.)

Allowing arbitrary domains would complicate the presentation somewhat, so we stick to the simple case. (The general case is mathematically doable, but tedious.)

Definition C.21 (Signature-unique interpolation)**.** *Let $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$. Consider the interpolation*

$$\begin{aligned}\mathcal{X} &\to \mathcal{V}^1([0, S]; \mathbb{R}^{2d+1})\\ \mathbf{x} &\mapsto x_{\mathbf{x}}.\end{aligned}$$

We call this a signature-unique interpolation *if $\mathbf{x} \mapsto x_{\mathbf{x}}$ is injective, and if $\{x_{\mathbf{x}} \mid \mathbf{x} \in \mathcal{X}\}$ has uniqueness of signatures in the sense of Definition C.7.*

Remark C.22. *Injectivity is included in the above definition only for emphasis – it is automatically true for any interpolation scheme. In the case of missing data, injectivity holds because of the extra $c_j(\mathbf{x})$ channels of Definition C.17.*

For example, $((t_0, x_0), (t_2, x_2))$ and $((t_0, x_0), (t_1, *), (t_2, x_2))$ might otherwise both be interpolated to produce the same result (perhaps a linear interpolation over $[t_0, t_2]$), and injectivity would have been lost.

Definition C.23 (Time series topologies)**.** *Given any particular interpolation, we will equip $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$ with the weakest topology for which that interpolation is continuous.*

Lemma C.24. *Let $\mathcal{X} \subseteq \mathcal{TS}(\mathbb{R}^d)$, and let*

$$\begin{aligned}\mathcal{X} &\to \mathcal{V}^1([0, T]; \mathbb{R}^{2d+1})\\ \mathbf{x} &\mapsto x_{\mathbf{x}}.\end{aligned}$$

be a bounded interpolation. Suppose there exists $C > 0$ such that for all $\mathbf{x} \in \mathcal{X}$ that $\omega(\mathbf{x}) < C$. Let $\mathfrak{X} = \{x_{\mathbf{x}} \mid \mathbf{x} \in \mathcal{X}\}$. Then $\mathfrak{X}$ is relatively compact (that is, its closure is compact) in $\mathcal{V}^1([0, T]; \mathbb{R}^{2d+1})$.

Proof. By boundedness of the interpolation then

$$\sup_{x \in \mathfrak{X}} \left(\|x\|_\infty + \left\| \frac{\mathrm{d}x}{\mathrm{d}t} \right\|_\infty + \left| \frac{\mathrm{d}x}{\mathrm{d}t} \right|_{\mathrm{BV}} \right) < \infty.$$

Now $\mathfrak{X}$ is bounded in $W^{1,\infty}([0, T]; \mathbb{R}^{2d+1})$ and so relatively compact in $L^\infty([0, T]; \mathbb{R}^{2d+1})$. Let $\mathfrak{X}' = \left\{ \frac{\mathrm{d}x}{\mathrm{d}t} \;\middle|\; x \in \mathfrak{X} \right\}$. Then $\mathfrak{X}'$ is bounded in $\mathrm{BV}([0, T]; \mathbb{R}^{2d+1})$ and so relatively compact in $L^1([0, T]; \mathbb{R}^{2d+1})$. Therefore $\mathfrak{X} \times \mathfrak{X}'$ is relatively compact in $L^\infty([0, T]; \mathbb{R}^{2d+1}) \times L^1([0, T]; \mathbb{R}^{2d+1})$.

Let $\mathbb{X} = \left\{(x, \frac{\mathrm{d}x}{\mathrm{d}t}) \,\middle|\, x \in \mathfrak{X}\right\}$. Then $\mathbb{X} \subseteq \mathfrak{X} \times \mathfrak{X}'$ so $\mathbb{X}$ is also relatively compact in $L^\infty([0,T];\mathbb{R}^{2d+1}) \times L^1([0,T];\mathbb{R}^{2d+1})$. This implies that $\mathfrak{X}$ is relatively compact with respect to the topology generated by $x \mapsto \|x\|_\infty + \left\|\frac{\mathrm{d}x}{\mathrm{d}t}\right\|_1$, and hence also with respect to the topology generated by $x \mapsto \|x\|_\infty + |x|_{\mathrm{BV}}$. □

Theorem C.25 (Universal approximation with neural CDEs on time series)**.** *Let $d_x, d_y, n \in \mathbb{N}$ and let $T > 0$.*

For all $d_y \in \mathbb{N}$, let $F_{d_y} \subseteq \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times (2d_x+1)})$ be dense in $C(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times (2d_x+1)})$. Likewise let $\xi_{d_y} \subseteq C(\mathbb{R}^{2d_x+1}; \mathbb{R}^{d_y})$ be dense in $C(\mathbb{R}^{2d_x+1}; \mathbb{R}^{d_y})$. (Typically these will both be classes of neural networks).

Let $\mathcal{X} \subseteq \mathcal{TS}\big(\mathbb{R}^{d_x}\big)$ be such that there exists $C > 0$ such that

$$\omega(\mathbf{x}) < C \tag{C.5}$$

for every $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n)) \in \mathcal{X}$. (With C independent of $\mathbf{x}$.)

Let

$$\begin{aligned} \mathcal{X} &\to \mathcal{V}^1([0,T];\mathbb{R}^{2d_x+1}) \\ \mathbf{x} &\mapsto x_{\mathbf{x}} \end{aligned}$$

be a bounded signature-unique interpolation.

Then

$$\bigcup_{d_y \in \mathbb{N}} \left\{\mathbf{x} \mapsto \ell(z_{f,\zeta,x_{\mathbf{x}}}(T)) \,\middle|\, f \in F_{d_y}, \zeta \in \xi_{d_y}, \ell \in L(\mathbb{R}^{d_y}; \mathbb{R}^{d_o})\right\}$$

is dense in $C(\mathcal{X}; \mathbb{R}^{d_o})$, where $z_{f,\zeta,x}$ is as defined in Definition C.4.

Proof. By equation (C.5) and boundedness of the interpolation, Lemma C.24 implies that $\mathfrak{X} = \{x_{\mathbf{x}} \,|\, \mathbf{x} \in \mathcal{X}\}$ is relatively compact in $\mathcal{V}^1([0,T];\mathbb{R}^{2d_x+1})$.

By Theorem C.12 and signature-uniqueness of the interpolation,

$$\bigcup_{d_y \in \mathbb{N}} \left\{x \mapsto \ell(z_{f,\zeta,x}(T)) \,\middle|\, f \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times (2d_x+1)}),\ \zeta \in C(\mathbb{R}^{2d_x+1}, \mathbb{R}^{d_y}),\ \ell \in L(\mathbb{R}^{d_y}; \mathbb{R}^{d_o})\right\}$$

is dense in $C(\overline{\mathfrak{X}}, \mathbb{R}^{d_o})$, where the overline denotes a closure.

For any $f \in F_{d_y}$, any $\zeta \in \xi_{d_y}$, any $f' \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times (2d_x+1)})$ and any $\zeta' \in C(\mathbb{R}^{2d_x+1}, \mathbb{R}^{d_y})$, the terminal values $z_{f,\zeta,x}(T)$ and $z_{f',\zeta',x}(T)$ may be compared by standard estimates, for example as commonly used in the proof of Picard's theorem. Classical universal approximation results for neural networks [Pin99; KL20b] then yield that

$$\bigcup_{d_y \in \mathbb{N}} \left\{x \mapsto \ell(z_{f,\zeta,x}(T)) \,\middle|\, f \in F_{d_y}, \zeta \in \xi_{d_y}, \ell \in L(\mathbb{R}^{d_y}; \mathbb{R}^{d_o})\right\}$$

is dense in $C(\overline{\mathfrak{X}}, \mathbb{R}^{d_o})$.

By the definition of the topology on $\mathcal{TS}\big(\mathbb{R}^{d_x}\big)$, then

$$\bigcup_{d_y\in\mathbb{N}} \big\{\mathbf{x}\mapsto \ell(z_{f,\zeta,x_{\mathbf{x}}}(T)) \,\big|\, f\in F_{d_y},\, \zeta\in\xi_{d_y},\, \ell\in L(\mathbb{R}^{d_y};\mathbb{R}^{d_o})\big\}$$

is dense in $C(\mathcal{X},\mathbb{R}^{d_o})$. □

It is now a relatively straightforward matter to determine boundedness and signature-uniqueness for any individual problem. Boundedness is typically obtained by demanding that $\mathcal{X}$ consist of time series of at most some length, of at most some value, and so on. Signature uniqueness is typically obtained via Lemma C.8, and the fact that time is included as a channel.

Remark C.26. *For example, both boundedness and signature-uniqueness are immediately true of linear interpolation.*

Likewise, [Kid+20a, Appendix B] demonstrates that these properties hold for natural cubic splines. There we fix $\tau < T$, consider $s_j = t_j$, and take $\mathcal{X}\subseteq\mathcal{TS}\big(\mathbb{R}^d\big)$ to be those time series for which $t_0=\tau$ and $t_n = T$.

C.2.2 Neural CDEs compared to alternative ODE models

Suppose if instead of equation (3.5), we replace $g_{\theta,x}(y,s)$ by $h_\theta(y,x(s))$ for some other vector field h_θ. This might seem more natural. Instead of having $g_{\theta,x}$ linear in $\frac{\mathrm{d}x}{\mathrm{d}s}$, then h_θ is potentially nonlinear in the control $x(s)$.

Have anything been gained by doing so? It turns out no, and in fact something has been lost. The neural CDE setup directly subsumes anything depending directly on x.

Theorem C.27. *Let $T>0$. Let $d_x, d_y\in\mathbb{N}$ with $d_x < d_y$. Let*

$$\mathcal{V}^1([0,T];\mathbb{R}^{d_x-1}) = C([0,T];\mathbb{R}^{d_x-1})\cap \mathrm{BV}([0,T];\mathbb{R}^{d_x-1}).$$

For all $x\in\mathcal{V}^1([0,T];\mathbb{R}^{d_x-1})$, let $\widehat{x}(t)=(t,x(t))$.

Let $\pi\colon\mathbb{R}^{d_y}\to\mathbb{R}^{d_y-d_x}$ be the orthogonal projection onto the first d_y-d_x coordinates.

Let

$$\mathcal{Y} = \big\{x\mapsto y_{h,\xi,x} \,\big|\, h\in\mathrm{Lip}(\mathbb{R}^{d_y-d_x}\times\mathbb{R}^{d_x};\mathbb{R}^{d_y-d_x}),\, \xi\in C(\mathbb{R}^{d_x};\mathbb{R}^{d_y-d_x})\big\},$$
$$\mathcal{Z} = \big\{x\mapsto \pi\circ z_{f,\zeta,x} \,\big|\, f\in\mathrm{Lip}(\mathbb{R}^{d_y};\mathbb{R}^{d_y\times d_x}),\, \zeta\in C(\mathbb{R}^{d_x};\mathbb{R}^{d_y})\big\},$$

where $y_{h,\xi,x}\colon[0,T]\to\mathbb{R}^{d_y-d_x}$ is the unique solution to

$$y_{h,\xi,x}(t) = y_{h,\xi,x}(0) + \int_0^t h(y_{h,\xi,x}(s),\widehat{x}(s))\,\mathrm{d}s \quad \text{for } t\in(0,T],$$

with $y_{h,\xi,x}(0) = \xi(\widehat{x}(0))$*, and* $z_{f,\zeta,x}\colon [0,T] \to \mathbb{R}^{d_y}$ *is the unique solution to*

$$z_{f,\zeta,x}(t) = z_{f,\zeta,x}(0) + \int_0^t f(z_{f,\zeta,x}(s))\,\mathrm{d}\widehat{x}(s) \quad \textit{for } t \in (0,T],$$

with $z_{f,\zeta,x}(0) = \zeta(\widehat{x}(0))$.
Then $\mathcal{Y} \subsetneq \mathcal{Z}$.

In the above statement, a practical choice of $f \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times d_x})$ or $h \in \mathrm{Lip}(\mathbb{R}^{d_y - d_x} \times \mathbb{R}^{d_x}; \mathbb{R}^{d_y - d_x})$ will be some trained neural network.

Note the inclusion of time via the augmentation $x \mapsto \widehat{x}$. Without it, the reparameterisation invariance property of CDEs (Section 3.3.3.2) would restrict the possible functions that CDEs can represent. This hypothesis is not necessary for the $\mathcal{Y} \neq \mathcal{Z}$ part of the conclusion.

Note also how the CDE uses a larger state space of d_y, compared to $d_y - d_x$ for the alternative ODE. The reason for this is that whilst f has no explicit nonlinear dependence on x, we may construct it to have such a dependence implicitly, by recording d into d_x of its d_y hidden channels, whereupon x is hidden state and may be treated nonlinearly. This hypothesis is also not necessary to demonstrate the $\mathcal{Y} \neq \mathcal{Z}$ part of the conclusion.

Proof.
That $\mathcal{Y} \neq \mathcal{Z}$:
Let $\zeta \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y})$ be arbitrary and let

$$f(z) = \underbrace{\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}}_{d_x} \Bigg\} d_y$$

Then for any $x \in \mathcal{V}^1([0,T]; \mathbb{R}^{d_x - 1})$, the corresponding CDE solution $z_{f,\zeta,x} \in \mathcal{Z}$ satisfies

$$z_{f,\zeta,x}(t) = z_{f,\zeta,x}(0) + \int_0^t f(z_{f,\zeta,x}(s))\,\mathrm{d}\widehat{x}(s),$$

and so the first component of its solution is

$$z_{f,\zeta,x,1}(t) = x_1(t) - x_1(0) + \zeta_1(\widehat{x}(0)),$$

whilst the other components are constant

$$z_{f,\zeta,x,i}(t) = \zeta_i(\widehat{x}(0))$$

for $i \in \{2, \ldots, d_y\}$.

Now suppose for contradiction that there exists $\xi \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y - d_x})$ and $h \in \mathrm{Lip}(\mathbb{R}^{d_y - d_x} \times \mathbb{R}^{d_x}; \mathbb{R}^{d_y - d_x})$ with a corresponding $y_{h,\xi,\cdot} \in \mathcal{Y}$, such that $y_{h,\xi,x} = \pi \circ z_{f,\zeta,x}$ for all $x \in \mathcal{V}^1([0,T]; \mathbb{R}^{d_x - 1})$. Now $y_{h,\xi,x}$ must satisfy

$$y_{h,\xi,x}(t) = y_{h,\xi,x}(0) + \int_0^t h(y_{h,\xi,x}(s), \widehat{x}(s))\,\mathrm{d}s,$$

and so

$$\begin{bmatrix} x_1(t) - x_1(0) + \zeta_1(\widehat{x}(0)) \\ \zeta_2(\widehat{x}(0)) \\ \cdots \\ \zeta_{d_y - d_x}(\widehat{x}(0)) \end{bmatrix}$$
$$= y_{h,\xi,x}(0) + \int_0^t h(\begin{bmatrix} x_1(s) - x_1(0) + \zeta_1(\widehat{x}(0)) \\ \zeta_2(\widehat{x}(0)) \\ \cdots \\ \zeta_{d_y - d_x}(\widehat{x}(0)) \end{bmatrix}, \widehat{x}(s))\,\mathrm{d}s.$$

Consider those x which are differentiable. Differentiating with respect to t and considering the first component now gives

$$\frac{\mathrm{d}x_1}{\mathrm{d}t}(t) = h_1(\begin{bmatrix} x_1(s) - x_1(0) + \zeta_1(\widehat{x}(0)) \\ \zeta_2(\widehat{x}(0)) \\ \cdots \\ \zeta_{d_y - d_x}(\widehat{x}(0)) \end{bmatrix}, \widehat{x}(t)). \tag{C.6}$$

That is, h_1 satisfies equation (C.6) for all differentiable x. This is clearly impossible: the right hand side is a function of t, $x(t)$ and $x(0)$ only, which is insufficient to determine $\frac{\mathrm{d}x_1}{\mathrm{d}t}(t)$.

That $\mathcal{Y} \subseteq \mathcal{Z}$:
Let $y_{h,\xi,x} \in \mathcal{Y}$ for some $\xi \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y - d_x})$ and $h \in \mathrm{Lip}(\mathbb{R}^{d_y - d_x} \times \mathbb{R}^{d_x}; \mathbb{R}^{d_y - d_x})$. Let $\sigma \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_x}$ be the orthogonal projection onto the last d_x coordinates. Let $\zeta \in C(\mathbb{R}^{d_x}; \mathbb{R}^{d_y})$ be such that $\pi \circ \zeta = \pi \circ \xi$ and $\sigma(\zeta(\widehat{x}(0))) = \widehat{x}(0)$. Then let $f \in \mathrm{Lip}(\mathbb{R}^{d_y}; \mathbb{R}^{d_y \times d_x})$ be defined by

$$f(z) = \begin{bmatrix} h_1(\pi(z), \sigma(z)) & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ h_{d_y - d_x}(\pi(z), \sigma(z)) & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \} d_y - d_x \\ \\ \\ \\ \\ \} d_x \\ \\ \end{matrix}$$
$$\underbrace{\qquad\qquad}_{1} \quad \underbrace{\qquad\qquad\qquad}_{d_x - 1}$$

Then for $t \in (0, T]$,

$$\begin{aligned} z_{f,\zeta,x}(t) &= \zeta(\widehat{x}(0)) + \int_0^t f(z_{f,\zeta,x}(s))\, \mathrm{d}\widehat{x}(s) \\ &= \zeta(\widehat{x}(0)) + \int_0^t \begin{bmatrix} h_1(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s))) & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ h_{d_y - d_x}(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s))) & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} \mathrm{d}s \\ \mathrm{d}x_1(s) \\ \vdots \\ \mathrm{d}x_{d_x - 1}(s) \end{bmatrix} \\ &= \zeta(\widehat{x}(0)) + \int_0^t \begin{bmatrix} h_1(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s)))\, \mathrm{d}s \\ \vdots \\ h_{d_y - d_x}(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s)))\, \mathrm{d}s \\ \mathrm{d}s \\ \mathrm{d}x_1(s) \\ \vdots \\ \mathrm{d}x_{d_x - 1}(s) \end{bmatrix} \\ &= \zeta(\widehat{x}(0)) + \int_0^t \begin{bmatrix} h(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s)))\, \mathrm{d}s \\ \mathrm{d}\widehat{x}(s) \end{bmatrix}. \end{aligned}$$

Thus

$$\sigma(z_{f,\zeta,x}(t)) = \sigma(\zeta(\widehat{x}(0))) + \int_0^t \mathrm{d}\widehat{x}(s) = \sigma(\zeta(\widehat{x}(0))) - \widehat{x}(0) + \widehat{x}(t) = \widehat{x}(t),$$

and so

$$\begin{aligned}\pi(z_{f,\zeta,x}(t)) &= \pi(\zeta(\widehat{x}(0))) + \int_0^t h(\pi(z_{f,\zeta,x}(s)), \sigma(z_{f,\zeta,x}(s)))\,\mathrm{d}s \\ &= \pi(\xi(\widehat{x}(0))) + \int_0^t h(\pi(z_{f,\zeta,x}(s)), \widehat{x}(s))\,\mathrm{d}s.\end{aligned}$$

We see that $\pi \circ z_{f,\zeta,x}$ satisfies the same differential equation as $y_{h,\xi,x}$. So by uniqueness of solution [LCL04, Theorem 1.3], $y_{h,\xi,x} = \pi \circ z_{f,\zeta,x} \in \mathcal{Z}$. □

C.2.3 Reparameterisation invariance of CDEs

Proposition 3.18. *Let $\psi\colon [0,S] \to [0,T]$ be differentiable, increasing, and such that $\psi(0) = 0$ and $\psi(S) = T$. Let y solve a CDE driven by a path x. Then $y \circ \psi$ solves the same CDE driven by $x \circ \psi$, and in particular their terminal values are the same: $(y \circ \psi)(S) = y(T)$.*

Proof. The proof is straightforward change of variables. For expository purposes we consider only differentiable paths; equivalent change-of-variable formulae may be used for bounded variation paths.

Let $\widetilde{t} \in [0, \widetilde{T}]$ and let $t = \psi(\widetilde{t})$. Let $\widetilde{x} = x \circ \psi$ and $\widetilde{y} = y \circ \psi$.

Then make the change of variables $s = \psi(\widetilde{s})$,

$$\begin{aligned}\widetilde{y}(\widetilde{t}) &= y(t) \\ &= y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s) \\ &= y(0) + \int_0^t f(y(s))\,\frac{\mathrm{d}x}{\mathrm{d}s}(s)\,\mathrm{d}s \\ &= y(\psi(0)) + \int_0^{\psi^{-1}(t)} f(y(\psi(\widetilde{s})))\,\frac{\mathrm{d}x}{\mathrm{d}s}(\psi(\widetilde{s}))\,\frac{\mathrm{d}\psi}{\mathrm{d}\widetilde{s}}(\widetilde{s})\,\mathrm{d}\widetilde{s} \\ &= (y \circ \psi)(0) + \int_0^{\psi^{-1}(t)} f((y \circ \psi)(\widetilde{s}))\,\frac{\mathrm{d}(x \circ \psi)}{\mathrm{d}\widetilde{s}}(\widetilde{s})\,\mathrm{d}\widetilde{s} \\ &= (y \circ \psi)(0) + \int_0^{\psi^{-1}(t)} f((y \circ \psi)(\widetilde{s}))\,\mathrm{d}(x \circ \psi)(\widetilde{s}) \\ &= \widetilde{y}(0) + \int_0^{\widetilde{t}} f(\widetilde{y}(\widetilde{s}))\,\mathrm{d}\widetilde{x}(\widetilde{s}).\end{aligned}$$

□

C.2.4 Comments

Surprisingly – despite it being a well-known part of the folklore for signatures – we could note find a direct statement of Lemma C.8 anywhere in the literature. (It is

easy to prove, at least.)

To the best of our knowledge all of the discussion on interpolation schemes is new here. We find this a little surprising as the use of differential equations to control dynamical systems is well-studied, as is discrete-time control via for example reinforcement learning. Despite this we have encountered almost nothing written about the formalities of embedding discrete observations into continuous time.

The proof for the comparison of neural CDEs against alternative ODE models is a variation on a standard trick in rough path theory, in which the control is 'recorded' into some additional state.

C.3 Backpropagation via optimise-then-discretise

We will now prove how to backpropagate via optimise-then-discretise for ODEs, CDEs, and SDEs.

In principle these may essentially all be thought of as special cases of the same general result (the one shown for SDEs), but in the interests of pedagogy each case is proved separately.

C.3.1 Optimise-then-discretise for ODEs

Recall backpropagation through ODEs via optimise-then-discretise.

Theorem 5.2. *Let $y_0 \in \mathbb{R}^d$ and $\theta \in \mathbb{R}^m$. Let $f_\theta \colon [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t, uniformly Lipschitz in y, and continuously differentiable in y. Let $y \colon [0, T] \to \mathbb{R}^d$ be the unique solution to*

$$y(0) = y_0, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)).$$

Let $L = L(y(T))$ be some (for simplicity scalar) function of the terminal value $y(T)$. Then $\frac{\mathrm{d}L}{\mathrm{d}y(t)} = a_y(t)$ and $\frac{\mathrm{d}L}{\mathrm{d}\theta} = a_\theta(0)$, where $a_y \colon [0, T] \to \mathbb{R}^d$ and $a_\theta \colon [0, T] \to \mathbb{R}^m$ solve the system of differential equations

$$\begin{aligned} a_y(T) &= \frac{\mathrm{d}L}{\mathrm{d}y(T)}, & \frac{\mathrm{d}a_y}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial y}(t, y(t)), \\ a_\theta(T) &= 0, & \frac{\mathrm{d}a_\theta}{\mathrm{d}t}(t) &= -a_y(t)^\top \frac{\partial f_\theta}{\partial \theta}(t, y(t)). \end{aligned} \tag{5.1}$$

The following proof is both simpler and more precise than those we have typically seen in the literature.

Proof. Without loss of generality we will prove the equation for a_y only. The equation for a_θ may be derived by replacing y with the $[y, \theta]$ and f_θ with $[f., 0]$.

Now y is continuous, and f_θ is continuously differentiable in y, so $t \mapsto \frac{\partial f_\theta}{\partial y}(t, y(t))$ is a continuous function on the compact set $[0, T]$, so it is bounded by some $C > 0$. Correspondingly for $a \in \mathbb{R}^d$ then $(t, a) \mapsto -a^\top \frac{\partial f_\theta}{\partial y}(t, y(t))$ is Lipschitz in a with Lipschitz constant C and this constant is independent of t. Therefore by Picard's existence theorem (Theorem 2.1) the solution a_y to equation (5.1) exists and is unique.

We still need to show that $a_y(t) = \frac{\mathrm{d}L}{\mathrm{d}y(t)}$.

For $s, t \in [0, T]$ with $s < t$ then

$$y(t) = y(s) + \int_s^t f_\theta(u, y(u)) \, \mathrm{d}u,$$

so

$$\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} = I_{d\times d} + \int_s^t \frac{\partial f_\theta}{\partial y}(u, y(u)) \frac{\mathrm{d}y(u)}{\mathrm{d}y(s)} \, \mathrm{d}u, \tag{C.7}$$

interchanging limits (Leibniz integral rule or dominated convergence theorem) as $\frac{\partial f_\theta}{\partial y}$ was assumed to be bounded. This is the forward sensitivity equation (Theorem 5.8), which is an ODE for the Jacobian $\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}$, the solution of which exists by Picard's existence theorem (Theorem 2.1).

For $s, t \in [0, T]$ with $s < t$ then

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\left(a_y(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right) &= \frac{\mathrm{d}a_y}{\mathrm{d}t}(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} + a_y(t)^\top \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right) \\
&= \frac{\mathrm{d}a_y}{\mathrm{d}t}(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} + a_y(t)^\top \frac{\partial f_\theta}{\partial y}(t, y(t)) \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} \\
&= \left(\frac{\mathrm{d}a_y}{\mathrm{d}t}(t) + a_y(t)^\top \frac{\partial f_\theta}{\partial y}(t, y(t))\right) \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} \\
&= 0
\end{aligned} \tag{C.8}$$

where the second line is obtained by differentiating (C.7) directly.

Therefore

$$a_y(t) = a_y(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(t)} = a_y(T)^\top \frac{\mathrm{d}y(T)}{\mathrm{d}y(t)} = \frac{\mathrm{d}L}{\mathrm{d}y(t)}$$

and in particular $\frac{\mathrm{d}L}{\mathrm{d}y_0} = \frac{\mathrm{d}L}{\mathrm{d}y(0)} = a_y(0)$. □

C.3.2 Optimise-then-discretise for CDEs

Theorem 5.9. *Let $f\colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be both Lipschitz and continuously differentiable. Let $x\colon [0, T] \to \mathbb{R}^{d_x}$ be continuous and of bounded variation. Let $L\colon \mathbb{R}^{d_y} \to \mathbb{R}$ be differentiable (and scalar just for simplicity). Let $y_0 \in \mathbb{R}^{d_y}$ and let $y\colon [0, T] \to \mathbb{R}^{d_y}$ solve*

$$y(0) = y_0, \qquad y(t) = y(0) + \int_0^t f(y(s)) \, \mathrm{d}x(s). \tag{5.4}$$

Then the adjoint process $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)}$ *satisfies the backwards-in-time linear CDE*

$$a(t) = a(T) + \int_T^t -a(s)^\top \frac{\partial f}{\partial y}(y(s))\,\mathrm{d}x(s), \tag{5.5}$$

starting from the terminal condition $a(T) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(T)}$*, and where the right hand side denotes a vector-Jacobian product.*

The following proof is precisely analogous to the one presented for ODEs in the previous section. The only difference is that the product rule is substituted for its integral equivalent, namely integration by parts.

Proof. First we will demonstrate existence and uniqueness of the adjoint process a. Analogous to the ODE case, we may wish to consider the vector field as a map $(s, a) \mapsto -a^\top \frac{\partial f}{\partial y}(y(s))$. However in the CDE setting we have restricted ourselves to vector fields that are a function of the state (in this case a) only.

The quickest resolution to this is to incorporate the time dependence into the control. That is, we reformulate the solution to (5.5) as the solution to

$$a(t) = a(T) + \int_T^t -a(s)^\top \,\mathrm{d}M(s) \tag{C.9}$$

where $M\colon [0,T] \to \mathbb{R}^{d_y \times d_y}$ is itself the value of the integral

$$M(t) = M(T) + \int_T^t \frac{\partial f}{\partial y}(y(s))\,\mathrm{d}x(s), \tag{C.10}$$

which we note is merely an integral and not a differential equation.

As f was assumed to have continuous derivative then $s \mapsto \frac{\partial f}{\partial y}(y(s))$ is continuous and so (C.10) exists and is of bounded variation as a Riemann–Stieltjes integral. Then by Picard's existence theorem (Theorem 3.3), (C.9) exists and is unique as the vector field $a \mapsto -a^\top$ is Lipschitz.

Next, let $J_s(t) = \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} \in \mathbb{R}^{d_y \times d_y}$, which by [FV10, Theorem 4.4] exists and satisfies the CDE

$$J_s(t) = J_s(0) + \int_0^t \frac{\partial f}{\partial y}(y(u))J_s(u)\,\mathrm{d}x(u).$$

(This CDE is the one we would expect, in analogy to the ODE case.)

For $s, t, \tau \in [0,T]$ with $s < t < \tau$, and using Einstein notation over indices k_1, k_2, k_3,

$$\begin{aligned}
&a_{k_1}(\tau)J_{s,k_1,k_2}(\tau) - a_{k_1}(t)J_{s,k_1,k_2}(t)\\
&= \int_t^\tau a_{k_1}(u)\,\mathrm{d}J_{k_1,k_2}(u) + \int_t^\tau J_{s,k_1,k_2}(u)\,\mathrm{d}a_{k_1}(u)\\
&= \int_t^\tau a_{k_1}(u)\frac{\partial f_{k_1}}{\partial y_{k_2}}(u, y(u))J_{s,k_2,k_3}(u)\,\mathrm{d}x(u) - \int_t^\tau a_{k_1}(u)\frac{\partial f_{k_1}}{\partial y_{k_2}}(u, y(u))J_{s,k_2,k_3}(u)\,\mathrm{d}x(u)\\
&= 0.
\end{aligned}$$

where the first equality is integration by parts for Riemann–Stieltjes integrals, and the second equality follows from substituting in the differential equations defining a and z.

Therefore

$$a(t) = a(t)^\top \frac{\mathrm{d}y(t)}{\mathrm{d}y(t)} = a(t)^\top J_t(t) = a(T)^\top J_t(T) = a(T)^\top \frac{\mathrm{d}y(T)}{\mathrm{d}y(t)} = \frac{\mathrm{d}L}{\mathrm{d}y(t)}.$$

□

C.3.3 Optimise-then-discretise for SDEs

We now provide a precise statement for optimise-then-discretise backpropagation through SDEs (originally stated informally in Theorem 5.10).

Classical SDE theory struggles to make sense of the backward-in-time SDE. This motivates our use of rough path theory.

We begin by outlining the rough path approach to SDEs. We assume familiarity with bounded variation paths, Riemann–Stieltjes integration, and the definition of Brownian motion. We will *not* assume familiarity with classical SDE theory – for such readers the following presentation should provide an introduction to SDEs that is (in this author's opinion) substantially more elegant than the classical approach.

C.3.3.1 Fundamentals

We begin by setting up a few abstract notions.

Notation. *For any $d \in \mathbb{N}$ and $k \in \{0, 1, 2\}$, let π_k denote the projection $\mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d} \to \mathbb{R}^{d^k}$.*

We will use $\|\cdot\|$ to denote any choice of norm on $\mathbb{R}$, $\mathbb{R}^d$, $\mathbb{R}^{d \times d}$; in finite dimensions all are equivalent so the choice of norm will not be important to us.

CDEs will appear several times. As such and for consistency with the usual way of writing down SDEs, we will switch from denoting solutions of CDEs by

$$y(t) = y(0) + \int_0^t f(y(s))\,\mathrm{d}x(s)$$

to denoting them by

$$\mathrm{d}y(t) = f(y(t))\,\mathrm{d}x(t).$$

Finally, we recall the standard notation collected at the end of this thesis, including in particular the definition of the tensor product $\otimes$.

Definition C.28 (Depth-2 signature)**.** *Let $x\colon [0,T] \to \mathbb{R}^d$ be continuous and of bounded variation. Then the depth-2 signature of x is defined as*

$$\begin{aligned}&\operatorname{sig}^2(x)\colon [0,T] \to \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d\times d},\\ &\operatorname{sig}^2(x)\colon t \mapsto \operatorname{sig}^2_{0,t}(x) = \left(1, x(t) - x(0), \int_0^t (x(s) - x(0)) \otimes \mathrm{d}x(s)\right). \end{aligned} \tag{C.11}$$

where the final term is defined via Riemann–Stieltjes integration. The constant term 1 is included by convention.

Note the use of the tensor (outer) product $\otimes$. This is a bilinear operator so by appropriately manipulating dimensions then the integral of equation (C.11) may be interpreted as a matrix-vector product as already introduced for controlled differential equations [Kid+20a, Definition B.4].

Note that the signature may be defined for arbitrary depths – indeed this was used elsewhere in this Appendix, see Definition C.6 – and the above is simply the special case of interest to us here.

Definition C.29 (Partition)**.** *A partition of $[0,T]$ is some finite sequence $\mathcal{D} = (t_0, \dots t_n)$ with $0 = t_0 < \cdots < t_n = T$.*

Definition C.30 (Inhomogeneous p-variation, [LCL04, Section 3.2.1], [FV10, Definition 8.6.(i)])**.** *Let $X_1, X_2\colon [0,T] \to \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d\times d}$.*

For $p \in [2,3)$, define

$$\rho_p(X_1, X_2) = \max_{k=0,1,2} \sup_{\mathcal{D}} \left(\sum_{t_i \in \mathcal{D}} \left\| \pi_k\Big(X_1(t_{i+1}) - X_1(t_i) - X_2(t_{i+1}) + X_2(t_i)\Big) \right\|^{p/k} \right)^{k/p},$$

where the supremum is taken over all partitions $\mathcal{D}$ of $[0,T]$.

*Then define the p-*variation metric *between X_1 and X_2 as*

$$d_p(X_1, X_2) + \max_{k=0,1,2} \|\pi_k(X_1 - X_2)\|_\infty + \rho_p(X_1, X_2).$$

Remark C.31. *Notions of p-variation are crucial to rough path theory. Correspondingly several remarks are in order.*

- *If we were to take $p = 1$, $X_2 \equiv 0$, and consider only $k = 1$, then $\rho_p(X_1, X_2)$ would recover the definition of the bounded/total variation seminorm of X_1. Indeed p-variation should be thought of as a generalisation of total variation.*
- *It is immediate from the definition that d_p convergence implies uniform convergence. However if $\|\pi_k(X_1 - X_2)\|_\infty$ is replaced with just $|\pi_k(X_1(0) - X_2(0))|$, then in fact d_p convergence still implies uniform convergence [LCL04, Definition 3.12].*

- *There are several quantities related to p-variation, often going by similar names [FV10, Chapter 8]. Take care not to trip up when reading the literature.*
- *p-variation is a subtly different notion to that of quadratic variation used in classical SDE theory. Where p-variation takes a supremum over all partitions, quadratic variation instead takes a limit. The quadratic variation of a path may be smaller than its 2-variation, and in particular almost all samples of Brownian motion have finite quadratic variation but infinite 2-variation.*
- *A path which is Hölder continuous with exponent $\alpha \in (\frac{1}{3}, \frac{1}{2}]$ has finite $\frac{1}{\alpha}$-variation. For example Brownian motion is Hölder continuous with exponent α for all $\alpha \in (0, \frac{1}{2})$, and correspondingly Brownian motion has finite p-variation for all $p \in (2, 3)$.*

Definition C.32. *We say that a sequence of continuous and bounded variation paths $x_n \colon [0,T] \to \mathbb{R}^d$ converge in p-variation to a continuous $X \colon [0,T] \to \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d}$ if*

$$d_p(\mathrm{sig}^2(x_n), X) \to 0 \text{ as } n \to \infty.$$

Whenever such a limit exists, we refer to X as a geometric p-rough path.

Theorem C.33 (Brownian motion as a geometric rough path, [FV10, Corollaries 13.20, 13.22])**.** *Let $w \colon [0,T] \to \mathbb{R}^d$ be a Brownian motion. Let $D_n = (t_0, \ldots, t_n)$ be a uniform partition of $[0,T]$. Let w_n be the unique continuous piecewise linear function with knots t_j such that $w_n(t_j) = w(t_j)$.*

Let $W \colon [0,T] \to \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d}$ be defined by

$$W(t) = \left(1, w(t) - w(0), \int_0^t (w(s) - w(0)) \otimes \circ \mathrm{d}w(s)\right)$$

with $\circ$ denoting that the integral is defined in the Stratonovich sense.

Let $p \in (2,3)$ (but not $p = 2$). Then w_n converges to W in p-variation almost surely. W is called Stratonovich Brownian motion, and it is almost surely a geometric p-rough path.

Summary Let us take stock of what has been introduced.

We have seen that for any continuous bounded variation path $x \colon [0,T] \to \mathbb{R}^d$, we may consider 'enhancing' it with $\int_0^t (x(s) - x(0)) \otimes \mathrm{d}x(s)$. This extra term is completely determined by the base path x.

Meanwhile for a Brownian path $w \colon [0,T] \to \mathbb{R}^d$, we may consider enhancing it with $\int_0^t (w(s) - w(0)) \otimes \circ \mathrm{d}w(s)$. This time the extra term is not completely determined by the base path, and we had to make a choice: what notion of integration to use. We chose Stratonovich integration, but could equally have chosen another form of integration, such as Itô integration. Because w is not of bounded variation, then there is not a single unique notion of integration.

One way or the other, we have lifted ourselves into the larger dimensional space $\mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d}$. In this lifted space we have defined the notion of convergence we are interested in, namely p-variation. In performing this lift, it transpires that we have *completely defined* what it means to integrate against a path $[0, T] \to \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d}$: in particular we have already made the choice of Stratonovich over Itô. As such we will sometimes think of *the lift (*$\mathrm{sig}^2(x)$ *or* W*) as the fundamental object*, and reverse what is defined by what, so that x or w is defined as the projection of $\mathrm{sig}^2(x)$ or W by π_1.

C.3.3.2 Rough differential equations and the universal limit theorem

We are now ready to define what is meant by a rough differential equation.

Definition C.34 ($\mathrm{Lip}(\gamma)$ functions)**.** *Let $\gamma > 1$. A function $f \colon \mathbb{R}^d \to \mathbb{R}^d$ is said to be* $\mathrm{Lip}(\gamma)$ *if it is bounded, $\lfloor \gamma \rfloor$-times differentiable, all derivative are bounded, and the highest derivative is $(\gamma - \lfloor \gamma \rfloor)$-Hölder continuous.*[2]

Theorem C.35 (Universal limit theorem, [LCL04, Theorem 5.3], [FV10, Theorems 10.29, 10.50, 10.57])**.** *Let $d_x, d_y \in \mathbb{N}$. Let $p \in (2, 3)$ and let $\gamma > p$. Let $f \colon \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_x}$ be either linear or* $\mathrm{Lip}(\gamma)$*.*

Let $\zeta_n \in \mathbb{R}^{d_y}$ be a sequence converging to $\zeta \in \mathbb{R}^{d_y}$.

Let $x_n \colon [0, T] \to \mathbb{R}^{d_x}$ be a sequence of continuous bounded variation paths, which converge in p-variation to a geometric p-rough path $X \colon [0, T] \to \mathbb{R} \times \mathbb{R}^{d_x} \times \mathbb{R}^{d_x \times d_x}$.

Let $y_n \colon [0, T] \to \mathbb{R}^{d_y}$ solve the CDEs

$$y_n(0) = \zeta_n, \qquad \mathrm{d}y_n(t) = f(y_n(t)) \, \mathrm{d}x_n(t).$$

Then there exists a unique geometric p-rough path $Y \colon [0, T] \to \mathbb{R} \times \mathbb{R}^{d_y} \times \mathbb{R}^{d_y \times d_y}$ such that $Y(0) = (1, \zeta, 0)$ and y_n converges to Y in p-variation.

Moreover, the limit Y depends only on X, f and ζ, and in particular not on the sequence x_n. As such it is referred to as the 'universal limit', and is said to solve the 'rough differential equation'

$$\mathrm{d}Y(t) = f(\pi_1(Y(t))) \, \mathrm{d}X(t).$$

Given drift μ, diffusion σ, and Brownian motion w, then we may now immediately deduce a corollary specifically for SDEs, by taking $f = \begin{bmatrix} \mu & 0 \\ 0 & \sigma \end{bmatrix}$ and $x_n(t) = [t, w_n(t)]$ in the above result.

[2] This notation is conventional in rough path theory; when other fields have needed this concept it is sometimes denoted in other ways, such as '$f \in C_b^{k+\alpha}$' with $k = \lfloor \gamma \rfloor$, $\alpha = \gamma - \lfloor \gamma \rfloor$, and b denoting boundedness.

Notation. *Let* $\mathcal{T}(t) = \mathrm{sig}^2_{0,t}(\mathrm{id}) \in \mathbb{R}\times\mathbb{R}\times\mathbb{R}$*, where* id *is the identity function. Where a stochastic integral* $\int \cdots \circ \mathrm{d}w(t)$ *will become* $\int \cdots \mathrm{d}W(t)$ *when lifting to the rough setting, a deterministic integral* $\int \cdots \mathrm{d}t$ *will become* $\int \cdots \mathrm{d}\mathcal{T}(t)$*.*

Corollary C.36 (Universal limit theorem for Stratonovich SDEs)**.** *Let* $d_w, d_y \in \mathbb{N}$*. Let* $p \in (2,3)$ *and let* $\gamma > p$*. Let* $\mu\colon \mathbb{R}\times\mathbb{R}^{d_y} \to \mathbb{R}^{d_y}$ *and* $\sigma\colon \mathbb{R}\times\mathbb{R}^{d_y} \to \mathbb{R}^{d_y\times d_w}$ *be either linear or* $\mathrm{Lip}(\gamma)$*.*

Let $\zeta_n \in \mathbb{R}^{d_y}$ *be a sequence converging to* $\zeta \in \mathbb{R}^{d_y}$*.*

Let $w_n\colon [0,T] \to \mathbb{R}^{d_w}$ *be as defined in Theorem C.33, converging to the Stratonovich Brownian motion* $W\colon [0,T] \to \mathbb{R}\times\mathbb{R}^{d_w}\times\mathbb{R}^{d_w\times d_w}$*.*

Let $y_n\colon [0,T] \to \mathbb{R}^{d_y}$ *solve the (random) CDEs*

$$y_n(0) = \zeta_n, \qquad \mathrm{d}y_n(t) = \mu(t, y_n(t))\,\mathrm{d}t + \sigma(t, y_n(t))\,\mathrm{d}w_n(t).$$

Then y_n *converge in* p*-variation almost surely to a unique geometric* p*-rough path* Y *solving the rough differential equation*

$$Y(0) = (1,\zeta,0), \qquad \mathrm{d}Y(t) = \mu(s,\pi_1(Y(s)))\,\mathrm{d}\mathcal{T}(t) + \sigma(s,\pi_1(Y(s)))\,\mathrm{d}W(s), \quad \text{(C.12)}$$

and moreover the process $y(t) = \pi_1(Y(t))$ *satisfies the Stratonovich SDE*

$$\mathrm{d}y(t) = \mu(t,y(t))\,\mathrm{d}t + \sigma(t,y(t)) \circ \mathrm{d}w(t)$$

defined in the classical sense.

See Section C.3.4.3 for an appendix on some technical points associated with Theorems C.35 and C.36.

Summary The key point of the universal limit theorem is that instead of defining a differential equation driven by some continuous path $[0,T] \to \mathbb{R}^d$ (for example as was done with CDEs), we have defined a differential equation driven by an enhanced path $[0,T] \to \mathbb{R}\times\mathbb{R}^d\times\mathbb{R}^{d\times d}$.

In particular we have defined integration against Stratonovich Brownian motion. Note the terminology of 'Stratonovich Brownian motion' rather than 'Stratonovich SDE': the rough path approach has entirely contained both the 'Stratonovich-ness' and the stochasticity to within the enhanced Brownian motion W. After that we simply sample W, and deterministically solve the RDE driven by this Brownian sample.

Rough objects are easily dealt with via the universal limit theorem: any time we encounter an RDE we may simply approximate it with a sequence of CDEs, perform the appropriate manipulations, and then take a limit.

Overall, we see that the notions of stochasticity, roughness, and control have been factored apart. This is in contrast to the classical approach to SDEs, which muddles together these three separate ideas.

In passing, note the relatively high regularity $\mathrm{Lip}(\gamma)$ assumed of the vector fields. This is needed to 'offset' the roughness of the driving signal.

C.3.3.3 Rough adjoints

Having established what is meant by a rough differential equation, and how it may be used as a notion of solution to a stochastic differential equation, it is now straightforward to derive our main result. This is the precise statement corresponding to the informal Theorem 5.10.

Theorem C.37 (Optimise-then-discretise for SDEs)**.** *Fix $d_y, d_w \in \mathbb{N}$ and $\gamma > 2$. Let $\mu\colon \mathbb{R} \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y}$ and $\sigma\colon \mathbb{R} \times \mathbb{R}^{d_y} \to \mathbb{R}^{d_y \times d_w}$ be linear or $\mathrm{Lip}(\gamma + 1)$.*

Let $W\colon \mathbb{R} \times \mathbb{R}^{d_w} \times \mathbb{R}^{d_w \times d_w}$ denote a Stratonovich Brownian motion as in Theorem C.33.

Let $L\colon \mathbb{R}^{d_y} \to \mathbb{R}$ be continuously differentiable (and scalar just for simplicity). Let $y_0 \in \mathbb{R}^{d_y}$ and let $Y\colon [0,T] \to \mathbb{R} \times \mathbb{R}^{d_y} \times \mathbb{R}^{d_y \times d_y}$ solve the rough differential equation

$$Y(0) = (1, y_0, 0), \qquad \mathrm{d}Y(t) = \mu(t, y(t))\,\mathrm{d}\mathcal{T}(t) + \sigma(t, y(t))\,\mathrm{d}W(t), \tag{C.13}$$

where $y(t) = \pi_1(Y(t))$.

Consider the adjoint process A solving the backwards-in-time linear rough differential equation

$$\begin{aligned} A(T) &= \left(1, \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(T)}, 0\right), \\ \mathrm{d}A(t) &= -a(t)^\top \frac{\partial \mu}{\partial y}(t, y(t))\,\mathrm{d}\mathcal{T}(t) - a(t)^\top \frac{\partial \sigma}{\partial y}(t, y(t))\,\mathrm{d}W(t), \end{aligned} \tag{C.14}$$

where $a(t) = \pi_1(A(t))$.

Then the solution A exists and is unique, and for almost all sample paths W, we have $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)} \in \mathbb{R}^{d_y}$.

For completeness we note that the non-rough (classical SDE) equivalent to (C.13) is

$$y(0) = y_0, \qquad \mathrm{d}y(t) = \mu(t, y(t))\,\mathrm{d}t + \sigma(t, y(t)) \circ \mathrm{d}w(t),$$

whilst the non-rough equivalent to (C.14) is

$$\mathrm{d}a_{k_1}(t) = -a_{k_2}(t)\frac{\partial \mu_{k_2}}{\partial y_{k_1}}(t, y(t))\,\mathrm{d}t - a_{k_2}(t)\frac{\partial \sigma_{k_2,k_3}}{\partial y_{k_1}}(t, y(t)) \circ \mathrm{d}w_{k_3}(t),$$

in Einstein notation is over the indices k_1, k_2, k_3. This latter equation is not technically defined as a Stratonovich SDE (a is not measurable with respect to the natural filtration of w), and so is best interpreted as the projection under π_1 of the rough differential equation.

Proof. Let $p \in (2, \gamma)$.

Let $w_n\colon [0,T] \to \mathbb{R}^{d_w}$ be as defined in Theorem C.33, converging in p-variation to the Stratonovich Brownian motion W.

Let $y_n\colon [0,T] \to \mathbb{R}^{d_y}$ solve the (random) CDEs

$$y_n(0) = y_0, \qquad \mathrm{d}y_n(t) = \mu(s, y_n(s))\,\mathrm{d}s + \sigma(s, y_n(s))\,\mathrm{d}w_n(s),$$

which by the universal limit theorem (Theorem C.36) converge to Y in p-variation almost surely.

By optimise-then-discretise for CDEs (Theorem 5.9), each adjoint process

$$a_n(t) = \frac{\mathrm{d}L(y_n(T))}{\mathrm{d}y_n(t)} \tag{C.15}$$

satisfies

$$\mathrm{d}a_{n,k_1}(t) = -a_{n,k_2}(t)\frac{\partial \mu_{k_2}}{\partial y_{k_1}}(t, y_n(t))\,\mathrm{d}t - a_{n,k_2}(t)\frac{\partial \sigma_{k_2,k_3}}{\partial y_{k_1}}(t, y_n(t))\,\mathrm{d}w_{n,k_3}(t) \tag{C.16}$$

starting from the terminal condition $a_n(T) = \frac{\mathrm{d}L(y_n(T))}{\mathrm{d}y_n(T)}$, and Einstein notation is used over the indices k_1, k_2, k_3.

As in the proof of Theorem 5.9, we interpret the solution (C.16) as the solution to the CDE

$$\mathrm{d}a_{n,k_1}(t) = -a_{n,k_2}(t)\,\mathrm{d}M_{n,k_2,k_1}(t),$$

where $M_n\colon [0,T] \to \mathbb{R}^{d_y \times d_y}$ is a bounded variation path satisfying

$$\mathrm{d}M_n(t) = \frac{\partial \mu}{\partial y}(t, y_n(s))\,\mathrm{d}s + \frac{\partial \sigma}{\partial y}(t, y_n(s))\,\mathrm{d}w_n(s). \tag{C.17}$$

We would like to take $n \to \infty$ in equation (C.17) via the universal limit theorem. The version we have stated here does not allow for n-dependent vector fields (note that the vector fields depend on y_n). This is resolved by the standard trick of replacing M_n with $[M_n, z_n]$ and $[s, w_n(s)]$ with $[s, w_n(s), y_n(s)]$, where

$$\begin{aligned}\mathrm{d}M_n(t) &= \frac{\partial \mu}{\partial y}(t, z_n(s))\,\mathrm{d}s + \frac{\partial \sigma}{\partial y}(t, z_n(s))\,\mathrm{d}w_n(s),\\ \mathrm{d}z_n(t) &= \mathrm{d}y_n(t),\end{aligned} \tag{C.18}$$

so that the vector fields are a function of the state $[M_n, z_n]$ only.

We may now take $n \to \infty$ in equation (C.18) by the universal limit theorem, as $\frac{\partial \mu}{\partial y}$, $\frac{\partial \sigma}{\partial y}$ are $\mathrm{Lip}(\gamma)$.

As such M_n converges in p-variation almost surely to a geometric p-rough path

$$\mathbb{M}\colon [0,T] \to \mathbb{R} \times \mathbb{R}^{d_y \times d_y} \times \mathbb{R}^{d_y \times d_y \times d_y \times d_y}$$

satisfying[3]

$$\mathrm{d}\mathbb{M}(t) = \frac{\partial \mu}{\partial y}(s, y(s))\,\mathrm{d}\mathcal{T}(s) + \frac{\partial \sigma}{\partial y}(s, y(s))\,\mathrm{d}W(s).$$

[3]Implying that the corresponding the non-rough $M(t) = \pi_1(\mathbb{M})$ satisfies

$$\mathrm{d}M(t) = \frac{\partial \mu}{\partial y}(s, y(s))\,\mathrm{d}s + \frac{\partial \sigma}{\partial y}(s, y(s)) \circ \mathrm{d}w(s).$$

By the universal limit theorem (with linear vector field), then a_n now converges in p-variation almost surely to a geometric p-rough path A satisfying

$$\mathrm{d}A(t) = -\pi_1(A(t))\,\mathrm{d}\mathbb{M}(t),$$

which we may rewrite as

$$\mathrm{d}A(t) = -a(t)^\top \frac{\partial \mu}{\partial y}(t, y(t))\,\mathrm{d}\mathcal{T}(t) - a(t)^\top \frac{\partial \sigma}{\partial y}(t, y(t))\,\mathrm{d}W(t),$$

with $a(t) = \pi_1(A(t))$. We are now halfway through the proof, and have derived our desired RDE.

Overall what we have done is very simple: just take the limit $n \to \infty$ in (C.16). *The argument until now has just been to shuffle things around so that the appropriate theorems may be applied.*

It remains to show that $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)}$. (Implying in particular the terminal condition $A(T) = \left(1, \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(T)}, 0\right)$.)

Fix $s, t \in [0, T]$ with $s < t$ and let $J_n(t) = \frac{\mathrm{d}y_n(t)}{\mathrm{d}y_n(s)}$. [FV10, Theorem 4.4] gives the 'forward sensitivity' result for CDEs (the forward-mode autodifferentiation counterpart to the reverse-mode autodifferentiation version we are currently deriving), and states that the Jacobian J_n evolves according to the CDE

$$\mathrm{d}J_n(t) = \frac{\partial \mu}{\partial y}(t, y_n(t))J_n(t)\,\mathrm{d}t + \frac{\partial \sigma}{\partial y}(t, y_n(t))J_n(t)\,\mathrm{d}w_n(t).$$

By the universal limit theorem (and applying the same trick as with $\mathbb{M}$, moving y_n into the state and control), this sequence converges in p-variation almost surely to a geometric p-rough path

$$\mathbb{J}\colon [0, T] \to \mathbb{R} \times \mathbb{R}^{d_y \times d_y} \times \mathbb{R}^{d_y \times d_y \times d_y \times d_y}$$

satisfying

$$\mathrm{d}\mathbb{J}(t) = \frac{\partial \mu}{\partial y}(t, y(t))\pi_1(\mathbb{J}(t))\,\mathrm{d}\mathcal{T}(t) + \frac{\partial \sigma}{\partial y}(t, y(t))\pi_1(\mathbb{J}(t))\,\mathrm{d}W(t).$$

We appeal to our final theorem. [FV10, Theorem 11.3] gives the 'forward sensitivity' result for RDEs, satisfied by the the lift of $\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}$. And unsurprisingly, this is the same equation we have just derived. So by uniqueness of solution $\pi_1(\mathbb{J}(t)) = \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}$.

In summary: we have shown that the Jacobian flow $t \mapsto \frac{\mathrm{d}y_n(t)}{\mathrm{d}y_n(s)}$ converges, and moreover it converges to (the lift of) $t \mapsto \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}$. (In each case with respect to p-variation almost surely, and therefore also uniformly almost surely and therefore also pointwise almost surely.)

Consequently and by continuous differentiability of L,

$$a_n(t) = \frac{\mathrm{d}L(y_n(T))}{\mathrm{d}y_n(t)} = \frac{\mathrm{d}L}{\mathrm{d}y}(y_n(T))\frac{\mathrm{d}y_n(T)}{\mathrm{d}y_n(t)} \to \frac{\mathrm{d}L}{\mathrm{d}y}(y(T))\frac{\mathrm{d}y(T)}{\mathrm{d}y(t)}$$

pointwise over t. As also $a_n(t) \to a(t)$ pointwise (as $a_n \to A$ in p-variation), then by uniqueness of limits $a(t) = \frac{\mathrm{d}L(y(T))}{\mathrm{d}y(t)}$. □

Remark C.38. *The above method of proof may be trivially extended to any RDE driven by a geometric p-rough path.*

C.3.4 Comments

C.3.4.1 On ODEs

Optimise-then-discretise for ODEs is also referred to as Pontryagin's Maximum Principle (PMP). Often only special cases of the result shown here are presented; frequently only the behaviour at an optimum is considered.

Our proof is new here – whilst combining flavours of various previous proofs – and much simpler than most versions found in the literature. The fact that it lacks any meaningful reliance on the differentiability of the forward or reverse sensitivities is what allows the later generalisation to the CDE case.

The basic idea of finding two processes (a, z) for which $\frac{\mathrm{d}}{\mathrm{d}t}(a(t)z(t)) = 0$ is the same notion used in duality of stochastic processes [JK14, Proposition 4.1.(ii)], and was inspired by the fact that PMP may be proved in a similar way [Li20a, Proof 2.7]. The discrete analogue is [GW08, Equation (3.4)]. That the proof proceeds by considering the interaction between the forward and reverse sensitivities is vaguely reminiscent of [Fro+21], who derive reverse sensitivities by combining forward sensitivities and transposition rules.

C.3.4.2 On CDEs

Optimise-then-discretise for CDEs was first shown in [Kid+20b, Appendix A.1], but this was never formally published. The proof we present here is substantially simpler than that of [Kid+20b], and is new here.

C.3.4.3 On SDEs

Optimise-then-discretise for CDEs was first shown in [Kid+20b, Appendix A.2, Appendix A.3], but this was never formally published. The presentation shown here follows essentially the same lines, whilst being a bit simpler and fixing some technical holes.

The following are technical notes (mostly for the expert) on the theorems given here.

The universal limit theorem (Theorem C.35) The statement given here is a slightly custom mish-mash of the different ways in which this theorem is sometimes expressed. The bulk of the statement comes from [LCL04, Theorem 5.3], although we have simplified the statements about p-variation from the general (potentially non-geometric) form given there to the geometric-only form considered here.

Surprisingly, we could not find a form of this theorem which explicitly included the convergence of the initial points ζ_n, as is stated here. This may be recovered from Davie's Lemma [FV10, Theorem 10.29].

For simplicity of presentation the statement given here has elided the usual notion by which integrator and integrand are coupled together into a single rough path.

The universal limit theorem for Stratonovich SDEs (Theorem C.36) It is possible to admit lower regularity on the drift μ than the $\mathrm{Lip}(\gamma)$ assumed here. This is because the drift is not integrated against Brownian motion, but is only integrated against time – for which classical ODE theory would demand only that μ be Lipschitz. See [FV10, Chapter 12].

Note the dependence on time in the vector fields (μ, σ), despite this not appearing in Theorem C.35. Technically speaking we have accomplished this by concatenating $\mathrm{d}t = 1\mathrm{d}t + 0\,\mathrm{d}w_n(t)$ to each y_n, so that time becomes part of the state. This is only possible because time is also part of the control.

Having taken the control to be the limit of $(t, w_n(t)) \in \mathbb{R}^{1+d_w}$, its rough lift will actually be in $\mathbb{R} \times \mathbb{R}^{1+d_w} \times \mathbb{R}^{(1+d_w)\times(1+d_w)}$, which is larger than the two separate $\mathcal{T}(t) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$, $W(t) \in \mathbb{R} \times \mathbb{R}^{d_w} \times \mathbb{R}^{d_w \times d_w}$. We have elided the reduction to two separate terms, and (C.12) may just be considered a formal notation for the 'true' RDE if the reader so prefers.

Optimise-then-discretise for Stratonovich SDEs (Theorem C.37) For all of the theorems we have seen in this section – ODEs, CDEs, and SDEs – we have for simplicity only considered derivatives of the solution with respect to the initial condition y_0. In general we may wish to also consider derivatives with respect to either the driving signal x or the vector field f.

To the best of our knowledge, no complete account of every case (both forward and backward sensitivities; derivatives with respect to all of y_0, x, f; ODEs, CDEs, SDEs or the general RDE case) yet exists in the literature. (Although the forward sensitivity with respect to x may be found in [FV10, Theorem 4.4, Theorem 11.3, Exercise 11.10].)

In practice the result we have shown is almost always the most important, from which important special cases of the other sensitivities may be derived. For example, derivatives with respect to θ for $f = f_\theta$ may be derived by replacing y with $[y, \theta]$, y_0 with $[y_0, \theta]$ and f_θ with $f(y, \theta) = [f_\theta, 0]$.

The assumed regularity of $\mathrm{Lip}(\gamma + 1)$ is for simplicity of presentation and is much

higher than is probably necessary. As with the universal limit theorem for Stratonovich SDEs, we expect to require only minimal regularity on the drift. Moreover substituting the universal limit theorem for [FV10, Theorem 17.1] when obtaining $\mathbb{M}$ would allow for only $\mathrm{Lip}(\gamma)$ regularity on the diffusion. Something similar could like be arranged when obtaining $\mathbb{J}$.

Likewise for simplicity of presentation, the proof leaves a few things implicit (including the inclusion of y_n in the control when taking the limit in Φ_n; that the Jacobian $\frac{\mathrm{d}y_n(T)}{\mathrm{d}y_n(t)}$ should be thought of as a solution map from t to T).

C.4 Convergence and stability of the reversible Heun method

Recall the definition of the reversible Heun method, as applied to ODEs.

$$\begin{aligned} t_{j+1} &= t_j + \Delta t, \\ \widehat{y}_{j+1} &= 2y_j - \widehat{y}_j + \mu_j \Delta t, \\ \mu_{j+1} &= \mu(t_{j+1}, \widehat{y}_{j+1}), \\ y_{j+1} &= y_j + \frac{1}{2}(\mu_j + \mu_{j+1})\Delta t. \end{aligned}$$

(See Section 5.3.2.2 and/or [Kid+21a, Appendix D] for the full SDE case.)

C.4.1 Convergence

Theorem 5.18. *The reversible Heun method, when applied to ODEs, is a second-order method.*

Proof. Consider a two-step update over the $\widehat{y}$ component. Then

$$\widehat{y}_{j+1} = \widehat{y}_{j-1} + 2\mu(t_j, \widehat{y}_j)\Delta t,$$

which is precisely the equation for the leapfrog/midpoint method [Sha09]; see also Section 5.3.2.4. This is a second-order method.

Therefore

$$\begin{aligned}
y_n &= y_0 + \sum_{j=0}^{n-1} \frac{1}{2} \left(\mu_j + \mu_{j+1}\right) \Delta t \\
&= y_0 + \sum_{j=0}^{n-1} \frac{1}{2} \left(\mu(t_j, y(t_j)) + \mu(t_{j+1}, y(t_{j+1})) + \mathcal{O}(\Delta t^2)\right) \Delta t \\
&= y_0 + \sum_{j=0}^{n-1} \left(\frac{1}{2} \left(\mu(t_j, y(t_j)) + \mu(t_{j+1}, y(t_{j+1}))\right) \Delta t + \mathcal{O}(\Delta t^3) \right) \\
&= y_0 + \sum_{j=0}^{n-1} \left(\int_{t_j}^{t_{j+1}} \mu(t, y(t)) \, \mathrm{d}t + \mathcal{O}(\Delta t^3) + \mathcal{O}(\Delta t^3) \right) \\
&= y(t_n) + \mathcal{O}(\Delta t^2).
\end{aligned}$$

□

C.4.2 Stability

Definition C.39. *Fix some numerical differential equation solver (we will consider just the reversible Heun method). Let* $\{y_{j,\lambda,\Delta t}\}_{j\in\mathbb{N}}$ *be the numerical approximation to the linear (Dahlquist) test equation*

$$y(0) \in \mathbb{R}, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = \lambda y(t) \qquad \text{for } t \in [0, \infty)$$

with $\lambda \in \mathbb{C}$*, numerical step size* $\Delta t > 0$ *and* $y_{j,\lambda,\Delta t} \approx y(j\Delta t)$*. We define the* region of stability *as*

$$\{\lambda \Delta t \in \mathbb{C} \mid \{y_{j,\lambda,\Delta t}\}_{j\in\mathbb{N}} \text{ is uniformly bounded over } j\} .$$

That is, there exists a constant C *depending on* λ *and* Δt *but independent of* j *for which* $|y_{j,\lambda,\Delta t}| < C$*.*

Remark C.40. *We have chosen to define stability in terms of the boundedness of the numerical solution. (Which is the behaviour of the analytical solution for* $\mathrm{Re}(\lambda) \leq 0$*.) Some authors define the region of stability in terms of the slightly stronger condition that the numerical solution converges to zero. (Which is the behaviour of the analytical solution for* $\mathrm{Re}(\lambda) < 0$*.)*

Theorem 5.20. *The region of stability for the reversible Heun method (for ODEs) is the complex interval* $[-i, i]$*.*

Proof. Consider a two-step update over the $\widehat{y}$ component. Then

$$\widehat{y}_{j+1} = \widehat{y}_{j-1} + 2\mu(t_j, \widehat{y}_j)\Delta t,$$

which is precisely the equation for the leapfrog/midpoint method [Sha09]; see also Section 5.3.2.4.

This is a difference equation for $\widehat{y}$, which may be solved explicitly to obtain

$$\widehat{y}_j = \alpha_1 \eta^j + \beta \kappa^j,$$

where

$$\begin{aligned}
\alpha &= \frac{1}{2} y_0 \left(1 + \frac{1}{\sqrt{1+\lambda^2\Delta t^2}}\right), \\
\beta &= \frac{1}{2} y_0 \left(1 - \frac{1}{\sqrt{1+\lambda^2\Delta t^2}}\right), \\
\eta &= \lambda\Delta t + \sqrt{1+\lambda^2\Delta t^2}, \\
\kappa &= \lambda\Delta t - \sqrt{1+\lambda^2\Delta t^2}.
\end{aligned}$$

(With $z \mapsto \sqrt{1+z^2}$ putting branch cuts down $(-i\infty, -i)$ and $(i, i\infty)$.)

Therefore

$$\begin{aligned}
y_n &= y_0 + \sum_{j=0}^{n-1} \frac{1}{2} \left(\mu_j + \mu_{j+1}\right) \Delta t \\
&= y_0 + \frac{1}{2}\lambda\Delta t\alpha(1+\eta) \sum_{j=0}^{n-1} \eta^j + \frac{1}{2}\lambda\Delta t\beta(1+\kappa) \sum_{j=0}^{n-1} \kappa^j \\
&= y_0 + \frac{1}{2}\lambda\Delta t\alpha \frac{1+\eta}{1-\eta}(1-\eta^n) + \frac{1}{2}\Delta t\beta \frac{1+\kappa}{1-\kappa}(1-\kappa^n). \qquad \text{(C.19)}
\end{aligned}$$

Consider when $\lambda\Delta t \in [-i, i]$. Then $\eta = \lambda\Delta t + \sqrt{1 - |\lambda\Delta t|^2}$ and $\kappa = \lambda\Delta t - \sqrt{1 - |\lambda\Delta t|^2}$, so that

$$\begin{aligned}
|\eta|^2 &= |\lambda\Delta t|^2 + (1 - |\lambda\Delta t|^2) = 1, \\
|\kappa|^2 &= |\lambda\Delta t|^2 + (1 - |\lambda\Delta t|^2) = 1,
\end{aligned}$$

and therefore by (C.19) $|y_n|$ is bounded independent of n.

Conversely consider when $\lambda\Delta t \notin [-i, i]$. Then $|\eta| \neq 1$. (A fact most easily verified via the usual 'proof by dodgy diagram'[4] typically used for determining the image of a composition of conformal functions.) Now $\eta\kappa = -1$ so one term in C.19 will decay and the other will blow up as $n \to \infty$; consequently $|y_n|$ is not bounded over n. □

Remark C.41. *This is the same region of stability as both the asynchronous leapfrog method [Zhu+21, Appendix A.4] and the leapfrog/midpoint method [Sha09, Section 2].*

[4] A term which we must thank Hilary Priestley for introducing to our lexicon.

C.5 Brownian Interval

This Appendix continues the discussion and definition of the Brownian Interval of Section 5.5.3.

C.5.1 Algorithmic definitions

See Algorithms 7–10 for the full description of how the binary tree is traversed, modified, and $w(s,t)$ subsequently sampled.

Let *List* be an ordered data structure that can be appended to, and iterated over sequentially. For example a linked list would suffice. Let *Node* denote a 5-tuple consisting of an interval, a seed, and three optional *Node*s, corresponding to the parent node, and two child nodes, respectively. (Optional as the root has no parent and leaves have no children.)

We let `split_seed` denote a splittable PRNG as above and `bridge` to denote equation (5.16). We use $*$ to denote an unfilled part of the data structure, equivalent to `None` in Python or a null pointer in C/C++; in particular this is used as a placeholder for the (nonexistent) children of leaf nodes and the (nonexistent) parent of the root node.

We use $=$ to denote the creation of a new local variable, and $\leftarrow$ to denote in-place modification of a variable.

We use $x : T$ to denote that x is a value with type T.

Algorithm 7: Sampling the Brownian Interval

Input: Interval $[s,t] \subseteq [0,T]$
State: Binary tree with elements of type *Node*, with root $\widehat{I} = ([0,T], \widehat{\rho}, *, \widehat{I}_{\texttt{left}}, \widehat{I}_{\texttt{right}})$. A *Node* $\widehat{J}$, which provides a hint for where to start the traversal from (for efficiency).
Result: Sample increment $W_{s,t}$

```
# The returned 'nodes' is a list of Nodes whose intervals partition [s, t].
# Practically speaking this will usually have only one or two elements.
nodes = traverse(Ĵ, [s, t])
Ĵ ← nodes[−1]        # last element of 'nodes'
return Σ_{I∈nodes} sample(I, Î)
```

C.5.2 Discussion

There are some further technical considerations worth mentioning. Recall that the context we are explicitly considering is when sampling Brownian motion to solve an

Algorithm 8: Definitions of `traverse` and `traverse_impl`. The argument nodes is passed by reference, that is to say it is mutated and these mutations are visible to the calling function.

```
def traverse(I : Node, [c, d] : Interval):
    Let nodes be an empty List.
    traverse_impl(I, [c, d], nodes)
    return nodes
def traverse_impl(I : Node, [c, d] : Interval, nodes : List[Node]):
    Decompose ([a, b], ρ, I_parent, I_left, I_right) = I

    # Outside our jurisdiction - pass to our parent
    if c < a or d > b then
        traverse_impl(I_parent, [c, d], nodes)
        return
    # It's I that is sought. Add I to the list and return.
    if c = a and d = b then
        nodes.append(I)
        return
    # Check if I is a leaf or not.
    if I_left is * then
        # I is a leaf
        if a = c then
            # Create children and add on the left child.
            bisect(I, d)      # I_left is created.
            nodes.append(I_left)
            return
        # Otherwise create children and pass on to our right child.
        bisect(I, c)      # I_right is created.
        traverse_impl(I_right, [c, d], nodes)
        return
    else
        # I is not a leaf.
        Decompose ([a, m], ρ_left, I, I_ll, I_lr) = I_left
        if d ≤ m then
            # Strictly our left child's problem.
            traverse_impl(I_left, [c, d], nodes)
            return
        if c ≥ m then
            # Strictly our right child's problem.
            traverse_impl(I_right, [c, d], nodes)
            return
        # A problem for both of our children.
        traverse_impl(I_left, [c, m], nodes)
        traverse_impl(I_right, [m, d], nodes)
        return
```

Algorithm 9: Definition of `bisect`

def `bisect`*(I : Node, $x : \mathbb{R}$)*:
    # Only called on leaf nodes
    Decompose $([a, b], \rho, I_{\texttt{parent}}, *, *) = I$
    $\rho_{\texttt{left}}, \rho_{\texttt{right}} = \texttt{split\_seed}(\rho)$
    $I_{\texttt{left}} = ([a, x], \rho_{\texttt{left}}, I, *, *)$
    $I_{\texttt{right}} = ([x, b], \rho_{\texttt{right}}, I, *, *)$
    $I \leftarrow ([a, b], \sigma, I_{\texttt{parent}}, I_{\texttt{left}}, I_{\texttt{right}})$
    return

Algorithm 10: Definition of `sample`

def `sample`(I : *Node*, $\widehat{I}$: *Node*)**:**
    if I is $\widehat{I}$ **then**
        Decompose $([a, b], \widehat{\rho}, *, \widehat{I}_{\text{left}}, \widehat{I}_{\text{left}}) = \widehat{I}$
        return $\mathcal{N}(0, T)$ sampled with seed $\widehat{\rho}$.
    Decompose $([a, b], \rho, I_{\texttt{parent}}, I_{\texttt{left}}, I_{\texttt{right}}) = I$
    Decompose $([a_p, b_p], \rho_p, I_{pp}, I_{lp}, I_{rp}) = I_{\texttt{parent}}$
    $w_{\texttt{parent}} = \texttt{sample}(I_{\texttt{parent}})$
    if I is I_{rp} **then**
        $w_{\texttt{left}} \sim \texttt{bridge}(a_p, b_p, a, w_{\texttt{parent}})$ sampled with seed ρ
        return $w_{\texttt{parent}} - w_{\texttt{left}}$
    else
        # I is I_{lp}
        return $\texttt{bridge}(a_p, b_p, b, w_{\texttt{parent}})$ sampled with seed ρ
`sample` = `LRUCache(sample)`

SDE forwards in time, then the adjoint backwards in time, and then discarding the Brownian motion. This motivates several of the choices here.

Small intervals First, the access patterns of SDE solvers are quite specific. Queries will be over relatively small intervals: the step that the solver is making. This means that the list of nodes populated by `traverse` is typically small: usually only consisting of a single element; occasionally two.

In contrast if the Brownian Interval has built up a reasonable tree of previous queries, and was then queried over $[0, s]$ for $s \gg 0$, then a long (inefficient) list would be returned. It is the fact that SDE solvers do not make such queries that means this is acceptable.

Search hints: starting from $\widehat{J}$ Moreover, the queries are either just ahead (fixed-step solvers; accepted steps of adaptive-step solvers) or just before (rejected steps of adaptive-step solvers) previous queries. Thus in Algorithm 7, we keep track of the most recent node $\widehat{J}$, so that we begin `traverse` near to the correct location. This ensures the modal time complexity of the search procedure is only $\mathcal{O}(1)$, and not $\mathcal{O}(\log(1/h))$ in the average step size h, which for example would be the case if searching commenced from the root on every query.

LRU cache The fact that queries are often close to one another is also what makes the strategy of using an LRU (least recently used) cache work. Most queries will correspond to a node that have a recently-computed parent in the cache.

Backward pass The queries are broadly made left-to-right (on the forward pass), and then right-to-left (on the backward pass). (Other than the occasional rejected adaptive step.)

Left to its own devices, the forward pass will thus build up a highly imbalanced binary tree. At any one time, the LRU cache will contain only nodes whose intervals are a subset of some contiguous subinterval $[s, t]$ of the query space $[0, T]$. Letting n be the number of queries on the forward pass, then this means that the backward pass will consume $\mathcal{O}(n^2)$ time – each time the backward pass moves past s, then queries will miss the LRU cache, and a full recomputation to the root will be triggered, costing $\mathcal{O}(n)$. This will then hold only nodes whose intervals are subsets of some contiguous subinterval $[u, s]$: once we move past u then this $\mathcal{O}(n)$ procedure is repeated, $\mathcal{O}(n)$ times. This is clearly undesirable.

This is precisely analogous to the classical problem of optimal recomputation for performing backpropagation, whereby a dependency graph is constructed, certain values are checkpointed, and a minimal amount of recomputation is desired; see [Gri92].

In principle the same solution may be applied: apply a snapshotting procedure in which specific extra nodes are held in the cache. This is a perfectly acceptable solution, but implementing it requires some additional engineering effort, carefully determining which nodes to augment the cache with.

Fortunately, we have an advantage that [Gri92] does not: we have some control over the dependency structure between the nodes, as we are free to prespecify any dependency structure we like. That is, we do not have to start the binary tree as just a stump. We may exploit this to produce an easier solution.

Let the size of the LRU cache be L and let ν be some estimate of the average step size of the SDE solver (which may be fixed and known if using a fixed step size solver, or estimated from the first few steps if using an adaptive step size solver). Then *before a user makes any further queries*, we simply make some queries of our own. These queries correspond to the intervals $[0, T/2], [T/2, T], [0, T/4], [T/4, T/2], \ldots$, so as to create a dyadic tree, such that the smallest intervals (the final ones in this sequence) are of size not more than νL. (In practice we use $\frac{4}{5}\nu L$ as an additional safety factor.)

Letting $[s, t]$ be some interval at the bottom of this dyadic tree, where $t \approx s + \frac{4}{5}\nu L$, then we are capable of holding every node within this interval in the LRU cache. Once we move past s on the backward pass, then we may in turn hold the entire previous subinterval $[u, s]$ in the LRU cache, and in particular the values of the nodes whose intervals lie within $[u, s]$ may be computed in only logarithmic time, due to the dyadic tree structure.

This is now analogous to the Virtual Brownian Tree of [GL97; Li+20a]. (Up to the use of intervals rather than points.) If desired, this approach may be loosely interpreted as placing a Brownian Interval on every leaf of a shallow Virtual Brownian Tree.

Recursion errors We find that for some problems, the recursive computations of `traverse` (and in principle also `sample`, but this is less of an issue due to the LRU cache) can occasionally grow very deep. In particular this occurs when crossing the midpoint of the pre-specified tree: for this particular query, the traversal must ascend the tree to the root, and then descend all the way down again. As such `traverse` should be implemented with trampolining and/or tail recursion to avoid maximum depth recursion errors.

CPU vs GPU memory We describe this algorithm as requiring only constant memory. To be more precise, the algorithm requires only constant GPU memory, corresponding to the fixed size of the LRU cache. As the Brownian Interval receives queries then its internal tree tracking dependencies will grow, and CPU memory will increase. For deep learning models, GPU memory is usually the limiting (and so more relevant) factor.

Appendix D

Experimental Details

D.1 Continuous normalising flows on images

This appendix provides the details for the example of Section 2.2.3.3.

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The experiment was implemented using the JAX, Equinox, Diffrax, and Optax software libraries [Bra+18; Kid21b; Kid21a; Hes+20]. (Providing autodifferentiation, neural networks, differential equation solvers, and optimisers respectively.)

The differential equation was solved from $t = 0$ to $t = 0.5$, with fixed timestep of size 0.05, using the Tsitouras 5(4) solver [Tsi11]. Backpropagation was performed via discretise-then-optimise (Section 5.1).

Every operation was performed at 32-bit floating point precision.

The optimiser used was 'AdamW' [KB15; Hes+20], with a batch size of 1000, a learning rate of 10^{-3}, and a weight decay of 10^{-5}. It was trained for 10 000 steps. Each step took a couple of seconds on an A100 GPU.

The vector field was parameterised as an MLP acting on the state y, except that each affine transformation was replaced by the variant layer of Section 2.3.2.4. (Which induces a time dependence.) The activation function was taken to be tanh.

For the 'target' dataset, the width of each hidden layer was 128, and 3 hidden layers were used. 2 CNFs were stacked on top of each other to produce the overall transformation. (Equivalently, it was a single CNF with piecewise vector field, split into two pieces, as in Section 2.3.2.1.)

For the 'cat' dataset, the width of each hidden layer was 64, and 3 hidden layers were used. 2 CNFs were stacked on top of each other to produce the overall transformation.

For the 'butterfly' dataset, the width of each hidden layer was 64, and 3 hidden

layers were used. 3 CNFs were stacked on top of each other to produce the overall transformation.

Each dataset is normalised to have zero mean and unit variance.

Due to the low dimensionality, exact trace-Jacobian calculations were used (not the approximate scheme of Section 2.2.3.4).

D.2 Latent ODEs on decaying oscillators

This appendix provides the details for the example of Section 2.2.4.1.

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The experiment was implemented using the JAX, Equinox, Diffrax, and Optax software libraries [Bra+18; Kid21b; Kid21a; Hes+20]. (Providing autodifferentiation, neural networks, differential equation solvers, and optimisers respectively.)

A dataset of 10 000 sample paths were produced, as solutions to the linear differential equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} -0.1 & 1.3 \\ -1 & -0.1 \end{bmatrix}\begin{bmatrix} y(t) \\ z(t) \end{bmatrix}.$$

Correspondingly the data dimensionality is $d = 2$.

The initial $y(0)$ was sampled from a two-dimensional standard normal distribution, independently for each sample path. The time interval solved over was taken to be $[0, T]$, where $T \sim \text{Uniform}[2, 3]$ independently for each sample path. Each sample path was observed at 20 time points independently sampled from $\text{Uniform}[0, T]$.

The differential equation was solved over each $[0, T]$ at train time, and over the larger interval $[0, 12]$ at test time. The solver used was Dormand–Prince 5(4) [DP80], with a fixed timestep of 0.4. Backpropagation was performed via discretise-then-optimise (Section 5.1).

Every operation was performed at 32-bit floating point precision.

The optimiser used was Adam [KB15], with a batch size of 256, and a learning rate of 10^{-2}. It was trained for 250 steps. Each step took about half a second on an A100 GPU.

The dimensionality of the evolving state y is taken to be $d_l = 16$. The vector field f_θ was taken to be autonomous and of the form

$$y \mapsto \alpha \tanh(\text{MLP}(y)),$$

where $\alpha \in \mathbb{R}$ is a learnt parameter initialised at one, and MLP is an MLP of width 16, with 3 hidden layers using softplus activation functions.

The initial noise-to-$y(0)$ network g_θ is taken to be an MLP of width 16 and 3 hidden layers, using ReLU activation functions. The latent space is taken of dimension $d_m = 16$.

The probability $p_{\theta,y}$ is parameterised as $\mathcal{N}(\ell_\theta(y), I_{d_l \times d_l})$, where $\ell_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}^d$ is learnt and affine.

The encoder ν_θ is parameterised as a single-layer GRU with hidden size 16. The final hidden state of size 16 is mapped into $\mathbb{R}^{d_m} \times \mathbb{R}^{d_m}$ by a learnt affine transformation, to produce the mean $\mu_{\theta,\mathbf{x}} \in \mathbb{R}^{d_m}$ and the log-standard deviation $\log \sigma_{\theta,\mathbf{x}} \in \mathbb{R}^{d_m}$.

D.3 Neural CDEs on spirals

This appendix provides the details for the example of Section 3.1.4.1.

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The experiment was implemented using the JAX, Equinox, Diffrax, and Optax software libraries [Bra+18; Kid21b; Kid21a; Hes+20]. (Providing autodifferentiation, neural networks, differential equation solvers, and optimisers respectively.)

The dataset is of size 256. Each time series consists of 100 regularly sampled points over the interval $[0, 4\pi] \ni t$ of

$$\begin{bmatrix} y(t) \\ z(t) \end{bmatrix} = \exp\left(t \begin{bmatrix} -0.3 & 2 \\ -2 & -0.3 \end{bmatrix}\right) \begin{bmatrix} y_0 \\ z_0 \end{bmatrix},$$

where $(y_0, z_0) = (\cos\theta, \sin\theta)$ with $\theta \sim \text{Uniform}[0, 2\pi]$. Half of the time series are then flipped in the y axis so that the dataset consists of 128 clockwise and 128 counter-clockwise spirals.

The neural CDE was solved by reducing it to an ODE as in Section 3.1.3, and using the Tsitouras 5(4) solver [Tsi11]. The step size is selected adaptively, and the initial step size is selected automatically, as in [HNW08, Section II.4]. Backpropagation was performed via discretise-then-optimise (Section 5.2).

Every operation was performed at 32-bit floating point precision.

The optimiser used was Adam [KB15], with a batch size of 32, and a learning rate of 10^{-2}. It was trained for just 20 steps. Each step took about 1.5 seconds on an A100 GPU.

The initial network ζ_θ is parameterised as an MLP with a single hidden layer of width 128 and ReLU activation functions. The vector field f_θ is parameterised as an MLP with a single hidden layer of width 128 and softplus activation functions. The output of the MLP is passed through a tanh as discussed in Section 3.4.1. The evolving hidden state y is taken to have $d_l = 8$ dimensions. The output of the model is given by applying a learnt affine transform $\ell_\theta \colon \mathbb{R}^{d_l} \to \mathbb{R}$, followed by a sigmoid to map the result into $(0, 1)$.

The interpolation scheme used is Hermite cubic splines with backward differences as discussed in Section 3.5.

The loss function used is binary cross-entropy.

The final model achieves 100% (test) accuracy.

D.4 Neural SDEs on time series

This appendix provides details for the examples of Section 4.5.

D.4.1 Brownian motion

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The experiment was implemented using the JAX, Equinox, Diffrax, and Optax software libraries [Bra+18; Kid21b; Kid21a; Hes+20]. (Providing autodifferentiation, neural networks, differential equation solvers, and optimisers respectively.)

The dataset is of size 8192. Each sample is of $v + w(t)$, where $v \sim \text{Uniform}[-1, 1]$ and $w \colon [0, 10] \to \mathbb{R}$ is a Brownian motion. Each time series consists of 11 regularly sampled points over the interval $[0, 10]$.

The neural SDE and neural CDE were both solved using the reversible Heun method (Section 5.3.2.2), with unit step size. Backpropagation was performed via discretise-then-optimise[1] (Section 5.2).

Every operation was performed at 32-bit floating point precision.

The optimiser used – for both generator and discriminator – was RMSprop (which is similar to Adadelta, and used for simplicity as Optax does not provide a built-in Adadelta optimiser). The batch size was 1024. It was trained for 10 000 steps. The generator and discriminator are trained via simultaneous gradient descent (rather than by alternating training steps for the generator and discriminator).

The initial network ζ_ϕ in the generator used a learning rate of 2×10^{-4}. The other components of the generator $(\mu_\theta, \sigma_\theta, \alpha_\theta, \beta_\theta)$ used a learning rate of 2×10^{-5}. The initial network ξ_ϕ of the discriminator used a learning rate of 10^{-3}. The other components of the generator (f_ϕ, g_ϕ, m_ϕ) used a learning rate of 10^{-4}.

All parameters (for both generator and discriminator) were initialised close to zero. In practice this was done by initialising them as per Equinox's default, and then multiplying every parameter by 0.01.

[1]Which is in any case essentially equivalent to optimise-then-discretise when using a reversible solver.

Remark D.1. *The discriminator uses a larger learning rate than the generator (by a factor of 5) as per [Heu+17]. In brief: the discriminator must be able to 'keep up' with the generator as it trains, so that it can always provide informative gradients. As such we may either train the discriminator for multiple steps for every step of the generator, or (since we use simultaneous gradient descent here) give the discriminator a larger learning rate.*

The initial networks were taken to be use a larger learning rate as this was found to improve the speed at which they converged to the true distribution, without creating any instability.

The initial networks ζ_θ and ξ_ϕ were taken to be MLPs with a single hidden layer of width 16 and ReLU activation function. The vector fields μ_θ and σ_θ were parameterised as

$$(t, y) \mapsto \gamma \tanh(\mathrm{MLP}_\theta(t, y)),$$

where γ is a learnt parameter randomly initialised from Uniform$[0.9, 1.1]$, and MLP_θ had a single hidden layer of width 16 with LipSwish activation function.[2] The vector fields f_ϕ and g_ϕ were parameterised as

$$(t, y) \mapsto \tanh(\mathrm{MLP}_\theta(t, y)),$$

where MLP_θ had a single hidden layer of width 16 with LipSwish activation function.

The dimensionality of the initial noise was taken to be $d_v = 5$. The dimensionality of the Brownian motion was taken to be $d_w = 3$. The dimensionality of the evolving hidden state y was taken to be $d_y = 16$.

Lipschitzness of the discriminator was maintained using careful clipping (Section 4.4.3.2). Both the real and the generated data were treated as time series and linearly interpolated before passing to the neural CDE (the discriminator).

The output of the discriminator was defined using the alternate formula $D = m_\phi \cdot h(0) + m_\phi \cdot h(T)$, to encourage better learning the initial distribution. (Contrast equation (4.5) and see also Section 4.4.2.2.)

D.4.2 Time-dependent Ornstein–Uhlenbeck process

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The dataset is taken to be samples from

$$y(0) \sim \mathrm{Uniform}[-1, 1], \qquad \mathrm{d}y(t) = (at - by)\,\mathrm{d}t + ct\,\mathrm{d}w(t), \qquad \text{for } t \in [0, 63],$$

with $a = 0.02$, $b = 0.1$, $c = 0.013$. Samples were obtained from this equation by solving using the Euler–Maruyama method and a step size of 0.1. Each time series consisted of 64 regularly spaced points over the interval $[0, 63]$.

[2]Which is not necessary for the generator – just the discriminator, see Section 4.4.3 – so this activation function was used just for simplicity.

In all other respects this example is identical to the Brownian motion example discussed above.

D.4.3 Damped harmonic oscillator

This experiment was implemented using the PyTorch, torchdiffeq, and torchsde libraries [Pas+19; Che18; Li20b]. (Providing autodifferentiation, ordinary differential equation solvers, and stochastic differential equation solvers respectively.)

The dataset is of size 8192. Each sample is of

$$y_1(0), y_2(0) \sim \text{Uniform}[-1, 1], \qquad \mathrm{d}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} -0.01 & 0.13 \\ -0.1 & -0.01 \end{bmatrix} \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} \mathrm{d}t,$$

for $t \in [0, 100]$. Samples were obtained from this equation by solving using the Dormand–Prince 5(4) method with an adaptively chosen step size. Each time series consists of 101 regularly sampled points over the interval $[0, 100]$. Prior to training the dataset was normalised to have zero mean and unit standard deviation.

The auxiliary neural SDE was trained, and the neural SDE sampled, with the reversible Heun method (Section 5.3.2.2) with unit step size. Backpropagation was performed via discretise-then-optimise (Section 5.2).

Every operation was performed at 32-bit floating point precision.

The optimiser used was Adam [KB15], with a batch size of 1024, and trained over 20 000 steps. The initial networks ζ_θ and ξ_ϕ used a learning rate of 10^{-3}. Every other component used a learning rate of 2×10^{-4}.

All parameters were initialised relatively small. In practice this was done by initialising them as per PyTorch's default, and then multiplying every parameter by 0.5. The parameters of the initial network ζ_θ were instead multiplied by 0.25.

The auxiliary network ν_ϕ was parameterised as $\nu_\phi(t, y, x) = \nu_{\phi,1}(t, y, \nu_{\phi,2}(x|_{[t,T]}))$, where $\nu_{\phi,1}$ was an MLP and $\nu_{\phi,2}$ is the evaluation function $\nu_{\phi,2}(x|_{[t,T]}) = x(t)$. (Rounded to the nearest discrete timestamp.)

Every neural network was parameterised as an MLP with a single hidden layer of width 32 and LipSwish activation function. The vector fields μ_θ and $\nu_{\phi,1}$ additionally had a final tanh nonlinearity. The vector field σ_θ produced a diagonal matrix as its output, and additionally had a $z \mapsto \text{sigmoid}(z) + 10^{-4}$ nonlinearity. (A simple way to avoid the numerical issues with the diffusion, as discussed in Section 4.4.2.1.)

The standard deviation outputted by the encoder ξ_ϕ (nominally in $(0, \infty)$) was performed by outputting a log-standard deviation (valued in $\mathbb{R}$), and then clipping the log-standard deviation to the range $(-10, 10)$, to promote better numerical stability.

The dimensionality of the initial noise was taken to be $d_v = 2$. The dimensionality of the Brownian motion was taken to be $d_w = 2$. The dimensionality of the evolving hidden state y was taken to be $d_y = 64$.

D.4.4 Lorenz attractor

This example considered samples from the Lorenz attractor

$$\begin{aligned} y &\sim \mathcal{N}\left(0, I_{3\times 3}\right), \\ \mathrm{d}y_1(t) &= a_1(y_2(t) - y_1(t))\,\mathrm{d}t + b_1 y_1(t)\mathrm{d}w(t), \\ \mathrm{d}y_2(t) &= (a_2 y_1(t) - y_1(t)y_3(t))\,\mathrm{d}t + b_2 y_2(t)\mathrm{d}w(t), \\ \mathrm{d}y_3(t) &= (y_1(t)y_2(t) - a_3 y_3(t))\,\mathrm{d}t + b_3 y_3(t)\mathrm{d}w(t), \end{aligned}$$

for $t \in [0, 2]$. We take specifically $a_1 = 10$, $a_2 = 28$, $a_3 = \frac{8}{3}$, $b_1 = 0.1$, $b_2 = 0.28$, $b_3 = 0.3$. Samples were obtained from this equation using Milstein's method and a step size of 0.1. Each time series consisted of 100 regularly spaced points over the interval $[0, 2]$.

Except as now otherwise stated, this was otherwise identical to the damped harmonic oscillator just discussed.

The latent SDE component continued to use the Adam optimiser. When training as an SDE-GAN, the Adadelta optimiser was used.[3]

The initial network ξ_ϕ of the discriminator (not to be confused with the same notation ξ_ϕ also being used in the latent SDE) was parameterised as an MLP with a single hidden layer of width 32 and LipSwish activation function. The vector fields f_ϕ and g_ϕ were parameterised as

$$(t, y) \mapsto \tanh(\mathrm{MLP}_\theta(t, y)),$$

where MLP_θ had a single hidden layer of width 32 with LipSwish activation function.

The dimensionality of the discriminator hidden state was taken to be $d_h = 64$.

Lipschitzness of the discriminator was maintained using careful clipping (Section 4.4.3.2). Both the real and the generated data were treated as time series and linearly interpolated before passing to the neural CDE (the discriminator).

The neural CDE of the discriminator was solved using the reversible Heun method (Section 5.3.2.2).

The output of the discriminator was defined using the alternate formula $D = \kappa_\phi(x(0)) + m_\phi \cdot h(T)$, to encourage better learning the initial distribution, where κ_ϕ is an MLP mapping $\mathbb{R}^{d_x} \to \mathbb{R}$, parameterised with a single hidden layer of width 32 and LipSwish activation function. (Contrast equation (4.5) and see also Section 4.4.2.2.)

[3]So that the components of the auxiliary neural SDE used in the latent SDE were associated only with an Adam optimiser, the discriminator was associated only with an Adadelta optimiser, and the generator was associated with both an Adam and an Adadelta optimiser independently of each other. This is arguably a little questionable – the statistics tracked in the Adam and the Adadelta optimisers no longer do exactly what is expected – but we found that training a latent SDE with Adadelta or training an SDE-GAN with Adam seemed to fail.

D.5 Symbolic regression on a nonlinear oscillator

This appendix provides details for the example of Section 6.1.3.

At time of writing, this experiment may be found implemented as an example in the Diffrax software package [Kid21a].

The experiment was implemented using the JAX, Equinox, Diffrax, Optax, and PySR software libraries [Bra+18; Kid21b; Kid21a; Hes+20; Cra20]. (Providing autodifferentiation, neural networks, differential equation solvers, gradient-based optimisers and regularised evolution algorithms respectively.)

The dataset is of size 256. Each time series consists of 100 regularly sampled points over the interval $[0, 10] \ni t$ of

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x \\ y \end{bmatrix}(t) = \begin{bmatrix} \frac{y(t)}{1+y(t)} \\ \frac{-x(t)}{1+x(t)} \end{bmatrix} \quad \text{for } t \in [0, T],$$

where $x(0), y(0) \sim \text{Uniform}[-0.6, 1]$.

The neural ODE was solved using the Tsitouras 5(4) solver [Tsi11] with a fixed step size of 0.1.

Every operation was performed at 32-bit floating point precision.

The gradient-based optimiser used was AdaBelief [Zhu+20a], with a batch size of 32, and a learning rate of 3×10^{-3}. It was trained for 5000 steps. Each step took about 0.9 seconds on an A100 GPU.

The first 500 steps of gradient-based optimisation were performed on only the first 10 sample points of each time series (so that approximately the interval $[0, 1]$ was considered instead). This helps to avoid local minima during training.

The neural vector field was parameterised as an MLP with two hidden layers, each of width 64.

The loss function used was L^2. The final loss was of order $\mathcal{O}(10^{-5})$.

Symbolic regression was performed by flattening all observations together into a single dataset of size 25600 and randomly selecting some 2000 samples. We then performed regularised evolution on this dataset. Regularised evolution comes with numerous hyperparameters; unless otherwise specified the PySR version 0.6.13 defaults were used. We used 100 populations each of size 20. 10 rounds of optimisation were performed; 100 mutations were performed in each round and between each round equations migrated between populations. Constants in each expression were optimised using 50 steps of BFGS.

Symbolic regression produces a Pareto front of equations, trading off loss against complexity. Each equation on the Pareto front was fine-tuned using full-batch gradient descent (possibly superfluously, given the earlier use of BFGS) and a learning rate of

3×10^{-4}. The best equation was then selected as being the one minimising

$$\log_2(\text{loss}) + \text{complexity},$$

where 'loss' is the L^2 loss when regressing $f_\theta(x(t), y(t))$ against $(x(t), y(t))$, and 'complexity' is the number of symbols in the symbolic expression: for example c is of size one, $c + x$ is of size three, and $a \times x + b \times y$ is of size seven. The use of base two in the logarithm serves as a quantitative measure of trading off loss against complexity: the use of an extra symbol must halve the loss if is to produce a 'better' expression.

The constants of the symbolic expression are then optimised by gradient descent, by plugging it back into the original (neural) optimisation problem. The gradient-based optimiser used was full-batch Adam, with a learning rate of 3×10^{-3}. It was trained for 500 steps.

Finally, the constants are rounded to the nearest multiple of 0.01.

D.6 Neural RDEs on BIDMC

This appendix provides details for the example of Appendix B.3.

This experiment was implemented using the PyTorch, torchdiffeq, torchcde, and Signatory libraries [Pas+19; Che18; Kid20; KL21]. (Providing autodifferentiation, differential equation solvers, and logsignature computations respectively.)

(This experiment predates the creation of the Diffrax software library, and in any case and at time of writing there does not exist a JAX library for computing logsignatures.)

The dataset is split into a 70%/15%/15% train/validation/test split. Each time series consists of 4 000 points sampled at 125 Hertz. Each channel is normalised to have zero mean and unit variance.

The RDE was solved by reducing it to an ODE and using the RK4 with 3/8 rule solver. The step size was fixed, and was equal to the 'step' hyperparameter (over either 8, 128 or 512 data points at once).

Every operation was performed at 32-bit floating point precision.

A batch size of 512 and a learning rate of 6.25×10^{-5} was used. If the validation loss failed to decrease over 15 epochs then the learning rate was reduce by a factor of 10. If the validation loss failed to improve over 60 epochs then training was terminated, and the model rolled back to the point at which it achieved the best validation loss.

Every neural network is parameterised as an MLP with three hidden layers of width 192 and ReLU activation functions. The evolving hidden state y is taken to have $d_l = 64$ dimensions. These were selected as a result of hyperparameter optimisation for the baseline (competing) neural CDE model.

The interpolation scheme used in the data space (necessarily) linear interpolation. This is necessary as the only choice for which the logsignature can be computed

efficiently. In addition linear interpolation was used to interpolate the sequence of logsignatures.

The loss function used is L^2 loss.

See also [Mor+21b, Appendix C] for details of this experiment.

Bibliography

[Agr+19] A. Agrawal et al. "Differentiable Convex Optimization Layers". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[ARF20] V. M. M. Alvarez, R. Roșca, and C. G. Fălcuțescu. "DyNODE: Neural Ordinary Differential Equations for Dynamics Modeling in Continuous Control". In: *arXiv:2009.04278* (2020).

[AF20] E. P. Alves and F. Fiuza. "Data-driven discovery of reduced plasma physics models from fully-kinetic simulations". In: *arXiv:2011.01927* (2020).

[Amo19] B. Amos. "Differentiable Optimization-Based Modeling for Machine Learning". PhD thesis. Carnegie Mellon University, May 2019.

[AK17] B. Amos and J. Z. Kolter. "OptNet: Differentiable Optimization as a Layer in Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 136–145.

[AXK17] B. Amos, L. Xu, and J. Z. Kolter. "Input Convex Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 146–155.

[Ara03] M. Arató. "A famous nonlinear stochastic equation (Lotka-Volterra model with diffusion)". In: *Mathematical and Computer Modelling* 38.7–9 (2003), pp. 709–726.

[ACB17] M. Arjovsky, S. Chintala, and L. Bottou. "Wasserstein Generative Adversarial Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 214–223.

[BKH16] J. L. Ba, J. R. Kiros, and G. E. Hinton. "Layer Normalization". In: *arXiv:1607.06450* (2016).

[BKK19] S. Bai, J. Z. Kolter, and V. Koltun. "Deep Equilibrium Models". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 690–701.

[BKK20] S. Bai, V. Koltun, and J. Z. Kolter. "Multiscale Deep Equilibrium Models". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 5238–5250.

[BKK21] S. Bai, V. Koltun, and Z. Kolter. "Stabilizing Equilibrium Models by Jacobian Regularization". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 554–565.

[Beh+19] J. Behrmann et al. "Invertible Residual Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 573–582.

[BBS21] A. Bellot, K. Branson, and M. van der Schaar. "Consistency of mechanistic causal discovery in continuous-time using Neural ODEs". In: *arXiv:2105.02522* (2021).

[Bez+17] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[Biń+18] M. Bińkowski et al. "Demystifying MMD GANs". In: *International Conference on Learning Representations*. 2018.

[BS73] F. Black and M. Scholes. "The Pricing of Options and Corporate Liabilities". In: *Journal of Political Economy* 81.3 (1973), pp. 637–654.

[Bla+09] S. Blanes et al. "The Magnus expansion and some of its applications". In: *Physics Reports* 470.5-6 (2009), pp. 151–238.

[Blo+21] M. Blondel et al. "Efficient and Modular Implicit Differentiation". In: *arXiv:2105.15183* (2021).

[Bol86] T. Bollerslev. "Generalized Autoregressive Conditional Heteroskedasticity". In: *Journal of Econometrics* 31.3 (1986), pp. 307–327.

[Bor+21] V. D. Bortoli et al. "Diffusion Schrödinger Bridge with Applications to Score-Based Generative Modeling". In: *arXiv:2106.01357* (2021).

[Bou+17] O. Bousquet et al. "From optimal transport to generative modeling: the VEGAN cookbook". In: *arXiv:1705.07642* (2017).

[Bra+18] J. Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: `http://github.com/google/jax`.

[BM01] D. Brigo and F. Mercurio. *Interest Rate Models: Theory and Practice*. Springer, Berlin, 2001.

[Bri+20] F.-X. Briol et al. "Statistical Inference for Generative Models with Maximum Mean Discrepancy". In: *arXiv:1906.05944* (2020).

[Bro+20] T. Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[BNK20] S. L. Brunton, B. R. Noack, and P. Koumoutsakos. "Machine Learning for Fluid Mechanics". In: *Annual Review of Fluid Mechanics* 52.1 (2020), pp. 477–508.

[BPK16] S. L. Brunton, J. L. Proctor, and J. N. Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (2016), pp. 3932–3937.

[But16] J. C. Butcher. *Numerical Methods for Ordinary Differential Equations*. Third Edition. 2016.

[CLX21] T. Cass, T. Lyons, and X. Xu. "General Signature Kernels". In: *2107.00447* (2021).

[Cha+21] B. Chamberlain et al. "GRAND: Graph Neural Diffusion". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 1407–1418.

[Cha+19a] K. Champion et al. "A unified sparse optimization framework to learn parsimonious physics-informed models from data". In: *arXiv:1906.10612* (2019).

[Cha+19b] K. Champion et al. "Data-driven discovery of coordinates and governing equations". In: *Proceedings of the National Academy of Sciences* 116.45 (2019), pp. 22445–22451.

[Cha+18] B. Chang et al. "Reversible Architectures for Arbitrarily Deep Residual Neural Networks". In: *AAAI* (2018).

[Che+18a] Z. Che et al. "Recurrent Neural Networks for Multivariate Time Series with Missing Values". In: *Scientific Reports* 8 (2018).

[Che+19] R. T. Q. Chen et al. "Residual Flows for Invertible Generative Modeling". In: *Advances in Neural Information Processing Systems 32*. 2019.

[Che18] R. T. Q. Chen. *torchdiffeq*. 2018. URL: `https://github.com/rtqichen/torchdiffeq`.

[Che20] R. T. Q. Chen. Private communication. 2020.

[CAN21a] R. T. Q. Chen, B. Amos, and M. Nickel. "Learning Neural Event Functions for Ordinary Differential Equations". In: *International Conference on Learning Representations*. 2021.

[CAN21b] R. T. Q. Chen, B. Amos, and M. Nickel. "Neural Spatio-Temporal Point Processes". In: *International Conference on Learning Representations*. 2021.

[Che+18b] R. T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 6571–6583.

[CK16] I. Chevyrev and A. Kormilitzin. “A primer on the signature method in machine learning”. In: *arXiv:1603.03788* (2016).

[Cho+14] K. Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Empirical Methods in Natural Language Processing* (2014).

[Cho+20] K. M. Choromanski et al. “Ode to an ODE”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 3338–3350.

[CS91] S. R. Chu and R. Shoureshi. “A Neural Network Approach for Identification of Continuous-Time Nonlinear Dynamic Systems”. In: *1991 American Control Conference*. 1991, pp. 1–5.

[CP13] K. Claessen and M. Pałka. “Splittable pseudorandom number generators using cryptographic hashing”. In: *ACM SIGPLAN Notices* 48 (2013), pp. 47–58.

[CKW12] W. T. Coffey, Y. P. Kalmykov, and J. T. Waldron. *The Langevin Equation: With Applications to Stochastic Problems in Physics, Chemistry and Electrical Engineering*. World Scientifc, 2012.

[CRW21] S. N. Cohen, C. Reisinger, and S. Wang. “Arbitrage-free neural-SDE market models”. In: *arXiv:2105.11053* (2021).

[Coo+17] T. Cooijmans et al. “Recurrent Batch Normalization”. In: *International Conference on Learning Representations* (2017).

[CIR85] J. C. Cox, J. E. Ingersoll, and S. A. Ross. “A theory of term structure of interest rates”. In: *Econometrica* 53.2 (1985), pp. 385–407.

[Cra20] M. Cranmer. *PySR: Fast & Parallelized Symbolic Regression in Python/Julia*. 2020. URL: `http://doi.org/10.5281/zenodo.4041459`.

[Cra21a] M. Cranmer. Private communication. 2021.

[Cra21b] M. Cranmer. Private communication. 2021.

[Cra+20a] M. Cranmer et al. “Discovering Symbolic Models from Deep Learning with Inductive Biases”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 17429–17442.

[Cra+20b] M. Cranmer et al. “Lagrangian Neural Networks”. In: *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2020.

[CKT20] C. Cuchiero, W. Khosrawi, and J. Teichmann. “A Generative Adversarial Network Approach to Calibration of Local Stochastic Volatility Models”. In: *Risks* 8.4 (2020).

[Dau+20] T. Daulbaev et al. “Interpolation Technique to Speed Up Gradients Propagation in Neural ODEs”. In: *Advances in Neural Information Processing Systems 33*. Curran Associates, Inc., 2020.

[De +19] E. De Brouwer et al. “GRU-ODE-Bayes: Continuous Modeling of Sporadically-Observed Time Series”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 7379–7390.

[Den+20] R. Deng et al. “Modeling Continuous Stochastic Processes with Dynamic Normalizing Flows”. In: *Advances in Neural Information Processing Systems 33*. 2020, pp. 7805–7815.

[Den+21] R. Deng et al. “Continuous Latent Process Flows”. In: *arXiv:2106.15580* (2021).

[Den+19] Z. Deng et al. “Continuous Graph Flow”. In: *arXiv:1908.02436* (2019).

[DN21] P. Dhariwal and A. Nichol. “Diffusion Models Beat GANs on Image Synthesis”. In: *arXiv:2105.05233* (2021).

[DSB17] L. Dinh, J. Sohl-Dickstein, and S. Bengio. “Density estimation using Real NVP”. In: *International Conference on Learning Representations* (2017).

[DP80] J. R. Dormand and P. J. Prince. “A family of embedded Runge–Kutta formulae”. In: *J. Comp. Appl. Math* 6 (1980), pp. 19–26.

[DFD20] J. Du, J. Futoma, and F. Doshi-Velez. “Model-based Reinforcement Learning for Semi-Markov Decision Processes with Neural ODEs”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 19805–19816.

[DDT19] E. Dupont, A. Doucet, and Y. W. Teh. “Augmented Neural ODEs”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 3140–3150.

[DIX19] K. Duraisamy, G. Iaccarino, and H. Xiao. “Turbulence Modeling in the Age of Data”. In: *Annual Review of Fluid Mechanics* 51.1 (2019), pp. 357–377.

[E17] W. E. “A Proposal on Machine Learning via Dynamical Systems”. In: *Commun. Math. Stat.* 5.1 (2017), pp. 1–11.

[EUD17] S. Elfwing, E. Uchibe, and K. Doya. “Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning”. In: *arXiv:1702.03118* (2017).

[Eng82] R. F. Engle. “Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation”. In: *Econometrica* 50.4 (1982), pp. 987–1007.

[FF20] L. Falorsi and P. Forré. “Neural Ordinary Differential Equations on Manifolds”. In: *2006.06663* (2020).

[Fan+19] J. Fang et al. “Neural Network Solution of Single-Delay Differential Equations”. In: *Mediterranean Journal of Mathematics* 17.1 (2019), p. 30.

[Fer+21] A. Fermanian et al. “Framing RNN as a kernel method: A neural ODE approach”. In: *arXiv:2106.01202* (2021).

[Fin+20a] C. Finlay et al. “How to Train Your Neural ODE: the World of Jacobian and Kinetic Regularization”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 3154–3164.

[Fin+20b] C. Finlay et al. “Learning normalizing flows from Entropy-Kantorovich potentials”. In: *arXiv:2006.06033* (2020).

[Flo+21] P. Florence et al. “Implicit Behavioral Cloning”. In: *arXiv:2109.00137* (2021).

[Fos20] J. Foster. “Numerical approximations for stochastic differential equations”. PhD thesis. University of Oxford, 2020.

[FV10] P. K. Friz and N. B. Victoir. “Multidimensional stochastic processes as rough paths: theory and applications”. In: *Cambridge University Press* (2010).

[Fro+21] R. Frostig et al. “Decomposing reverse-mode automatic differentiation”. In: *LAFI workshop, POPL* (2021).

[Fun+21] S. W. Fung et al. “Fixed Point Networks: Implicit Depth Models with Jacobian-Free Backprop”. In: *arXiv:2103.12803* (2021).

[GL97] J. Gaines and T. Lyons. “Variable step size control in the numerical solution of stochastic differential equations”. In: *SIAM Journal on Applied Mathematics* 57.5 (1997), pp. 1455–1484.

[Gér17] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow*. Sebastopol, CA: O’Reilly Media, 2017.

[GKB19] A. Gholami, K. Keutzer, and G. Biros. “ANODE: Unconditionally Accurate Memory-Efficient Gradients for Neural ODEs”. In: *arXiv:1902.10298* (2019).

[Gho+20] A. Ghosh et al. “STEER : Simple Temporal Regularization For Neural ODE”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 14831–14843.

[Gid+19] G. Gidel et al. “Negative Momentum for Improved Game Dynamics”. In: *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*. Vol. 89. Proceedings of Machine Learning Research. PMLR, 2019, pp. 1802–1811.

[Gie+20] P. Gierjatowicz et al. “Robust Pricing and Hedging via Neural SDEs”. In: *arXiv:2007.04154* (2020).

[GG06] M. Giles and P. Glasserman. “Smoking adjoints: fast Monte Carlo Greeks”. In: *Risk* (2006).

[Gom+17] A. N. Gomez et al. “The Reversible Residual Network: Backpropagation Without Storing Activations”. In: *arXiv:1707.04585* (2017).

[Goo+14] I. Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 2672–2680.

[Gra+19] W. Grathwohl et al. “FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models”. In: *International Conference on Learning Representations* (2019).

[Gre+13] A. Gretton et al. “A kernel two-sample test”. In: *Journal of Machine Learning Research* 13.1 (2013), pp. 723–773.

[GDY19] S. Greydanus, M. Dzamba, and J. Yosinski. “Hamiltonian Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[Gri92] A. Griewank. “Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation”. In: *Optimization Methods and Software* 1.1 (1992), pp. 35–54.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Second Edition. Society for Industrial and Applied Mathematics (SIAM), 2008.

[Gui+20] R. Guimerà et al. “A Bayesian machine scientist to aid in the solution of challenging scientific problems”. In: *Science Advances* 6.5 (2020).

[Gul+17] I. Gulrajani et al. “Improved Training of Wasserstein GANs”. In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 5767–5777.

[HR17] E. Haber and L. Ruthotto. “Stable Architectures for Deep Neural Networks”. In: *Inverse Problems* 34.1 (2017).

[Hab+19] E. Haber et al. “IMEXnet A Forward Stable Deep Neural Network”. In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 2525–2534.

[Hag00] W. W. Hager. “Runge-Kutta methods in optimal control and the transformed adjoint system”. In: *Numerische Mathematik* 87.2 (2000), pp. 247–282.

[HNW08] E. Hairer, S. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I Nonstiff problems*. Second Revised Edition. Berlin: Springer, 2008.

[HW02] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II Stiff and Differential-Algebraic Problems*. Second Revised Edition. Berlin: Springer, 2002.

[HL10] B. M. Hambly and T. J. Lyons. “Uniqueness for the signature of a path of bounded variation and the reduced path group”. In: *Annals of Mathematics* 171.1 (2010), pp. 109–167.

[HJE18] J. Han, A. Jentzen, and W. E. "Solving high-dimensional partial differential equations using deep learning". In: *Proceedings of the National Academy of Sciences* 115.34 (2018), pp. 8505–8510.

[HS17] B. Hanin and M. Sellke. "Approximating Continuous Functions by ReLU Nets of Minimal Width". In: *arXiv:1710.11278* (2017).

[HR82] E. J. Hannan and J. Rissanen. "Recursive Estimation of Mixed Autoregressive-Moving Average Order". In: *Biometrika* 69 (1982), pp. 81–94.

[He+15] K. He et al. "Deep Residual Learning for Image Recognition". In: *arXiv:1512.03385* (2015).

[HG16] D. Hendrycks and K. Gimpel. "Gaussian Error Linear Units (GELUs)". In: *arXiv:1606.08415* (2016).

[Hes+20] M. Hessel et al. *Optax: composable gradient transformation and optimisation, in JAX!* Version 0.0.1. 2020. URL: `http://github.com/deepmind/optax`.

[Heu+17] M. Heusel et al. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[Hin+21] A. C. Hindmarsh et al. *User Documentation for CVODES v5.7.0*. 2021.

[Ho+21] J. Ho et al. "Cascaded Diffusion Models for High Fidelity Image Generation". In: *arXiv:2106.15282* (2021).

[HS97] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[Hod+20] L. Hodgkinson et al. "Stochastic Normalizing Flows". In: *arXiv:2002.09547* (2020).

[Hol57] C. Holt. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *ONR Research Memorandum, Carnegie Institute of Technology* 52 (1957).

[HLC21] C.-W. Huang, J. H. Lim, and A. Courville. "A Variational Perspective on Diffusion-Based Generative Models and Score Matching". In: *arXiv:2106.02808* (2021).

[Hui07] T. Huillet. "On Wright-Fisher diffusion and its relatives". In: *Journal of Statistical Mechanics: Theory and Experiment* 11 (2007).

[Hut89] M. F. Hutchinson. "A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines". In: *Communications in Statistics-Simulation and Computation* 18.3 (1989), pp. 1059–1076.

[Hwa+21] J. Hwang et al. "Climate Modeling with Neural Diffusion Equations". In: *arXiv:2111.06011* (2021).

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv:1502.03167* (2015).

[Izm+18] P. Izmailov et al. “Averaging Weights Leads to Wider Optima and Better Generalization”. In: *Conference on Uncertainty in Artificial Intelligence* (2018).

[JGH18] A. Jacot, F. Gabriel, and C. Hongler. “Neural Tangent Kernel: Convergence and Generalization in Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018.

[JK14] S. Jansen and N. Kurt. “On the notion(s) of duality for Markov processes”. In: *Probability Surveys* 11 (2014), pp. 59–120.

[Jhi+21] S. Y. Jhin et al. “Attentive Neural Controlled Differential Equations for Time-series Classification and Forecasting”. In: *arXiv:2109.01876* (2021).

[Ji+21] W. Ji et al. “Autonomous Kinetic Modeling of Biomass Pyrolysis using Chemical Reaction Neural Networks”. In: *arXiv:2105.11397* (2021).

[JB19] J. Jia and A. R. Benson. “Neural Jump Stochastic Differential Equations”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 9847–9858.

[JSP19] I. Jordan, P. A. Sokol, and I. M. Park. “Gated recurrent units viewed through the lens of continuous time dynamical systems”. In: *arXiv:1906.01005* (2019).

[Kah+19] K. Kaheman et al. “Learning Discrepancy Models From Experimental Data”. In: *Conference on Decision and Control* (2019).

[Kal+19] D. Kalimeris et al. “SGD on Neural Networks Learns Functions of Increasing Complexity”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[Kap+21] A. A. Kaptanoglu et al. “Promoting global stability in data-driven models of quadratic nonlinear dynamics”. In: *arXiv:2105.01843* (2021).

[Kar+18] T. Karras et al. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. In: *International Conference on Learning Representations*. 2018.

[Kar+19] T. Karras et al. “Analyzing and Improving the Image Quality of StyleGAN”. In: *arXiv:1912.04958* (2019).

[Kel+21] J. Kelly et al. “Learning Differential Equations that are Easy to Solve”. In: *Advances in Neural Information Processing Systems 34*. Curran Associates, Inc., 2021.

[Kid20] P. Kidger. *torchcde*. 2020. URL: `https://github.com/patrick-kidger/torchcde`.

[Kid21a] P. Kidger. *Diffrax*. 2021. URL: `https://github.com/patrick-kidger/diffrax`.

[Kid21b] P. Kidger. *Equinox*. 2021. URL: `https://github.com/patrick-kidger/equinox`.

[Kid21c] P. Kidger. *sympytorch*. 2021. URL: https://github.com/patrick-kidger/sympytorch.

[Kid21d] P. Kidger. *torchtyping*. 2021. URL: https://github.com/patrick-kidger/torchtyping.

[KCL21] P. Kidger, R. T. Q. Chen, and T. J. Lyons. ""Hey, that's not an ODE": Faster ODE Adjoints via Seminorms". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 5443–5452.

[KL20a] P. Kidger and R. Luo. *FromFile.jl*. 2020. URL: https://github.com/roger-luo/FromFile.jl.

[KL20b] P. Kidger and T. Lyons. "Universal Approximation with Deep Narrow Networks". In: *Conference on Learning Theory* (2020).

[KL21] P. Kidger and T. Lyons. "Signatory: differentiable computations of the signature and logsignature transforms, on both CPU and GPU". In: *International Conference on Learning Representations* (2021).

[KML20] P. Kidger, J. Morrill, and T. Lyons. "Generalised Interpretable Shapelets for Irregular Time Series". In: *arXiv:2005.13948* (2020).

[Kid+19] P. Kidger et al. "Deep Signature Transforms". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[Kid+20a] P. Kidger et al. "Neural Controlled Differential Equations for Irregular Time Series". In: *Neural Information Processing Systems* (2020).

[Kid+20b] P. Kidger et al. "Neural SDEs Made Easy: SDEs are Infinite-Dimensional GANs". In: *OpenReview (unpublished)* (2020).

[Kid+21a] P. Kidger et al. "Efficient and Accurate Gradients for Neural SDEs". In: *Advances in Neural Information Processing Systems 34*. Curran Associates, Inc., 2021.

[Kid+21b] P. Kidger et al. "Neural SDEs as Infinite-Dimensional GANs". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 5453–5463.

[Kil+20] T. W. Killian et al. "An Empirical Study of Representation Learning for Reinforcement Learning in Healthcare". In: *Proceedings of the Machine Learning for Health NeurIPS Workshop*. Ed. by E. Alsentzer et al. Vol. 136. Proceedings of Machine Learning Research. PMLR, 2020, pp. 139–160.

[Kim+21a] S. Kim et al. "Stiff neural ordinary differential equations". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9 (2021).

[Kim+21b] T. D. Kim et al. "Inferring Latent Dynamics Underlying Neural Population Activity via Neural Differential Equations". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 5551–5561.

[KB15] D. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: *International Conference on Learning Representations* (2015).

[Kin+21] D. P. Kingma et al. “Variational Diffusion Models”. In: *arXiv:2107.00630* (2021).

[KP92] P. E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*. Springer, 1992.

[KDJ20] Z. Kolter, D. Duvenaud, and M. Johnson. *Deep Implicit Layers - Neural ODEs, Deep Equilibirum Models, and Beyond*. `https://implicit-layers-tutorial.org/`. 2020.

[KSZ20] L. Kong, J. Sun, and C. Zhang. “SDE-Net: Equipping Deep Neural Networks with Uncertainty Estimates”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5405–5415.

[Kos18] A. Kosiorek. *Normalizing Flows*. `https://akosiorek.github.io/ml/2018/04/03/norm_flows.html`. 2018.

[LLF97a] I. E. Lagaris, A. Likas, and D. I. Fotiadis. “Artificial Neural Networks for Solving Ordinary and Partial Differential Equations”. In: *arXiv:9705023* (1997).

[LLF97b] I. Lagaris, A. Likas, and D. Fotiadis. “Artificial neural network methods in quantum mechanics”. In: *Computer Physics Communications* 104.1 (1997), pp. 1–14.

[Lar+15] A. B. L. Larsen et al. “Autoencoding beyond pixels using a learned similarity metric”. In: *arXiv:1512.09300* (2015).

[LS16] X. Lelièvre and G. Stoltz. “Partial differential equations and stochastic methods in molecular dynamics”. In: *Acta Numerica* 25 (2016), pp. 681–880.

[Les+93] M. Leshno et al. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural Networks* 6.6 (1993), pp. 861–867.

[LLN13] D. Levin, T. Lyons, and H. Ni. “Learning from the past, predicting the statistics for the future, learning an evolving system”. In: *arXiv:1309.0260* (2013).

[Li+17] C.-L. Li et al. “MMD GAN: Towards Deeper Understanding of Moment Matching Network”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017, pp. 2203–2213.

[Li+19] L. Li et al. “Neural-Guided Symbolic Regression with Asymptotic Constraints”. In: *NeurIPS 2019 Workshop on Knowledge Representation & Reasoning Meets Machine Learning* (2019).

[Li20a] Q. Li. “Dynamical Systems and Machine Learning”. In: *Peking University Summer School* (2020).

[LLS19] Q. Li, T. Len, and Z. Shen. "Deep Learning via Dynamical Systems: An Approximation Perspective". In: *arXiv:1912.10382* (2019).

[Li20b] X. Li. *torchsde*. 2020. URL: `https://github.com/google-research/torchsde`.

[Li+20a] X. Li et al. "Scalable Gradients and Variational Inference for Stochastic Differential Equations". In: *AISTATS* (2020).

[Li+20b] Z. Li et al. "Multipole Graph Neural Operator for Parametric Partial Differential Equations". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 6755–6766.

[Li+20c] Z. Li et al. "Neural Operator: Graph Kernel Network for Partial Differential Equations". In: *arXiv:2003.03485* (2020).

[Li+21] Z. Li et al. "Fourier Neural Operator for Parametric Partial Differential Equations". In: *International Conference on Learning Representations*. 2021.

[LKT16] J. Ling, A. Kurzawski, and J. Templeton. "Reynolds averaged turbulence modelling using deep neural networks with embedded invariance". In: *Journal of Fluid Mechanics* 807 (2016), pp. 155–166.

[Liu+19] X. Liu et al. "Neural SDE: Stabilizing Neural ODE Networks with Stochastic Noise". In: *arXiv:1906.02355* (2019).

[Lor+21] J. Lorraine et al. "Complex Momentum for Optimization in Games". In: *arXiv:2102.08431* (2021).

[Lou+20] A. Lou et al. "Neural Manifold Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 17548–17558.

[Lu+21] C. Lu et al. "Implicit Normalizing Flows". In: *International Conference on Learning Representations*. 2021.

[Lu+17a] Y. Lu et al. "Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations". In: *arXiv:1710.10121* (2017).

[Lu+17b] Z. Lu et al. "The Expressive Power of Neural Networks: A View from the Width". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[Luc+18] M. Lucic et al. "Are GANs Created Equal? A Large-Scale Study". In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018.

[LKB18] B. Lusch, J. N. Kutz, and S. L. Brunton. "Deep learning for universal linear embeddings of nonlinear dynamics". In: *Nature Communications* 9.1 (2018), p. 4950.

[Lut+21] M. Lutter et al. “Value Iteration in Continuous Actions, States and Time”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 7224–7234.

[Lyo04] T. Lyons. “Rough paths, Signatures and the modelling of functions on streams”. In: *Proceedings of the International Congress of Mathematicians* (2004).

[LCL04] T. Lyons, M. Caruana, and T. Lévy. *Differential equations driven by rough paths*. École d’Été de Probabilités de Saint-Flour XXXIV, 2004.

[Lyo98] T. J. Lyons. “Differential equations driven by rough signals.” In: *Revista Matemática Iberoamericana* 14.2 (1998), pp. 215–310.

[MQH18] M. Magill, F. Qureshi, and H. de Haan. “Neural Networks Trained to Solve Differential Equations Learn General Representations”. In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018.

[Man+16] N. M. Mangan et al. “Inferring Biological Networks by Sparse Identification of Nonlinear Dynamics”. In: *IEEE Transactions on Molecular, Biological and Multi-Scale Communications* 2.1 (2016), pp. 52–63.

[MRO20] D. Maoutsa, S. Reich, and M. Opper. “Interacting particle solutions of Fokker-Planck equations through gradient-log-density estimation”. In: *arXiv:2006.00702* (2020).

[ML16] G. Martius and C. H. Lampert. “Extrapolation and learning equations”. In: *arXiv:1610.02995* (2016).

[Mas+20] S. Massaroli et al. “Dissecting Neural ODEs”. In: *Advances in Neural Information Processing Systems 33*. Curran Associates, Inc., 2020.

[Mas+21] S. Massaroli et al. “Differentiable Multiple Shooting Layers”. In: *Advances in Neural Information Processing Systems 34*. Curran Associates, Inc., 2021.

[MN20] E. Mathieu and M. Nickel. “Riemannian Continuous Normalizing Flows”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 2503–2515.

[Mau+19] R. Maulik et al. “Subgrid modelling for two-dimensional turbulence using neural networks”. In: *Journal of Fluid Mechanics* 858 (2019), pp. 122–144.

[Men+21] C. Meng et al. “SDEdit: Image Synthesis and Editing with Stochastic Differential Equations”. In: *arXiv:2108.01073* (2021).

[Miy+18] T. Miyato et al. “Spectral Normalization for Generative Adversarial Networks”. In: *International Conference on Learning Representations*. 2018.

[Mor+20] J. Morrill et al. “A Generalised Signature Method for Time Series”. In: *arXiv:2006.00873* (2020).

[Mor+21a] J. Morrill et al. “Neural Controlled Differential Equations for Online Prediction Tasks”. In: *arXiv:2106.11028* (2021).

[Mor+21b] J. Morrill et al. “Neural Rough Differential Equations for Long Time Series”. In: *International Conference on Machine Learning* (2021).

[Mut13] U. Mutze. “An asynchronous leapfrog method II”. In: *arXiv:1311.6602* (2013).

[New36] I. Newton. *The Method of Fluxions and Infinite Series; with its Application to the Geometry of Curve-Lines*. Translated from the Author’s Latin Original not yet made publick. To which is subjoin’d, A Perpetual Comment upon the whole Work, Consisting of Annotations, Illustrations, and Supplements, In order to make this Treatise A compleat Institution for the use of Learners. By John Colson. The Lamb without Temple-Bar, London: Printed by Henry Woodfall and sold by John Nourse, 1736.

[Nor+20] A. Norcliffe et al. “On Second Order Behaviour in Augmented Neural ODEs”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 5911–5921.

[Nor+21] A. Norcliffe et al. “Neural ODE Processes”. In: *International Conference on Learning Representations*. 2021.

[OVV20] V. Oganesyan, A. Volokhova, and D. Vetrov. “Stochasticity in Neural ODEs: An Empirical Study”. In: *arXiv:2002.09779* (2020).

[OR20] D. Onken and L. Ruthotto. “Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and Continuous Normalizing Flows”. In: *arXiv:2005.13420* (2020).

[Onk+21] D. Onken et al. “OT-Flow: Fast and Accurate Continuous Normalizing Flows via Optimal Transport”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.10 (2021), pp. 9223–9232.

[Ott+21] K. Ott et al. “ResNet After All: Neural ODEs and Their Numerical Solution”. In: *International Conference on Learning Representations*. 2021.

[Pal+21] A. Pal et al. “Opening the Blackbox: Accelerating Neural Differential Equations by Regularizing Internal Solver Heuristics”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 8325–8335.

[Par+21] S. Park et al. “Minimum Width for Universal Approximation”. In: *International Conference on Learning Representations*. 2021.

[Pas+19] A. Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[Pav14] G. A. Pavliotis. *Stochastic Processes and Applications: Diffusion Processes, the Fokker-Planck and Langevin Equations*. Springer, New York, 2014.

[Per18] I. Perez Arribas. “Derivatives pricing using signature payoffs”. In: *arXiv:1809.09466* (2018).

[Pin99] A. Pinkus. “Approximation theory of the MLP model in neural networks”. In: *Acta Numer.* 8 (1999), pp. 143–195.

[PSW19] M. L. Piscopo, M. Spannowsky, and P. Waite. “Solving differential equations with neural networks: Applications to the calculation of cosmological phase transitions”. In: *Phys. Rev. D* 100 (1 2019).

[Pol+19] M. Poli et al. “Graph Neural Ordinary Differential Equations”. In: *arXiv:1911.07532* (2019).

[Pol+20] M. Poli et al. “Hypersolvers: Toward Fast Continuous-Depth Models”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 21105–21117.

[Pol+21] M. Poli et al. “Neural Hybrid Automata: Learning Dynamics with Multiple Modes and Stochastic Transitions”. In: *Advances in Neural Information Processing Systems 34*. Curran Associates, Inc., 2021.

[Pol21] F. Pollock. Private communication. 2021.

[Pon+62] L. S. Pontryagin et al. *The mathematical theory of optimal processes*. 1962.

[Por+19] G. Portwood et al. “Turbulence forecasting via Neural ODE”. In: *Machine Learning and the Physical Sciences, NeurIPS Workshop* (2019).

[QWX19] T. Qin, K. Wu, and D. Xiu. “Data driven governing equations approximation using deep neural networks”. In: *Journal of Computational Physics* 395 (2019), pp. 620–635.

[Que+21] A. F. Queiruga et al. “Continuous-in-Depth Neural Networks”. In: *arXiv:2008.02389* (2021).

[Rac21a] C. Rackauckas. *Notes on Algorithms*. `https://devdocs.sciml.ai/dev/internals/notes_on_algorithms/`. 2021.

[Rac21b] C. Rackauckas. *Timestepping Method Descriptions*. `https://diffeq.sciml.ai/stable/extras/timestepping/`. 2021.

[Rac+20a] C. Rackauckas et al. “A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions”. In: *arXiv:1812.01892* (2020).

[Rac+20b] C. Rackauckas et al. “Universal Differential Equations for Scientific Machine Learning”. In: *arXiv:2001.04385* (2020).

[Rad+21] A. Radford et al. “Learning Transferable Visual Models From Natural Language Supervision”. In: *arXiv:2103.00020* (2021).

[RPK19] M. Raissi, P. Perdikaris, and G. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[Rai18] M. Raissi. “Deep Hidden Physics Models: Deep Learning of Nonlinear Partial Differential Equations”. In: *Journal of Machine Learning Research* 19.25 (2018), pp. 1–24.

[RPK18] M. Raissi, P. Perdikaris, and G. E. Karniadakis. “Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems”. In: *arXiv:1801.01236* (2018).

[RZL17] P. Ramachandran, B. Zoph, and Q. V. Le. “Searching for Activation Functions”. In: *arXiv:1710.05941* (2017).

[Ram+20a] A. Ramadhan et al. “Capturing missing physics in climate model parameterizations using neural differential equations”. In: *arXiv:2010.12559* (2020).

[Ram+20b] H. Ramsauer et al. “Hopfield Networks is All You Need”. In: *arXiv:2008.02217* (2020).

[Rea+19] E. Real et al. “Regularized Evolution for Image Classifier Architecture Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (2019), pp. 4780–4789.

[Rei17] J. Reizenstein. “Calculation of Iterated-Integral Signatures and Log Signatures”. In: *arXiv:1712.02757* (2017).

[Rei18] J. Reizenstein. “The iisignature library: efficient calculation of iterated-integral signatures and log signatures”. In: *arXiv:1802.08252* (2018).

[RY13] D. Revuz and M. Yor. *Continuous martingales and Brownian motion*. Vol. 293. Springer Science & Business Media, 2013.

[RM15] D. Rezende and S. Mohamed. “Variational Inference with Normalizing Flows”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 1530–1538.

[RAK94] R. Rico-Martinez, J. Anderson, and I. Kevrekidis. “Continuous-time nonlinear signal processing: a neural network based approach for gray box identification”. In: *Proceedings of IEEE Workshop on Neural Networks for Signal Processing*. 1994, pp. 596–605.

[RK93] R. Rico-Martinez and I. Kevrekidis. “Continuous time modeling of nonlinear systems: a neural network-based approach”. In: *IEEE International Conference on Neural Networks*. Vol. 3. 1993, pp. 1522–1525.

[Ric+92] R. Rico-Martínez et al. “Discrete-vs. continuous-time nonlinear signal processing of Cu electrodissolution data”. In: *Chemical Engineering Communications* 118.1 (1992), pp. 25–48.

[RRS21] E. Roesch, C. Rackauckas, and M. P. H. Stumpf. “Collocation based training of neural ordinary differential equations”. In: *Statistical Applications in Genetics and Molecular Biology* 20.2 (2021), pp. 37–49.

[RFB15] O. Ronneberger, P. Fischer, and T. Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *MICCAI* (2015).

[Ros+17] M. Rosca et al. “Variational Approaches for Auto-Encoding Generative Adversarial Networks”. In: *arXiv:1706.04987* (2017).

[Roz+21] N. Rozen et al. “Moser Flow: Divergence-based Generative Modeling on Manifolds”. In: *arXiv:2108.08052* (2021).

[RCD19] Y. Rubanova, R. T. Q. Chen, and D. K. Duvenaud. “Latent Ordinary Differential Equations for Irregularly-Sampled Time Series”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 5320–5330.

[Rud+17] S. H. Rudy et al. “Data-driven discovery of partial differential equations”. In: *Science Advances* 3.4 (2017).

[Sal+11] J. K. Salmon et al. “Parallel random numbers: as easy as 1, 2, 3.” In: *Proc. High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[SLG21] C. Salvi, M. Lemercier, and A. Gerasimovics. “Neural Stochastic Partial Differential Equations”. In: *arXiv:2110.10249* (2021).

[Sal+20] C. Salvi et al. “The Signature Kernel is the solution of a Goursat PDE”. In: *2006.14794* (2020).

[San+21] M. E. Sander et al. “Momentum Residual Neural Networks”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 9276–9287.

[Sch+21] A. Schwarzschild et al. “The Uncanny Similarity of Recurrence and Depth”. In: *arXiv:2102.11011* (2021).

[SH05] R. Serban and A. Hindmarsh. “Cvodes, the sensitivity-enabled ode solver in sundials”. In: *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 6. 2005.

[Sha09] L. Shampine. “Stability of the leapfrog/midpoint method”. In: *Applied Mathematics and Computation* 208.1 (2009), pp. 293–298.

[Shi+21] C. Shi et al. “Learning Gradient Fields for Molecular Conformation Generation”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 9558–9568.

[SM21] R. Shi and Q. Morris. "Segmenting Hybrid Trajectories using Latent ODEs". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 9569–9579.

[SM19] S. N. Shukla and B. Marlin. "Interpolation-Prediction Networks for Irregularly Sampled Time Series". In: *International Conference on Learning Representations*. 2019.

[SP03] T. K. Soboleva and A. B. Pleasants. "Population Growth as a Nonlinear Stochastic Process". In: *Mathematical and Computer Modelling* 38.11–13 (2003), pp. 1437–1442.

[Son+21a] Y. Song et al. "Maximum Likelihood Training of Score-Based Diffusion Models". In: *arXiv:2101.09258* (2021).

[Son+21b] Y. Song et al. "Score-Based Generative Modeling through Stochastic Differential Equations". In: *International Conference on Learning Representations*. 2021.

[SAV20] E. Stevens, L. Antiga, and T. Viehmann. *Deep Learning with PyTorch*. Manning Publications Co., 2020.

[Tan+20] C. W. Tan et al. *Monash University, UEA, UCR Time Series Regression Archive*. 2020. URL: `http://timeseriesregression.org/`.

[Tes+20] T. Teshima et al. "Universal Approximation Property of Neural Ordinary Differential Equations". In: *Differential Geometry meets Deep Learning, NeurIPS 2020 workshop* (2020).

[Thu+21] N. Thuerey et al. *Physics-based Deep Learning*. WWW, 2021. URL: `https://physicsbaseddeeplearning.org`.

[TBO21] C. Toth, P. Bonnier, and H. Oberhauser. "Seq2Tens: An Efficient Representation of Sequences by Low-Rank Tensor Projections". In: *International Conference on Learning Representations*. 2021.

[Tsi11] C. Tsitouras. "Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2 (2011), pp. 770–775.

[TR19a] B. Tzen and M. Raginsky. "Neural Stochastic Differential Equations: Deep Latent Gaussian Models in the Diffusion Limit". In: *arXiv:1905.09883* (2019).

[TR19b] B. Tzen and M. Raginsky. "Theoretical guarantees for sampling and inference in generative models with latent diffusions". In: *Conference on Learning Theory* (2019).

[Vas+17] A. Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[Wal21] B. Walker. Private communication. 2021.

[WWX17] J.-X. Wang, J.-L. Wu, and H. Xiao. "Physics-informed machine learning approach for reconstructing Reynolds stress modeling discrepancies based on DNS data". In: *Phys. Rev. Fluids* 2 (3 2017).

[Wan+21] R. Wang et al. "Bridging Physics-based and Data-driven modeling for Learning Dynamical Systems". In: *Proceedings of the 3rd Conference on Learning for Dynamics and Control*. Vol. 144. Proceedings of Machine Learning Research. PMLR, 2021, pp. 385–398.

[WGY18] G. Weiss, Y. Goldberg, and E. Yahav. "On the Practical Computational Power of Finite Precision RNNs for Language Recognition". In: *Association for Computational Linguistics* (2018).

[WG10] A. G. Wilson and Z. Ghahramani. "Copula Processes". In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010.

[WK20] E. Winston and J. Z. Kolter. "Monotone operator equilibrium networks". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 10718–10728.

[Win60] P. Winters. "Forecasting sales by exponentially weighted moving averages". In: *Management Science* 6 (1960), pp. 324–342.

[WC13] T. Worm and K. Chiu. "Prioritized Grammar Enumeration: Symbolic Regression by Dynamic Programming". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1021–1028.

[XZW21] H. Xie, L. Zhang, and L. Wang. "Ab-initio study of interacting fermions at finite temperature with neural canonical transformation". In: *arXiv:2105.08644* (2021).

[Yan+19] G. Yang et al. "PointFlow: 3D Point Cloud Generation with Continuous Normalizing Flows". In: *arXiv:1906.12320* (2019).

[Yaz+19] Y. Yazıcı et al. "The Unusual Effectiveness of Averaging in GAN Training". In: *International Conference on Learning Representations*. 2019.

[YHL19] C. Yildiz, M. Heinonen, and H. Lahdesmaki. "ODE2VAE: Deep generative second order ODEs with Bayesian neural networks". In: *Advances in Neural Information Processing Systems 32*. 2019.

[Yin+21] Y. Yin et al. "Augmenting Physical Models with Deep Networks for Complex Dynamics Forecasting". In: *International Conference on Learning Representations*. 2021.

[YJS19] J. Yoon, D. Jarrett, and M. van der Schaar. "Time-series Generative Adversarial Networks". In: *Advances in Neural Information Processing Systems 32*. 2019.

[Zei12] M. D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *arXiv:1212.5701* (2012).

[Zha+20] H. Zhang et al. “Approximation Capabilities of Neural ODEs and Invertible Residual Networks”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 11086–11095.

[ZC21] Q. Zhang and Y. Chen. “Diffusion Normalizing Flow”. In: *Thirty-Fifth Conference on Neural Information Processing Systems*. 2021.

[Zha+19] T. Zhang et al. “ANODEV2: A Coupled Neural ODE Framework”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[Zha+17] X. Zhang et al. “PolyNet: A Pursuit of Structural Diversity in Very Deep Networks”. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2017).

[ZDC20a] Y. D. Zhong, B. Dey, and A. Chakraborty. “Dissipative SymODEN: Encoding Hamiltonian Dynamics with Dissipation and Control into Deep Learning”. In: *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations* (2020).

[ZDC20b] Y. D. Zhong, B. Dey, and A. Chakraborty. “Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control”. In: *International Conference on Learning Representations*. 2020.

[ZDC21] Y. D. Zhong, B. Dey, and A. Chakraborty. “Extending Lagrangian and Hamiltonian Neural Networks with Differentiable Contact Models”. In: *arXiv:2102.06794* (2021).

[Zhu+20a] J. Zhuang et al. “AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients”. In: *Neural Information Processing Systems* (2020).

[Zhu+20b] J. Zhuang et al. “Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 11639–11649.

[Zhu+21] J. Zhuang et al. “MALI: A memory efficient and reverse accurate integrator for Neural ODEs”. In: *International Conference on Learning Representations*. 2021.

[Zub+21] K. Zubov et al. “NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations”. In: *arXiv:2107.09443* (2021).

Notation

$\mathbb{R}^{d_1 \times d_2}$	The space of real matrices with d_1 rows and d_2 columns.
$\mathbb{R}^{d_1 \times \cdots \times d_k}$	The space of tensors of shape $(d_1, \ldots, d_k)$.
$I_{d \times d}$	The $d \times d$ identity matrix.
$\operatorname{tr}(A)$	The trace of a square matrix $A \in \mathbb{R}^{d \times d}$.
$\operatorname{diag}(v)$	The diagonal matrix in $\mathbb{R}^{d \times d}$ whose diagonal entries are given by the vector $v \in \mathbb{R}^d$.
$\operatorname{Re}(\lambda)$	The real part of a complex number $\lambda \in \mathbb{C}$.
$L(X; Y)$	The space of linear (not affine) functions $X \to Y$. If Y is omitted then $Y = \mathbb{R}$.
$L_b(X; Y)$	The space of affine (not linear) functions $X \to Y$. If Y is omitted then $Y = \mathbb{R}$.
$C(X; Y)$	The space of continuous functions $X \to Y$. (With respect to some topologies on X and Y.) If Y is omitted then $Y = \mathbb{R}$.
$L^p(X; Y)$	The space of p-integrable functions $X \to Y$. If Y is omitted then $Y = \mathbb{R}$.
$W^{1,p}(X; Y)$	The space of p-integrable functions $X \to Y$ with p-integrable first derivative. If Y is omitted then $Y = \mathbb{R}$.
$\operatorname{BV}(X; Y)$	The space of (possibly discontinuous) functions $X \to Y$ with bounded variation. If Y is omitted then $Y = \mathbb{R}$.
$\operatorname{Lip}(X; Y)$	The space of Lipschitz functions $X \to Y$. If Y is omitted then $Y = \mathbb{R}$.
$a \mapsto b$	The function mapping a to b. That is, $a \mapsto b$ denotes the function f such that $f(a) = b$. (Sometimes referred to as an 'anonymous' or 'lambda' function.)
$\lVert \cdot \rVert_p$	The L^p norm.
$\lvert \cdot \rvert_{\operatorname{BV}}$	The bounded variation seminorm.
$\mathbb{1}_A$	The indicator function with value 1 when the condition A is true, and value 0 when A is false.

$\sim$	Denotes sampling from a probability distribution: $x \sim \mu$ denotes a sample x from probability distribution μ.
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ and variance σ^2.
Uniform$[a, b]$	Uniform distribution over $[a, b] \subseteq \mathbb{R}$.
$\mathbb{P}$	A probability measure; the probability of an event occurring or a statement being true.
$\mathrm{KL}(\mathbb{P} \Vert \mathbb{Q})$	The Kullback–Leibler divergence between two probability measures $\mathbb{P}$ and $\mathbb{Q}$. Will also be written $\mathrm{KL}(X \Vert Y)$ to denote the KL divergence between the laws of two random variables X,Y, or $\mathrm{KL}(p \Vert q)$ to denote the KL divergence between the two probability measures corresponding to the densities p, q.
$\int f(z(s)) \, \mathrm{d}X(s)$	A Riemann–Stieltjes integral driven by X. '$f \, \mathrm{d}X$' refers to a matrix-vector product.
$\circ \, \mathrm{d}w(t)$	Used to denote integration as a Stratonovich SDEs. (As opposed to just '$\mathrm{d}w(t)$' denoting an Itô SDE.)
$\otimes$	The tensor product, defined by $$\otimes \colon \mathbb{R}^{d_1 \times \cdots \times d_k} \times \mathbb{R}^{d_{k+1} \times \cdots \times d_m} \to \mathbb{R}^{d_1 \times \cdots \times d_m},$$ $$\otimes \colon (a_{i_1, \ldots, i_k}, b_{i_{k+1}, \ldots, i_m}) \to a_{i_1, \ldots, i_k} b_{i_{k+1}, \ldots, i_m}.$$ When applied to two vectors, this is the outer product.

In addition, following the convention of the machine learning literature (and with apologies to the mathematicians), we sometimes use 'max' and 'min' where 'sup' and 'inf' would be technically correct.

Abbreviations

Ordered alphabetically:

API	Application programming interface
CDE	Controlled differential equation
CNF	Continuous normalising flows
CNN	Convolutional neural network
DAG	Directed acyclic graph
DEQ	Deep equilibrium model
GAN	Generative adversarial network
GRU	Gated recurrent unit
JAX	JAX (not an abbreviation)
KL	Kullback–Leibler (divergence)
LASSO	Least absolute shrinkage and selection operator
LRU	Least recently used (a form of caching)
LSTM	Long-short term memory
MLP	Multi-layer perceptron; feedforward neural network
MMD	Maximum mean discrepancy
NDE	Neural differential equation
NCDE	Neural controlled differential equation
NODE	Neural ordinary differential equation
NRDE	Neural rough differential equation
NSDE	Neural stochastic differential equation
ODE	Ordinary differential equation
PRNG	Pseudo-random number generator
RDE	Rough differential equation
RL	Reinforcement learning
RMS	Root-mean-squared

RNN	Recurrent neural network
SDE	Stochastic differential equation
SGD	Stochastic gradient descent
SINDy	Sparse identification of nonlinear dynamics
UDE	Universal differential equation

Index