Imperial College
London

Imperial College London

Department of Mathematics

Neural Rough Differential Equations for Long Time Series and Classification of Default Risk

Author: Hinrik Bergs (CID: 00449457)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2020-2021

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Abstract

In this thesis we provide an exposition on the key concepts, motivations and milestones underpinning machine learning models that combine Neural Networks with the modelling paradigm of differential equations, with the aim of discussing the Neural Rough Differential Equation (RDE) model [1]. The Neural RDE model combines the continuously updated dynamics and solution methods of differential equations with the ability of Recurrent Neural Networks to capture sequential dynamics. Furthermore, Neural RDEs utilise a mathematical concept from Rough Path Theory called the (log-)signature to summarise the input signal to reduce the effective length of a time series and a method called log-ODE to solve the differential equation that drives the hidden state. The model is compared to benchmark models and shown to offer superior performance in classifying longer time series of multivariate data in which sequential dynamics contain orthogonal information important to the outcome of the task. We provide an illustrative example of how the Neural RDE model can be coded up [2] and used to classify longer time series. Finally, we propose how the Neural RDE model could be used to classify default risk in a financial setting by extending some very recent work on sequential deep learning for credit risk classification undertaken at American Express, AI Research [3].

Acknowledgements

I would like to thank Dr. Thomas Cass for suggesting the topic, providing invaluable guidance and introducing me to the fascinating field of Signatures and Rough Paths in Machine Learning by teaching a course on the topic in a pedagogic manner.

Furthermore I would like to thank the authors of [1] and [4] for developing these very interesting concepts and being so thoughtful in their presentation of these often difficult to understand writings, providing rich supplementary materials on the underlying theory and detailed instructions for implementing their approach.

Contents

List of Figures

List of Tables

Introduction

In this thesis we will discuss Neural Rough Differential Equations (RDE), an extension of Neural Controlled Differential Equations that uses the log-signature path to summarise a (possibly) irregularly sampled time series. The Neural Controlled Equation may be thought of as a continuous time equivalent of a Recurrent Neural Network, a deep learning approach to capture the sequential dynamics of various types of data.

Chapter by chapter, we construct an understanding of the key concepts that underpin Neural RDE and finally combine them to disuss the model and its suitability to learn the sequential dynamics of a long time series.

We illustrate a working example on a time series of length 5,000 and discuss the potential applications of the model, e.g., to classify credit default risk of a borrower.

Traditional Feedforward Neural Networks do not incorporate sequential dynamics. Recurrent Neural Networks capture sequential dynamics but have drawbacks such as being discrete time, requiring regularly sampled time series and incurring a vast number of computations for modest time series leading to vanishing/exploding gradients and reduced accuracy.

The Neural ODE model introduces a new family of deep neural network models [7] by replacing the discrete sequence of hidden layers with appropriately constructed ODEs. The Neural ODE model introduces continuous time transformations for the hidden state of a Feedforward Neural Network and applies a vastly more memory efficient method for backpropagation called the adjoint sensitivity method. The introduction of the Neural ODE model allows Neural Networks to be viewed through the lens of differential equation modelling. As a result, it becomes natural to input data that is irregularly sampled, it becomes possible to trade precision for speed and existing tools to handle differential equations become available.

[4] extend the Neural ODE model by incorporating a concept from Rough Path Theory called a Controlled Differential Equation (CDE) to introduce the Neural CDE model. A controlled differential equation is driven by a continuous path interpolated between the observables. The hidden state is dependent on this path and so becomes continuously varying. The Neural CDE model may be thought of as the continuous time equivalent of Recurrent Neural Networks as it captures sequential dynamics in a dataset. Further to the abilities of the Neural ODE model, the Neural CDE can process new information without interruption and thus capture sequential dynamics. As the path that drives the differential equation that determines the

hidden state is interpolated to be continuous, data may just as well be irregularly sampled and partially observed. Much alike its discrete step equivalent the Recurrent Neural Network, the Neural CDE model only works well with relatively short time series (max. low hundreds). The number of forward operations for each forward pass becomes unmanageable, loss/accuracy suffers and training times become prohibitively long. However, the Neural CDE model appears to be quite effective in learning time series where the sequential dynamics of a dataset contain non-trivial information and the data is irregularly sampled and or partially observed. Indeed, it is particularly noteworthy how effective the Neural CDE model is in dealing with missing data and [4] showed that the model remained remarkably accurate for time series where 70% of the data had been dropped at random.

This motivated [1] to develop the Neural Rough Differential Equation model in which the effective length of the input signal or time series is dramatically reduced by summarising the input signal over small time intervals using a concept called the log-signature. The log-signature is a compressed set of ordered statistics that describe how the input signal drives the controlled differential equation and therefore the hidden state. By combining rough path theory, the log-signature and previously existing methods for solving ODEs, [1] create the log-ODE method which allows the differential equation that drives the hidden state to be solved with integration steps much larger than the discretisation of the data where again accuracy may be traded for speed. The rough approach allows the effectiveness of the Neural CDE formulation to be applied to a much longer time series, shown by [1] to be effective up to a length of c. 17k making the whole approach feasible for a much larger range of applications.

We provide a simple illustrative example of how the Neural RDE method is scripted into code and highlight the results.

Finally, we dissuss a potential application of the Neural RDE model in predicting default risk by capturing the sequential dynamics of a long time series of irregularly sampled data on borrowers to classify credit default risk. Our discussion of this potential application builds upon the very recent work of researchers at American Express AI Research [3] which shows that discrete time sequential models such as RNNs offer measurable improvements over non-sequential models in predicting default risk. Due to the nature of lending, small improvements in predicting default risk can lead to large savings.

Chapter 1

Recurrent Neural Networks

A staple of deep learning is the Feedforward Neural Network (FNN) which operates by taking a fixed and static input to produce a fixed and static ouput, i.e., there is no inherent notion of sequence or time. The activations flow only in one direction from input to output. As sequential data arises in many settings, e.g. financial time series, machine translation, trajectories of moving objects etc. it is important to have a model that can be used for learning sequential data. This is particularly true when a whole time series is to be classified. Going from the static Neural Network model to capture sequential dynamics can be done using Recurrent Neural Networks (RNN), a type of Artificial Neural Network with feedback connections allowing for sequential dynamics behaviour. In an RNN, the network output and its computations are not only a function of the input at a particular time step but also the past. The Neural Differential equation models [4, 1] that form the focal point of this thesis can in some ways be thought of as a continuous time equivalent of the RNN. Therefore we begin by giving a brief account of the concepts underlying RNNs, following the outlines of [5, Chapter 14, pages 470-505] which provides a more complete and practical description of RNNs than summarised here.

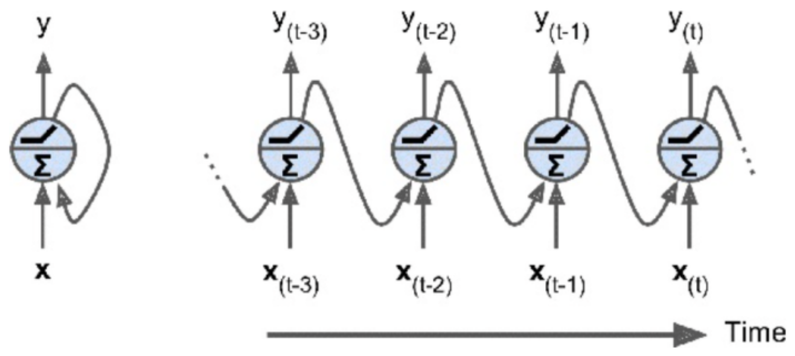


Figure 1.1: A single recurrent neuron (left), unrolled through time (right) [5, Figure 14-1, pg. 470]

1.1 Recurrent Neurons

The simplest example of an RNN would be a single neuron that is fed inputs, produces an output and then feeds that output back to itself along with a new input at the subsequent time steps as is shown in figure 1.1 (left). If we unroll these individual time steps across time as shown in figure 1.1 (right) we can see the neuron/s can be described via a recurrence relation. This means that at each time step (a.k.a. a frame) the cell state depends on the current input $x_{(t)}$ and on the prior cell state (output) $y_{(t)}$.

RNNs can otherwise be constructed in much the same way as FNNs, with varying numbers of layers and neuron units. Figure 1.2 depicts a single layer of five recurrent neurons. The input is now in vector format in order to specifiy the input for each individual neuron in the layer. Therefore, at each indvidual time step, each individual neuron is simultaneously fed as input, a combination from the input vector $x_{(t)}$ and the previous output vector $y_{(t-1)}$. Weight vectors W_x and W_y are assigned to each neuron to determine the combination of inputs from the input vector and the output vector of the previous time step respectively.

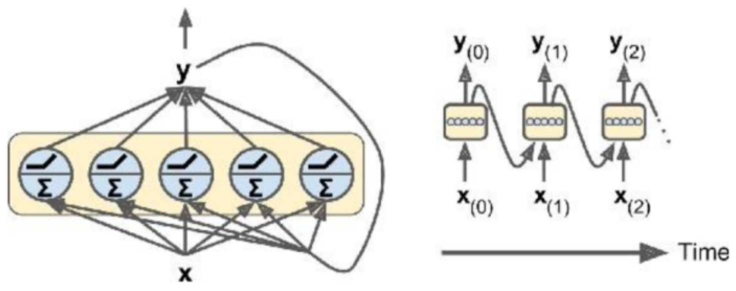


Figure 1.2: A single layer of five neurons (left), unrolled through time (right) [5, Figure 14-2, pg. 470]

Definition 1.1.1. The output of a single layer of recurrent neurons is given by equation (1.1.1)

$$y_{(t)} = \phi\left(x_{(t)}^T \cdot w_x + y_{(t-1)}^T \cdot w_y + b\right) \tag{1.1.1}$$

where b is the neuron's bias term and $\phi(\cdot)$ is the activation function. Equation (1.1.1) can be extended to vectorised form to calculate the output of a recurrent layer for a number of instances, for example a mini-batch (see [5, Ch. 14]). This gives the below equation for outputs of a layer of recurrent neurons for multiple instances,

$$\begin{aligned} y_{(t)} &= \phi\left[X_{(t)} \cdot X_x + Y_{(t-1)} \cdot W_y + b\right) && (1.1.2) \\ &= \phi\left(\begin{bmatrix} X_{(t)} & Y_{(t-1)} \end{bmatrix} \cdot W + b\right), \qquad W = \begin{bmatrix} W_x \\ W_{y,} \end{bmatrix} && (1.1.3) \end{aligned}$$

1.2 Memory Cells

As seen above, the output of an RNN at a particular time step depends not only on the input at that particular time step but on all of the inputs up until that time. This sequential nature can be viewed as a form of memory. A *memory cell* or simply a *cell* is the part of an RNN that carries some state (information) between time steps. So far we have looked at the two most basic cells that would constitute an RNN, the single recurrent neuron and a single layer of recurrent neurons. However, an RNN can be constructed with multiple layers and varying numbers of neurons to give deep RNNs in much the same way as FNNs.

The cells in an RNN have a hidden state denoted by $h_{(t)}$ that is updated at each time step as a sequence is processed. The hidden state $h_{(t)}$, like the output, is a function of the inputs at that time step and the hidden state at the previous time step, $h(t) = f(h_{(t-1)}, x_(t))$. For the simple cells discussed above, the output was simply equal to the cell state, however this is often not the case and the RNN unrolled through time takes the form seen in figure 1.3

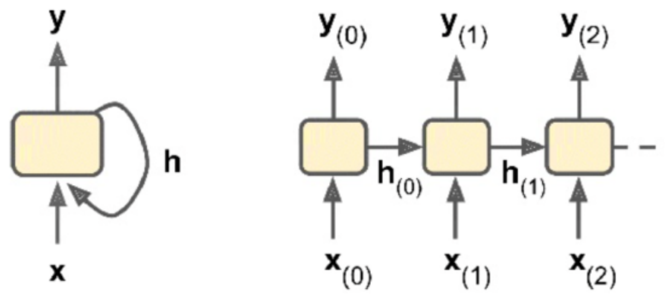


Figure 1.3: An RNN where the cells output and hidden state are not the same [5, Figure 14-3, pg. 471]

1.3 Input and Output Sequences

A benefit of RNNs is that they can have variable length sequences as both inputs and outputs. This is in contrast to feedforward neural networks (FNN) that only work with sequences of pre-determined length. Figure 1.4 shows four common combinations of inputs and outputs used with RNN models. The top left network is an example of the type that could be used for time series prediction, e.g. stock prices. The input would be the stock price at each of the last N time steps, and the output could be the stock price at the next time steps. This would be an example of a sequence of inputs to produce a sequence of outputs (many to many). The top right network takes a sequence of inputs to produce only one output, i.e. a sequence (of vectors) to vector network (many to one), used for example in sentiment classification. The bottom left network takes in a single input vector and outputs a sequence (one to many), an example use would be text generation or image captioning. Finally, the type of network seen in the bottom right is another example of a many

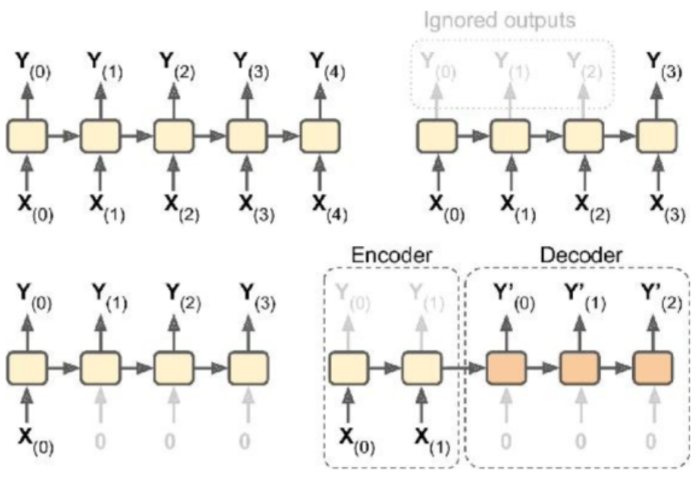


Figure 1.4: Many to many (top left), many to one (top right), one to many (bottom left) and encoder-decoder (bottom right) [5, Figure 14-4, pg. 473]

to many network that is composed first of a sequence-to-vector (encoder) and then a vector-to-sequence (decoder). Machine translation is an example where this is particularly useful. The original sentence is processed into a single vector representation by the encoder part of the network that is then processed into a sequence of outputs, i.e. a translated sentence, by the decoder. This provides much improved translations as the sequential nature of sentences means that the meaning of the words within a sentence is highly dependant on the other words in a sentence and their positioning.

1.4 Training RNNs

In Feedforward Neural Network (FNN) models, a backpropagation algorithm involves first taking the derivative (gradient) of the cost or loss with respect to each parameter and second, shifting the parameters until the loss is minimised. Similarly, an RNN is trained using a method called backpropagation through time (BPTT). This is done by unrolling the RNN through time and using backpropagation as in FNN models. A schematic representation of BPTT is shown in figure 1.5. From the figure, we can intuitively see the vast number of computations and the large memory cost associated to this for even a modest number of time steps, a drawback of RNN models.

1.4.1 Backpropagation through time

The training of an RNN model begins like backpropagation for an FNN with a first forward pass through the unrolled network as represented by the dashed arrows in figure 1.5. The error between predicted values and expected values in the output is evaluated using some cost function,

$$C\left(Y_{(t_{min})}, Y_{(t_{min}+1)}, ..., Y_{t_{max}}\right), \tag{1.4.1}$$

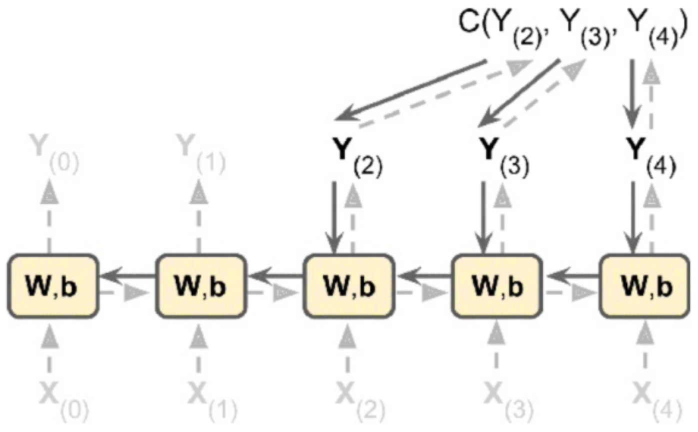


Figure 1.5: BPTT: Unrolling an RNN through time and minimise cost/loss w.r.t. each parameter [5, Figure 14-5, pg. 480]

where $Y_{(t_{min})}$ is the first output that is not ignored and $Y_{(t_{max})}$ is the last output. The reason why the method is called backpropagation through time (BPTT) is that now the gradients of the cost function are propagated backwards through each individual time step of the unrolled RNN all the way from where we currently are in the sequence and back to the beginning. All the weights and biases (model parameters) are updated to minimise the cost function using the gradients computed during BPTT.

Gradient Issues

If we look at how gradients flow across the unrolled RNN in figure 1.5, we can see that between each time step a matrix multiplication is performed that involves the weight matrix W_h of $h_{(t)}$ as a parameter. Therefore, computing the gradient with respect to the initial cell state $h_{(t_{min})}$ requires a large number of multiplications of W_h and an equal number of computations of the gradients with respect to the weight matrix which can lead to either vanishing gradients (many factors smaller than 1) and exploding gradients (many factors larger than 1). With exploding gradients, training explicitly breaks down as the gradients become too large to handle and are therefore easier to deal with, e.g. by pre-defining a threshold. On the other hand, vanishing gradients are harder to handle as they are not as easily detected, training takes too long and the results are incorrect. Therefore, we are more concerned with the problem of vanishing gradients, that is, gradients that become so small that we are not able to effictively train the network.

Memory cost with long time series

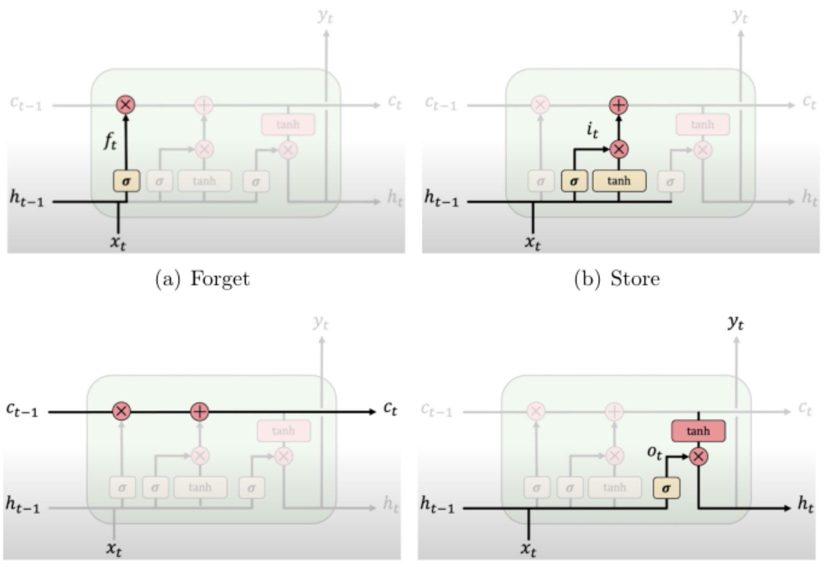
As the number of time steps grows, the number of computations that must be carried out quickly becomes prohibitive and because all of the intermediate quantities must be stored, training will incur a very large memory footprint. The memory cost associated to training RNNs is often the bottleneck for the length of time series that can be processed.

The methods used to handle these problems are only so efficient at dealing with the issues posed by long time series and so even for modest time series, training still takes disproportionately long and incurs huge memory costs and/or precision suffers.

Truncated Backpropagation through time (TBPTT)

It is possible to perform truncated backpropagation through time where the RNN is unrolled over a manageable number of time steps during training. As a result the RNN will only learn patterns exhibited across those specific time steps excluding longer-term patterns. Alternatively it is possible to unroll the RNN over a combination of time steps that are both near and further away in the sequence, e.g. using frequent data points for recent time steps and sparse data points for time steps further away. This can provide some improvement but still has the drawback of excluding potentially significant patterns from the more distant past that are revealed only in frequent data.

1.5 LSTM(Long-short term memory)/LSTM Cell



(a) Forget (b) Store

(c) Update (d) Output

Figure 1.6: A schematic representation of an LSTM cell. Forget irrelevant parts of previous state (top left), Stores relevant new information into the cell state (top right), selectively updates the cell state (bottom left) and output gate controls what information is sent to the next time step (bottom right) [6, Lecture 2]

Another issue with training RNNs on long time series is that the network eventually "forgets" inputs from earlier time steps. The computations at each time step erode information from earlier time steps when taking in new inputs. At some point, the effects of inputs from early time steps on the cell state of the RNN becomes negligible whilst effects of more recent inputs become dominant. For example, if a financial time series contains cyclical periods of extreme volatility during its early time steps followed by a period of low volatility, the RNN will predominantly predict continuing low volatility even though it may be intuitive that high volatility will return.

To combat this, gated cells have been designed that can keep a certain amount of information from previous (and more distant) time steps (long term memory) and balance it with a certain amount of new information contained in each new time step (short term memory). The most commonly used type of such cells is called long short term memory (LSTM) cells originally proposed by [8] and have since replaced the more basic cells described above for most practical purposes. There is also a simplified version, the GRU cell, (see [5, Ch. 14] for an introduction). Figure 1.6, shows how an LSTM cell gates the flow of information.

Chapter 2

Neural ODEs

In [7] a new family of deep neural network models is introduced, Neural Ordinary Differential Equations (ODE). The Neural ODE model links together Machine Learning's Neural Networks with the modelling capabilities of differential equations. To avoid confusion, note that although continuous in time, the Neural ODE model does not capture sequential dynamics of data like an RNN. In traditional (non-recurring) Neural Networks (such as ResNet) the hidden state of a cell undergoes a discrete sequence of transformations. In contrast, the Neural ODE model of [7] parameterises the continuous dynamics of these hidden states using an ordinary differential equation (ODE) that is specified by a neural network [7]. Note that the Neural ODE model, thanks to its continuous dynamics is able to use data sampled at irregular times unlike recurrent neural networks that require data to be evenly spaced in time (or pre processed in some way so as to be). Being able to naturally incorporate data sampled at irregular times is a major contribution of these new models involving Neural Differential Equations.

As the evolution of the hidden state is described by an ODE that in turn is speficified by a neural network, [7] were able to use black-box ODE solvers to compute the output of the neural network that specifies the ODE that describes the dynamics of the hidden state. Through the use of ODE solvers the Neural ODE model can then be evaluted adaptively and allows accuracy to be traded for speed.

A key feature the Neural ODE model is its ability to train via the adjoint sensitity method for backpropagation. This can reduce memory cost by an order of magnitude compared to backpropagating directly through the ODE solver. Deep models tend to require a prohibitively large memory footprint for a reasonably sized dataset as the intermediate quantities of a forward pass must be stored.

2.1 From Discrete Transformations to ODE Dynamics

Consider as an example how the decoder of a Recurrent Neural Network shown in figure 1.4 repeatedly computes a discrete sequence of transformations to a hidden state,

$$h_{t+1} = h_t + f(h_t, \theta_t), \tag{2.1.1}$$

where $t \in \{0...T\}$ and $\mathbf{h}_t \in \mathbb{R}^D$. In [9] and [10] it is argued that these repeated computations may be treated as a Euler discretisation of a continuous process [7]. Imagine that we increase the number of hidden layers in a Neural Network so that each partition in the iterative procedure becomes smaller. As the size of the partition tends to zero, using the definition of the derivative and rearranging equation (2.1.1) the dynamics of these hidden units can be described by the below ordinary differential equation (ODE),

$$\frac{d\mathbf{h}_t}{dt} = f(\mathbf{h}(t), t, \theta), \tag{2.1.2}$$

where f is a neural network. It is now possible to set up an ODE initial value problem by defining the output layer $\mathbf{h}(T)$ as the solution to the above ODE at a time T with the initial condition given by the input layer $\mathbf{h}(0)$. In [7], a black-box differential equation solver is used to compute the hidden unit dynamics $f(\mathbf{h}(t), t, \theta)$ from which it is possible to derive $\mathbf{h}(T)$.

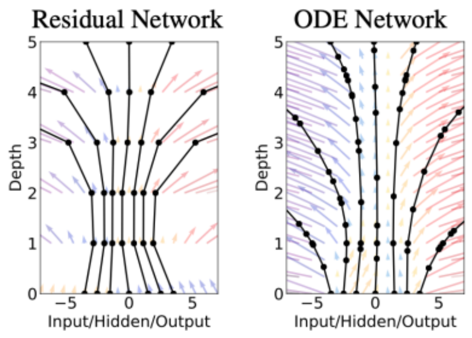


Figure 2.1: Left: Finite transformations in a discrete sequence as defined by a Residual Neural Network. Right: Continuous transformations of the state by a vector field define by an ODE network [7, Figure 1, pg. 1]

Figure 2.1 highlights the transition from a discrete sequence of transformations such as those found in a Residual Network (ResNet) to the ODE network that continuously transforms the state. We mention ResNets [11] here as [7] consider their Neural ODE model to be a continuous time analogy to a ResNet but the details of the ResNet are not important beyond the fact that it is a non-recurring feedforward neural network.

2.1.1 Advantages of Modelling with Differential Equations

By describing the dynamics of the hidden state with an ODE, the Neural ODE model introduced in [7] is able to incorporate black-box ODE solvers as a part of the model and this carries forward into the Neural CDE and RDE models discussed later. This use of ODE solvers has some significant advantages that are listed in [7] and recounted here.

Memory Efficiency

[7] demonstrates that it is possible to construct a method for computing the gradients of a scalar-valued loss function with respect to a hidden state $z(t)$ (or $h(t)$) at each instant, without backpropagating through the computations of the black-box ODE solver. This is achieved using the adjoint sensitivity method, described in section 2.1.2 below. This means that it is no longer necessary to store the intermediate quantities computed at each step in the forward pass. As a result, it is possible to train the ODE based model with constant memory cost as a function of depth [7].

As the memory cost associated to storing intermediate quantities of the forward pass quickly becomes prohibitively large it has been a limitation on the length of time series that deep models such as Recurrent Neural Networks can be trained on. As a result, using the adjoint method vastly broadens the scope of possible applications.

Adaptive Computation

The ODE solvers available today are able to change their evaluation strategy without interrupting a run to provide a chosen level of accuracy, i.e. they can readily trade precision for speed. Such ODE solvers can give reliable information about how approximation errors vary with other quantities. An example of a practical use might be to train a model and evaluate with high numerical precision and then lower the precision to achieve greater speed in a production version.

Easier to compute change of variables

Although not directly applicable to the Neural Rough Differential Equations we aim to consider, it is worth mentioning that it is easier to compute the change of variable formulas when the transformations are continuous as they are in the ODE model. A more detailed discussion can be found in [7, pg. 4]

Continuous time series models

The ability to deal with missing data and data points at arbitrary times is of great importance. Recurrent Neural Networks can not deal with irregularly sampled data without some preprocessing. Either the data must be discretised such that is becomes regularly sampled or some generative time-series model must be used to create a regular data sample. With continuously defined dynamics it is straightforward to incorporate irregularly sampled data.

On [7, pg. 6] a generative approach to time-series modelling that is continuous in time is presented. Each time series is represented as a trajectory beginning from a local initial state $\mathbf{z}_{t_0}$ and described by a set of hidden dynamics that are common to all of the time series. Now consider a chosen set of regularly spaced observation times $t_0, t_1, ...t_N$ that may or may not have an associated hidden state and an initial state $\mathbf{z}_{t_0}$. From this the ODE solver will produce a set of hidden states $\mathbf{z}_{t_1}, ..., \mathbf{z}_{t_N}$ corresponding to each observation time. This generative approach to modeling time series can be described by the following sampling procedure,

$$\mathbf{z}_{t_0} \sim p(\mathbf{z}_{t_0}) \tag{2.1.3}$$

$$\mathbf{z}_{t_1}, ..., \mathbf{z}_{t_N} = \text{ODESolve}(\mathbf{z}_{t_0}, f, \theta_f, t_0, ..., t_N) \tag{2.1.4}$$

$$\text{for each} \quad \mathbf{x}_{t_i} \sim p(\mathbf{x}|\mathbf{z}_{t_i}, \theta_{\mathbf{x}}) \tag{2.1.5}$$

and is visualised on the right hand side of figure 2.2. The function f maps the values of the hidden state to the gradient, $f(\mathbf{z}(t), \theta_f) = \frac{\partial \mathbf{z}(t)}{\partial t}$. In [7], the function f is parameterised by a neural network. A result of f being time invariant is that each latent time series is uniquely defined given an associated hidden $z(t)$. The unique definition of the latent time series allows for it to be extrapolated forwards and backwards in time to make predictions as required. A schematic representation that compares computations in an RNN encoder with that of the latent ODE model

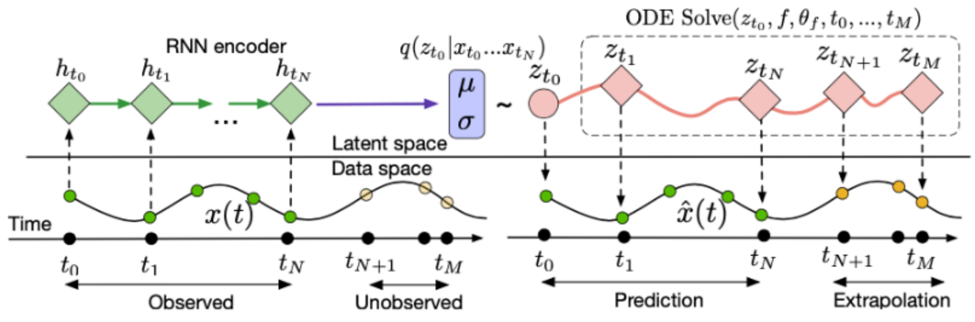


Figure 2.2: RNN encoder vs. generative latent ODE model [7, Fig 6, pg. 6]

is given in figure 2.2. Note how the Neural ODE model is able to update the hidden state at the times of the extrapolated observations (i.e. the predictions).

2.1.2 Adjoint Method for Memory Efficiency

For these continuous Neural ODE models, the equivalent of backpropagation through time (BPTT) described in chapter 1 would be to perform reverse mode differentiation through the operations of the ODE solver to compute the gradients of a loss function. Performing the reverse mode differentiation for each operation of the forward pass also requires storing the intermediate quantities and so incurs a prohibitively large memory cost that becomes unmanageable for longer time series as well as exacerbating numerical errors analogous to the vanishing/exploding gradients problem described above.

Originally developed by [12], the adjoint sensitivity method was introduced by [7] as a method to compute the gradients of a scalar valued loss with respect to all inputs of the black-box ODE solver without storing the intermediate quantities. A simple contrast between the BPTT approach and the adjoint method can be seen in figure 2.3. As the tools developed by [7] for Neural ODEs can also be used for Neural CDEs and Neural RDEs and are paramount to memory efficiency we will describe them here.

When using the adjoint sensitivity method, the gradients of a scalar value loss function are computed by solving another (augmented) ODE that describes the dynamics of the *adjoint*. This ODE can also be used with most ODE solvers. The memory requirements of the adjoint method are $\mathcal{O}(H)$, i.e. it varies linearly with the complexity of evaluating each step whereas for a BPTT approach it would be of $\mathcal{O}(HT)$, where H would be the memory cost of each computation and T is time. Furthermore the adjoint sensitivity method is not numerically unstable like the BPTT approach with its vanishing/exploding gradients. As the adjoint sensitivity method is also making use of an ODE solver it too can provide information on the level of accuracy, adapt its approach and can trade accuracy for speed [7, pg. 2].

In the adjoint sensitivity method the scalar valued loss function maps the output of the (augmented) ODE solver to a loss for a given hidden state $\mathbf{z}(t_1)$ [7, pg. 2].

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)) \tag{2.1.6}$$

To minimise L, we require gradients with respect to the dynamics parameters θ. To find gradients with respect to θ it is necessary to begin by determining how the loss L varies with the hidden state $\mathbf{z}(t)$ at any given time. This is the **adjoint** $a(t)$,

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{z}(t)}, \tag{2.1.7}$$

the dynamics of which are described by the following ODE,

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}. \tag{2.1.8}$$

The derivation of the above ODE relies on the chain rule applied to the gradient between hidden layers and the definition of the derivative applied to the adjoint. A detailed proof of equation (2.1.8) is given in [7, Appendix B, pg. 15-16].

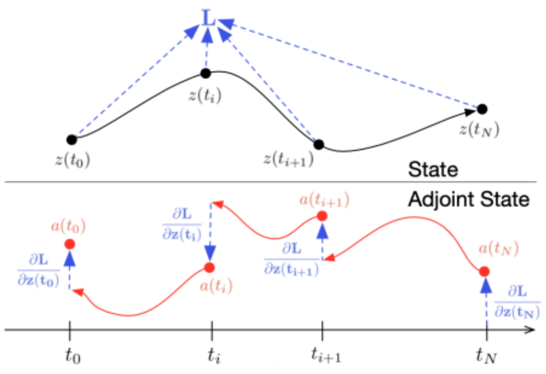

L
$z(t_i)$
$z(t_0)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial z(t_i)}$
$a(t_{i+1})$
$a(t_N)$
$\frac{\partial L}{\partial z(t_0)}$
$a(t_i)$
$\frac{\partial L}{\partial z(t_{i+1})}$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N

Figure 2.3: Using the adjoint sensitivity method to backpropagate through the ODE solver involves solving a modified ODE backwards in time. The new modified ODE 2.1.8 depends on both the underlying hidden state $\mathbf{z}(t)$ and the adjoint $\mathbf{a}(t)$. The value of adjoint state $\mathbf{a}(t)$ moves in the direction of the sensitivity of the loss L with respect to the underlying state $\mathbf{z}(t)$ at each observation time for which the loss function is affected. [7, Fig. 2, pg. 2]

In order to solve the above ODE, the value of $\mathbf{z}(t)$ must be fully known along each time series. This is achieved by recomputing $z(t)$ backwards in time together with the adjoint from its final value $\mathbf{z}(t_i)$ [7]. That is, by calling the ODE solver again and running it backwards from the final value of $\frac{\partial L}{\partial \mathbf{z}(t_1)}$ all the way back to the initial value $\frac{\partial L}{\partial(\mathbf{z}(t_0))}$. A schematic representation of the backpropagation of an ODE solution can be seen in figure 2.3.

A third integral that depends on the adjoint $\mathbf{a}(t)$ and the hidden state $\mathbf{z}(t)$ is then evaluated to determine the gradients with respect to the dynamics parameters θ.

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{2.1.9}$$

Computational differentiation can be used to evaluate the vector-Jacobian products $\mathbf{a}(t)^T \frac{\partial f}{\partial z}$ and $\mathbf{a}(t)^T \frac{\partial f}{\partial \theta}$ from equations (2.1.8) and (2.1.9) respectively. Doing so does not significantly add to the computational and memory cost associated to evaluating $f(\mathbf{z}(t), t, \theta)$. It then becomes possible to evaluate all the integrals required to solve for $\mathbf{z}$, $\mathbf{a}$ and $\frac{\partial L}{\partial \theta}$ with one call to an ODE solver. The ODE solver will then combine, the original hidden state $\mathbf{z}(t)$, the adjoint $\mathbf{a}$, and the partial derivatives from above into a single vector.

The below algorithm is taken from [7, Appendix C, Page 16] and demonstrates in pseudo code how the above described dynamics are constructed and subsequently the ODE solver is called to determine the gradients with respect to each parameter simultaneously.

Algorithm 2 Complete reverse-mode derivative of an ODE initial value problem

```
Input: dynamics parameters $\theta$, start time $t_0$, stop time $t_1$, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$
  $\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^{\intercal} f(\mathbf{z}(t_1), t_1, \theta)$        ▷ Compute gradient w.r.t. $t_1$
  $s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}, -\frac{\partial L}{\partial t_1}]$        ▷ Define initial augmented state
  def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot, \cdot], t, \theta$):        ▷ Define dynamics on augmented state
      return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^{\intercal} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^{\intercal} \frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^{\intercal} \frac{\partial f}{\partial t}]$        ▷ Compute vector-Jacobian products
  $[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$        ▷ Solve reverse-time ODE
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$        ▷ Return all gradients
```

Pitfalls of adjoint backpropagation

Applying the adjoint sensitivity method for Neural Network models is not completely without faults and as an example [13] highlights the following pitfalls that may be encountered:

Using the adjoint method may result in numerical instability for ReLu/non-ReLU activation functions and general convolutional operators.
Training may diverge because of the inconsistent gradients associated to small time step sizes that can appear as a result of the *optimise then discretise* approach.

Chapter 3

Neural CDEs

In [4], the Neural ODE model is extended by using a mathematical concept from Rough Path Theory called Controlled Differential Equations (CDEs) to capture sequential dynamics readily incorporate incoming data. [4] points out that as they capture sequential dynamics, the Neural CDE may be thought of as the continuous time analogue of the Recurrent Neural Networks.

In this chapter, we will discuss the key contributions of [4] to highlight how the sequential nature of data can be captured through the use of CDEs which are driven or **controlled** by a continuously updated approximation of an underlying process or data process called the path X.

3.1 Temporal Dynamics vs. Sequential Dynamics

In order to fully appreciate the significance of going from the Neural ODE model to the Neural CDE model, it is important to make note of the subtle difference between **temporal dynamics** and **sequential dynamics**. Temporal dynamics do indeed map variables to observation times or equivalent concepts such as the position of a non-spatial observable etc. but it does so in a static manner dependent only on intitial conditions. That is, the temporal dynamics do not develop in continuous time and do not capture the information inherent in the sequence of events. Sequential dynamics on the other hand are continuously updated as if the data points arrrived one by one on a conveyor belt and are dependent on the sequence of data that has come before it and its relation to each new observation. Again, we may think of the heuristic example of a ball thrown. The way you naturally watch and anticipate its trajectory is sequential modelling of sorts. In contrast temporal dynamics would contain only observations of its spatial coordinates at given observation times processed all at once after the ball had landed. For example, with only temporal dynamics, there would be no way of knowing whether the ball flew in a curved trajectory from A to B or was thrown vertically upwards at intermittent points in time. The difference seems very subtle but is extremely important.

As discussed above, Neural ordinary differential equations (Neural ODEs) are effective at modelling temporal dynamics and bridge the gap between traditional neural networks in machine learning and describing dynamical systems using differential equations. However, the Neural ODE models described in the above chapter are not able to incorporate the sequential nature of data, e.g., a time series because the solution of a given ODE depends only on its initial condition. Incoming information cannot be incorporated as there is no mechanism to adjust the hidden state without interruption. In a Neural ODE model, the observation times $\mathbf{t_i}$ are set to certain values which are passed to the ODE solver and thus become a static part of the model. Recapping from the previous chapter, Neural ODE models [7] approximate a map $x \mapsto y$ by learning a function f_θ and linear maps $\ell_\theta^1, \ell_\theta^2$ such that,

$$y \approx \ell_\theta^1(z_T), \quad \text{where} \quad z_t = z_0 + \int_0^t f_\theta(z_s)ds \qquad \text{and } z_0 = \ell_\theta^2(x). \tag{3.1.1}$$

Bear in mind, that in the Neural ODE model f_θ does not explicitly depend on time s but rather on the hidden state z_s which can include an extrapolated dependence on time as an extra (artificial) dimension. [4] aimed to orientate the added time t dimension with the natural ordering of the data by extending the $z_0 = \ell_\theta^2(x)$ condition in equation (3.1.1) to a condition along the lines of $\ell(x_0), ..., z_n = \ell(x_n)$ given some sequential data $(x_0, ..., x_n)$.

Obviously equation, (3.1.1) is the solution to an ODE so as mentioned above, once the parameters θ have been learnt, the initial condition at z_0 determines the solution or output of (3.1.1). In line with intuition, [14] rigorously establishes that it is not possible to incorporate new incoming data without interrupting a run and updating the initial conditions when using an ODE model.

[4] observed that the existing mathematics of Controlled Differential Equations (CDEs), a concept from, Rough Path Theory offered well developed tools for incorporating incoming information. Specifically, Controlled Differential Equations operate by driving differential equations with continuously updated rough signals [15] in the form of a path X that approximates the underlying process.

3.2 The Path, the CDE and the Underlying Process

Following the work of [15, 16] as presented in [4]: Let $\tau, T \in \mathbb{R}$ with $\tau < T$, and let, $v, w \in N$. Let $X : [\tau, T] \to \mathbb{R}^v$ be a continuous function of bounded variation. Let $\zeta \in \mathbb{R}^w$ and $f : \mathbb{R}^w \to \mathbb{R}^{w \times v}$ be continuous.

Then a continous path $z : [\tau, T] \to \mathbb{R}^w$ with sensitivity to time given by $\frac{dz}{dt}(t) = f(z(t), X(t))$ may be defined as the following Riemann-Stieltjes integral.

$$z_t = z_\tau + \int_\tau^T f(z_s)dX_s \quad \text{for } t \in (\tau, T] \text{ and} \quad z_\tau = \zeta. \tag{3.2.1}$$

The $f(z_s)dX_s$ on the RHS of equation (3.2.1) is a matrix-vector product since $f(z_s) \in \mathbb{R}^{w \times v}$ and $X_s \in \mathbb{R}^v$. Please note that the subscript s refers to evaluating the function X at s, a common notation in stochastic analysis.

According to [16, Theorem 1.3], equation (3.2.1) establishes global existence and uniqueness given a global Lipschitz condition on f. Equation (3.2.1) is the solution to a Controlled Differential Equation, $\frac{dz}{dt}(t) = f(z(t), X(t))$ which is controlled or driven by the continuously updated process X. In practise, X is interpolated between data points so as to be continuously updated.

3.3 Application of the Neural CDE model

Consider that we observe some time series $\mathbf{x} = ((t_0, x_0), (t_1, x_1), ..., (t_n, x_n))$ and that at each time t_i, we observe a value x_i which may or may not be missing. That is, the time series is fully observed but may be irregularly sampled. We then interpolate to produce the **path** $X : [t_0, t_n] \to \mathbb{R}^{v+1}$ such that $X_{t_i} = (x_i, t_i)$. Usually, the time series is assumed to be a discretisation of a continuous underlying process observed through $\mathbf{x}$. The path X is then an approximation for that process. [4]

To interpolate between the observables, X is taken to be a Natural Cubic Spline, which is a piece-wise cubic polynomial that is twice continuously differentiable. Using a piece-wise low-degree polynomial works to reduce oscillations and mitigate non-convergence when fitting a large number of data points [17]. According to [4], Natural Cubic splines have the minimum required regularity to handle certain edge cases but we will not discuss them here. Technical details may be found in [4, Appendix A].

Time as a channel & Initial value networks

Equipped with the approximation X of the underlying process, we seek to learn a map from the time series $\mathbf{x}$ to some object y by learning the functions ζ_θ, f_θ and a linear map l_θ. Note that the initial condition depends on the first element of the time series and z is modified continuously according to the following differential equation presented in [4, 18]

$$\text{(Initial condition)} \qquad z(0) = \zeta_\theta(t_0, x_0) \tag{3.3.1}$$

$$\text{(CDE)} \qquad \frac{dz}{dt}(t) = f_\theta(z(t))\frac{dX}{dt}(t), \tag{3.3.2}$$

$$\text{(Result)} \qquad y \approx \ell_\theta(z(T)) \tag{3.3.3}$$

$$\text{or} \tag{3.3.4}$$

$$y(t) \approx \ell_\theta(z(t)). \tag{3.3.5}$$

Bear in mind that the RHS of (3.3.2) is a matrix-vector product between $f_\theta(z(t))$ and $\frac{dX}{dt}(t)$. It follows that the hidden state z has a dependence on X because the local dynamics of the system have a dependence on X. To obtain our result or ouput y, we can project from a terminal time T if y is a fixed value and if $y(t)$ has time dependence of its own, we can continuously apply the projection. Projecting a time dependent $y(t)$ gives a way of producing sequence to sequence models, analogous to the one shown in figure 1.4.

Figure 3.1 gives a visual representation of how we begin with a fully observed (but perhaps irregularly sampled) time series or data process. At each time t_i, we observe a value x_i and interpolate between them to produce the path X wich in turn continuously affects the hidden state z. Please compare this to figure 2.3

which depicts how for a Neural ODE the hidden state evolves continuously only between observations but is discontinuously modified at each observation time. As the Neural CDE model can be thought of as a continuous time analogue of a Recurrent Neural Network (RNN), lets examine the connection between the Neural CDE model and the discrete time RNNs described in chapter 1 and how CDEs can be applied to neural network models.

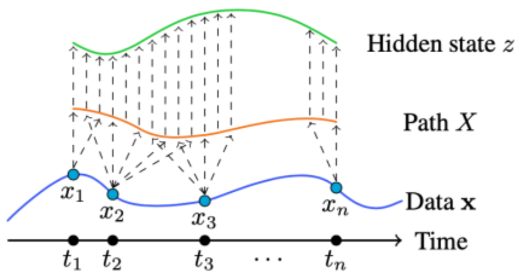


Figure 3.1: In the Neural CDE model the path X continuously drives the differential equations that updates the hidden state z. Therefore the evolution of the hidden state is continuous. [4, Fig. 1, pg. 3]

The functions to be learned in the above CDE model, $f_\theta : \mathbb{R}^w \to \mathbb{R}^{w\times(v+1)}$ and $\zeta_\theta : \mathbb{R}^{v+1} \to \mathbb{R}^w$ can be taken to be any neural network models with parameters θ. w may be taken to be a hyperparameter whose value controls the number of features of the hidden state for an RNN. Applying the above description of CDEs to a neural network yields the **Neural Controlled Differential Equation** model in which the hidden state z_t is given as the solution of a CDE,

$$z_t = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)dX_s \quad \text{for } t \in (t_0, t_n] \tag{3.3.6}$$

and the initial condition $z_{t_0} = \zeta_\theta(x_0, t_0)$ as before [4]. Just as for conventional RNN models, the output or hidden state of the model can be either a single value z_{t_n} at a terminal time t_n or be a time dependent process z itself. The final results or predictions are then typically given by linear maps applied to these hidden states as seen in equation (3.3.5)

Note the similarity between equation (3.3.6) for Neural Controlled Differential Equations and equation (3.1.1) for Neural Ordinary Differential Equations. Indeed, equation (3.3.6) is also an ODE in some sense and so can be solved with the same tools that were developed for Neural ODEs [4, 18]. The main difference being that equation (3.3.6) is driven or "controlled" by the continosly updated path X whereas the Neural ODE in equation (3.3.6) is not driven at all, i.e. the identity function $\iota : \mathbb{R} \to \mathbb{R}$ takes the place of the process X. This is what allows Neural CDEs to continuously incorporate new data, as changes in X change the local dynamics of the system and so also the hidden state [4].

CDEs as Universal Approximation

It has been established that CDEs represent general functions on streams [19, Proposition A.6], [20, Theorem 4.2]. Here "Streams" are simply sequences of data elements that become available over time and can be thought of as observables arriving one by one rather than being processed in larger batches of existing data. [4, Theorem B.14, Appendix B] states and proves a Universal Approximation Theorem that Neural CDEs are able to approximate any functional sequence which they summarise in the following informal theorem [4, Pg. 3]

Theorem 3.3.1 (Informal)**.** *The action of a linear map on the terminal value of a Neural CDE is a universal approximator from* $\{$*sequences in* $\mathbb{R}^v\}$ *to* $\mathbb{R}$*.*

3.3.1 Solving the Neural CDE

[4] formulate their problem in such a way that X is not only of bounded variation but also (twice) differentiable making it possible to write

$$\frac{dz}{ds}(s) = f_\theta(z(t))\frac{dX}{ds}(s), \quad \text{to define,} \quad g_{\theta,X}(z,s) = f_\theta(z)\frac{dX}{ds}(s), \tag{3.3.7}$$

so for $t \in (t_0, t_n]$ it is possible to use the derivative of the control path to write the CDE in equation (3.3.6) in the form of an ODE,

$$z_t = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)dX_s = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)\frac{dX_s}{ds}(s) = z_{t_0} + \int_{t_0}^{t} g_{\theta,X}(z_s,s)ds \tag{3.3.8}$$

Because, this takes the form of an ODE, [4] are able to solve it with excactly the same tools as developed for Neural ODEs by [7] even using the same software and code, namely the *torchdiffeq* package [21].

3.3.2 Generalises alternative ODE models

[4] points out that for those unfamiliar with CDEs, it could be tempting to replace $g_{\theta,X}(z,s)$ with an ODE type model of the form,

$$h_\theta(z, X_s) = \frac{dz}{ds}(s) = f_\theta(z(s), X(s)), \tag{3.3.9}$$

in which $h_\theta(z, X_s)$ is directly applied to and is possibly nonlinear in X_s. [4, Thm. C.1, App. C] states and proves that the Neural Control Differentiation model is strictly more general than such models which is summarised in an informal statement on [4, Pg. 4].

Theorem 3.3.2 (Informal)**.** *Any equation of the form,*

$$z_t = z_0 + \int_{t_0}^{t} h_\theta(z_s, X_s)ds, \tag{3.3.10}$$

may be represented excactly by a Neural CDE of the form,

$$z_t = z_0 + \int_{t_0}^{t} f_\theta(z_s)dX_s. \tag{3.3.11}$$

However the converse statement is not true

The gist of the theorem is that a Neural CDE can readily represent the identity function between paths but the alternative ODE models cannot. In the experiments of [4], Neural CDE models measurably outperformed a selection of ODE type models, which according to [4] might be a consequence of this generality.

3.3.3 Training CDEs via the adjoint method

In the previous chapter on Neural ODEs, we describe how Neural ODE models are able to train using the memory efficient adjoint sensitivity method for backpropagation (see figure 2.1.8) [7]. The advantage of the adjoint method is that if H is the memory cost of evaluating one step of the model and $T = t_n - t_0$ is the time horizon, the memory cost of adjoint backpropagation is still only $\mathcal{O}(H)$ whilst typical backpropagation uses $\mathcal{O}(HT)$ memory.

In an effort to apply Neural ODE models to time series, previous work such as [22] has interrupted the ODE solver at each observation to update the initial conditions to accommodate the new observation. Interrupting at each observation requires a memory of $\mathcal{O}(H)$ for each time step time, for a total memory cost of $\mathcal{O}(HT)$. Because the adjoint method of backpropagation cannot be used across the discontinuity at the observation it cannot be used for this setting.

This is yet another way in which the Neural CDEs ability to naturally adapt to incoming data proves advantageous. The function,

$$g_{\theta,X}(z,s) = f_\theta(z)\frac{dX}{ds}(s), \tag{3.3.12}$$

incorporates new data continuously through the process X without having to interrupt the ODE solver. Therefore it is possible to use the adjoint method of backpropagation just as for the uninterrupted Neural ODE models.

As mentioned above the adjoint method of backpropagation in the Neural ODE model has a memory cost of $\mathcal{O}(H)$ where H is the complexity of each step. Because the Neural CDE is continuously updating through the time horizon of the time series, it incurs an additional memory cost of $\mathcal{O}(T)$. The combined memory cost of training in the Neural CDE model is therefore only $\mathcal{O}(H + T)$ as opposed to the typical $\mathcal{O}(HT)$ in alternative models such as Recurrent Neural Networks or ODE models that interrupt the solver to incorporate new data.

3.3.4 Partially observed data

The first step in a Neural CDE model is to interpolate between potentially irregularly sampled and/or partially observed data $\mathbf{x}$ to construct $X : [t_0, t_n] \to \mathbb{R}^{v+1}$, such that, $X_{t_i} = (x_i, t_i)$. As a result the model is agnostic to whether the data is irregularly sampled and/or partially observed. Furthermore the Neural CDE model readily handles multivariate data where observations from the different dimensions are missing at the same observation time.[18]

3.3.5 Batching

One of the challenges posed by irregularly sampled time series/data is how to split the data into batches when the observation times t_i may not line up between the batches. Within the framework of the CDE model it is possible to process the whole data set prior to feeding into the model to produce the continuous-time interpolations (paths) X [18]. As described in section 3.3, we can construct a continuous path X from an irregularly sampled and or partially observed time series $\mathbf{x} = ((t_0, x_0), (t_1, x_1), ..., (t_n, x_n))$ by interpolating between the $\mathbf{x} = (x_0, ..., x_n)$ such that $X_{t_i} = (x_i, t_i)$. Since X is already an interpolation for irregularly sampled and or partially observed time series, it is possible to construct a path X for each batch of training samples and an invidual path so constructed can be thought of as representing each individual batch.

3.4 Experimental Results

We will not include a detailed discussion of the experiements carried out in [4] as Neural CDEs are not the main focus of this paper but rather stepping stone towards understanding Neural RDEs. However we highlight their results on a CharacterTrajectories dataset mainly to highlight how memory efficient the model is compared to other sequential models and how powerful it is in dealing with missing data.

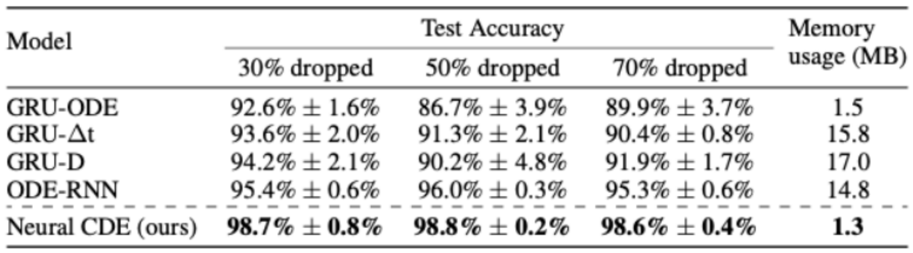
[4] compare their Neural CDE model to a few existing models so chosen as to be representative of the class of ODE and RNN based models to which the Neural CDE model belongs. [4] determined hyperparameters by performing a grid search to optimise the performance of the benchmark ODE-RNN model and then used equivalent hyperparameters when running the other models. Further results and precise experimental details may be found in [4, Chapter 4 and Appendix D]

3.4.1 Character Trajectories

To illustrate the usefulness of Neural CDEs for acting on irregularly sampled time series, [4] process the 'CharacterTrajectories' dataset from the UEA time series classification archive [23]. This data set captures the human hand writing letters of the latin alphabet in a single stroke and consists of 2858 time series each of length 182 that record the x, y position and pen tip force. The task is to figure out which of twenty different letters is represented by the time series.

To see the Neural CDEs model effectiveness in dealing with increasingly irregularly sample data [4] run three experiments, in which they (uniformly) randomly drop 30%, 50% and 70% of the observations. To keep the randomly reduced datasets irregularly sampled but fully observed, observations at a given time are removed across channels (x, y position and pen tip force). All of the models and all of the runs use the same randomly reduced datasets.

The results can be seen in figure 3.2. The Neural CDE model outperforms the other models chosen from its class. Note, that as the percentage of missing observations increases, the performance of the Neural CDE model is remarkably



Model	Test Accuracy			Memory usage (MB)
	30% dropped	50% dropped	70% dropped	
GRU-ODE	$92.6\% \pm 1.6\%$	$86.7\% \pm 3.9\%$	$89.9\% \pm 3.7\%$	1.5
GRU-Δt	$93.6\% \pm 2.0\%$	$91.3\% \pm 2.1\%$	$90.4\% \pm 0.8\%$	15.8
GRU-D	$94.2\% \pm 2.1\%$	$90.2\% \pm 4.8\%$	$91.9\% \pm 1.7\%$	17.0
ODE-RNN	$95.4\% \pm 0.6\%$	$96.0\% \pm 0.3\%$	$95.3\% \pm 0.6\%$	14.8
Neural CDE (ours)	**$98.7\% \pm 0.8\%$**	**$98.8\% \pm 0.2\%$**	**$98.6\% \pm 0.4\%$**	**1.3**

Figure 3.2: Test accuracy (mean $\pm$ std, computed across five runs) and memory usage on CharacterTrajectories. Memory usage is independent of repeats and of amount of data dropped [4, Table 1:, pg. 6]

constant compared to the other models, highlighting the strong adaptiveness to missing data built into the formulation of the Neural CDE model. Furthermore, the memory cost of the Neural CDE model is an order of magnitude less than for the other models thanks to the adjoint method for backpropagation.

Last but not least, note that the table in figure 3.2 does not show the time taken to train. As the path X is driven by a sequence of observations from a time series and the hidden state z_t is the solution of a CDE dependant on X, a longer time series will incur a large number of forward computations that result in long training times and reduced accuracy despite the memory efficiency. The next chapter contains an example of such training times and accuracy. The large number of forward computations for a longer time series required by the Neural CDE model is what motivates the development of the Neural Rough Differential Equation model (RDE) the key consideration of this paper.

Chapter 4

Rough Path Theory & the Path Signature

In [15, 16, 24], it is detailed how the dynamics of Controlled Differential Equations are described by a mathematical concept called the **signature transform** or **path signature** or simply **the signature** of the control process or path X. The signature is a collection of real valued integrals that summarise a stream of data. The collection of integrals form a series where each additional term corresponds to a higher order approximation.

In [15, 16, 25] it is established that a CDE can be solved by using a compressed version of the signature called the **log-signature** of the control path X. Each log-signature summarises the path X only over a short time interval. A **Rough Differential Equation** is a CDE where the path X is summarised with a log-signature and it is solved with a numerical method called the **log-ODE** method. This effectively reduces the length of the time series as the input signal is summarised for a chosen length of the time intervals unlike in the Neural CDE model that requires pointwise evaluation. The size of the time interval chosen becomes another way in which accuracy may be traded for speed. Figure 4.1 provides a simplified visual contrast between driving a differential equation with a rough path (left) and the Neural CDE model from the previous chapter (right).

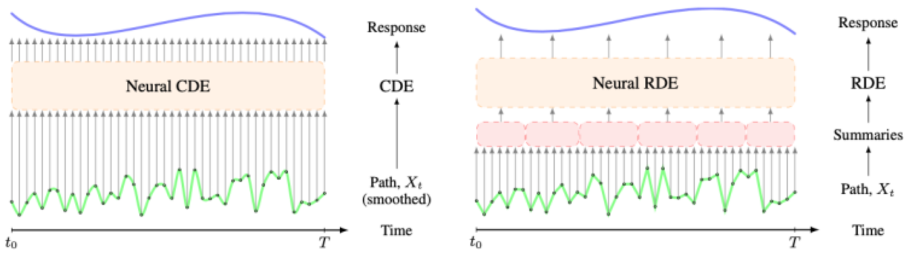


Figure 4.1: A simplified contrast between the pointwise Neural CDE model and Rough Path solution theory: **Left:** The path is interpolated to be smooth (differantiable) and pointwise evaluation is used to drive the CDE. **Right:** Rough path solution theory where the path X is summarised with a (log-)signature thus effectively reducing the number of evaluations required or shortening the time series [1, Figure 1, pg. 2]

A key consideration of the work on Neural Rough Differential Equations in [1] is how to solve the problem of too many forward operations in the Neural CDE model by applying Rough Path Theory and the log-Signature. We therefore discuss Rough Path Theory and the log-ODE method here. Our discussion of Rough Path Theory follows the outlines of how it is presented in [1, Section 2 & Appendix A] as it contains a concise summary of the key concepts from rough path theory that underpin their Neural RDE model. Key definitions are restated. This in turn relies on material found in the following books on Rough Path Theory [25, Sections 7,10], [24, Chapters 1-4 & 6-8], [16], and [1, Appendix A]. We would recommend [24] as an introduction to the subject of Rough Path Theory or for a more concise introduction, [16] by Terry Lyons who introduced the topic of driving differential equations by rough signals in [15].

4.1 Log-ODE method to approximate CDEs

In this section, we give a breakdown of the log-ODE method as presented in [1]. The log-ODE method is a numerical method constructed of various concepts from stochastic analysis and rough path theory [16][25]. For the log-ODE method, a CDE is converted to an ODE format using a concept called the log-signature to summarise the control path. As a result the control path no longer has to be differentiable. This is in contrast to the method from the previous chapter on Neural CDEs that uses the derivative of the control path $\dot{X}$ to convert the CDE to an ODE. The log-ODE method is then used to give a high order approximation of the solution of that CDE by the solution to an ODE.

The log-ODE method is applied by [1] to approximate Controlled Differential Equations (CDEs) of the form:

$$dY_t = f(Y_t)dX_t, \quad Y_0 = \epsilon, \tag{4.1.1}$$

where the following equality holds for all $t \in [0, T]$,

$$Y_t = \epsilon + \int_0^t f(Y_s)dX_s. \tag{4.1.2}$$

The path $X : [0, T] \to \mathbb{R}^d$ is of finite length and bounded variation. The initial condition $\epsilon \in \mathbb{R}^n$ is n-dimensional and the function $f : \mathbb{R}^n \to L(\mathbb{R}^d, \mathbb{R}^n)$ is assumed sufficiently smooth for the CDE in equation (4.1.1) to be well formulated. Using **signatures** or their compressed format **log-signatures** to drive differential equations, the solutions to corresponding CDEs can be accurately approximated with the *log-ODE method*. In the following text, we aim to construct an understanding how this rough path solution theory is constructed.

$L(U, V)$ is a continuous linear map between the vector spaces U and V such that if the path X_t maps $X : [0, T] \to U$, and the output Y_t maps $Y : [0, T] \to V$. In the above case f is called the **vector field** of the CDE and assumed to have $Lip(\gamma)$ regularity [25, definition 10.2]. (In a Neural model, f becomes the Neural Network). [1] make one of the following assumptions on the vector field:

- f is bounded and has N bounded derivatives

- f is linear

Following the presentation of key concepts found in [1] we begin by defining the tensor algebra, a pre-requisite for the signature.

Definition 4.1.1 (Tensor Algebra)**.** $T(\mathbb{R}^d) := \mathbb{R} \oplus \mathbb{R}^d \oplus \mathbb{R}^{\otimes 2} \oplus ...$ is said to be the tensor algebra of $\mathbb{R}^d$ and $T((\mathbb{R}^d)) := \{\mathbf{a} = (a_0, a_1, ...) : a_k \in (R^d)^{\otimes k}\ \forall k \geq 0\}$ is the set of a formal series of tensors of $\mathbb{R}^d$. Furthermore, $T(\mathbb{R}^d)$ and $T((\mathbb{R}^d))$ can be equipped with the operations of addition and multiplication, If $\mathbf{a} = (a_0, a_1, ...)$ and $\mathbf{b} = (b_0, b_1, ...)$, then,

$$\mathbf{a} + \mathbf{b} = (a_0 + b_0, a_1 + b_1, ...), \tag{4.1.3}$$

$$\mathbf{a} \otimes \mathbf{b} = (c_0, c_1, c_2, ...), \tag{4.1.4}$$

in which the n-th term $c_n \in (\mathbb{R}^d)^{\otimes n}$ is given by,

$$c_n := \sum_{k=0}^{n} a_k \otimes b_{n-k} \quad \text{for } n \geq 0. \tag{4.1.5}$$

In equation 4.1.5, $\otimes$ represents the usual tensor product. The $\otimes$ symbol in equation 4.1.4 is also called a ”tensor product” but the equation is a generalisation. If all but one a_i and one b_i are equal to zero, then 4.1.4 becomes the usual tensor product.

Definition 4.1.2 (Signature Transform). If the path $X = (X^1, ..., X^d) : [0, T] \to \mathbb{R}^d$ is continuous and piecewise differentiable then the signature of a finite length path $X : [0, T] \to \mathbb{R}^d$ over the interval $[s, t]$ is defined as the following collection of iterated (Riemann-Stieltjes) integrals:

$$S_{s,t}(X) : \left(1, x_{s,t}^{(1)}, x_{s,t}^{(2)}, x_{s,t}^{(3)}...\right) \in T((\mathbb{R}^d)), \tag{4.1.6}$$

where for $n \geq 1$,

$$X_{s,t}^{(n)} := \int \cdots \int_{s<u_1<...<u_n<t} dX_{u_1} \otimes ... \otimes dX_{u_n} \in (\mathbb{R}^d)^{\otimes n}. \tag{4.1.7}$$

It is possible to truncate this signature to define the depth-N signature of the path X on $[s, t]$ as,

$$S_{s,t}^N(X) := \left(1, x_{s,t}^{(1)}, x_{s,t}^{(2)}, x_{s,t}^{(3)}...x_{s,t}^{(N)}\right) \in T^N((\mathbb{R}^d)), \tag{4.1.8}$$

where $T^N(\mathbb{R}^d) := \mathbb{R} \oplus \mathbb{R}^d \oplus \mathbb{R}^{\otimes 2} \oplus ... \oplus (\mathbb{R}^d)^{\otimes N}$ is an N-truncated tensor algebra.

A simplified version of the definition of a depth-N signature is presented in [1] as follows: If $X = (X^1, ..., X^d) : [0, T] \to \mathbb{R}^d$ is continuous and piecewise differentiable as before and,

$$S_{a,b}^{i_1,...,k}(X) = \int \cdots \int_{a<t_1<...<t_k<b} \prod_{j=1}^{k} \frac{dX^{i_j}}{dt}(t_j) dt_j, \tag{4.1.9}$$

then the (simplified) depth-N signature transform of the path X can be written as

$$\mathrm{Sig}^N_{a,b}(X) = \left(\{S^i_{a,b}(X)^{(i)}\}^d_{i=1}, \{S^i_{a,b}(X)\}^d_{i,j=1}, ...\{S^i_{a,b}(X)\}^d_{i_1,\dots,i_N=1}\right) \tag{4.1.10}$$

The simplified definition is independent of the choice of T and t_i due to the change of variables that is possible to make in equation (4.1.9).

The depth-N signature gives an ordered set of statistics that summarise how the path X effects the dynamics of a system modelled by a controlled differential equation. The same is also true of the more compact log-signature.

In [26] the depth-N signature is extended under mild conditions to show that $Sig^\infty(X)$ completely describes the path X up to translation, provided time is included as a channel in the path [1]. What this means here is that adding more terms to the (log-Signature) alway equates to characterising the path in more detail, i.e. higher order terms include more substep information.

Log-signature The log-signature can be thought of as a compressed format of the path signature containing the same information. It is possible to remove certain algebraic redundancies from the signature to derive the log-signature. Using integration by parts and some algebraic manipulation gives that,

$$\int_0^t \int_0^s dX^i_u dX^j_s + \int_0^t \int_0^s dX^j_u dX^i_s = X^i_t X^j_t \quad \text{for } i,j \in \{1,\dots,d\}, \tag{4.1.11}$$

or in simplified form,

$$S^{1,2}_{a,b}(X) + S^{2,1}_{a,b}(X) = S^1_{a,b}(X) S^2_{a,b}(X). \tag{4.1.12}$$

Therefore, in the above equation any one of the four quantities can be obtained by knowing the other three and so it is possible to remove quantities whilst retaining the full information contained in the signature.

We then compute the signature transform (up to depth-N), discard redundant terms and obtain some (non-unique) minimal collection to produce the log-signature transform. Lets examine how we may fix such set of redundancies (somewhat corresponding to a choice of basis) to define a log-signature transform $\mathrm{LogSig}_{s,t}(X)$. This is formally constructed through the following three definitions which begins with the logarithm map on the depth-N truncated tensor algebra $T^N(\mathbb{R}^d) := \mathbb{R} \otimes \mathbb{R}^d \otimes \cdots \otimes (\mathbb{R})^{\otimes N}$.

Definition 4.1.3 (The logarithm of a formal series). For $\mathbf{a} = (a_0, a_1, ...) \in T((\mathbb{R}^d))$ with $a_0 > 0$, define $\log(\mathbf{a})$ to be the element of $T((\mathbb{R}^d))$ given by the following series:

$$\log(\mathbf{a}) := \log(a_0) + \sum_{n=1}^{\infty} \frac{(-1)^n}{n} \left(\mathbf{1} - \frac{\mathbf{a}}{a_0}\right)^{\otimes n}, \tag{4.1.13}$$

where $\mathbf{1} = (1, 0, \dots)$ is the unit element of $T((\mathbb{R}^d))$ and $\log(a_0)$ is viewed as $\log(a_0)\mathbf{1}$.

Definition 4.1.4 (The logarithm of a truncated series). For $\mathbf{a} = (a_0, a_1 \dots a_N) \in T((\mathbb{R}^d))$ with $a_0 > 0$, define $\log^N(\mathbf{a})$ to be the element of $T^N(\mathbb{R}^d)$ defined from the logarithm map (4.1.13) as

$$\log^N(\mathbf{a}) := P_N(\log(\tilde{\mathbf{a}})), \tag{4.1.14}$$

where $\tilde{\mathbf{a}} := (a_0, a_1, \dots, a_N, 0, \dots) \in T((\mathbb{R}^d))$ and P_N denotes the standard projection map from $T((\mathbb{R}^d))$ onto $T^N(\mathbb{R}^d)$.

Definition 4.1.5 (The log-signature). The log-signature of finite length path $X : [0, T] \to \mathbb{R}^d$ over the interval [s,t] is defined as,

$$\text{LogSig}_{s,t}(X) := \log(S_{s,t}(X)), \tag{4.1.15}$$

where $\log(S_{s,t}(X))$ denotes the path signature of X as given by definition 4.1.2. Similarly to the truncated path signature the depth-N log-signature of the path X is defined as,

$$\text{LogSig}^N_{s,t}(X) := \log^N(S^N_{s,t}(X)) \quad \text{for each } N \geq 1. \tag{4.1.16}$$

In an effort to simplify the definition of the log-ODE method, each log-signature, $\text{LogSig}_{s,t}(X) := \log(S_{s,t}(X))$ is taken to be an element of the truncated tensor algebra $T^N(\mathbb{R}^d)$. This is equivalent to the definition that we will use in the next chapter that defines the log-signature as a map from the finite length path $X : [0, T] \to \mathbb{R}^d$ to $\mathbb{R}^{\beta(d,N)}$ where $\beta(d, N)$ is the dimension of the log-signature and is given by,

$$\beta(d, N) = \sum_{k=1}^{N} \frac{1}{k} \sum_{i|k} \mu\left(\frac{k}{i}\right) d^i, \tag{4.1.17}$$

where μ is the Möbius function. From this we can see that the number of log-signature channels, $\beta(d, N)$ grows exponentially with the number of input channels d and so the log-ODE clearly becomes unusable for a large number of input channels. It is essentially by this mapping that the log-signature trades the length of the input sequence for an increased number of input channels.

Terms of the (log-)Signature

The depth-N log-signature is composed of N terms, each corresponding to a certain level of depth. The first two levels of the log-signature have intuitive geometric interpretations that are shown in figure 4.2. The depth 1 terms can be thought of as a length of sorts, measuring coordinate changes in each channel (variable) over a given interval, shown as ΔX_1 and ΔX_2 in figure 4.2. The depth 2 terms correspond to the signed area called **Lévy area** above A_+ and below A_- the straight line joining the end points of the path over the interval shown in figure 4.2. In other words the log signature contains summarised information about the path over the interval, e.g. for a stock price this might be thought of as quantifying the price change and a cumulative volatility of sorts (not volatility as normally defined though).

Deeper levels ($N > 2$) are computed using a correspondingly larger collection of integrals that are iterated areas in higher dimensional space (not volume etc.) and contain increasingly detailed information about the path X but are perhaps less intutitive to visualise. Depth-2 and depth-3 already characterise the path X to a sufficient degree for many practical purposes. Even deeper levels bring with them an increasingly higher computational cost that can be dealt with by parallelising.

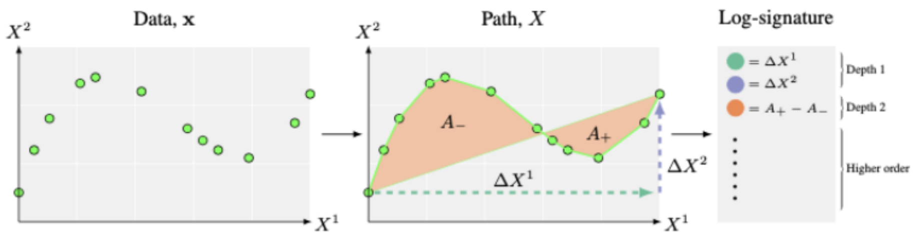


Figure 4.2: Visualisation of the depth-2 terms of a log-signature for a two-dimensional path. Depth-1 terms are a length of sorts corresponding to directional displacement in that channel, ΔX_i. The depth-2 term(s) correspond to a *Lévy area*, i.e, it is the signed area between the path X and a straight line joining its endpoints on the interval [1, Figure 2, pg. 3]

Comparing the Taylor expansion of a CDE to the signature

Recall how $Z : [a, b] \to \mathbb{R}^w$ is the unique solution to the controlled differential equation,

$$Z_a = \epsilon, \quad Z_t = Z_a + \int_a^t f(Z_s) dX_s \quad \text{for } t \in (a, b], \tag{4.1.18}$$

where $f(z_s)dX_s$ is a matrix vector product and dX_s is a Riemann-Stieltjes integral. Expanding the CDE as a first order Taylor Expansion, gives the below equations as derived in [1, Page. 4].

$$\begin{aligned}
Z_t &= Z_a + \int_a^t f(Z_s) dX_s && (4.1.19)\\
&\approx Z_a + \int_a^t \big(f(Z_a) + D_f(Z_a)(Z_s - Z_a)\big)\frac{dX}{dt}(s)ds && (4.1.20)\\
&= Z_a \int_a^t f(Z_a)\frac{dX}{dt}(s)ds + \int_a^t \left(D_f(Z_a)\int_a^s f(Z_u)\frac{dX}{dt}(u)du\right)\frac{dX}{dt}(s)ds && (4.1.21)\\
&\approx Z_a + f(Z_a)\int_a^t \frac{dX}{dt}(s)ds + D_f(Z_a)f(Z_a)\int_a^t\int_a^s \frac{dX}{dt}(u)du\frac{dX}{dt}(s)ds && (4.1.22)\\
&= Z_a + f(Z_a)\{S(X)^{(i)}\}_{i=1}^d + D_f(Z_a)f(Z_a)\{S(X)^{(i,j)}\}_{i,j=1}^d, && (4.1.23)
\end{aligned}$$

where D_f is the Jacobian of the vector field (and soon to be Neural Network) f. Comparing the first and last equations shows that each term of the signature corresponds to a term in the Taylor expansion of the CDE. This establishes they way in which signatures are connected to the solutions of CDEs and verifies that signatures may be used to approximate the solution to CDEs. Including higher order terms in the Taylor expansion would call for a correspondingly greater level of depth of signature terms [27]. A formal definition of the Taylor method to any order is as follows,

Definition 4.1.6. (The Taylor method) Given, a CDE of the form $dY_t = f(Y_t)dX_t,\ Y_0 = \epsilon$, the path signature of X can be used to approximate the solution Y on an interval $[s, t]$ via its Truncated Taylor expansion,

$$\text{Taylor}(Y_s, f, S^N_{s,t}(X)) := \sum_{k=0}^{N} f^k(Y_s)\pi_k(S^N_{s,t}(X)). \tag{4.1.24}$$

The above Taylor expansion is an approximation for Y_t where each $\pi_k : T^N(\mathbb{R}^d) \to (\mathbb{R}^d)^{\otimes k}$ is the projection map (e.g. a Jacobian) onto $(\mathbb{R}^d)^{\otimes k}$.

Log-ODE method as an ordered approximation

We have seen how the terms in a Taylor expansion of a CDE are equivalent to the terms in a signature. As the log-signature is simply a compressed format of the signature it follows that the terms of the log-signature can used to approximate a CDE.

Building on the formal definition for the Taylor expansion above, the *Log-ODE method* may then be defined

Definition 4.1.7 (The Log-ODE method). Begin by defining a function,

$$\widehat{f} : \mathbb{R}^n \to L(T^N(\mathbb{R}^d), \mathbb{R}^n) \quad \text{as} \quad \widehat{f}(z) : \text{Taylor}(z, f, \cdot). \tag{4.1.25}$$

By applying $\widehat{f}$ to the truncated log-signature of the path X over an interval $[s, t]$, [1] could define the following ODE on $[0, 1]$,

$$z(0) = Y_s, \quad \frac{dz}{du} = \widehat{f}(z)\text{LogSig}^N_{s,t}(X), \tag{4.1.26}$$

It follows that if Y_s and $\text{LogSig}^N_{s,t}$ are known, the ordered log-ODE approximation of Y_t can be defined as,

$$\text{LogODE}(Y_s, f, \text{LogSig}^N_{s,t}(X)) := z(1). \tag{4.1.27}$$

The ODE is then solved on $[0, 1]$ to given an approximation to the output, $Y_t := z(1)$ [1, Appendix A] and [16, Section 2].

Log-ODE method for the Neural RDE model

[1] found that the log-ODE method was especially useful for long time series because the log-ODE method is able to trade the length of the input sequence for an increased number of channels. The log-ODE method facilitates the training of the model by allowing the black box ODE-solver to take integration steps larger than the discretisation of the data but allowing us to recover a chosen amount from the substep information with higher order approximations.

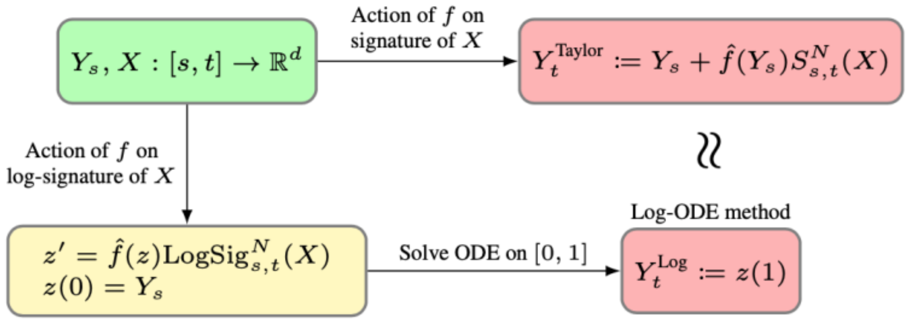

$Y_s, X : [s,t] \to \mathbb{R}^d$
Action of f on signature of X
$Y_t^{\text{Taylor}} := Y_s + \hat{f}(Y_s) S^N_{s,t}(X)$
$\wr\wr$
Action of f on log-signature of X
Log-ODE method
$z' = \hat{f}(z)\text{LogSig}^N_{s,t}(X)$
$z(0) = Y_s$
Solve ODE on $[0, 1]$
$Y_t^{\text{Log}} := z(1)$

Figure 4.3: A comparison between applying f on the signature of X and using the Talyor method and applying f to the log-signature of X and solving the ODE on [0,1] [1, Figure 6, Appendix A]

Reverting back to the form of the CDE as defined in equation (4.1.18) above, the log-ODE method implies that Z_b can be approximated with $Z_b \approx \widehat{Z}_b$ such that,

$$\widehat{Z}_u = \widehat{Z}_a + \int_a^u \widehat{f}(\widehat{Z}_s) \frac{\text{LogSig}^N_{a,b}(X)}{b-a} ds, \quad \text{for } u \in (a,b], \tag{4.1.28}$$

and $\widehat{Z}_a = Z_a$ [1]. What this means is that the log-ODE method can provide a high order approximation of a CDE by the solution to an ODE. As mentioned before, obtaining an ODE format is important as a wealth of previously existing methods to handle ODEs become available.

In [1], the interval $[a,b]$ is partitioned into smaller more manageable intervals such that $a = r_0 < r_1 < \cdots < r_m = b$. The solution to the CDE in equation 4.1.18 is then split into an integral over each $[r_i, r_{i+1}]$ in order to apply the log-ODE method to each interval seperately and the results added up. In [1] a CDE whose solution is approximated using the above log-ODE method is called a **Rough Differential Equation**. Hence the name Neural Rough Differential Equations when extended to Neural Network machine learning.

Chapter 5

Neural Rough Differential Equations

In [1], the rough path theory described in chapter 4 is used to extend the Neural CDE model to introduce their Neural Rough Differential Equation (RDE) model. The central contribution of [1] is to extend the Neural CDE model of [4] by effectively reducing the length of the time series by summarising the path X_t over appropriately chosen intervals using the log-signature. As a result the hidden sate Z_t is only updated once for each interval $[t_i, t_i + 1]$ rather than at each data point.

As described in chapter 3, previously established methods for computing the forward pass of a Neural CDE, e.g. [4] summarised the possibly irregular input signal as an interpolated path $X : [t_0, t_n] \to \mathbb{R}^{v+1}$. [1] uses rough path theory to extend the CDE model to represent the input signal over small partitions of the time interval through its log-signature, which is composed of real valued terms that approximate how the input signal drives the CDE. This approach then uses the log-ODE method for solving Rough Differential Equations as described in the previous chapter. This approach is termed Neural Rough Differential Equations by [1].

In the Neural CDE model, the path X is a pointwise interpolation between the observations and the hidden state is the solution of a CDE dependant on X. Due to these pointwise evaluations, a longer time series will incur a prohibitive number of forward computations resulting in long training times and reduced accuracy. By generalising the Neural CDE model to work with a broader class of driving signals, [1] is able to mitigate these performance issues by summarising the path X with its log-signature over appropriately chosen time intervals. This effectively reduces the length of the time series and thus reduces the number of integration steps required to solve the CDE. This allows longer time series to be handled effectively and [1] were able to show efficacy for time series with up to 17,000 observations. [1] also observed considerably reduced training times, increased accuracy and lowered memory costs.

Discussing the work found in *Neural Rough Differential Equations for Long Time Series*[1], building the pre-requisite understanding of its key concepts and applying their Neural RDE model and finding a potential use for the model in a financial setting is the key objective of this thesis.

5.1 Pointwise evaluation of the CDE vs. log-signature & RDE

From the field of Rough Path Theory [15],[16] [25], it was already known that it is possible to summarise the path X with the log-signature to drive the CDE and then solve the CDE via the log-ODE method. Indeed, this solution theory for CDEs was developed in the 90's by one of the authors of [1] and [4], Terry Lyons. The rough path solution theory for CDEs is in contrast to other methods that require a pointwise evaluation of the control path such as in the Neural CDE model. A CDE driven using the rough path solution theory is thus a rough differential equation, and the log-ODE method is used to solve it numerically. Figure 4.1 on page 32 provided a simplified visual representation of the RDE (log-ODE method) vs CDE (pointwise)

5.2 Neural Rough Differential Equation

Now that we have built the required understanding of the Rough Path solution theory for CDEs and discussed its relevance to solving CDEs, we discuss the method developed by [1] to apply Rough Differential Equations with a neural network.

By piecewise linear interpolation, we have constructed an approximation, $X : [t_0, t_n] \to \mathbb{R}^v$ for a continuous underlying process that is observed through some potentially irregularly sampled time series $\mathbf{x} = ((t_0, x_0), (t_1, x_1), ..., (t_n, x_n))$, with $x_i \in \mathbb{R}^{v-1}$ and so $X_{t_i} = (t_i, x_i)$.

[1] introduces as hyperparameters for the Neural RDE model, the step size (length of the time interval) and truncation depth-N. As described in chapter 4 we choose points r_i to split the time horizon into smaller intervals, $t_0 = r_0 < r_1 < \cdots < r_m = t_n$. In [1] these are equally spaced but it is possible to have intervals of differing length. We would be interested in seeing how varying the interval size so that each interval contained a similar number of actual obervations might affect model performance. Next, the truncation depth $N \geq 1$ is chosen for the depth-N log-signature $\mathrm{LogSig}^N_{r_i, r_{i+1}}(X) \in \mathbb{R}^{\beta(v,N)}$ introduced in chapter 4. For $X : [t_0, t_n] \to \mathbb{R}^v$ and $t_0 \leq r_i \leq r_{i+1} \leq t_n$, the log-signature of X over the interval $[r_i, r_{i+1}]$ is a collection of integral terms each returning a real value to form an N term sequence of statistics that characterise how the path X drives the CDE over the interval $[r_i, r_{i+1}]$

5.2.1 Neural RDE Formulation

[1] builds the following analogy with CDEs: Let Z be the hidden state and Y be the output of a Neural CDE driven by X. Then the Neural CDE formulation,

$$Z_t = Z_{t_0} + \int_{t_0}^{t} f_\theta(Z_s) dX_s, \tag{5.2.1}$$

where $Z_{t_0} = \epsilon_\theta(t_0, x_0)$ and $Y_t = \ell_\theta(Z_t)$ for $t \in (t_0, t_n]$ can be solved by assuming X to be differentiable and rewriting the above CDE as an ODE of the form,

$$Z_t = Z_{t_0} + \int_{t_0}^{t} g_{\theta,X}(Z_s, s)ds, \tag{5.2.2}$$

where $g_{\theta,X}(Z_s, s) = f_\theta(Z)\dot{X}_s$. Obtaining the above equation in the form of an ODE allowed the use of the existing tools developed for ODEs such as black box solvers and the adjoint method for backpropagation to be used.

The path X that approximates the data process has been embedded in the new vector field $g_{\theta,X}(Z_s, s)$ as before. Applying the Rough Path solution theory, we replace $g_{\theta,X}(Z_s, s)$ with the below piecewise approximation as justified by definition 4.1.7 in chapter 4

$$\widehat{g}_{\theta,X}(Z, s) = \widehat{f}_\theta(Z)\frac{\text{LogSig}^N_{r_i,r_{i+1}}(X)}{r_{i+1} - r_i} \text{ for } s \in [r_i, r_{i+1}) \tag{5.2.3}$$

where $\widehat{f}_\theta : R^{w\times\beta(v,N)}$ is an arbitrary neural network. The RHS is a matrix-vector product, between the neural network $\widehat{f}_\theta$ and the log-signature $\text{LogSig}^N_{r_i,r_{i+1}}(X)$.

It then becomes possible to write an ODE in the same form as equation 5.2.2 for the rough path approach,

$$Z_t = Z_{t_0} + \int_{t_0}^{t} \widehat{g}_{\theta,X}(Z_s, s)ds, \tag{5.2.4}$$

which also allows the existing tools for Neural ODEs to be used.

Figure 5.1 from [1] visualises this process and contrasts it to the pointwise evaluation required for CDEs.

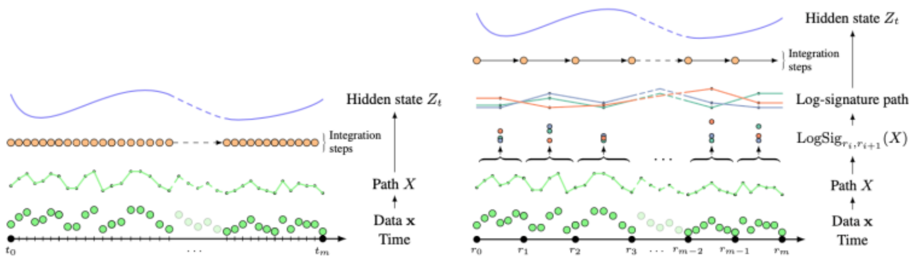


Figure 5.1: **Left:** The Neural CDE approach of pointwise evaluation leads to a high number of integration steps. **Right:** The Neural RDE approach summarises the input signal with the log-signature path leading to much fewer integration steps that updates the hidden state Z_t only for each interval [1, Figure 4, pg. 5]

Neural RDEs Generalise Neural CDEs

[1] notes that the Neural RDE may be reduced to a Neural CDE model and as such generalises the Neural CDE approach: Consider a log-signature term,

$$\frac{\text{LogSig}^N_{r_i,r_{i+1}}(X)}{t_{i+1} - t_i}, \tag{5.2.5}$$

where the time intervals have been chosen such that $r_i = t_i$ and $r_{i+1} = t_{i+1}$. We explained in chapter 4 how the depth-1 log signature is the change in coordinates (e.g. distance) over the interval. That is, $\text{LogSig}^1_{r_i,r_i+1}(X) = \Delta X_{[t_i,t_{i+1}]}$ and so we have,

$$\frac{\Delta X_{[t_i,t_{i+1}]}}{t_{i+1} - t_i} = \frac{dX^{linear}}{dt}(s) \quad \text{for} s \in [t_i, t_{i+1}). \tag{5.2.6}$$

This is the same as it would be for the Neural CDE model if using linear interpolation. However, previous work such as [4] used a cubic spline for integration for reasons of differentiability so strictly speaking this is not the same.

5.3 Implementation

We highlight a few advantages noted by [1] regarding the implementation of the Neural RDE model

Source of speed-ups

If the time horizon $[t_0, t_n]$ has been split into an $m << n$ number of intervals, we have a sequence of m log-signatures rather than n observation points. Therefore the *Log-signature path* is slowly varying compared to the input data. A differential equation driven by the log-signature path is therefore only updated at larger intervals allowing a reduction in the number of integration steps required. This correspondingly reduces the number of forward operations required in a forward pass and is the source of up to $10x$ speed ups in training times. Furthermore, even greater speed can be achieved by increasing step size and thus reducing accuracy

Adjoint Method for Backpropagation

As previously dicussed in section 2.1.2, using the memory efficient adjoint method for backpropagation dramatically reduces the memory cost of training a model. As for Neural CDEs and ODEs, it is possible to use the adjoint method for backpropagation with the log-ODE method. As described in section 3.3.3 on training CDEs with the adjoint method, if H is the memory cost of evaluating a single step for the vector field f_θ and T is the time horizon the adjoint method for backpropagation requires $\mathcal{O}(H + T)$. This is in contrast to the typical $\mathcal{O}(HT)$ required to backpropagate through the ODE solver [4].

Computing the Log-signatures upfront

Because the (log-)signatures are summaries of the input data (rather than a response) they only have to be computed once for a given choice of interval size $[r_i, r_{i+1}]$ and can be computed as part of a data preprocessing step. This also allows (log-)signatures to be computed on cloud based software and could essentially be made to be a part of a data-set.

Implementation Tools [1] use Signatory [2] to compute the log-signature transform but there are other standard libraries. [21] developed the 'torchdiffeq' tool to solve the Neural ODEs introduced in [7]. As equation (5.2.4) has been written in the form of an ODE 'torchdiffeq' can be used to solve it.

However, [1] developed another method by noting that (5.2.3) is of the same form as $g_{\theta,X}(Z_s, s) = f_\theta(Z)\dot{X}_s$ with the pointwise control path X_t replaced by a piecewise linear log-signature path. This formulation allows the log-signatures to be computed as a pre-processing step as described above. Once the log-signature control path has been constructed as a pre-processing step, existing tools for Neural CDEs (that also build on ODE-solvers) developed by [4] can be applied. The authors of [1] make the tools to implement this available at: https://github.com/jambo6/neuralRDEs

5.4 Applications

The Neural RDE model was formulated by [1] to address the Neural CDEs [4] shortcomings in coping with longer times series. As the Neural RDE model is a generalisation of the Neural CDE model it can in theory be applied to solve the same problems as a Neural CDE. In comparing the models, [1] saw little upside to using the Neural RDE model on short time series as the Neural CDE model performed "well enough" with little room for improvement in either speed or accuracy.

For longer time series this is an all together different story. As with RNNs, their continuous time equivalent, the Neural CDE model begins to break down as the length of the time series increases and the number of forward operations becomes prohibitively large resulting in unmanageable training times and reduced accuracy.

For the longer time series, pre-processing the input data to effectively reduce the length of a time series over the time horizon $[t_0, t_n]$ from n to $m << n$ is instrumental and a major contribution of [1]. Another source of efficiency is that by removing redundancies from the signature, the log-signature represents the same information in a compressed format. [1] maintains that closely sampled points will typically be strongly correlated and therefore little is lost by summarising over small intervals. Many times this is true, however, there might be significant cases where it is not e.g. in the advent of financial crashes.

Both the Neural CDE and RDE models are able to utilise the adjoint method for backpropagation, which dramatically reduces memory cost compared to backpropagating through an ODE solver. This is helpful to deal with longer time series as memory requirements would otherwise become prohibitively large. As an example of this, [1] highlight an experiment where the memory footprint is reduced from 3.6GB to 47MB by using the adjoint method of backpropagation.

Continuous time models such as the Neural CDE and RDE allow the number of steps used for the ODE solver to be chosen so as to obtain the desired level of accuracy vs speed. This step size can be chosen irrespective of the sampling rate of the data summary rate of the log-signature. Rather, the steps may be chosen so that they reflect the complexity of the data. For example, a large step size can be chosen for a slowly-varying but densely sampled path whereas when the path becomes more fast-varying the step size of the ODE solve must be closer to the sampling rate of the data to reflect the changes.

5.5 Experiments using the Neural RDE model

[1] experiemented with the Neural RDE on four datasets, one of which describes the movement of round worms and is named the Eigenworm dataset. We will discuss the Eigenworm experiment to illustrate the efficiency of the Neural RDE model applied to longer time series of multivariate data. The other three datasets were medical type data which we will not discuss for the sake of brevity and focus on discussing the Eigenworm classification as it offers a more intuitive and concise illustration of the application of the Neural RDE model presented in [1, Section 4 & Appendix D]

To examine model performance with respect to the choice of the two new hyper-parameters, [1] ran the Neural RDE model for log signature depths $N = 2, 3$ and all step sizes $(r_{i+1} - r_i)$ in $2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$. As discussed in section 5.2.1, a Neural RDE model with depth-1 and step size 1 reduces to a Neural CDE with a piecewise liner interpolation for the path.

5.5.1 Classifying EigenWorms

In this experiment the goal was to classify each roundworm as either a wild-type or one of four mutant types using the EigenWorms dataset from the UEA archive [28]. Each time series corresponds to the movement of a single roundworm as measured in relation to one of the six base shapes (i.e. the Eigenworms) of a roundworm on an agar plate. The time series are regularly sampled, are all of length 17,984 with 6 channels (including time). The Eigenworm dataset is regularly sampled and so [1] used $t_i = i$, i.e. a time corresponding to each observation In figure 5.2, the results of [1] in applying the Neural RDE model to the EigenWorm classification task are shown. The Neural RDE model is benchmarked against two models: As the work in [1] is an extension of the Neural CDE model from [4], a comparison with the Neural CDE model is most interesting. Also included is an ODE-RNN model introduced by [22] that generalises RNNs to have continuous-time hidden dynamics defined by ordinary differential equations (and so serves as a baseline of sorts) but does not use the adjoint method for backpropagation as seen in its very large memory footprint.

Model	Step	Accuracy (%)	Time (Hrs)	Mem (Mb)
ODE-RNN (folded)	1	–	–	–
	4	35.0 ± 1.5	0.8	3629.3
	32	32.5 ± 1.5	0.1	532.2
	128	47.9 ± 5.3	0.0	200.8
NCDE	1	62.4 ± 12.1	22.0	176.5
	4	66.7 ± 11.8	5.5	46.6
	32	64.1 ± 14.3	0.5	8.0
	128	48.7 ± 2.6	0.1	3.9
NRDE (depth 2)	4	$\mathbf{83.8 \pm 3.0}^*$	2.4	180.0
	32	67.5 ± 12.1	0.7	28.1
	128	$\mathbf{76.1 \pm 5.9}$	0.2	7.8
NRDE (depth 3)	4	76.9 ± 9.2	2.8	856.8
	32	$\mathbf{75.2 \pm 3.0}$	0.6	134.7
	128	68.4 ± 8.2	0.1	53.3

Figure 5.2: Eigenworms dataset [1, Table 1, pg. 7]

In the Neural CDE model, using a step size of 1 equates to "normal" use, i.e. a pointwise evaluation of the whole time series which took 22 hours to train. Increasing the step size for the Neural CDE model equates to naively subsampling the time series at every 4, 32, 128 etc. steps. Increasing the step size to 4 in the Neural CDE model reduces the training time to 5.5 hours without reducing the accuracy, perhaps highlighting how correlated adjacent observations are.

If we apply the Neural RDE model with the same step size of 4, the accuracy increases (from 66.7%) to 83.8%, standard deviation of the test reults is much lower and the training time is reduced to only 2.4 hours. As step size is increased the difference in model performance between Neural CDEs and Neural RDEs becomes accentuated, highlighting the superiority of the Neural RDE model in learning from long time series.

As expected, the ODE-RNN model incurs a huge memory cost and for this reason couldn't be trained for a step size of 1. Training times are modest compared to both the Neural CDE and Neural RDE model because there are much fewer forward operations to compute within each training epoch but accuracy is significantly lower, most likely due to the same reason.

Results for even higher step sizes than those shown, show a reduction in all of accuracy, training time and memory cost in a predictable manner. These are not all shown here but can be seen in [1, Appendix D]. Furthermore, the process for hyperparameter selection, optimiser, normalisation, architecture, etc. is given in [1, Appendix C].

Chapter 6

A Simple Implementation and Potential Financial Applications

In this chapter, we present a simple implementation of the Neural RDE method. The full code is made available at, https://github.com/patrick-kidger/torchcdeby by two of the authors of [4] and [1], Patrick Kidger and James Morrill. The code we used can be found in the form of a Jupyter Python notebook in A.1. The implementation summarises the input signal with a log-signature to drive a Neural CDE and then solves it with the log-ODE method to make it a Neural RDE model.

The implementation uses randomly generated data that is transformed into spirals that either rotate clockwise or anti-clockwise through time using a combination of a Uniform and Gaussian distribution. The vertical coordinates and horizontal coordinates are a channel each and time is included as a channel, so three channels in total. The experiment is run for log-signatures of depth-1, -2, -3 which results in a log-signature path of dimensions (channels) 3, 6 and 14 respectively. The log-signature summarises the input signal or path X, thus effectively reducing the time series but has higher dimensionality (more channels) than the underlying data. Each channel can be computed in parallel so reducing the length of a signal to introduce more channels increases possible parallelism, one of the key benefits of using the log-signature to summarise data.

A time series capturing the sequential dynamics of spirals rotating in two dimensions through time is perhaps a little too simplistic to showcase the full potential of Neural RDE models. However, the task of classifying the rotating spirals provides a very intuitive and easily understood example of how the Neural RDE model can be used to classify longer time series. Furthermore a rotating spiral is the kind of visual and intuitive example often used to showcase the (log-)signatures effectiveness at capturing a data trajectory.

A time series of length 5,000 is used as it is long enough to be computed with relative ease but long enough so that without the log-signature and the log-ODE method, the Neural CDE model would struggle to train and disregard early observations. Furthermore a rotating spiral is the kind of visual and intuitive example often used to showcase the (log-)signatures effectiveness at capturing a data trajectory.

We are particularly interested in applications of how the Neural RDE model could be used to classify a long time series of multivariate (and potentially irregularly sample) data relating to whether a positive or negative outcome was likely to occur in a financial setting. In particularly, we think it would be interesting to apply the Neural RDE model to long historical sequences of data relating to credit risk with a view to classify whether a debtor is likely to default or not. This could be particular useful in the setting of various lending arrangements such as Private Credit Funds, Leveraged Finance Teams, Credit Ratings, Mortgages, Credit Cards etc.

As the data for such an effort is not readily available but rather the guarded property of various types of lenders it has not been possible to use such data. However, the model is agnostic to the nature of the underlying data and the rotating spirals provide a good enough illustration of how the Neural RDE model can used to capture the sequential dynamics in a longer time series and classify it as one or the other.

In the table below, we present the main results of the experiment. The full results with losses and training times for each training epoch can be found at the end of appendix A.1

Final results	
Log-signature depth-1	Accuracy on test set: 93.0 Time per epoch: 11.7s
Log-signature depth-2	Accuracy on test set: 100.0% Time per epoch: 10.3s
Log-signature depth-3	Accuracy on test set: 100.0% Time per epoch: 13.5s

Table 6.1: Accuracy on test set for each of log-signature depths [1, 2, 3] and the time taken for each training epoch

The depth-1 log-signature is already quite accurate at 93% accuracy. As the rotating spirals are quite a simple geometric structure, the depth-1 log-signature already captures most of it and is therefore quite accurate in its classifications.

We can see that both the depth-2 and depth-3 log-signatures are 100% correct in classifying whether a spiral is rotating clockwise or anti-clockwise. However training losses are measurably smaller for the depth-3 log-signature. Note that all the log-signature depths take a similar time to train. As there are only three input channels and therefore only [3, 6, 14] channels in the log-signature, the parallelism mentioned above is not exhausted and so using a deeper log-signature does not measurably affect the time taken to train. In general, this parallelism can be taken of advantage of by summarising input signals with a deeper log-signature and dealing with the increased computations with a larger number of CPUs/GPUs.

6.1 Application in Finance

Machine learning is already widely used in the financial services industry and a body of work (some of which is publicly available) exists examining the use of machine learning methods to quantify or classify credit risk. In particular some research has gone into using Deep Learning and Convolutional Neural Networks for credit scoring [29]. However very little has been done so far in applying Machine Learning methods to capture the sequential dynamics of a time series relating to credit risk. As the data on various types of borrowers usually exists over a very long time window, some observations are missing and the sequence of events is highly informative, it feels intuitive that a model such as the Neural RDE model would offer improvement on previous approaches.

A recent paper [3] by researchers at American Express AI Research and Rutgers University dives into the topic of using sequential deep learning, namely RNNs with LSTM cells for credit default risk. In the paper they consider credit risk as a binary classification task of whether default (non-payment) occurs on credit card debt within a certain time frame. [3] find that the sequential models outperform non-sequential models, stating,

> ...suggesting that **historical information provides orthogonal information that is predictive of risky financial behavior**. While these performance improvements may seem modest, it is important to keep in mind the large volume of card members that exists in the dataset, implying that small improvements lead to significant savings (in our case, an annual savings of tens of millions of US dollars).

which is in line with intuition and solidifies the notion that modelling sequential dynamics is a superior approach for classifying default risk.

[3] examined the use of both an RNN model with LSTM cells and a Temporal Convolutional Network (TCN) to capture the sequential dynamics. As the data looks at a long time window, has dimensionality and is highly irregularly sampled, [3] run into all of the issues these sequential models have with long time series and irregularly sampled data as described throughout earlier chapters of this thesis. The methods they use to circumvent these issues such as generative sampling etc. are naive mathematically in comparison with the Neural RDE model. It is our believe that the Neural RDE model could have a highly valuable use in this setting as even the slightest improvements in the accuracy of predicting default can save lenders large amounts of capital.

Conclusion

This thesis provided an exposition on the theory underlying Neural Rough Differential Equations (Neural RDEs). The mathematical components are broken down and discussed both with the aim of highlighting what motivates the existence of the field of research, higlights its milestones and ultimately bringing the reader to an understanding the Neural RDE model.

We provide a simple illustrative example of how the Neural RDE method is scripted into code and highlight the results.

Finally, we dissuss a potential application of the Neural RDE model in predicting default risk that builds upon the very recent work of researchers at American Express AI Research [3].

Future work on this topic might focus on carrying out a similar classification task as in [3] with the Neural RDE model from [1].

Appendix A

Implementation

A.1 Code for Classifying Rotating Spirals

The full code along with more detailed documentation on the proprietary libraries is made available at, https://github.com/patrick-kidger/torchcdeby by two of the authors of [4] and [1], Patrick Kidger and James Morrill. The code from Kidger and Morrill was developed first to showcase the Neural CDE model and then the Neural RDE model and is spread across a number of python files containing the respective models, classes and functions. We have singled out the parts of their code neccessary to our example and pieced it together to form a Jupyter notebook. Aside from edits for the code to work in a Jupyter notebook and further annotations, the code below is the same as that presented by Kidger and Morrill. Below, we present the code we ran as a Jupyter Python notebook on Google Colab. The code should run without any editing on a Linux-based machine. The code summarises the input signal with a log-signature to drive a Neural CDE and then solves it with the log-ODE method to make it a Neural RDE model.

Please be careful to run each install before the next.

We begin by installing PyTorch which is an optimised tensor library used for various applications with Neural Networks

```
pip install torch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2
```

The package torchcde is made available by one of the authors of [1], Patrick Kidger and can be used to construct a path by interpolation, parameterises the CDE and solves it.

```
pip install torchcde
```

The 'signatory' package is also made available by Patrick Kidger at the above github site. The signatory package can construct both signatures and log-signatures of paths with parameters such as size of time intervals t_i, t_{i+1}

```
pip install signatory==1.2.4.1.7.1 --no-cache-dir --force-reinstall
```

Import the neccessary packages

```python
import math
import signatory
import time
import torch
import torchcde
```

A Controlled Differential Equation (CDE) takes the form, $z_t = z_0 + \int_0^t f_\theta(z_s)dX_s$. Where X is the path and the function f_θ is a neural network as described in the main body of the paper.

The below class defines a neural network f_θ with a single hidden layer of width 128,

```python
class CDEFunc(torch.nn.Module):
    def __init__(self, input_channels, hidden_channels):
        ######################
        # The number of input channels is determined by the data X,
        # in our case this will be 3 channels, time and x, y coordinates.
        # The number of hidden channels is the number of channels for the
        # hidden state z_t. This is chosen by the user.
        super(CDEFunc, self).__init__()
        self.input_channels = input_channels
        self.hidden_channels = hidden_channels

        self.linear1 = torch.nn.Linear(hidden_channels, 128)
        self.linear2 = torch.nn.Linear(128, input_channels * hidden_channels)

    ######################
    # The t argument is included here so that the CDE can be made to behave
    # differently at different times. However, according to [Kidger] this is unusual
    # making the t argument redudant for most practical purposes.
    ######################
    def forward(self, t, z):
        # z has shape (batch, hidden_channels)
        z = self.linear1(z)
        z = z.relu()
        z = self.linear2(z)
        ######################
        # According to [Kidger], the best results tend to be obtained by adding a
        # final tanh nonlinearity.
        ######################
        z = z.tanh()
        ######################
        # We require that the output tensor represents a linear map from
        # R^input_channels to R^hidden_channels. Therefore the output tensor
        # must be a matrix
        ######################
        z = z.view(z.size(0), self.hidden_channels, self.input_channels)
        return z
```

The below class constructs a Neural CDE model with the class 'CDEFunc' above defining the neural network ($f_\theta(z_s)$).

```python
class NeuralCDE(torch.nn.Module):
    def __init__(self, input_channels, hidden_channels, output_channels, interpolation="cubic"):
        super(NeuralCDE, self).__init__()

        self.func = CDEFunc(input_channels, hidden_channels)
        self.initial = torch.nn.Linear(input_channels, hidden_channels)
        self.readout = torch.nn.Linear(hidden_channels, output_channels)
        self.interpolation = interpolation

    def forward(self, coeffs):
        if self.interpolation == 'cubic':
            X = torchcde.CubicSpline(coeffs)
        elif self.interpolation == 'linear':
            X = torchcde.LinearInterpolation(coeffs)
        else:
            raise ValueError("Only 'linear' and 'cubic' interpolation methods are implemented.")

        ######################
        # The initial hidden state should be a function of the first observation.
        ######################
        X0 = X.evaluate(X.interval[0])
        z0 = self.initial(X0)

         ######################
        # Compute the integral to solve the CDE, z_t = z_0 + \int_0^t f_\theta(z_s) dX_s.
        ######################
        z_T = torchcde.cdeint(X=X,
                              z0=z0,
                              func=self.func,
                              t=X.interval)

        ######################
        # .cdeint returns both the initial value and the final value.
        # We extract only the final value of the hidden state.
        # To get the output (prediction), apply the linear map
        # self.readout(z_T) defined above
        ######################
        z_T = z_T[:, 1]
        pred_y = self.readout(z_T)
        return pred_y
```

The below function generates spirals that rotate clockwise and anti clockwise in time. The argument num_timepoints is effectively the length of the time series. Here initiated at 100 but later changed to $5,000$.

```python
def get_data(num_timepoints=100):
    # Create a 1-D tensor of size 'num_timepoints'
    # with values evenly spaced between 0 and 4 * math.pi
    t = torch.linspace(0., 4 * math.pi, num_timepoints)

    # torch.rand creates a tensor filled with random numbers from a uniform distribution
    # on the interval [0,1), in this case of size 128
    start = torch.rand(128) * 2 * math.pi

    # torch.cos returns a new tensor with the cosine of the elements inside
    # its input, this forms the x coordinates
    x_pos = torch.cos(start.unsqueeze(1) + t.unsqueeze(0)) / (1 + 0.5 * t)
    x_pos[:64] *= -1

    #similarly y coordinates generated with torch.sin
    y_pos = torch.sin(start.unsqueeze(1) + t.unsqueeze(0)) / (1 + 0.5 * t)

    # torch.randn_like Returns a tensor with the same size as input that
    # is filled with random numbers from a normal distribution with
    # mean 0 and variance 1. x and y coordinates redefined as
    x_pos += 0.01 * torch.randn_like(x_pos)
    y_pos += 0.01 * torch.randn_like(y_pos)

    ######################
    # According to Kidger, Neural CDEs in the model need to be explicitly
    # told the rate at which time passes so time is included as a channel
    # This randomly generated dataset is by default regularly sampled and
    # therefore incorporating time as a channel is relatively straightforward
    ######################

    X = torch.stack([t.unsqueeze(0).repeat(128, 1), x_pos, y_pos], dim=2)
    y = torch.zeros(128)
    y[:64] = 1

    perm = torch.randperm(128)
    X = X[perm]
    y = y[perm]

    ######################
    # X is a tensor of observations, of shape (batch=128, sequence=num_timepoints, channels=3)
    # y is a tensor of labels, of shape (batch=128,), either 0 or 1 corresponding to
    # anticlockwise or clockwise respectively.
    ######################
    return X, y
```

In the following code, the input signal is turned into a log-signature, a Neural CDE driven by the log-signature is constructed (using the above classes) and becomes a Neural RDE. The Neural RDE is solved and the model trained and tested on a time series of length 5000 so chosen as to be manageable to compute but long enough so as to be unmanageable for the Neural CDE model.

Documentation on the torchcde library that is used in the code below may be found at https://github.com/patrick-kidger/torchcde.

```python
import signatory
import time
import torch
import torchcde


# The function below defines the training and testing procedure
def _train(train_X, train_y, test_X, test_y, depth, num_epochs, window_length):
    # Start time of the training process
    start_time = time.time()

    # Using the 'torhcde' and 'signatory' libraries to compute the logsignature
    train_logsig = torchcde.logsig_windows(train_X, depth, window_length=window_length)
    print("Logsignature shape: {}".format(train_logsig.size()))

    model = NeuralCDE(
        input_channels=train_logsig.size(-1), hidden_channels=8, output_channels=1, interpolation="linear"
    )
    optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

    train_coeffs = torchcde.linear_interpolation_coeffs(train_logsig)

    train_dataset = torch.utils.data.TensorDataset(train_coeffs, train_y)
    train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=32)
    for epoch in range(num_epochs):
        for batch in train_dataloader:
            batch_coeffs, batch_y = batch
            pred_y = model(batch_coeffs).squeeze(-1)
            loss = torch.nn.functional.binary_cross_entropy_with_logits(pred_y, batch_y)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        print("Epoch: {}   Training loss: {}".format(epoch, loss.item()))

    # Compute the logsignature of the test data
    test_logsig = torchcde.logsig_windows(test_X, depth, window_length=window_length)
    test_coeffs = torchcde.linear_interpolation_coeffs(test_logsig)
    pred_y = model(test_coeffs).squeeze(-1)
    binary_prediction = (torch.sigmoid(pred_y) > 0.5).to(test_y.dtype)
    prediction_matches = (binary_prediction == test_y).to(test_y.dtype)
    proportion_correct = prediction_matches.sum() / test_y.size(0)
    print("Test Accuracy: {}".format(proportion_correct))

    # Time of each training epoch
    elapsed = time.time() - start_time

    return proportion_correct, elapsed


def print_heading(message):
    # Print a message inbetween rows of #'s
    string_sep = "#" * 50
    print("\n" + string_sep + "\n{}\n".format(message) + string_sep)


def main(num_epochs=15):
   ######################
    # Call the get_data() function defined above. Set the length of each time
    # series to 5,000 so chosen as to be manageable for these computations but
    # long enough so that the Neural CDE model would struggle to train and forget
    # earlier observations.
    ######################
    num_timepoints = 5000
    train_X, train_y = get_data(num_timepoints=num_timepoints)
    test_X, test_y = get_data(num_timepoints=num_timepoints)
    ######################
    # Note that the model is trained on whole time series and tested on whole
    # time series with a view to classify
    # This is not train on part of a time series, test on part of a time series
    # Therefore the training and testing data sets are generated in the same way
    ######################

    ######################
    # The model is tested for log-signatures of depth-1, -2 and -3. The time
    # interval is 50 steps so that the length of the log-signature path is 100.
    # The torchcde library offers torchcde.logsis_windows to set step size
    ######################
    # The spirals data has 3 channels, time and x, y coordinates. As a result,
    # the log-signature path is of dimensions (i.e. the number of channels of
    # the log-signature is) [3, 6, 14] for depth-1, -2 and -3 respectively.
    ######################
    # One of the major benefits of summarising the path X with log-signatures
    # is that the increased number of dimensions(channels) of the log-signature
    # can be computed in paralell.
    ######################

    depths = [1, 2, 3]
    window_length = 50
    accuracies = []
    training_times = []
    for depth in depths:
        print_heading('Running for logsignature depth: {}'.format(depth))
        acc, elapsed = _train(
            train_X, train_y, test_X, test_y, depth, num_epochs, window_length
        )
        training_times.append(elapsed)
        accuracies.append(acc)

    # Finally log the results to the console for a comparison
    print_heading("Final results")
    for acc, elapsed, depth in zip(accuracies, training_times, depths):
        print(
            "Depth: {}\n\tAccuracy on test set: {:.1f}%\n\tTime per epoch: {:.1f}s".format(
                depth, acc * 100, elapsed / num_epochs
            )
        )

if __name__ == "__main__":
    main()
```

The full results for all training epochs for log-signature depths-1, -2, and -3 may be seen below.

```
###################################################
Running for logsignature depth: 1
###################################################
Logsignature shape: torch.Size([128, 101, 3])
Epoch: 0   Training loss: 1.5701801776885986
Epoch: 1   Training loss: 1.8828034400939941
Epoch: 2   Training loss: 0.6248658299446106
Epoch: 3   Training loss: 0.90535569190979
Epoch: 4   Training loss: 0.6393105387687683
Epoch: 5   Training loss: 0.6126113533973694
Epoch: 6   Training loss: 0.5433077216148376
Epoch: 7   Training loss: 0.6375104188919067
Epoch: 8   Training loss: 0.4952704906463623
Epoch: 9   Training loss: 0.4968360960483551
Epoch: 10   Training loss: 0.558059811592102
Epoch: 11   Training loss: 0.47762250900268555
Epoch: 12   Training loss: 0.3704424798488617
Epoch: 13   Training loss: 0.44517382979393005
Epoch: 14   Training loss: 0.32703888416290283
Test Accuracy: 0.9296875

###################################################
Running for logsignature depth: 2
```

```
#################################################
Logsignature shape: torch.Size([128, 101, 6])
Epoch: 0   Training loss: 2.077491283416748
Epoch: 1   Training loss: 1.9196913242340088
Epoch: 2   Training loss: 0.8045685291290283
Epoch: 3   Training loss: 0.6398347020149231
Epoch: 4   Training loss: 0.9980854392051697
Epoch: 5   Training loss: 0.6584505438804626
Epoch: 6   Training loss: 0.8339160084724426
Epoch: 7   Training loss: 0.5205453634262085
Epoch: 8   Training loss: 0.6050302386283875
Epoch: 9   Training loss: 0.4231453537940979
Epoch: 10   Training loss: 0.3232039213180542
Epoch: 11   Training loss: 0.3679076135158539
Epoch: 12   Training loss: 0.28672972321510315
Epoch: 13   Training loss: 0.24821045994758606
Epoch: 14   Training loss: 0.2578696608543396
Test Accuracy: 1.0

#################################################
Running for logsignature depth: 3
#################################################
Logsignature shape: torch.Size([128, 101, 14])
Epoch: 0   Training loss: 0.543790876865387
Epoch: 1   Training loss: 1.2751718759536743
Epoch: 2   Training loss: 0.5648033618927002
Epoch: 3   Training loss: 0.6132439970970154
Epoch: 4   Training loss: 0.8236994743347168
Epoch: 5   Training loss: 0.5192417502403259
Epoch: 6   Training loss: 0.6903497576713562
Epoch: 7   Training loss: 0.5125799179077148
Epoch: 8   Training loss: 0.49508941173553467
Epoch: 9   Training loss: 0.5253514051437378
Epoch: 10   Training loss: 0.3784427046775818
Epoch: 11   Training loss: 0.35363882780075073
Epoch: 12   Training loss: 0.4072670638561249
Epoch: 13   Training loss: 0.3648272156715393
Epoch: 14   Training loss: 0.19591942429542542
Test Accuracy: 1.0

#################################################
Final results
#################################################
Depth: 1
        Accuracy on test set: 93.0%
        Time per epoch: 11.7s
Depth: 2
        Accuracy on test set: 100.0%
        Time per epoch: 10.3s
Depth: 3
        Accuracy on test set: 100.0%
        Time per epoch: 13.5s
```

Bibliography

[1] J. Morrill, C. Salvi, K. Kidger, J. Foster, and T. Lyons. Neural rough differential equations for long time series. Preprint, arXiv:2009.08295, 2020.

[2] P. Kidger and T Lyons. Signatory: differentiable computa- tions of the signature and logsignature transforms, on both cpu and gpu. arxiv:2001.00706, 2020b. Preprint, arXiv:2001.00706, URL: https: //github.com/patrick-kidger/signatory, 2020.

[3] J. M., D. Xu, N. Yousefi, and D. Efimov. Sequential deep learning for credit risk monitoring with tabular financial data. Preprint, arXiv:2012.15330, 2018.

[4] P. Kidger, J. Morrill, J. Foster, and T. Lyons. Neural controlled differential equations for irregular time series. Preprint, arXiv:2005.08926, 2019.

[5] A. Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

[6] A. Amini and A. Soleimany. Introduction to deep learning. Lecture notes available at: http://introtodeeplearning.com, 2021.

[7] R.T.Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. Preprint, arXiv:1806.07366, 2018.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9, 1997.

[9] Y. Lu, A. Zhong, Q. Li, and B. Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. Preprint, arXiv:1710.10121, 2017.

[10] L. Ruthotto and E. Haber. Deep neural networks motivated by partial differential equations. Preprint, arXiv:1804.04272, 2018.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. Preprint, arXiv:1512.03385, 2015.

[12] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishechenko. The mathematical theory of optimal processes. *ZAMM - Journal of Applied Mathematics and Mechanics*, 43:514–515, 1962.

[13] A. Gholami, K. Keutzer, and G. Biros. Anode: Unconditionally accurate memory-efficient gradients for neural odes. Preprint, arXiv:1902.10298, 2019.

[14] M. Ciccone, M. Gallieri, J. Masci, C. Osendorfer, and F. Gomez. The mathematical theory of optimal processes. *Advances in Neural Information Processing Systems*, pages 3025–3035, 2018.

[15] T.J. Lyons. Differential equations driven by rough signals. *Revista Matematica Iberoamericana*, 14:215–310, 1998.

[16] T. Lyon, M. Caruana, and T. Levy. *Differential equations driven by rough paths, Ecole d'été de probabilités de Saint-Flour XXIV-2004*. Springer, Berlin, Heidelberg, 2007.

[17] A. Kvarving. Natural cubic splines. Available at:https://www.math.ntnu.no/emner/TMA4215/2008h/cubicsplines.pd, 2008.

[18] P. Kidger. Neural controlled differential equations. Presentation available at:https://www.math.ntnu.no/emner/TMA4215/2008h/cubicsplines.pd, 2008.

[19] P. Bonnier, P. Kidger, I. P. Arribas, C. Salvi, and T. Lyons. Deepsignaturetransforms. *Advances in Neural Information Processing Systems*, page 3099–3109, 2019.

[20] I. P. Arribas. Derivatives pricing using signature payoffs. Preprint, arXiv:1809.09466, 2018.

[21] R.T.Q. Chen. "torchdiffeq". https://github.com/rtqichen/torchdiffeq, 2018.

[22] Y. Rubanova, T. Q. Chen, and D. K. Duvenaud. Latent ordinary differential equations for irregularly- sampled time series. *Advances in Neural Information Processing Systems 32*, page 5320–5330, 2019.

[23] A. Bagnall, H. A. Dau, J. Lines, M. Flynn, J. Large, A. Bostrom, P. Southam, and E. Keogh. The uea multivariate time series classification archive. arXiv:1811.00075, 2018.

[24] P. K. Friz and M. Hairer. *A Course on Rough Paths with an Introduction to Regularity Structures, Second Edition.* Springer, Cham, Switzerland, 2020.

[25] P. K. Friz and N. B. Victoir. *Multidimensional stochas- tic processes as rough paths: theory and applications, volume 120. Cambridge University Press.* Cambridge University Press, Cambridge, UK, 2010.

[26] B. Hambly and T. Lyons. Uniqueness for the signature of a path of bounded variation and the reduced path group. *Annals of Mathematics*, 171, 2010.

[27] Y. Boutaib, L. G. Gyorku, T. Lyons, and D. Yang. Dimension-free euler estimates of rough differential equations. *Revue Roumaine de Mathmatiques Pures et Appliques*, 59, 2014.

[28] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh. The great time series classification bake off: a re- view and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*, 31:606–660, 2017.

[29] B.R. Gunnarsson, Broucke S.V., B. Baesens, Óskarsdóttir, and W. Lemahieua. Deep learning for credit scoring: Do or don't? *European Journal of Operational Research*, 295:292–305, 2021.

HINRIK_BERGS_00449457

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61