IMPERIAL

IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

SECOND-YEAR GROUP RESEARCH PROJECT

Scientific Machine Learning: Neural Networks and Neural Differential Equations

Authors:
Jiaru (Eric) Li (CID: 02216531)
James Tay (CID: 02015786)
Xinyan Wang (CID: 02205857)
Jiankuan Liu (CID: 02215415)
Tianshi Liu (CID: 02218664)

Supervisor:
Sheehan Olver

June 17, 2024

Abstract

This paper delves into the integration of neural networks with differential equations, known as neural differential equations, within the field of scientific machine learning. They offer a continuous-time model that is particularly effective for handling time series data, surpassing traditional methods that operate in discrete time. We explore the core principles of neural networks, their architecture, and training techniques, and then introduce neural ordinary differential equations, highlighting their formulation and practical applications as part of larger neural networks. As an extension, we briefly discuss neural controlled differential equations. This study not only emphasises the theoretical importance of neural differential equations but also showcases their potential in various applications.

Contents

1 Introduction **3**

2 Neural Networks **4**

2.1 History . . . 4

2.2 Concepts . . . 4

2.2.1 Structure . . . 4

2.2.2 Activation Function . . . 4

2.2.3 Loss Function . . . 5

2.3 Optimisation Techniques . . . 6

2.3.1 Gradient Descent . . . 6

2.3.2 Automatic Differentiation and Discrete Backpropagation . . . 8

2.4 Example: Regression . . . 9

2.5 Example: Classification . . . 11

3 Neural ODEs **14**

3.1 Motivation . . . 14

3.2 Comparing Neural ODEs with ResNet . . . 15

3.3 Various Neural ODE Models . . . 16

3.3.1 Augmented Neural ODEs . . . 16

3.3.2 Neural ODEs with Scientific Machine Learning . . . 16

3.4 Solving ODEs Numerically . . . 16

3.5 Backpropagation in Neural ODEs . . . 19

3.5.1 Discretise-then-Optimise (DO) . . . 19

3.5.2 Optimise-then-Discretise (OD) . . . 19

3.6 Adjoint Sensitivity Method . . . 20

3.6.1 Continuous Backpropagation . . . 20

3.6.2 Gradient wrt. θ and t . . . 23

4 Application: Digit Classifier with Neural ODEs **25**

4.1 Implementation . . . 25

4.2 Comparison . . . 26

5 Extension: Neural CDEs **28**

5.1 Motivation . . . 28

5.2 Controlled Differential Equations . . . 28

5.3 Neural CDEs . . . 29

5.4 Solving Neural CDEs . . . 30

5.5 Application: Time Series Modelling 31
5.5.1 Cubic Splines 31
5.5.2 Implementation 35

6 Conclusion **36**

Acknowledgements **36**

References **37**

A Proof of Convergence of Gradient Descent Algorithm **39**

B Comparison of ODE Solvers **42**

C A Simple Example of the Adjoint Method **43**

D Data Preprocessing of Financial Time Series **45**

1 Introduction

In recent years, the intersection of machine learning and scientific computing has given rise to a novel field known as scientific machine learning. It takes advantage of the power of both disciplines to address complex problems that might be difficult to solve using conventional methods. A particularly exciting development in this area is the concept of neural differential equations, integrating neural networks with differential equations [1].

The motivation behind neural differential equations stems from the limitations of traditional neural networks, such as their inability to handle continuous time series data. Neural differential equations address these issues by providing a continuous-time analogue to discrete-layer architectures, such as residual neural networks. Additionally, neural differential equations are advantageous because of their efficient memory use [2].

In the next section, we describe some foundational notions of neural networks. We review the history and basic structure of neural networks, including key concepts such as layers, activation functions, and loss functions. Optimisation strategies are also discussed, specifically gradient descent and automatic differentiation for efficient calculations. We then explore some simple applications to further illustrate these ideas and help us to discover how neural networks work in terms of real-world scenarios, such as regression and classification problems.

Subsequently, we dive into the formulation and implementation of neural ordinary differential equations (ODEs) and discuss their connection to residual neural networks. Most importantly, the backpropagation of gradients needs to be derived in the gradient descent algorithm process. We discuss the methods to do so with neural ODEs, elaborating on numerical ODE solvers and the adjoint method.

Furthermore, we provide two applications for neural ODEs. We first extend the MNIST handwritten digit classification task from classical neural networks to neural ODEs, improving the accuracy from 80% to around 95% with a faster running time and a smaller number of parameters, thereby demonstrating the capabilities of neural ODEs.

Finally, building on the framework of neural ODEs to handle more complex and irregular data streams, we extend our discussion to neural controlled differential equations (CDEs). We examine the definition of neural CDEs and discuss and apply methods to model time series data using neural CDEs and a technique from numerical analysis called cubic splines.

Throughout this section, we use the Julia programming language. Julia is a modern, compiled, high-level, open-source language developed at MIT. It is becoming increasingly important in scientific machine learning and is widely used in high-performance computing and artificial intelligence, including by Astrazeneca, Moderna, and Pfizer in drug development and clinical trial acceleration, IBM for medical diagnosis, MIT for robot locomotion, and elsewhere [3]. Where appropriate, we supplement suitable Julia code to accompany the ideas presented. All code can be found in the project's GitHub repository.

Overall, we aim to provide a thorough understanding of neural differential equations and their significance in scientific machine learning.

2 Neural Networks

2.1 History

Before diving into neural ordinary and control differential equations, we first explore classic neural networks. Generally speaking, a *neural network* is a group of units called *neurons* connected by signals called *synapses*.

In 1943, Warren McCulloch and Walter Pitts published a seminal paper, in which they proposed the first mathematical model of an *artificial neural network*, inspired by biological neural networks in animal brains [4]. This model was named the *McCulloch-Pitts neuron*, or *perceptron*. In some sense, it could 'learn' from given data and make decisions or predictions based on some parameters. The process of optimising the choice of such parameters is called *training* the neural network.

Later in 1958, Frank Rosenblatt described an implementation of perceptron in detail [5]. The field of neural networks has expanded rapidly and is nowadays widely used in various fields such as machine learning and artificial intelligence. We explore the fundamental concepts of neural networks and the optimisation strategies, before conducting two examples in this section.

2.2 Concepts

2.2.1 Structure

An *(artificial) neural network* ((A)NN) consists of interconnected neurons, each of which can be thought of as a function $\mathbb{R} \to \mathbb{R}$. These neurons are typically organised into *layers* consisting of

- an *input layer* that receives the input,
- one or more *hidden layers* that process the data, and
- an *output layer* that produces the output.

The number of layers is usually called the *depth* of the neural network. The whole neural network can be thought of as a function $f : \mathbb{R}^m \to \mathbb{R}^n$ that takes an input vector $\mathbf{x}$ of dimension m and an output vector $\hat{\mathbf{y}}$ of dimension n. Each connection between neurons has a *weight* w, and possibly a *bias* b, which are adjusted in the optimisation process, allowing the model to generalise to unseen data. They are usually given as a *weight matrix* $\mathbf{W}$ and a *bias vector* $\mathbf{b}$. The *parameter* of the model, therefore, consists of the weights and biases of the model, which are wrapped in a vector θ.

A simple neural network with an input layer (1 neuron), a hidden layer (5 neurons), and an output layer (1 neuron) is shown in Figure 1. We note that every layer is *dense*, that is, every neuron in one layer is connected to every other neuron in the next layer.

2.2.2 Activation Function

For neural networks with multiple layers, an *activation function* σ is usually used. This is because it helps to decide whether a neuron should be 'activated' and introduces non-linearity into the model. The output of each layer is therefore calculated as $\mathbf{h}_{t+1} = \sigma(\mathbf{W}_t \mathbf{h}_t + \mathbf{b}_t)$, where σ acts

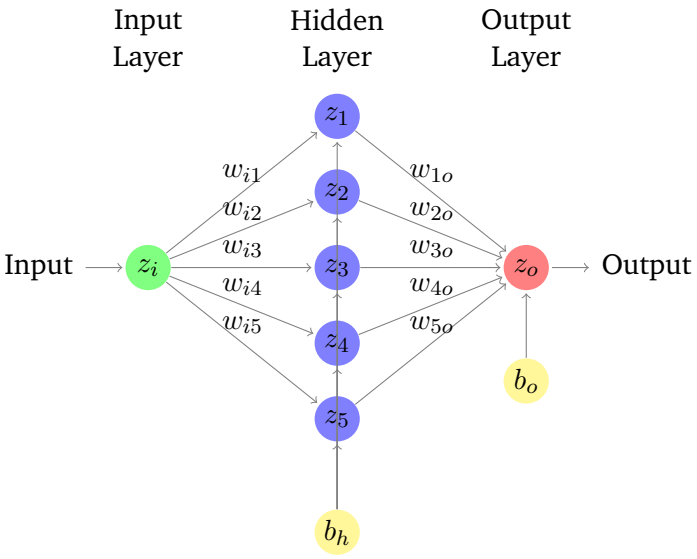


Figure 1 A simple neural network model.

element-wise, $\mathbf{h}_t$ is the output vector for layer t, $\mathbf{W}_t$ is the weight matrix connecting layer t to layer $t+1$, and $\mathbf{b}_t$ is the bias vector for layer t.

Regarding the choice of the activation function, the S-shaped *sigmoid function* has some nice properties: it is bounded, differentiable with a non-negative derivative, and has exactly one point of inflection [6]. Two familiar examples of sigmoid functions are the *logistic function* $\sigma(x) = 1/(1+e^{-x})$ and the hyperbolic tangent function $\sigma(x) = \tanh x$. Another commonly used activation function is the *rectifier*, or *rectified linear unit* (ReLU), defined as $\sigma(x) = \max(0, x)$. The graphs of these three functions are shown in Figure 2.

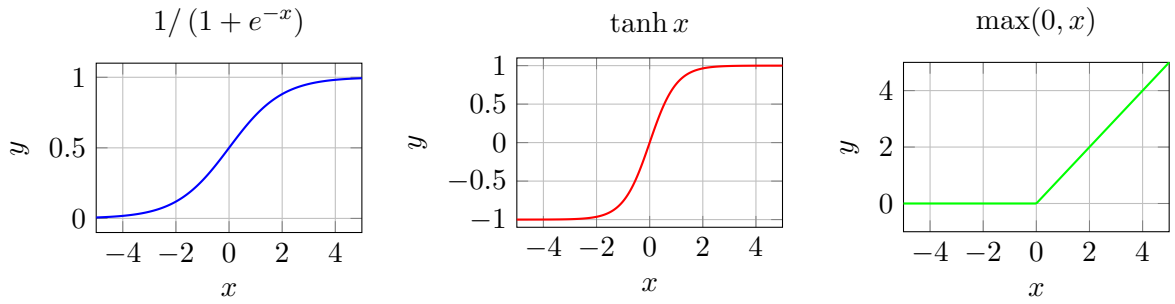


Figure 2 Three activation functions.

2.2.3 Loss Function

The ultimate purpose of *training* a neural network is to minimise the error between the output predicted by the neural network $\hat{\mathbf{y}}$ and some objective. We typically define this 'error' as the loss function $\mathcal{L} : \mathbb{R}^p \times \mathbb{R}^n \to [0, +\infty)$, where p represents the dimension of the parameter vector θ. The loss function is the so-called 'metric' that determines the performance of the neural network for any given parameter θ - minimising the value of $\mathcal{L}(\theta, \hat{\mathbf{y}})$ with respect to θ is crucial in the optimisation process. As such, the definition of the loss function is an integral step in defining the optimisation problem.

The definition of the loss function usually comprises of some constraints to the model, and how the output of the neural network performs against them. This could take on physical signif-

icance, such as in *physics-informed neural networks (PINNs)* where physical constraints naturally arise. In situations where training data with a *ground truth output* is provided, typically defined by $\mathbf{y}$, the loss function is defined to reflect the difference between the predicted output $\hat{\mathbf{y}}$ by the neural network and $\mathbf{y}$.

Of course, the precise definition of the loss function depends on the optimisation problem. For example, a regression problem would employ a loss function different from a classification problem. Even within the class of optimisation problems, different loss functions provide varying perfomances for convergence. We introduce several common loss functions used in regression and classification problems in Table 1, where we define $\mathbf{y} = (y_1, \ldots, y_n)^T$, $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_n)^T$ and $\epsilon > 0$ for entry 2.

	Problem	Loss Function	Definition
1	Regression	Absolute Value	$\frac{1}{n}\sum_{i=1}^{n} \lvert y_i - \hat{y}_i \rvert$
2	Regression	ϵ-insensitive	$\frac{1}{n}\sum_{i=1}^{n} \max\left(\lvert y_i - \hat{y}_i \rvert - \epsilon, 0\right)$
3	Regression/Classification	L^2-norm	$\frac{1}{n}\sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$
4	Classification	Hinge	$\frac{1}{n}\sum_{i=1}^{n} \max\left(1 - y_i\hat{y}_i, 0\right)$
5	Classification	Logistic	$\frac{1}{n}\sum_{i=1}^{n} \left(\ln 2\right)^{-1} \ln(1 + e^{-y_i\hat{y}_i})$
6	Classification	Cross-Entropy	$-\frac{1}{n}\sum_{i=1}^{n} y_i \ln \hat{y}_i$

Table 1 Commonly used loss functions $\mathcal{L}(\theta, \hat{\mathbf{y}})$ for regression and classification problems. We note that these loss functions assume the availability of training data with outputs $\mathbf{y}$ - in general, not all optimisation problems involving neural networks have access to this and other loss functions are defined to measure a neural network's performance.

Given the context of optimisation problems, we want to find the optimal value of θ (which shall be defined as $\hat{\theta}$), giving the global minimum of $\mathcal{L}(\theta, \hat{\mathbf{y}})$. Thus, choosing a loss function that can produce convergence to the global minimum efficiently is of utmost importance. This means that convex functions (which, by definition, can only have one minimum, the global minimum) are generally favoured over non-convex functions, which have multiple (local) minima that an optimisation algorithm might get stuck in. We note that only (6) from Table 1 is non-convex, which will require more sophisticated methods to more efficiently locate the global minimum. Such methods are discussed in Section 2.3.1.

2.3 Optimisation Techniques

Having now introduced the general structure of optimisation problems involving neural networks, we turn our focus towards the optimisation process, in particular, training the neural network and producing the optimum parameters $\hat{\theta}$ that minimise the loss function. This typically involves a method called *gradient descent* that utilises the gradient of $\mathcal{L}$ to iteratively generate stronger approximations of $\hat{\theta}$. Section 2.3.1 elaborates on this process, while Section 2.3.2 discusses the methods in which gradients can be efficiently and accurately calculated in the context of neural networks.

2.3.1 Gradient Descent

As described in Section 2.2.3, we want to find the parameters θ that minimise $\mathcal{L}$. In essence, we want to find $\hat{\theta} = \operatorname{argmin}_\theta \mathcal{L}(\theta, \hat{\mathbf{y}})$. The most conventional strategy to achieve this is to perform *gradient descent*: that is, to use the gradient of $\mathcal{L}$ with respect to some θ, say θ_n, to calculate the next iteration θ_{n+1}, which will then be used to calculate $\hat{\mathbf{y}}$ and therefore the value of $\mathcal{L}(\theta_{n+1}, \hat{\mathbf{y}})$.

This generates a sequence $(\theta_n)_N$, ideally with $N = \infty$, but usually up to some N determined by a stopping condition (typically a hard limit on the number of iterations or when the difference in $\mathcal{L}$ evaluated between two consecutive estimates of $\hat{\theta}$ is sufficiently close).

In order to represent the dependence of a given θ_n, we rewrite the loss function as $\mathcal{L}(\theta, \hat{\mathbf{y}}) = \mathcal{L}(\theta_n, \hat{\mathbf{y}})$. The basic gradient descent method can be described as the following iterative process, given an initial guess θ_0, $\forall n < N$:

$$\theta_{n+1} = \theta_n - \eta \nabla \mathcal{L}(\theta_n, \hat{\mathbf{y}}),$$

where $\nabla \mathcal{L}(\theta_n, \hat{\mathbf{y}})$ is the vector of gradients of $\mathcal{L}$ with respect to each parameter in θ_n, and η represents the *learning rate* of the algorithm. This algorithm intuitively makes sense: $\nabla \mathcal{L}(\theta_n, \hat{\mathbf{y}})$ points toward the steepest ascent. By subtracting a multiple of the learning rate from it, we try to find new parameters θ_{n+1} that give $\mathcal{L}(\theta_{n+1}, \hat{\mathbf{y}}) < \mathcal{L}(\theta_n, \hat{\mathbf{y}})$ [7].

There are several conditions on η and $\mathcal{L}$ to be fulfilled for the basic gradient descent algorithm to converge to $\hat{\theta}$. In particular, we require that $\nabla \mathcal{L}$ is Lipschitz continuous (with constant L) and convex for the convergence of $(\theta_n)_\infty$ to $\hat{\theta}$ (the proof of which can be found in Appendix A). It is also helpful to have the learning rate $\eta := \frac{1}{L}$, which has a linear convergence at a rate proportional to $\frac{1}{L}$ [8].

In practice, most loss functions are Lipschitz continuous and convex, and thus the gradient descent algorithm will converge. However, there are several considerations that might affect the effectiveness of the basic gradient descent algorithm:

1. In optimisation problems where training data with ground truth outputs are available, if the training data is sufficiently large, the computation of gradients (no matter the algorithm, the most efficient of which are discussed further in 2.3.2 and 3.6.1) might be too costly. A stronger method of calculating gradients in such a context would be to conduct *stochastic gradient descent* by choosing a random subset of the training data to evaluate in the loss function. We note that each loss function $\mathcal{L}$ can be described as finding the average of the errors of each training point, say $\mathcal{L}(\theta_n, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, \hat{y}_i)$. As such, we can think of $\mathcal{L}$ as the *expected loss* of the function l at each point i. Suppose we define some discrete random variable K with $\mathrm{P}(K = j) = \frac{1}{n}$ for $j = 1, \ldots, n$. It can be easily proven that for any $k \ll n$, we can estimate $\mathcal{L}$ by taking the average of the loss function l at training points $\{y_{K_1}, \ldots, y_{K_k}\}$, where $\{K_1, \ldots, K_k\}$ are i.i.d. copies of K [9]. These subsets of data points can alternatively define a partition on the whole dataset and be trained individually - these are known as *mini-batches*. This greatly reduces the computational cost of gradient calculations for gradient descent in particularly large training datasets.

2. The learning rate $\eta = \frac{1}{L}$ is idealistic in practice. Such a small (and constant) learning rate, while guaranteeing convergence, takes far too long to converge. However, having too large of a learning rate might not allow the sequence $(\theta_n)_\infty$ to converge. As such, more effort is required to estimate the *hyperparameter* η. There are multiple methods that attempt to remedy this issue. A straightforward method is to decrease the learning rate according to the size of the recalculation of the loss function. Suppose in a certain iteration θ_n, the current learning rate is $\eta = 1$. A possible algorithm is to check whether $\theta_{n+1} = \theta_n - \eta \nabla \mathcal{L}(\theta_n, \hat{\mathbf{y}})$ satisfies $\mathcal{L}(\theta_{n+1}, \hat{\mathbf{y}}) \leq \mathcal{L}(\theta_n, \hat{\mathbf{y}})$. If not, this suggests that the current learning rate overcompensates too much - and we can halve the learning rate and check whether the equality is achieved. This reduces the need to calculate a 'perfect' constant for η, instead treating η as some function of t. Other (more sophisticated) methods include adaptive gradient techniques such as `AdaGrad`, `RMSProp` and `Adam`.

3. As seen in Section 2.2.3, there exists some non-convex loss functions (such as the commonly used cross-entropy loss function in classification problems). This complicates the

gradient descent process, since the choice of starting parameters θ_0 would determine whether the sequence $(\theta_n)_\infty$ would provide a local or global minimum of $\mathcal{L}$. There is a need to 'escape' local minima so that the algorithm can give $\hat{\theta}$ as required. The stochastic gradient descent mentioned above actually works to solve this issue: the stochasticity creates *noisy gradients* that helps to nudge θ_n out of problematic spots [7].

With all these considerations in mind, a popular choice for gradient descent algorithms is the *adaptive moment estimation* (Adam) algorithm. It makes use of previous gradients in the algorithm to determine the next parameter θ_{n+1} as well as a separate learning rate for each parameter, scaled to the gradients of the other parameters. With such a combination of methods, it has been empirically the most robust on real datasets [7].

2.3.2 Automatic Differentiation and Discrete Backpropagation

Having discussed the importance of deriving parameters $\hat{\theta}$ that produce the global minimum of $\mathcal{L}$ through gradient descent, it becomes imperative to use algorithms that derive gradients and derivatives efficiently.

There are a number of methods that do so with varying computational costs. The conventional ideas brought forward by divided differences tend to incur large errors in computation, making gradient calculations inaccurate and thus unreliable. Calculating symbolic expressions of gradients, on the other hand, tends to be rather memory and time consuming [10]. The computational cost of such a method is further compounded by the fact that functions derived from neural networks tend to be rather complicated in nature. A method that has proven to be time and memory efficient is *automatic differentiation* (AD).

There are two distinct types of AD. *Forward-mode AD* computes the derivative of the function along with the function evaluation. A familiar (albeit limited) concept that implements this is the commutative ring of *dual numbers* $\mathbb{D}$ over $\mathbb{R}$ generated by 1 and ϵ, where $\epsilon^2 = 0$. In the context of neural networks and loss functions, instead of simply evaluating functions with the value of θ_n, evaluating them using dual numbers allows both $\mathcal{L}(\theta_n, \hat{\mathbf{y}})$ and its derivative with respect to θ_n to be calculated at once. *Reverse-mode AD*, and in particular *backpropagation*, instead evaluates the function first before propagating the derivatives from the output to the input.

Computationally speaking, reverse-mode AD is a 'cheaper' alternative to forward-mode AD. We note that this is especially true for neural networks, which typically have a large number of parameters (that is, θ has dimension $p \gg 1$). This means that, using forward-mode AD, we need to calculate p forward passes to find the gradients of each parameter [11]. This section aims to discuss how backpropagation, in particular, makes for a more efficient method.

Backpropagation can be used to derive the gradients of $\mathcal{L}(\theta, \hat{\mathbf{y}})$ with respect to θ. Here we define $\theta := (\theta^1, \ldots, \theta^p)$. Suppose that we calculate $\hat{y}_i = g(\theta)$, where g is made up of a composition of m elementary operators $g_1, \ldots, g_m$. Additionally, suppose that we know the value of $\frac{\partial \mathcal{L}}{\partial \hat{y}_i}$, which we will represent with $\overline{\hat{y}_i}$. These assumptions are valid: First, the functions that define the neural network are (in most circumstances) differentiable. Next, the derivative of the loss function with respect to $\hat{y}_i$ is readily available in practice, as $\mathcal{L}$ is typically explicitly defined and differentiable. This process can thus be conducted using the following algorithm:

1. *Forward Pass.* Evaluate g with the given values of θ, recording the intermediate values after each g_l. Thus, we receive a set of values $\{v_1, v_2, \ldots, v_m\}$, where $v_m = \hat{y}_i$. These intermediate values (and the functions that connect them) can be represented by a directed acyclic graph (DAG) (an example of which is found in Figure 3) that flows (via the functions g_l) from each of the θ^j.

2. *Chain Rule Application.* By the chain rule, we can easily calculate the values of $\overline{v_l}$, $q = 1, \ldots, m$. If v_l contributes directly to some v_q via the function g_q, then

$$\overline{v_l} = \overline{v_q} \cdot \frac{\partial v_q}{\partial v_l} = \overline{v_q} \cdot \dot{g}_q(v_l). \tag{1}$$

3. *Backward Pass.* Suppose we want to consider the value of $\overline{\theta^j}$. Given $\overline{\hat{y}_i}$, we can work our way up the DAG to θ^j, calculating the intermediate derivatives $\overline{v_q}$ along the way. This can be done for all θ^j.

An example of such an algorithm can be found in Figure 3. Such an algorithm to calculate derivatives only requires a single forward pass and a reverse pass. This means that the algorithm scales well even for more complex models with high dimensional θ [12].

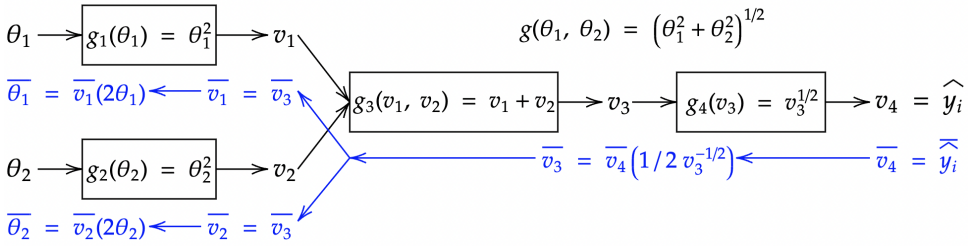


Figure 3 A directed acyclic graph representing the intermediate values and the derivatives of $\mathcal{L}$ with respect to each v_l. The function used to illustrate this property is $g(\theta_1, \theta_2) = (\theta_1^2 + \theta_2^2)^{\frac{1}{2}}$. Note that the θ_1 in the figure represents θ^1 in the algorithmic process described above for clearer notation. The forward pass is represented in black, while the backward pass is represented in blue.

Much work has been put into the improvement of discrete backpropagation algorithms, including the use of *static single assignment* (SSA) forms to increase efficiency [13]. These advanced implementations of backpropagation can be found in many Julia modules such as `Zygote`, which computes $\overline{v_l}$ as defined in (1) as *pullbacks* under the hood of `Zygote.gradient()`.

2.4 Example: Regression

Now we turn our attention to two examples that utilise neural networks. We consider *regression* first, which is a technique that approximates the data and estimates the relationship between variables. Linear regression and polynomial regression are basic examples of regression discussed at the undergraduate level in statistical modelling and numerical analysis courses. With the help of neural networks, we take these ideas further and explore nonlinear regression. Some of the ideas presented in this section are taken from [3].

Consider a simple example of a neural network that performs regression-based analysis on $y = \sin x$. Assume that we are given n data points between $[-\pi, \pi]$, denoted by $\mathbf{x}$. We try to minimise the difference between the predicted output $f(\mathbf{x})$ and the target output $\sin(\mathbf{x})$, where f is the neural network being considered.

We now present an implementation of this neural network in Julia. The file containing all the code below can be found in the project's GitHub repository at `Neural_Network.ipynb`.

We first need to import some packages for later use.

```
using Lux, Random, Optimization, OptimizationOptimisers,
      ComponentArrays, Zygote, Plots, LinearAlgebra
```

We then define the data. Let us select 100 evenly spaced points from $[-\pi, \pi]$ and store them in `x`. We will also define the target output in `y`.

```
x = range(-pi, pi; length = 100)
y = sin.(x)
```

To create the neural network model, a Julia package `Lux` is usually used. For the purpose of this example, we use three layers, all of which are dense. For 100 data points, 10 neurons in the hidden layer would be enough. We can create the composition of such layers using the `Chain` command. The input and hidden layers will have ReLU as the activation function.

```
model = Chain(
    Dense(1 => 10, relu),
    Dense(10 => 10, relu),
    Dense(10 => 1)
)
```

After that, we implement the loss function based on `norm` from the `LinearAlgebra` package. For simplicity, we consider our loss function to be the L^2-norm of $f(\mathbf{x}) - \sin(\mathbf{x})$, i.e., the square root of the sum of squares of differences between the predicted output and the target output.

```
function regression_loss(ps, (model, st, (x, y)))
    return norm(vec(model(x', ps, st)[1]) - y)
end
```

We now turn our attention to training the neural network. This is essentially equivalent to solving an optimisation problem. We can use `setup` from the package `Lux` and `MersenneTwister` from the package `Random` to create a random initial guess of the parameters.

```
ps, st = Lux.setup(MersenneTwister(), model)
```

To define the optimisation problem, we use the `Optimization` package. Note that we need to wrap `ps` in an array type supplied in the `ComponentArrays` package, as the `Optimization` package requires optimisation to be over an array type. To calculate the gradients needed, `AutoZygote` from the `Zygote` package is used for automatic differentiation.

```
prob = OptimizationProblem(OptimizationFunction(regression_loss,
       Optimization.AutoZygote()), ComponentArray(ps), (model, st, (x, y)))
```

To solve the optimisation problem, we use the gradient descent method supplied by the aforementioned `Adam` in `OptimizationOptimisers` with appropriate parameters. We fix the learning rate at a constant $\eta = 0.03$ in this scenario and let the maximum number of iterations be $N = 250$.

```
ret = solve(prob, Adam(0.03), maxiters = 250)
```

Lastly, we visualise our results by plotting supplied in the `Plots` package:

```
plot(x, y, label = "Target")
plot!(x, vec(model(x', ret.u, st)[1]), label = "Predicted")
```

The result of the regression is found in Figure 4. This is a pretty impressive result that demonstrates the power of neural networks in terms of regression.

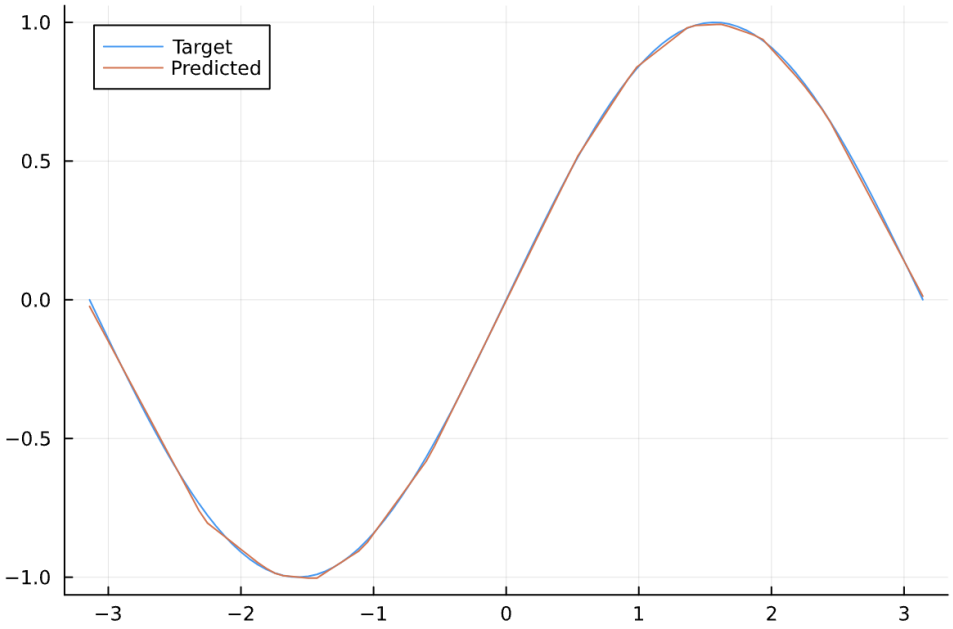


Figure 4 Regression of $y = \sin x$.

2.5 Example: Classification

Neural networks can also be applied to *classification* problems. They involve using a neural network to assign labels to input data based on some patterns learnt during the training process. A classic example is digit recognition based on the *Modified National Institute of Standards and Technology database* (MNIST database). This database presents a large collection of handwritten digits given in pixels. The task is to learn from the database and classify the digits. As we can see, it is basically another form of regression; therefore, neural networks would be a suitable model.

A sample image from MNIST is shown below. We immediately see that it is a 6. We want our model to recognise it as a 6 as well, that is, the model thinks that the probability that this digit is a 6 is higher than that of any other digit.

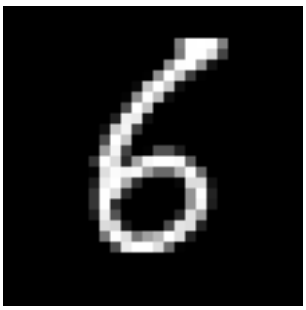


Figure 5 A sample image from MNIST.

We now present a simple implementation of this neural network in Julia. At this point, we will not dive into the details too much; the details will be discussed in Section 4 where more sophisticated methods are used to perform a stronger analysis. Some of the ideas presented here are taken from [14]. The file containing all the code below can be found in the project's GitHub repository at `Neural_Network.ipynb`.

We first need to import some packages for later use. For this task, we introduce another

package `Flux`, which is similar to the aforementioned `Lux`. We also want to import the MNIST database from the `MLDatasets` package.

```
using Flux
using MLDatasets: MNIST
```

First, we extract data from the database. We split the data into the *training set* and the *test set*, where the former is used to train the model and the latter is used to evaluate its performance on unseen data.

```
x_train, y_train = MNIST(split=:train)[:]
x_test, y_test = MNIST(split=:test)[:]
```

We then need to 'flatten' the images so that they become vectors and would be easier to process. For labels, i.e., the input images, we need to use a technique called *one-hot encoding* to transform categorical variables into *one-hot* form, i.e., a group of bits with one and only one `1` and all other bits `0`.

```
x_train_flat = Flux.flatten(x_train)
x_test_flat = Flux.flatten(x_test)
dataset = [(x_train_flat, Flux.onehotbatch(y_train, 0:9))]
```

After that, we define the neural network for our use. The input layer consists of $28 \times 28 = 784$ neurons. We employ two hidden layers, each of which has $1/4$ the number of neurons compared to the previous layer. Finally, the output layer consists of 10 neurons corresponding to the digits 0 to 9. As usual, all layers are dense and the ReLU activation function is used.

```
model = Chain(
    Dense(784, 196, relu),
    Dense(196, 49, relu),
    Dense(49, 10)
)
```

The loss function we are using is a modified form of the *cross-entropy*. It measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

```
loss(x, y) = Flux.Losses.logitcrossentropy(model(x), y)
```

We are now ready to train our neural network. For this task, it is better to train the neural network several times, or *epochs*. To optimise the performance, we shall use 25 epochs. We use a smaller learning rate $\eta = 0.003$ here.

```
@time for epoch in 1:25
    Flux.train!(loss, Flux.params(model), dataset, Adam(0.003))
end
```

We see that the training process takes 13 seconds. By inverting the one-hot encoding using `onecold` and comparing with the target output, we might see the test accuracy given as

```
sum(Flux.onecold(model(x_test_flat)) .== (y_test .+ 1)) / length(y_test)
```

where `+1` is added as Julia uses 1-based indexing and `y_test` starts at 0. We note that the result is normally around 80% - satisfactory but not perfect. This inspires us to consider a better model that incorporates other structures such as neural differential equations, as detailed in the following sections. In Section 4, we will show that implementing a network with a neural ODE improves the test accuracy to around 95%.

3 Neural ODEs

3.1 Motivation

The method of *neural ordinary differential equations* (neural ODEs) was first proposed in a 2018 paper titled "Neural Ordinary Differential Equations" [1]. This paper won the Best Paper Award at NeurIPS in the same year. The inspiration for neural ODEs came from the observation of a specific neural network model called the *residual neural network* (ResNet) which was introduced by a Microsoft research team in 2015 [15]. Unlike traditional neural networks, ResNet incorporates residual connections by adding a *residual term* to the output of each layer, as shown in the equation below:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + \sigma\left(\mathbf{W}_t\mathbf{h}_t + \mathbf{b}_t\right). \tag{2}$$

The second term in equation (2) is in the form of a typical neural network. To obtain the state at layer $t+1$, we perform a calculation using the current state $\mathbf{h}_t$, the weight matrix $\mathbf{W}_t$ and the bias vector $\mathbf{b}_t$ at layer t, and an activation function σ. The term $\mathbf{h}_t$ here is the residual term, representing the identity of the hidden state in the layer t.

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f\left(\mathbf{h}_t, \theta_t\right). \tag{3}$$

We write equation (2) in the form of equation (3), where f is a function that depends on the state of layer t and some parameters θ_t related to this layer (which belong to the larger parameter vector θ). Then, by transforming equation (3), we get

$$\frac{\mathbf{h}_{t+1} - \mathbf{h}_t}{1} = f\left(\mathbf{h}_t, \theta_t\right),$$

which gives

$$\left.\frac{\mathbf{h}_{t+\Delta t} - \mathbf{h}_t}{\Delta t}\right|_{\Delta t=1} = f\left(\mathbf{h}_t, \theta_t\right).$$

Such a form inspires us to make Δt infinitesimally small, allowing us to transform this discrete form into a continuous one:

$$\lim_{\Delta t \to 0} \frac{\mathbf{h}_{t+\Delta t} - \mathbf{h}_t}{\Delta t} = f(\mathbf{h}_t, \theta_t, t).$$

This motivates us to consider ordinary differential equations, which can be written as

$$\frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = f(\mathbf{h}(t), \theta, t).$$

This is the most essential equation for defining a neural ODE. The function f here on the right is a function derived from a neural network, with θ representing the parameters of that network. In practice, however, f does not necessarily need to be a function derived from a neural network. By convention, since there is no clear concept of hidden layers in neural ODEs, we usually write this function as

$$\dot{z} = f(z(t), \theta, t). \tag{4}$$

3.2 Comparing Neural ODEs with ResNet

In this section, we compare neural ODEs as defined above with ResNets. Within a ResNet with a finite number of layers, the transition between states from one layer to the next is discrete, and each layer has its own parameters to alter the current state. However, in an ODE network, it defines a continuous vector field which essentially represents a neural network with infinitely many layers. The state change can thus be interpreted as a *flow* within this vector field.

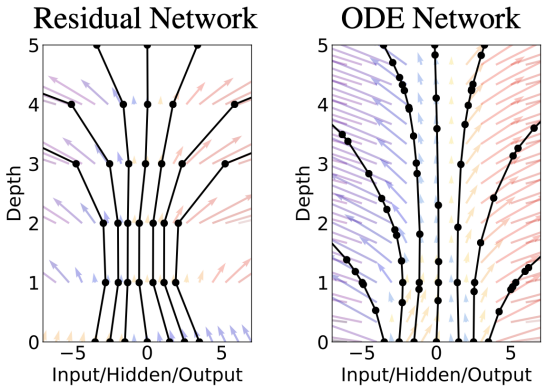


Figure 6 A residual network typically represents discrete transformations, whereas an ODE network defines a continuous vector field. This figure is taken from [1].

Note that the points in both graphs in Figure 6 represent the evaluation points. In the residual network, these points correspond to different layers, whereas in the ODE network, the points are taken according to an increasing sequence of time stamps $(t_i)_{i=1,\dots,N}$, and $(z(t_0), t_0)$ is the initial condition. The values given by the model at these points are $z(t_i)_{i=1,\dots,N}$. These can be obtained by solving the ODE:

$$z(t_i) = \text{ODESolve}\left(z(t_0), f, t_0, t_i\right), \quad i = 1, \dots, N.$$

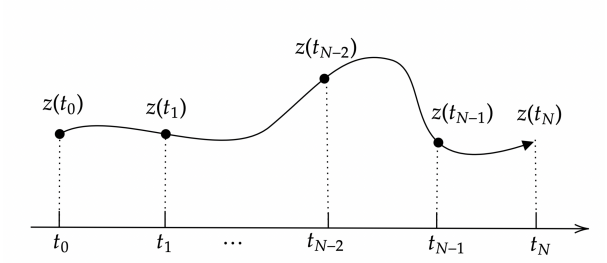


Figure 7 Trajectory for obtaining the output value.

Then the time span here for defining a neural ODE would be $[t_0, t_N]$. Since all these points are in the same space, each output has the same dimension as the input vector.

Unlike neural networks, where depth refers to the number of layers, the depth of neural ODEs can be considered as the number of observation points. For neural networks, more layers can help the model fit the training dataset better in general. Similarly, in neural ODEs, adding more observation points is the most direct way to help the model fit the training set better.

3.3 Various Neural ODE Models

3.3.1 Augmented Neural ODEs

From the previous part, we find that the input space and the output space of a neural ODE always have the same dimension. It poses a problem when simulating certain types of data. For instance, if we want to use a neural ODE to simulate a sine function, where we want the model to output $\sin x$ for a given input x, we might choose to use a simple neural ODE with a one-dimensional input space. In that case, for the sine function, if we input $\frac{\pi}{2}$ and $\frac{3\pi}{2}$, we would expect the model to output 1 and -1, respectively, since the idea of the neural ODE model here is to follow a trajectory from the input x and try to reach the expected output value $\sin x$ after some time t (t can be either fixed or trainable as the parameter θ). And the function f used to define a neural ODE is Lipschitz continuous [1], meaning that the different trajectories cannot intersect. This makes it difficult for a simple neural ODE to handle such problems.

This is why we might use the *augmented neural ODE model* [16]. The idea is to avoid conflicting trajectories by adding more dimensions to the input. In practice, we usually concatenate some zeros to the input and then drop the excess dimensions to the output [17]. This method allows the model to learn a more complex function in a higher-dimension space.

3.3.2 Neural ODEs with Scientific Machine Learning

As mentioned above, we usually define the right-hand side of (4) with a neural network represented by a function f. However, if we have some prior knowledge about the model, we can introduce some interpretable terms to define f, and use a neural ODE to approximate the remaining part. This means that we can represent the model as $\dot{z} = g(u,t) + f(u,\theta,t)$, where we already know g (if $g = 0$, it is just the neural ODE discussed earlier), and approximate f with a neural network. For example, in a Lotka-Volterra system, the dynamic would be

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha x - \beta xy,\\ \frac{\mathrm{d}y}{\mathrm{d}t} &= -\delta y - \gamma xy\end{aligned}$$

for some parameters $\alpha, \beta, \gamma, \delta$. If we only have some (but not all) information about the model, we can state these terms explicitly as g and add a neural network f as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha x \\ -\delta y \end{pmatrix} + f(x,y),$$

where α and δ are known constants and f is an unknown function, then we can use a neural network only to approximate the function f.

Recent research has shown that integrating information from physical laws and scientific models is particularly useful when there is a limited amount of training data. Additionally, it has the potential to automate the discovery of explicit dynamic equations [18].

3.4 Solving ODEs Numerically

As with all neural networks, when using neural ODEs as part of a larger network, we are required to optimise the parameters to train the network. This requires the calculation (and backpropagation) of gradients throughout the network, including through the neural ODE. For

standard neural networks, the training methods are standard as described in Section 2.3.2. On the other hand, the calculation of gradients in a neural ODE requires an additional step: to solve the ODE in question.

Perhaps the most straightforward approach to numerically approximate a solution of ODE $\dot{z} = f(z(t), \theta, t)$, on some interval $[a, b]$, is to look at the Taylor series expansion of z. We first discretise the system and consider the points $t_i = a + ih \in [a, b]$, where $i = 0, 1, \ldots, N-1$ and $h = \frac{b-a}{N}$. We can therefore use the Taylor series expansion iteratively, centred at each point t_i, to find the value of the next point t_{i+1}. This is known as *Euler's method*.

Definition 3.1 (Euler's method)**.** Given points $t_i = a + ih \in [a, b]$, $i = 0, 1, \ldots, N-1$, $h = \frac{b-a}{N}$, *Euler's method* is an iterative process to solve the initial value problem $\dot{z} = f(z(t), \theta, t)$, $z(t_0) = z_0$, where

$$\begin{aligned} z(t_0) &= z_0 \\ z(t_{i+1}) &= z(t_i) + f(z(t_i), \theta, t_i)h + \mathcal{O}(h^2), \quad \forall i = 1, \ldots, N-1. \end{aligned}$$

The final term in the iterative process is known as the *error term*. We do not include this in the numerical approximation. We note several characteristics of Euler's method:

- It is a *first order* approximation, that is, the error of the approximation to the true value is at most linear with respect to h. This can be seen from the iterative step as $\frac{z(t_{i+1})-z(t_i)}{h} = f(z(t_i), \theta, t_i) + \mathcal{O}(h)$.
- It is a *1-stage* method, in which we only require 1 computation of f at a single point, namely t_i in the iterative step. In a perhaps unintuitive way, we can redefine the iterative step as

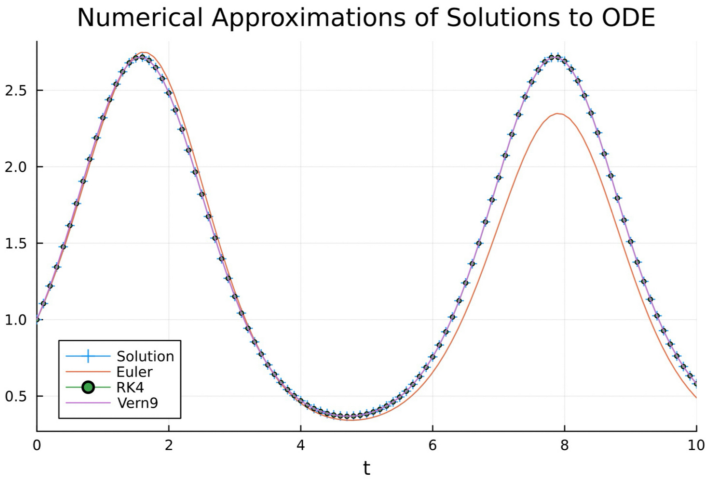
$$\begin{aligned} z(t_{i+1}) &= z(t_i) + hb_1k_1, \\ k_1 &= f\left(z(t_i) + ha_{11}k_1, \theta, t_i + hc_1\right), \end{aligned}$$

  where $a_{11} = 0, b_1 = 1, c_1 = 0$.

These characteristics point toward possible limitations of Euler's method in approximating solutions to initial value problems. The errors of each iterative step can accumulate and result in large errors further away from the initial value. This necessitates having a sufficiently small h for an accurate computation, increasing the computational cost. Furthermore, the assumption of evaluating the derivative (and thus f) in the current step t_i might not be an accurate and reliable method to approximate the next step t_{i+1}. These limitations call for more robust and computationally efficient algorithms to approximate such solutions. One such class of methods, which extends fairly nicely from Euler's method, are the *Runge-Kutta methods* [19].

Definition 3.2 (Runge-Kutta methods)**.** Given points $t_i = a + ih \in [a, b]$, $i = 0, 1, \ldots, N-1$, $h = \frac{b-a}{N}$, we define an *s-stage Runge-Kutta method* that solves the initial value problem $\dot{z} = f(z(t), \theta, t)$, $z(t_0) = z_0$ with iterative step

$$\begin{aligned} z(t_{i+1}) &= z(t_i) + h\sum_{j=1}^{s} b_j k_j, \\ k_j &= f\left(z(t_i) + h\sum_{l=1}^{s} a_{jl}k_l, \theta, t_i + hc_j\right), \end{aligned}$$




```
Sum of Errors of Euler approximation: 9305.807927541871
Sum of Errors of RK4 approximation: 2.6304972084667533e-5
Sum of Errors of Vern9 approximation: 6.4161453927624734e-12
```

Figure 8 A comparison of ODE Solvers used to solve the initial value problem $\dot{z} = \cos(t)z$, $z(0) = 1$. The errors were calculated by adding the absolute differences of the actual solution and the approximations at discrete times $\{0, 0.1, \ldots, 9.9, 10.0\}$.

where the constants in the matrix/vectors $\mathbf{A} = (a_{jl}), \mathbf{b} = b_j, \mathbf{c} = c_j$ are derived from the Taylor series expansions of $z(t_i)$. These constants are usually represented in a *Butcher's tableau*:

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

The Runge-Kutta methods can be categorised into two types: *explicit*, where the matrix $\mathbf{A}$ defined by the Butcher tableau of the method is lower triangular, and *implicit*, where there is no restriction on the structure of the matrix $\mathbf{A}$. The most common example of a higher order Runge-Kutta method is the widely studied *RK4 method* [20]. It is a 4-stage explicit Runge-Kutta method and is a fourth-order approximation. Its Butcher's tableau is specified as

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

The characteristics specified above make the RK4 method a much stronger algorithm than Euler's method. Given a fixed h, while more stages need to be calculated at each iterative step, we note a much better approximation of the solution. Of course, many more sophisticated numerical methods have been developed (such as `Vern9` [21]) to tackle the approximation of such solutions accurately. These methods give differing levels of accuracy, as seen in Figure 8 (the code for which can be found in Appendix B), with higher order approximations clearly giving better approximations. Many of these algorithms can be implemented via packages such as `DifferentialEquations` in Julia.

With such a wide range of numerical methods, there is much consideration in which method is the best. One such consideration is the *absolute stability* of the numerical method. This is

an especially important consideration when dealing with stiff ODEs, where the eigenvalues of the linear approximations of f are all negative, with a large ratio between the maximum and minimum real values of the set of eigenvalues. These give rise to numerical instabilities to the numerical methods introduced thus far - essentially requiring an extremely (and thus inefficiently) small value of h. A possible alternative to this is to use implicit methods instead. In the context of the Runge-Kutta methods, this means removing the restriction that the matrix $\mathbf{A}$ must be lower triangular. It can be proven that such methods have an *unbounded region of absolute stability*, as compared to explicit Runge-Kutta methods which have *bounded regions of absolute stability* [19]. In the context of solving ODEs numerically, this means that larger values of h still give accurate approximations of the actual solution. There is, however, a trade-off in using such implicit methods. The derivation of $k_1, \ldots, k_s$ in implicit methods requires solving the s equations simultaneously, instead of in succession as seen in explicit methods. This requires more computational power and might make computation inefficient.

Through the Runge-Kutta methods, we derive equations that iteratively define evaluations of z at every time point $t_N, \ldots, t_0$. This allows us to easily differentiate and perform backpropagation throughout the solution of the neural ODE as prescribed in Section 2.3.2.

3.5 Backpropagation in Neural ODEs

Consider a neural ODE defined by a neural network f. This gives us the differential equation of the system $\dot{z} = f(z(t), \theta, t)$. There are two predominant methods for determining gradients through the neural ODE, known as *Discretise-then-Optimise* (DO) and *Optimise-then-Discretise* (OD) [22].

3.5.1 Discretise-then-Optimise (DO)

The Discretise-then-Optimise (DO) method can be described in the following steps:

1. *Solve the neural ODE numerically with current parameters.* This involves choosing an initial 'guess' of parameters θ and solving the neural ODE with the guessed parameters. The Runge-Kutta methods described in Section 3.4 are most commonly used to achieve this.
2. *Perform backpropagation with the chain rule with respect to parameters θ.* This is done using the methods detailed in 2.3.2, going from the final time t_N of the neural ODE to the initial time t_0.

However, we see a trade-off between the accuracy of the solution and the computational cost of backpropagation through this method. With more time steps to calculate in the neural ODE, we improve the accuracy of the ODE solver's approximation to the true solution of the ODE problem. However, this requires many more calculations in both the forward pass and backward pass (through the ODE solver), thus requiring more time and computational power to calculate these values. This motivates us to use a method that calculates gradients without needing to store all the intermediate states while doing backpropagation.

3.5.2 Optimise-then-Discretise (OD)

In contrast to DO, the Optimise-then-Discretise (OD) method can be described in the following steps:

1. *Formulate the continuous adjoint equations of neural ODE.* The resulting problem is a backwards differential equation known as the *adjoint ODE*.

2. *Solve the adjoint ODE.* Once the adjoint ODE has been formulated, the numerical methods for solving ODEs (as seen in 3.4) can be utilised to find the relevant gradients required.

We note that the OD method does not require to store any intermediate states through backpropagation, because it makes use of the *adjoint method*, in comparison to the one described in Section 2.3.2. This method is described below.

3.6 Adjoint Sensitivity Method

The adjoint sensitivity method, first proposed by Pontryagin in 1962 [23], offers a way to find the desired gradients by solving another ODE. This method is a special approach to perform backpropagation in neural ODE. Consequently, we do not need to store too many state values for backpropagation, which helps reduce the computational cost.

An important point to note is that in the adjoint method we treat θ as a constant that does not change over time. Therefore, the adjoint method can only be used to solve these specific classes of neural ODEs.

Since there will be N evaluation points and all values at these points contribute to the loss function, we need to obtain the gradient of the parameters with respect to the loss function at each evaluation point; we then aggregate these gradients to get the overall gradient of each parameter for the model.

3.6.1 Continuous Backpropagation

In this section, we provide a proof of the adjoint sensitivity method. The proof references the method from [24]. However, we adjust the choice of the Lagrange multiplier to ensure that the adjoint states adjust in the same direction as outlined in the original neural ODE paper [1]. Additionally, we use clearer notation to illustrate the adjustment process at each evaluation point.

We know that there will be multiple evaluation points $z(t_i)_{i=1,\dots,N}$ on the trajectory and these points are obtained as the output of the model - thus contributing to the loss function. Our first goal is to find the gradient of the loss function with respect to the parameter θ. We can turn this into an optimisation problem:

$$\min_{\theta} \mathcal{L}\left(z(t_1), \dots, z(t_N)\right),$$

$$\text{with} \quad \dot{z} = f(z, \theta, t).$$

Then we introduce a Lagrange multiplier $\lambda(t)$ which is a function of time, and use λ to define the Lagrange cost as

$$\mathcal{J}(\theta) = \mathcal{L}(z(t_1), \dots, z(t_N)) - \int_{t_0}^{t_N} \lambda(t) \left(\dot{z}(t) - f(z(t), \theta, t)\right) \mathrm{d}t.$$

Note that we also impose a constraint on $\lambda(t)$ such that $\lambda(t_N) = \lambda(t_N^+) = 0$. To find the gradient with respect to θ, we introduce a small perturbation to θ to see how it affects the loss function. Therefore, we define the perturbed parameter as

$$\theta^* = \theta + \epsilon\zeta,$$

where $\zeta = \zeta(t)$ is the perturbation and ϵ is a small perturbation factor. Consequently, the value $z(t)$ will also change according to θ. We define the perturbed trajectory $z^*(t)$ as

$$z^*(t) = z(t) + \epsilon\eta(t) + \mathcal{O}(\epsilon^2),$$

and we assume that $z(t_0) = z^*(t_0)$. The difference in the Lagrange cost due to the added perturbation is then:

$$\begin{aligned}\mathcal{J}(\theta^*) - \mathcal{J}(\theta) = &(\mathcal{L}(z^*(t_1), \dots, z^*(t_N)) - \mathcal{L}(z(t_1), \dots, z(t_N))) \\ &- \int_{t_0}^{t_N} \lambda(t)(\dot{z}^*(t) - \dot{z}(t))\mathrm{d}t \\ &+ \int_{t_0}^{t_N} \lambda(t)(f(z^*(t), \theta^*, t) - f(z(t), \theta, t))\mathrm{d}t. \end{aligned} \tag{5}$$

The first term of (5) can be written as

$$\mathcal{L}\left(z(t_1), \dots, z(t_N)\right) - \mathcal{L}\left(z^*(t_1), \dots, z^*(t_N)\right) = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial z(t_i)} \epsilon\eta(t_i) + \mathcal{O}\left(\epsilon^2\right).$$

Assuming that $\lambda(t)$ is continuous in each pair of successive time steps $[t_i, t_{i+1}]_{i=0,\dots,N-1}$, and that the value of λ at each discrete point is equal to the right-hand limit at that point, the second term of (5) yields

$$\begin{aligned}-\int_{t_0}^{t_N} \lambda(\dot{z}^*(t) - \dot{z}(t))\mathrm{d}t &= -\sum_{i=1}^{N} [\lambda(t)(z^*(t) - z(t))]_{t_{i-1}^+}^{t_i^-} + \int_{t_0}^{t_N} \dot{\lambda}(t)(z^*(t) - z(t))\mathrm{d}t \\ &= -\sum_{i=1}^{N} [\lambda(t)\epsilon\eta(t)]_{t_{i-1}^+}^{t_i^-} + \int_{t_0}^{t_N} \dot{\lambda}(t)\epsilon\eta(t)\mathrm{d}t + \mathcal{O}(\epsilon^2) \\ &= -\sum_{i=1}^{N} \left(\lambda(t_i^-)\epsilon\eta(t_i) - \lambda(t_{i-1}^+)\epsilon\eta(t_{i-1})\right) + \int_{t_0}^{t_N} \dot{\lambda}(t)\epsilon\eta(t)\mathrm{d}t + \mathcal{O}(\epsilon^2) \\ &= \sum_{i=1}^{N} (\lambda(t_i^+) - \lambda(t_i^-))\eta(t_i)\epsilon + \lambda(t_0)\eta(t_0)\epsilon + \int_{t_0}^{t_N} \dot{\lambda}(t)\epsilon\eta(t)\mathrm{d}t + \mathcal{O}(\epsilon^2).\end{aligned}$$

To transform the third term of (5) using the same method, we first need to get the value of $\dot{\eta}$, and from the definition of $z^*(t)$ we know $\eta(t) = \frac{\mathrm{d}z^*(t)}{\mathrm{d}\epsilon}\big|_{\epsilon=0}$. Then we have

$$\begin{aligned}\dot{\eta}(t) &= \left.\frac{\mathrm{d}z^*(t)^2}{\mathrm{d}\epsilon\mathrm{d}t}\right|_{\epsilon=0} \\ &= \left.\frac{\mathrm{d}}{\mathrm{d}\epsilon}\right|_{\epsilon=0} \frac{\mathrm{d}z^*(t)}{\mathrm{d}t} \\ &= \left.\frac{\mathrm{d}}{\mathrm{d}\epsilon}\right|_{\epsilon=0} f\left(z(t) + \epsilon\eta(t) + \mathcal{O}(\epsilon^2), \theta + \epsilon\zeta, t\right) \\ &= \frac{\partial f}{\partial z}\epsilon\eta + \frac{\partial f}{\partial \theta}\epsilon\zeta.\end{aligned}$$

So the third term of (5) yields

$$\begin{aligned}\int_{t_0}^{t_N} \lambda(f(z^*(t), \theta^*, t) - f(z(t), \theta, t))\mathrm{d}t &= \int_{t_0}^{t_N} \lambda(t)(f(z(t) + \epsilon\eta(t) + \mathcal{O}(\epsilon^2), \theta + \epsilon\zeta, t) - f(z(t), \theta, t))\mathrm{d}t \\ &= \int_{t_0}^{t_N} \lambda(t)\left(\frac{\partial f}{\partial z}\epsilon\eta + \frac{\partial f}{\partial \theta}\epsilon\zeta\right)\mathrm{d}t + \mathcal{O}(\epsilon^2).\end{aligned}$$

So the difference as described in (5) becomes

$$\begin{aligned}\mathcal{J}(\theta^*) - \mathcal{J}(\theta) = &\int_{t_0}^{t_N} \left(\lambda(t)\frac{\partial f}{\partial z} + \dot{\lambda}(t)\right)\epsilon\eta(t)\mathrm{d}t + \lambda(t_0)\eta(t_0)\epsilon \\ &+ \sum_{i=1}^{N}\left(\lambda(t_i^+) - \lambda(t_i^-) + \frac{\partial \mathcal{L}}{\partial z(t_i)}\right)\epsilon\eta(t_i) \\ &+ \int_{t_0}^{t_N} \lambda(t)\frac{\partial f}{\partial \theta}\epsilon\zeta\mathrm{d}t + \mathcal{O}(\epsilon^2). \end{aligned} \tag{6}$$

This motivates us to choose λ as follows to eliminate the first three terms in the difference:

$$\dot{\lambda}(t) = \lambda(t)\frac{\partial f}{\partial z} \qquad \lambda(t_i^-) = \lambda(t_i^+) + \frac{\partial \mathcal{L}}{\partial z(t_i)}, \quad i = N, N-1, \ldots, 1.$$

Now we introduce the *adjoint state* $a(t)$ as the solution to these N initial value problems:

$$\dot{a}(t) = a(t)\frac{\partial f}{\partial z}, \tag{7}$$

$$a^{\text{initial}}(t_i) = a(t_i) + \frac{\partial \mathcal{L}}{\partial z(t_i)}, \quad i = N, N-1, \ldots, 1.$$

Figure 9 illustrates the graph of $a(t)$. We see that the value of the adjoint state needs to be adjusted at each evaluation point.

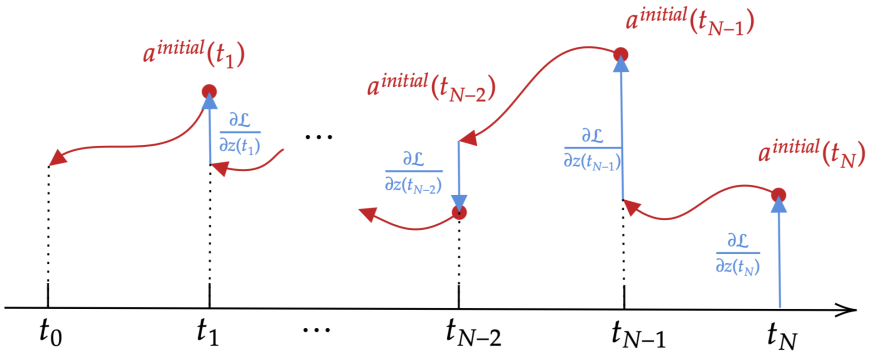


Figure 9 The graph of the function $a(t)$.

By substituting this $a(t)$ as λ in (7), we get

$$\mathcal{J}(\theta^*) - \mathcal{J}(\theta) = \int_{t_0}^{t_N} \lambda(t)\frac{\partial f}{\partial \theta}\epsilon\zeta\mathrm{d}t + \mathcal{O}(\epsilon^2).$$

Recall the assumption that $\zeta(t) = \zeta$. Then the gradient of the Lagrange loss with respect to θ is thus

$$\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\theta} = \int_{t_0}^{t_N} a(t)\frac{\partial f}{\partial \theta}\mathrm{d}t.$$

3.6.2 Gradient wrt. θ and t

For this part of the proof, we assume that the loss function $\mathcal{L}$ depends only on the final time point t_N. If the loss function also depends on intermediate time points $t_1, \ldots, t_{N-1}$, we can simply apply the following method for each interval $[t_{N-1}, t_N], \ldots, [t_1, t_0]$ in reverse order and sum up the gradients of the parameters at each evaluation point.

Compared to the simplified derivation in the original paper [1], here we provide a detailed derivation of the choice of the initial values and how to obtain each derivative.

To calculate the gradient with respect to θ and t, similarly, we try to turn the problem into solving two initial value problems.

We first try to find the value of the gradient of loss function with respect to θ at the start time t_0. Notice that the loss $\mathcal{L}$ is the same as the Lagrange loss $\mathcal{J}$. We define a_θ as a function such that $\frac{\mathrm{d}a_\theta}{\mathrm{d}t} = -a(t)\frac{\partial f}{\partial \theta}$. Thus, we have

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} &= \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\theta} \\
&= \int_{t_0}^{t_N} a(t)\frac{\partial f}{\partial \theta}\mathrm{d}t \\
&= \int_{t_N}^{t_0} -a(t)\frac{\partial f}{\partial \theta}\mathrm{d}t \\
&= a_\theta(t_0) - a_\theta(t_N).
\end{aligned}$$

To simplify the calculation, we assume that $a_\theta(t_N) = 0$. We can make this assumption because if $a_\theta(t_N) \neq 0$, we can then define $a'_\theta(t) = a_\theta(t) - a_\theta(t_N)$:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \int_{t_N}^{t_0} -a(t)\frac{\partial f}{\partial \theta}\mathrm{d}t = a'_\theta(t_0).$$

Then the first initial value problem with respect to θ is

$$\begin{aligned}
a_\theta(t_N) &= 0 \\
\frac{\mathrm{d}a_\theta(t)}{\mathrm{d}t} &= -a(t)\frac{\partial f}{\partial \theta}.
\end{aligned} \tag{8}$$

Next, we consider the gradient with respect to t. We define a function $a_t(t) := -\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}$ - note that the minus sign here is added to simplify future calculations.

$$a_t(t) = -\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z(t)}\frac{\mathrm{d}z(t)}{\mathrm{d}t} = -a(t)f\left(z(t), \theta, t\right) \implies a_t(t_N) = -a(t_N)f\left(z(t_N), \theta, t\right)$$

$$\begin{aligned}
\frac{\mathrm{d}a_t}{\mathrm{d}t} &= -\frac{\mathrm{d}a(t)}{\mathrm{d}t}f(z, \theta, t) - a(t)\left(\frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}t} + \frac{\partial f}{\partial t}\right) \\
&= a(t)\frac{\partial f}{\partial z}f(z, \theta, t) - a(t)\frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}t} - a(t)\frac{\partial f}{\partial t} \\
&= -a(t)\frac{\partial f}{\partial t}.
\end{aligned}$$

So the second initial value problem with respect to t is

$$a_t(t_N) = -a(t)f\left(z(t), \theta, t\right)$$

$$\frac{\mathrm{d}a_t(t)}{\mathrm{d}t} = -a(t)\frac{\partial f}{\partial t}. \tag{9}$$

We notice that equations (7), (8), (9) follow a similar pattern. By adding a minus sign when defining $a_t(t)$, we ensure that equation (9) aligns with the pattern of equations (7), (8). Accordingly, we define the augumented state a_{aug} and the corresponding differrential equation f_{aug} as follows

$$a_{\text{aug}} := \begin{pmatrix} a \\ a_\theta \\ a_t \end{pmatrix} \qquad f_{\text{aug}}\left([z, \theta, t]\right) := \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} z \\ \theta \\ t \end{pmatrix} = \begin{pmatrix} f(z, \theta, t) \\ 0 \\ 1 \end{pmatrix}$$

$$\frac{\partial f_{\text{aug}}}{\partial(z, \theta, t)} = \begin{pmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

This step follows the approach suggested by the origional neural ODE paper [1], which allows us to concatenate all three equations (7), (8), (9) together.

Plugging a_{aug} into (7), we have

$$\frac{\mathrm{d}a_{\text{aug}}}{\mathrm{d}t} = -[a(t), a_\theta(t), a_t(t)]\frac{\partial f_{\text{aug}}}{\partial[z, \theta, t]}(t) = -\left[a\frac{\partial f}{\partial z}, a\frac{\partial f}{\partial \theta}, a\frac{\partial f}{\partial t}\right](t)$$

$$a_{\text{aug}}(t_N) = \left[a^{\text{initial}}(t_N), 0, -a(t_N)f(z(t_N), \theta, t_N)\right].$$

Thus, we only need to solve this initial value problem. By calling the ODE solver on the augmented adjoint once, we can calculate all necessary gradients with respect to $z(t_0)$, θ, t_0 and t_N.

In Appendix C, we provide a simple example and use the adjoint method to perform back-propagation. A solution is provided to illustrate how this method can be applied.

4 Application: Digit Classifier with Neural ODEs

In this section, we return our attention to the example we considered in Section 2.5, in which we implemented a digit classifier with around 80% accuracy using classic neural networks. With the help of neural ODEs, we shall implement a better version of this, aiming to boost the test accuracy to approximately 95%. The idea is largely taken from Lab 6 of [3]. The file containing all the code below can be found in the project's GitHub repository at `Neural_ODE.ipynb`.

4.1 Implementation

To start, we need to import some essential packages. We will use the `DifferentialEquations` and `DiffEqFlux` packages for neural ODEs, and despite the name, `DiffEqFlux` uses `Lux` instead of `Flux`. We also want to load the MNIST database from the `MLDatasets` package, and the `Images` package is useful for plotting images given by pixels.

```
using DifferentialEquations, DiffEqFlux, Lux, Random, Optimization,
      OptimizationOptimisers, ComponentArrays, Zygote, Statistics, Images
using MLDatasets: MNIST
imgs, nums = MNIST().features, MNIST().targets
```

We want to create a neural network with a neural ODE layer to map from an image to a 10-vector where the entry with the largest value is the number itself (plus 1 since Julia indices start from 1), and other entries give us extra information about the chance that it is that number. This reminds us of the one-hot encoding we used before. We can do this with the following function.

```
function onehot(nums::AbstractVector)
    n = length(nums)
    ret = zeros(Int, 10, n)
    for j = 1:n
        ret[nums[j]+1, j] = 1
    end
    ret
end
```

We then 'flatten' the images and setup the input layer of the neural network $\mathbb{R}^{28\times 28} \to \mathbb{R}^{20}$. Throughout the neural network, we will use $\tanh$ as the activation function.

```
down = Chain(FlattenLayer(), Dense(784, 20, tanh))
down_p, down_st = Lux.setup(MersenneTwister(), down)
```

The hidden layers $\mathbb{R}^{20} \to \mathbb{R}^{20}$ are given as a map from initial conditions to final values before passing through the output layer $\mathbb{R}^{20} \to \mathbb{R}^{10}$ since the output must be a 10-vector. Here, `Tsit5` is a numerical ODE solver which is a modified form of RK4 mentioned before [25].

```
nn = Chain(Dense(20, 10, tanh), Dense(10, 10, tanh), Dense(10, 20, tanh))
nn_ode = NeuralODE(nn, (0.0f0, 1.0f0), Tsit5(); save_everystep = false,
                   reltol = 1e-3, abstol = 1e-3, save_start = false)
fc = Dense(20, 10)
```

Now we put everything together, where the `convert` layer maps a solution to an ODE to its final value which we need to wrap in order for it to work with `Lux`.

```
m = Chain(; down, nn_ode, convert = WrappedFunction(last), fc)
ps, st = Lux.setup(rng, m)
ps = ComponentArray(ps)
```

We use the same loss function as before since we simply want to measure that the largest components are in the same spot, not necessarily whether the k^{th} entry is close to 1 and all other entries are close to 0.

```
logitcrossentropy(yˆ, y) = mean(-sum(y .* logsoftmax(yˆ; dims = 1); dims = 1))
function loss_function(ps, x, y)
    pred, st_ = m(x, ps, st)
    return logitcrossentropy(pred, y), pred
end
```

After that, we setup the optimisation problem.

```
opt_func = OptimizationFunction((ps, _, x, y) -> loss_function(ps, x, y),
                                AutoZygote())
opt_prob = OptimizationProblem(opt_func, ps)
```

Finally, since the sample size is large, we want to work with several images at the same time for efficiency reasons, a process called *batching*. Here, we prepare data in batches of 100 for efficiency and train the model. We take the last batch as the test set and all the others as the training set.

```
const N = 100
M = length(nums) ÷ N - 1
data = [(imgs[:, :, N*(b-1)+1:N*b], onehot(nums[N*(b-1)+1:N*b])) for b = 1:M]
@time res = solve(opt_prob, Adam(0.003), data)
```

4.2 Comparison

We see that the time used is around 8 seconds, which is around 38.5% shorter than 13 seconds before with the neural network model.

The test accuracy on the last batch is then given as

```
classify(x) = [argmax(x[:,j]) - 1 for j = 1:size(x,2)]
sum(classify(m(imgs[:,:,end-N+1:end], res.u, st)[1]) .== nums[end-N+1:end])/N
```

which is around 95%, a huge improvement from the previous result of 80% before.

We also compare the number of parameters we used in this model with the model that we used in Section 2.5. We know that the number of parameters in a neural network includes weights and biases for each layer. The structure of the model in the first section is

```
model = Chain(
    Dense(784, 196, relu),
    Dense(196, 49, relu)
    Dense(49, 10)
)
```

The number of parameters related to the weights is $784 \times 196 + 196 \times 49 + 49 \times 10$, and the total number of biases is $196 + 49 + 10 = 255$. So, the total number of parameters in this model is $163758 + 255 = 164013$.

However, for the network we used here which embeds a neural ODE layer, since we are using a neural network to initailise the neural ODE layer, the model would have the same parameter as a neural network with the following structure:

```
model = Chain(
    Dense(784, 20, tanh),
    Dense(20, 10, tanh),
    Dense(10, 10, tanh),
    Dense(10, 20, tanh),
    Dense(20, 10)
)
```

Using the same method, we find that the number of parameters in this model is 16450.

This demonstrates that by embedding a neural ODE layer into the network, we use nearly one-tenth of the parameters compared to a simple neural network while still improving the results.

5 Extension: Neural CDEs

5.1 Motivation

As we have seen before, neural networks and differential equations seem to be irrelevant but, in fact, they have been proven to be two sides of the same coin. In particular, neural ODEs can be seen as a continuous-time analogue to the ResNet. However, there exist certain real-life applications such as time series forecasting where neural ODEs are no longer effective in solving the problem. One such example is in irregularly sampled time series data. Fortunately, a British mathematician Terry Lyons developed a novel and exciting field of mathematics, rough path theory, in the 1990s [22]. The mathematical insights from this theory can be used to develop a new concept called *neural controlled differential equations* (neural CDEs).

5.2 Controlled Differential Equations

Before defining neural CDEs, it is essential to understand the definition of CDEs. To formally define them, we need to introduce a new notion of integration called *Riemann-Stieltjes integral* and a new concept of real functions called *bounded variation*.

Definition 5.1 (Riemann-Stieltjes integral)**.** Let f and g be two bounded, real functions defined on $[a, b]$, and let $P = \{a = t_0 < t_1 < \cdots < t_n = b\}$ be an arbitrary partition of $[a, b]$. $\forall i = 1, \ldots, n$, let $\Delta g_i = g(t_i) - g(t_{i-1})$, $M_i = \sup f(t)$, and $m_i = \inf f(t)$, where $t \in [t_{i-1}, t_i]$.

Define the upper and lower 'Darboux sums' similar to those in the construction of Riemann integrals as $U(P, f, g) = \sum M_i \Delta g_i$ and $L(P, f, g) = \sum m_i \Delta g_i$.

If $\sup L(P, f, g)$ and $\inf U(P, f, g)$ exist and are equal, then denote their common value by $\int_a^b f \mathrm{d}g$, or $\int_a^b f(t)\mathrm{d}g(t)$, which is the *Riemann-Stieltjes integral* of f with respect to g on $[a, b]$.

In short, the Riemann-Stieltjes integral is just a simple extension of the Riemann integral.

Definition 5.2 (Bounded variation)**.** The total variation of a real-valued function f in an interval $[a, b]$, denoted by $V_a^b(f)$, is defined as

$$\sup_{P \in \mathbb{P}} \sum_{i=0}^{n_P - 1} \left| f(t_{i+1}) - f(t_i) \right|,$$

where $\mathbb{P} = \{P = \{a = t_0 < t_1 < \cdots < t_{n_P} = b\}\}$, the set of all partitions on $[a, b]$.

Then f is a function of *bounded variation* iff $V_a^b(f) < \infty$.

We have the following theorem about bounded variation.

Theorem 5.1. *If $f : [a, b] \to \mathbb{R}$ is continuously differentiable, then f is of bounded variation.*

Proof. Since $f : [a, b] \to \mathbb{R}$ is differentiable,

$$\begin{aligned} V_a^b(f) &= \sup_{P \in \mathbb{P}} \sum_{i=0}^{n_P - 1} \left| \int_{t_i}^{t_{i+1}} f'(x) \mathrm{d}x \right| \\ &\leq \sup_{P \in \mathbb{P}} \sum_{i=0}^{n_P - 1} \int_{t_i}^{t_{i+1}} \left| f'(x) \right| \mathrm{d}x \\ &= \int_a^b \left| f'(x) \right| \mathrm{d}x \\ &< \infty. \end{aligned}$$

where the last inequality follows from the fact that $f'(x)$ is continuous on a closed interval and thus is bounded. This implies that f is of bounded variation. □

Definition 5.3 (Controlled differential equations (CDEs))**.** Let $a, b \in \mathbb{R}$ with $a < b$ and let $v, w \in \mathbb{N}$. Let $x : [a, b] \to \mathbb{R}^v$ be a continuous function of bounded variation. Let $f : \mathbb{R}^w \to \mathbb{R}^{w \times v}$ be Lipschitz continuous. Let $y_a \in \mathbb{R}^w$.

A continuous function $y : [a, b] \to \mathbb{R}^w$ is said to be a solution of the following initial value problem if:

$$y(a) = y_a, y(t) = y(a) + \int_a^b f(y(s))\mathrm{d}x(s),$$

where $t \in (a, b]$ [22], $\int_a^b f(y(s))\mathrm{d}x(s)$ is a Riemann-Stieltjes integral and $f(y(s))\mathrm{d}x(s)$ is a matrix vector multiplication.

The control integral equation can be rewritten as a *control differential equation* (CDE):

$$y(a) = y_a, \mathrm{d}y(t) = f(y(t))\mathrm{d}x(t).$$

Note that the continuity of f and the bounded variation of x guarantee the existence of the Riemann-Stieltjes integral, and the Lipschitz continuity of f satisfies the condition of the Picard-Lindelöf theorem for controlled differential equations. Hence, the solution to this initial value problem is unique. The existence of the integral and the uniqueness of the solution will not be investigated in detail. For interested readers, see [22] and [26] for details.

From now on $y(t)$ will be referred to as y_t, which is a notation commonly used to describe CDEs.

5.3 Neural CDEs

Before we introduce neural CDEs, we first explore the idea of a *recurrent neural network* (RNN). In fact, so far, we have been discussing *feedforward neural networks* (FNNs) only, whose flow of information is in one direction only, from the input layer to the output layer, without any cycles or loops. FNNs are sufficient for ordinary tasks such as regression and classification. However, for more complicated problems, RNNs play an important role. In RNNs, the output of previous steps can be fed as input to the current step, allowing it to maintain a form of 'memory'. This makes RNNs particularly effective for tasks such as time series prediction, natural language processing, and speech recognition.

One fundamental assumption of neural CDEs is that the data we observe should form a continuous path $X : [a, b] \to \mathbb{R}^v$ with bounded variation. This assumption is made simply for theoretical purposes, i.e. the continuous data correspond to an evolutionary process or control path in the neural CDE. However, it is worthwhile to note that it is highly unlikely to observe data in such a form in real-life applications, and thus a trick that converts discrete observations to a continuous control path would be introduced in later parts of this extension. Let us now formulate the definition of neural CDEs rigorously, and then it will be clear that neural CDE is the continuous analogue to the recurrent neural network.

Definition 5.4 (Neural CDEs)**.** Let $f_\theta : \mathbb{R}^w \to \mathbb{R}^{w \times v}$ be an arbitrary neural network with parameters θ that represents the dynamics between hidden states. Note that v represents the dimension of the input X_t, and w represents the dimension of the hidden states Z_t. Let $g_\theta : \mathbb{R}^v \to \mathbb{R}^w$ be a neural network that maps the input X_t to hidden states Z_t.

Then, we define the initial value problem of a neural CDE to be the following:

$$Z_a = g_\theta(X_a), Z_t = Z_a + \int_a^t f_\theta(Z_s)\mathrm{d}X_s,$$

where $t \in (a, b]$, or in the differential form

$$Z_a = g_\theta(X_a), \mathrm{d}Z_t = f_\theta(Z_t)\mathrm{d}X_t,$$

where $t \in (a, b]$. [22]

From the above CDE, we may conclude that the hidden state Z_t of the recurrent neural network can be continuously updated as the input X_t evolves over time. Thus, neural CDEs are a continuous analogue of recurrent neural networks. Finally, let $l_\theta : \mathbb{R}^w \to \mathbb{R}^p$ be a linear map mapping the hidden states Z_t to the output Y_t, where p is the dimension of the output Y_t. The difference between a neural network and a linear map is that a neural network contains an activation function, and a linear map does not. Figure 10 illustrates the difference between neural ODEs and neural CDEs in terms of modelling time series data. For a detailed discussion of the comparison between two models, we refer the reader to [27].

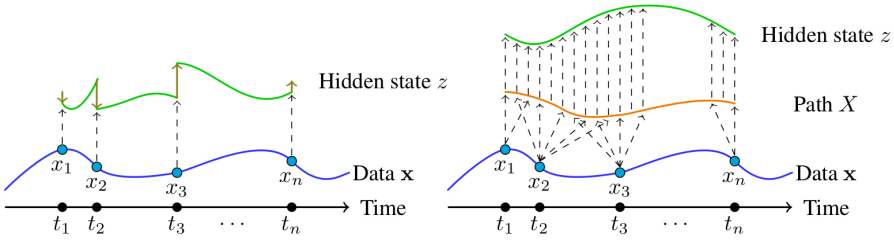


Figure 10 The comparison of neural ODEs and neural CDEs, respectively, when modelling time series data. This figure is taken from [27].

5.4 Solving Neural CDEs

It is theoretically possible to directly solve a CDE $\mathrm{d}Z_t = f(Z_t)\mathrm{d}X_t$ with a given initial condition by discretisation. In particular, one could use the following modification of Euler's method from Section 3.4 to obtain a numerical solution: $Z_{i+1} = Z_i + f_\theta(Z_i)(X(t_{i+1}) - X(t_j))$.

However, CDEs are much harder to solve than ODEs since there is a control term introduced in the equation. For the general case, we do not know whether the control path X_t is differentiable or not. Using powerful tools from rough path theory, one can derive an elegant result called the *log-ODE method*, which approximates the CDE by an ODE containing the log-signature of the control path [28]. The details of the log-ODE method are omitted here, since the underlying theory is beyond the scope of the paper.

Despite the fact that the log-ODE method can be used to solve CDEs under any circumstances, the log-ODE method is quite complicated to be implemented numerically. Fortunately, it can be cleverly avoided in terms of application. By a standard treatment common in numerical analysis, we can guarantee that the control path X_t is continuously differentiable. Thus, by Theorem 5.1, it is of bounded variation and satisfies the condition of a path in CDEs. We defer the discussion of this technique to later parts of this paper. In this case, the CDE can be shown to be equivalent to the following non-autonomous ODE: $\mathrm{d}Z_t = g(t, Z_t)\mathrm{d}t$, where $g(t, Z_t) = f(Z_t)\frac{\mathrm{d}X_t}{\mathrm{d}t}$. Then, we can solve this ODE using methods explained in Section 3.4 of this paper.

It can also be noted that the backpropagation through neural CDEs is similar to that through neural ODEs. In particular, the Discretise-then-Optimise method is exactly the same as that of neural ODEs, and the Optimise-then-Discretise method is similar to that of neural ODEs. Since the proofs are very similar, we omit them here.

5.5 Application: Time Series Modelling

It is possible to use neural ODEs to model time series data. However, neural ODEs become less useful when it comes to irregularly sampled time series data. In this final subsection, we propose a naive implementation of using neural CDEs to model financial time series data.

In the context of financial time series, the input data consists of a sequence of discrete data points (t_i, x_i). As mentioned in the previous subsection, it is possible to construct a continuous control path based on these discrete data points. To do so, we need to employ a standard technique from numerical analysis, called *cubic splines*.

5.5.1 Cubic Splines

Lagrange basis polynomials and how they can be used to interpolate data are standard ideas introduced in undergraduate numerical analysis courses that provide a continuous path between data points. However, whenever an additional data point is added to the original set of data points, the existing Lagrange polynomials would have to be recalculated. This makes Lagrange basis polynomials computationally inefficient. We seek an alternative set of basis polynomials for interpolation, called the *Newton basis polynomials*. The following lemma is not included in [29], but we state and prove it for completeness in defining the newton basis polynomials.

Lemma 5.2. *Let $n_0(t) = 1$, and $\forall j = 1, \dots, k$, let $n_j(t) = \prod_{i=0}^{j-1}(t - t_i)$. Let P_k denote the space of polynomials in $\mathbb{R}$ with degree less than or equal to k. Then $B = \{n_0(t), n_1(t), \dots, n_k(t)\}$ forms a basis of P_k.*

Proof. Since $|B| = k + 1 = \dim(P_k)$, it suffices to show that B is linearly independent.

We prove by induction on $k \in \mathbb{N}$. The result is trivial for $k = 0$. Suppose that the result is true for $k < m$. Then, for $k = m$, suppose that $\exists a_0, a_1, \dots, a_m$ such that $\sum_{i=0}^{m} a_i n_i(t) = 0$.

If $a_0 = a_1 = \cdots = a_m = 0$, then B is linearly independent. Otherwise, let j be the largest integer such that $a_j \neq 0$. By the induction hypothesis, $\{n_0(t), n_1(t), \dots, n_{j-1}(t)\}$ forms a basis of P_{j-1}, and thus $\sum_{i=0}^{m} a_i n_i(t) \neq 0$, which is a contradiction. Hence, the result is true for $k = m$.

By induction, we have that $\forall k \in \mathbb{N}, B = \{n_0(t), n_1(t), \dots, n_k(t)\}$ forms a basis of P_k. □

After proving this lemma, we are confident in making the following definitions.

Definition 5.5 (Newton basis polynomials)**.** Given a sample of $k+1$ data points $\{(t_0, x_0), (t_1, x_1), \dots, (t_k, x_k)\}$, the *Newton basis polynomials* corresponding to this sample are defined as $n_0(t) = 1$ and $\forall j = 1, \dots, k, n_j(t) = \prod_{i=0}^{j-1}(t - t_i)$.

Definition 5.6 (Newton polynomial)**.** The *Newton polynomial* that interpolates a sample of data points $\{(t_0, x_0), (t_1, x_1), \dots, (t_k, x_k)\}$ is defined as $N_k(t) = \sum_{i=0}^{k} a_i n_i(t)$, where $n_i(t)$ is taken from Definition 5.5.

As the Newton polynomial passes through data points $(t_0, x_0), (t_1, x_1), \dots, (t_k, x_k)$, the coefficients of the Newton polynomial can be uniquely determined by solving a system of linear equations. The details of this computation are left for the interested readers to check. In other words, given a sample of $k + 1$ data points $(t_0, x_0), (t_1, x_1), \dots, (t_k, x_k)$, the corresponding Newton polynomial is unique.

Since a_i is a linear combination of $x_0, x_1, \dots, x_i$, the coefficients are dependent on $t_0, t_1, \dots, t_i$. We denote a_i as $[t_0, t_1, \dots, t_i]f$, where $i \in [0, k]$, and let f be some unknown function that passes through these points and needs to be estimated using the Newton interpolatory

polynomial. The right-hand side is also denoted as the i^{th} *divided difference* of f relative to $t_0, t_1, \ldots, t_i$. The motivation behind this name can be immediately seen from the following lemma.

Lemma 5.3. *$\forall k \in \mathbb{N}_{>0}$, we have*

$$[t_0, t_1, t_2 \ldots, t_k]f = \frac{[t_1, t_2 \ldots, t_k]f - [t_0, t_1, \ldots, t_{k-1}]f}{t_k - t_0}.$$

Proof. Let $r(t) = N_{k-1}(f; t_1, t_2, \ldots, t_k; t)$ and $s(t) = N_{k-1}(f; t_0, t_1, \ldots, t_{k-1}; t)$, where $N_{k-1}(f; t_1, t_2, \ldots, t_k; t)$ means that it is approximating f that passes through $(t_1, x_1), (t_2, x_2), \ldots, (t_k, x_k)$, and other Newton basis polynomials are expressed in a similar way [29].

We claim that

$$N_k(f; t_0, t_1, \ldots, t_k; t) = r(t) + \frac{t - t_k}{t_k - t_0}(r(t) - s(t)). \qquad (*)$$

By the definition of Newton basis polynomials, it is obvious that the degree of RHS is less than or equal to k. Since Newton interpolatory polynomials are unique, it suffices to show that RHS passes through $(t_0, x_0), (t_1, x_1), \ldots, (t_k, x_k)$.

Substituting t_0 into both sides of $(*)$ gives LHS $= x_0$ and RHS $= r(t_0) + \frac{t_0 - t_k}{t_k - t_0}(r(t_0) - s(t_0)) = s(t_0) = x_0$.

Substituting t_k into both sides of $(*)$ gives LHS $= x_k$ and RHS $= r(t_k) + \frac{t_k - t_k}{t_k - t_0}(r(t_k) - s(t_k)) = r(t_k) = x_k$.

Substituting t_i into both sides of $(*)$, where $i = 1, \ldots, k-1$, gives LHS $= x_i$ and RHS $= r(t_i) + \frac{t_i - t_k}{t_k - t_0}(r(t_i) - s(t_i)) = r(t_i) = x_i$.

In all cases, we have LHS $=$ RHS, and thus the claim is true.

Equating the leading coefficient of both sides of $(*)$ yields the result. □

Using this result, we can establish a powerful tool called the *table of divided differences*.

Definition 5.7 (Table of divided differences)**.** Under the same setting as in the previous definition of Newton polynomials, we create a *table of divided differences* as follows:

$$\begin{array}{cccccc} t & f & & & & \\ t_0 & x_0 & & & & \\ t_1 & x_1 & [t_0, t_1]f & & & \\ t_2 & x_2 & [t_1, t_2]f & [t_0, t_1, t_2]f & & \\ t_3 & x_3 & [t_2, t_3]f & [t_1, t_2, t_3]f & [t_0, t_1, t_2, t_3]f & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{array}$$

where the $(i+1)^{\text{th}}$ row of the table is defined as

$$t_i \quad x_i \quad [t_{i-1}, t_i]f \quad [t_{i-2}, t_{i-1}, t_i]f \quad \ldots \quad [t_0, t_1, \ldots, t_i]f.$$

The table of divided differences provides a visualisation of the process of computing the coefficients in the Newton polynomial. In particular, the coefficients $a_0, a_1, \ldots, a_k$ are the first $k+1$ diagonal entries in the table of divided differences. Using Lemma 5.3, one can calculate following this rule: **Each entry is the difference between the entry immediately to the left and the one above it, divided by the difference between the t value at the right end of the bracket term in this entry and the t value at the left end of the bracket term in this entry**. One can immediately see that the order of the calulation of the coefficients is from the left of the table to the right of it.

Lemma 5.4. $[t_0, t_0, \ldots, t_0]f = \frac{1}{k!}f^{(k)}(t_0)$, *where the* LHS *contains* $k+1$ t_0*'s.*

Proof. [29] Let s be an arbitrary node (a *time point* in the context of time series) not equal to any one of $t_0, t_1, \ldots, t_k$.

By the definition of the Newton polynomial, we have

$$N_{k+1}(f; t_0, t_1, \ldots, t_k, s; t) = N_k(f; t) + [t_0, t_1, \ldots, t_k, s]f \prod_{i=0}^{k}(t - t_i).$$

Inserting $t = s$ yields

$$f(s) = N_k(f; s) + [t_0, t_1, \ldots, t_k, s]f \prod_{i=0}^{k}(s - t_i).$$

As the choice of s is arbitrary, we can change the notation from s back to t, and thus

$$f(t) - N_k(f; t) = [t_0, t_1, \ldots, t_k, s]f \prod_{i=0}^{k}(t - t_i).$$

Now we quote without proof a classical result about the error of the interpolation:

$$f(t) - N_k(f; t) = \frac{f^{(k+1)}(\xi(t))}{(k+1)!} \prod_{i=0}^{k}(t - t_i),$$

where $\xi(t)$ is strictly between the smallest and the largest of these nodes. See [29] for details.

Comparing the two results, we have

$$[t_0, t_1, \ldots, t_k, s]f = \frac{f^{(k+1)}(\xi(t))}{(k+1)!}.$$

By letting $t = t_{k+1}$ and replacing $k+1$ by k, we obtain

$$[t_0, t_1, \ldots, t_k]f = \frac{1}{k!}f^{(k)}(\xi),$$

where ξ is defined similarly as before.

Finally, we let $t_1, \ldots, t_k \to t_0$. Then, by the definition of ξ, $\xi \to t_0$, which yields the result. □

Now we are ready to state and explain the method of cubic spline interpolation formally.

Definition 5.8. (Cubic splines) Given a sample of $k+1$ data points $\{(t_0, x_0), (t_1, x_1), \ldots, (t_k, x_k)\}$, the *cubic spline* X_t corresponding to this sample is an interpolatory curve such that it is twice continuously differentiable.

That is, if $p_i(t) = X_t|_{[t_i, t_{i+1}]}$, then the following conditions are satisfied:

$$p_i(t_i) = x_i, p_i(t_{i+1}) = x_{i+1}, \quad \forall i = 0, 1, \ldots, k-1; \tag{A}$$
$$p_i'(t_i) = m_i, p_i'(t_{i+1}) = m_{i+1}, \quad \forall i = 0, 1, \ldots, k-1; \tag{B}$$
$$p_{i-1}''(t_i) = p_i''(t_i), \quad \forall i = 1, \ldots, k-1, \tag{C}$$

where $\forall i = 0, 1, \ldots, k$, m_i are chosen constants.

One might be sceptical about whether the condition (C) is necessary, since in time series modelling we only ensure that the control path X_t is continuously differentiable. However, if condition (C) is dropped, it means that the curvature of the resultant path (which is related to the second derivative of the control path) might not be continuous, resulting in a possibly drastic change in the curvature of the control path that leads to unrealistic modelling.

Using the table of divided differences and Lemma 5.4, we obtain that the Newton polynomial defined on each time segment $[t_i, t_{i+1}]$ is

$$p_i(t) = x_i + (t - t_i)m_i + (t - t_i)^2 \frac{[t_i, t_{i+1}]f - m_i}{\Delta t_i} + (t - t_i)^2(t - t_{i+1}) \frac{m_{i+1} + m_i - 2[t_i, t_{i+1}]f}{(\Delta t_i)^2},$$

where $\Delta t_i = t_{i+1} - t_i$.

The above polynomial can be reformulated in terms of a Taylor polynomial:

$$p_i(t) = x_i + m_i(t - t_i) + a_i(t - t_i)^2 + b_i(t - t_i)^3,$$

where $a_i = \dfrac{[t_i, t_{i+1}]f - m_i}{\Delta t_i} - b_i \Delta t_i$ and $b_i = \dfrac{m_{i+1} + m_i - 2[t_i, t_{i+1}]f}{(\Delta t_i)^2}$.

Since $\forall i = 1, \ldots, k-1, p'_{i-1}(t_i) = p'_i(t_i)$, we have $2a_{i-1} + 6b_{i-1}(t_i - t_{i-1}) = 2a_i$, which simplifies to $a_{i-1} + 3b_{i-1}\Delta t_i = a_i$.

Using the coefficients of the Taylor polynomial, we obtain a system of linear equations:

$$\frac{[t_{i-1}, t_i]f - m_{i-1}}{\Delta t_{i-1}} + 2\frac{m_i + m_{i-1} - 2[t_{i-1}, t_i]f}{\Delta t_{i-1}} = \frac{[t_i, t_{i+1}]f - m_i}{\Delta t_i} - \frac{m_{i+1} + m_i - 2[t_i, t_{i+1}]f}{\Delta t_i},$$

where $i = 1, \ldots, k-1$.

Rearranging terms of the above equation yields

$$\Delta t_i m_{i-1} + 2(\Delta t_{i-1} + \Delta t_i)m_i + \Delta t_{i-1} m_{i+1} = 3(\Delta t_i [t_{i-1}, t_i]f + \Delta t_{i-1}[t_i, t_{i+1}]f),$$

where $i = 1, \ldots, k-1$.

There are $k-1$ linear equations with $k+1$ unknowns $m_0, m_1, \ldots, m_k$. If m_0 and m_k are chosen, the system can then be reduced to a tridiagonal system and thus can be solved using Gaussian elimination.

In terms of applications, the most commonly used cubic spline is the *natural cubic spline*.

Definition 5.9. (Natural cubic splines) Given a sample of $k+1$ data points $(t_0, x_0), (t_1, x_1), \ldots, (t_k, x_k)$, a *natural cubic spline* is a cubic spline X_t such that $X''(t_0) = X''(t_k) = 0$.

Using these additional conditions and the Taylor polynomial, we obtain the following system of linear equations:

$$2\frac{[t_0, t_1]f - m_0 - m_1 - m_0 + 2[t_0, t_1]f}{\Delta t_0} = 0,$$
$$2\frac{[t_{k-1}, t_k]f - m_{k-1}}{\Delta t_{k-1}} + 4\frac{m_k + m_{k-1} - 2[t_{k-1}, t_k]f}{\Delta t_{k-1}} = 0,$$

which simplifies to

$$2m_0 + m_1 = 3[t_0, t_1]f,$$
$$m_{k-1} + 2m_k = 3[t_{k-1}, t_k]f.$$

Adding these equations to the system of $k-1$ linear equations, we can readily solve for the constants $m_0, m_1, \ldots, m_k$, and thus we can obtain an explicit expression for the control path X_t.

5.5.2 Implementation

In this section, we propose a possible naive implementation of neural CDEs to model irregularly sampled time series data.

Description of the time series data:

The time series data used are adapted from Kaggle. It records the daily adjusted closing price of the Amazon stock AMZN from 15/05/1997 to 02/08/2017 excluding holidays, which is reasonable since the stock market is closed during holidays. Hence, it is an irregularly sampled time series requiring some data pre-processing.

Data preprocessing

The data preprocessing process contains several steps. Firstly, the time series data are normalised using min-max scaling. In particular, the time series data are now set to be in the range $[0, 1]$ using the formula $p_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$. Secondly, all dates are reorganised into the unified format "yyyy/mm/dd". Then, we transform each date into the relative date, which is the difference in days between the initial date and itself. Finally, we use the time series data of the form (t_i, x_i), where $t_i \in \mathbb{N}, x_i \in [0, 1]$ to create a natural cubic spline. The relevant code can be found in Appendix D.

Training

In this part, we only give a brief overview of the training process.

Recall that the entire forecasting process consists of three linear maps. In particular, $g_\theta : \mathbb{R}^v \to \mathbb{R}^w$ is a neural network that maps the input X_t to hidden states Z_t, $f_\theta : \mathbb{R}^w \to \mathbb{R}^{w \times v}$ is an arbitrary neural network with parameters θ that represents the dynamics between hidden states. Note that v represents the dimension of the input X_t, and w represents the dimension of the hidden states Z_t. Finally, $l_\theta : \mathbb{R}^w \to \mathbb{R}^p$ is a linear map that maps the hidden states Z_t to the output Y_t, where p is the dimension of the output Y_t.

The main goal is to find the optimal parameters for each of the neural networks and the final linear map. In the training process of the second neural network, we need to incorporate the derivative of the control path obtained in the previous section into the CDE. In other words, use the method mentioned previously to convert the neural CDE to an ODE.

There exist online Python libraries that uses neural CDEs to achieve time series forecasting. See [30] for details.

6 Conclusion

In this paper, we discussed the power of neural ODEs in scientific machine learning. We focussed on three main areas. Firstly, an initial introduction to neural networks and several optimisation strategies was conducted. We trained a standard neural network model to recognise handwritten digits from the MNIST dataset. Next, we introduced neural ODEs and explained their use in the machine learning process, incorporating them into the training of the same MNIST dataset and showing the improvement in test accuracy. Finally, we extended our discussion to neural CDEs, discussing their power and application in time series data analysis.

The research conducted through this paper has provided valuable learning opportunities in the area of scientific machine learning. We have gained a better appreciation of the methods taught and described in various undergraduate modules that provided the bridge to the complex ideas brought forward in this paper. Finally, the exposure to neural networks and neural differential equations has been beneficial in understanding more complicated theories in future courses and career paths.

At the same time, however, it is important to recognise and acknowledge the limitations of the paper. Given the context and timeline of the project, there was a limit to the extent of exploration in this area. Neural differential equations, while being new fields, are extremely rich and require much more time to be fully utilised. Additionally, the discussion was largely theoretical, with several simple applications brought forward. Further applications could examine more subtle ideas, such as the optimisation of layer dimensions for better results or a more extensive comparison between neural ODEs and neural CDEs.

This gives rise to future research that could be built on this paper. Further elaboration of the improvements in accuracy and efficiency could enrich the understanding of neural differential equations. Another area worth pursuing is studying perhaps more complex neural differential equations and how they better represent other data. These could thus be extended to other fields, such as studying neural differential equations in physics-informed neural networks (PINNs). These could give a more holistic understanding on the power of neural differential equations.

The field of scientific machine learning is a highly active one, attracting scientists and mathematicians alike. Various libraries are being developed, the most important of which in Julia is the `SciML` package. There is much anticipation surrounding the improvements and advancements in scientific machine learning - neural differential equations are merely the start. It remains to see just how powerful these tools can be in scientific machine learning and beyond.

Acknowledgments

We would like to thank Dr Olver, our project supervisor, and his PhD students Mr VandenHeuvel and Mr Pu for their continued support and guidance throughout the project. Also, we would like to thank Dr Salvi and Prof Cass for kindly sharing their lecture notes which inspire the neural CDE extension section in this paper.

References

[1] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural Ordinary Differential Equations. *Advances in Neural Information Processing Systems*, 31, 2018.

[2] H. Li. The Advance of Neural Ordinary Differential Ordinary Differential Equations. *Applied and Computational Engineering*, 6:1283–1287, 06 2023.

[3] S. Olver. SciMLSANUM2024: SciML Workshop at the South African Numerical and Applied Mathematics 2024 Conference. https://github.com/dlfivefifty/SciMLSANUM2024, June 2024.

[4] W. S. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[5] F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386–408, 1958.

[6] J. Han and C. Moraga. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. In *From Natural to Artificial Neural Computation*, pages 195–201. Springer Berlin Heidelberg, 1995.

[7] Y. Nazarathy and H. Klok. *Statistics with Julia: Fundamentals for Data Science, Machine Learning and Artificial Intelligence*. Springer Series in the Data Sciences. Springer International Publishing, 2021.

[8] R. M. Gower. Convergence Theorems for Gradient Descent, 2015. Lecture notes.

[9] D. Kroese, Z. Botev, T. Taimre, and R. Vaisman. *Data Science and Machine Learning: Mathematical and Statistical Methods*. CRC Press, Boca Raton, 2019.

[10] W. Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, 2011.

[11] A. Griewank and A. Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, 2008.

[12] N. Ketkar. *Deep Learning with Python*. Professional and Applied Computing, Apress Access Books, Professional and Applied Computing (R0). Apress, Berkeley, CA, 1st edition, 2017.

[13] M. Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs, 2019.

[14] P. Mikuła. Simple MNIST in Julia. https://github.com/piotrek124-1/Simple_MNIST_Julia/, June 2024.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[16] E. Dupont, A. Doucet, and Y. W. Teh. Augmented Neural ODEs. *Advances in Neural Information Processing Systems*, 32, 2019.

[17] L. H. Nguyen and A. Malinsky. Exploration and Implementation of Neural Ordinary Differential Equations. *Capstone Showcase*, 8, 2020.

[18] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman. Universal Differential Equations for Scientific Machine learning, 2020.

[19] E. Süli and D. F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

[20] C. Rackauckas, Q. Nie, J. Chen, R. Supekar, V. Vijay, J. Revels, V. Ivaturi, V. Shah, S. Gowda, Y. Ma, and A. Edelman. Ordinary Differential Equations: Applications and Discretizations, 2022.

[21] J. H. Verner. Numerically Optimal Runge–Kutta Pairs with Interpolants. *Numerical Algorithms*, 53(2):383–396, 2010.

[22] P. Kidger. On Neural Differential Equations, 2022.

[23] L. S. Pontryagin. *Mathematical Theory of Optimal Processes*. Routledge, 2018.

[24] P. Hu. A note on the adjoint method for neural ordinary differential equation network. *arXiv preprint arXiv:2402.15141*, 2024.

[25] C. Tsitouras. Runge–Kutta Pairs of Order 5(4) Satisfying only the First Column Simplifying Assumption. *Computers and Mathematics with Applications*, 62(2):770–775, 2011.

[26] W. Rudin. *Principles of Mathematical Analysis*. McGraw Hill, 1953.

[27] P. Kidger, J. Morrill, J. Foster, and T. Lyons. Neural Controlled Differential Equations for Irregular Time Series. *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, 2020.

[28] J. Morrill, C. Salvi, P. Kidger, J. Foster, and T. Lyons. Neural Rough Differential Equations for Long Time Series. *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[29] W. Gautschi. *Numerical Analysis*. Birkhäuser, 2012.

[30] S. Y. Jhin, H. Shin, S. Kim, S. Hong, M. Jo, S. Park, N. Park, S. Lee, H. Maeng, and S. Jeon. Attentive Neural Controlled Differential Equations for Time-Series Classification and Forecasting. *Knowledge and Information Systems*, 2023.

A Proof of Convergence of Gradient Descent Algorithm

In this appendix, we prove the convergence of the gradient descent algorithm (as presented in Section 2.3.1) that generates the sequence $(\theta_n)_\infty$, taking ideas from [8].

We first note that since $\hat{\mathbf{y}}$ depends on the values of θ, we can simply consider a univariate function $\mathcal{L}(\theta)$. We can thus define the concepts of L-Lipschitz continuity and convexity:

Definition A.1 (L-Lipschitz continuity)**.** A function f is *L-Lipschitz continuous* in interval $\Theta \subset \mathbb{R}^p$ if $\exists L > 0$, $\forall \theta_x, \theta_y \in \Theta$, we have

$$\|f(\theta_x) - f(\theta_y)\| < L\|\theta_x - \theta_y\|.$$

Definition A.2 (L-smoothness)**.** A function f is *L-smooth* if ∇f is L-Lipschitz continuous.

Definition A.3 (Convexity[1])**.** A function f is *convex* on $\mathbb{R}^p$ if $\forall \theta_x, \theta_y \in \mathbb{R}^p$, $\forall t \in [0, 1]$, we have

$$f(t\theta_x + (1-t)\theta_y) \leq tf(\theta_x) + (1-t)f(\theta_y).$$

In order to prove the convergence of the gradient descent algorithm, we need the following lemmas.

Lemma A.1. *If $\mathcal{L}$ is L-smooth, then the following inequalities hold $\forall \theta_x \in \mathbb{R}^p$:*

$$\mathcal{L}(\theta_x - \frac{1}{L}\nabla\mathcal{L}(\theta_x)) - \mathcal{L}(\theta_x) \leq -\frac{1}{2L}\|\nabla\mathcal{L}(\theta_x)\|_2^2, \tag{10}$$

$$\mathcal{L}(\hat{\theta}) - \mathcal{L}(\theta_x) \leq -\frac{1}{2L}\|\nabla\mathcal{L}(\theta_x)\|_2^2. \tag{11}$$

Proof. We note that

$$\begin{aligned}\mathcal{L}(\theta_x - \frac{1}{L}\nabla\mathcal{L}(\theta_x)) &\leq \mathcal{L}(\theta_x) - \frac{1}{L}\langle\nabla\mathcal{L}(\theta_x), \nabla\mathcal{L}(\theta_x)\rangle + \frac{L}{2}\|\frac{1}{L}\nabla\mathcal{L}(\theta_x)\|_2^2 \\ &= \mathcal{L}(\theta_x) - \frac{1}{2L}\|\nabla\mathcal{L}(\theta_x)\|_2^2\end{aligned}$$

by L-smoothness, giving (10). We note that for (11), $\mathcal{L}(\theta_x) \geq \mathcal{L}(\hat{\theta})$ $\forall \theta_x$. This completes the proof. □

Lemma A.2. *If $\mathcal{L}$ is convex and L-smooth, then*

$$\mathcal{L}(\theta_y) - \mathcal{L}(\theta_x) \leq \langle\nabla\mathcal{L}(\theta_y), \theta_y - \theta_x\rangle - \frac{1}{2L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|_2^2, \tag{12}$$

$$\langle\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x), \theta_y - \theta_x\rangle \geq \frac{1}{L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|. \tag{13}$$

Proof. (12) can be proven by stating

$$\begin{aligned}\mathcal{L}(\theta_y) - \mathcal{L}(\theta_x) &= \mathcal{L}(\theta_y) - \mathcal{L}(\theta_z) + \mathcal{L}(\theta_z) - \mathcal{L}(\theta_x) \\ &\leq \langle\nabla\mathcal{L}(\theta_y), \theta_y - \theta_z\rangle + \langle\nabla\mathcal{L}(\theta_x), \theta_z - \theta_x\rangle + \frac{L}{2}\|\theta_z - \theta_x\|_2^2\end{aligned}$$

[1] It should be noted that the 'univariate' f takes in a vector of parameters θ, as such the definition of convexity is slightly more complicated - requiring the space for which the function's convexity is defined on to be a *convex set*. We implicitly assume that $\mathbb{R}^p$ is a convex set.

by convexity and L-smoothness. We can minimise $\theta_z = \theta_x - \frac{1}{L}(\nabla\mathcal{L}(\theta_x) - \nabla\mathcal{L}(\theta_y))$, which gives

$$\begin{aligned}\mathcal{L}(\theta_y) - \mathcal{L}(\theta_x) &\leq \langle \nabla\mathcal{L}(\theta_y), \theta_y - \theta_x\rangle - \frac{1}{L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|_2^2 + \frac{1}{2L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|_2^2 \\ &= \langle \nabla\mathcal{L}(\theta_y), \theta_y - \theta_x\rangle - \frac{1}{2L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|_2^2.\end{aligned}$$

Finally, (13) follows since θ_x and θ_y are interchangeable, and adding up both versions of (12) gives the inequality

$$0 \leq \langle \nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x), \theta_y - \theta_x\rangle - \frac{1}{L}\|\nabla\mathcal{L}(\theta_y) - \nabla\mathcal{L}(\theta_x)\|.$$

□

We can now state the theorem for convergence, requiring that $\mathcal{L}(\theta, \hat{\mathbf{y}})$ is L-smooth and that $\mathcal{L}$ is convex. This produces a convergent sequence $(\theta_n)_\infty$ to $\hat{\theta} = \arg\min_\theta \mathcal{L}(\theta, \hat{\mathbf{y}})$ through the gradient descent algorithm described in Section 2.3.1:

Theorem A.3 (Convergence of Gradient Descent Algorithm)**.** *Let $\mathcal{L}$ be L-smooth and let θ_n for $n = 0, 1, \ldots$ be the sequence of iterates generated by the gradient descent algorithm*

$$\theta_{n+1} = \theta_n - \eta\nabla\mathcal{L}(\theta_n).$$

It follows that the sequence $(\theta_n)_\infty$ converges to $\hat{\theta}$, where $\hat{\theta} = \operatorname{argmin}_\theta \mathcal{L}(\theta)$.

Proof. Since $\mathcal{L}$ is convex and L-smooth, and by (13),

$$\begin{aligned}\|\theta_{t+1} - \hat{\theta}\|_2^2 &= \|\theta_t - \hat{\theta} - \eta\nabla\mathcal{L}(\theta_t)\|_2^2 \\ &= \|\theta_t - \hat{\theta}\|_2^2 - \frac{2}{L}\langle\theta_t - \hat{\theta}, \nabla\mathcal{L}(\theta_t)\rangle + \frac{1}{L^2}\|\nabla\mathcal{L}(\theta_t)\|_2^2 \\ &\leq \|\theta_t - \hat{\theta}\|_2^2 - \frac{1}{L^2}\|\nabla\mathcal{L}(\theta_t)\|_2^2.\end{aligned}$$

We see that $(\|\theta_t - \hat{\theta}\|_2^2)_\infty$ is a decreasing sequence in t (which already means it is convergent). By (10), we have that

$$\mathcal{L}(\theta_{t+1}) - \mathcal{L}(\theta_t) \leq \mathcal{L}(\theta_t) - \mathcal{L}(\hat{\theta}) - \frac{1}{2L}\|\nabla\mathcal{L}(\theta_t)\|_2^2. \tag{14}$$

Applying convexity, we have that

$$\begin{aligned}\mathcal{L}(\theta_t) - \mathcal{L}(\hat{\theta}) &\leq \langle\nabla\mathcal{L}(\theta_t), \theta_t - \hat{\theta}\rangle \\ &\leq \|\nabla\mathcal{L}(\theta_t)\|_2\|\theta_t - \hat{\theta}\| \\ &\leq \|\nabla\mathcal{L}(\theta_t)\|_2\|\theta_0 - \hat{\theta}\|.\end{aligned}$$

Combining the above statement with (14) gives

$$\mathcal{L}(\theta_{t+1}) - \mathcal{L}(\theta_t) \leq \mathcal{L}(\theta_t) - \mathcal{L}(\hat{\theta}) - \frac{1}{2L}\frac{1}{\|\theta_0 - \hat{\theta}\|^2}\left(\mathcal{L}(\theta_t) - \mathcal{L}(\hat{\theta})\right)^2. \tag{15}$$

We define $\delta_t = \mathcal{L}(\theta_t) - \mathcal{L}(\hat{\theta})$, $\beta = \frac{1}{2L}\frac{1}{\|\theta_0 - \hat{\theta}\|^2}$, and manipulating (15) gives:

$$\begin{aligned}\delta_{t+1} &\leq \delta_t - \beta\delta_t^2 \\ \beta\frac{\delta_t}{\delta_{t+1}} &\leq \frac{1}{\delta_{t+1}} - \frac{1}{\delta_t} \\ \beta &\leq \frac{1}{\delta_{t+1}} - \frac{1}{\delta_t}.\end{aligned}$$

Summing both sides over $t = 1, \ldots, n-1$ gives

$$(n-1)\beta \leq \frac{1}{\delta_n} - \frac{1}{\delta_1} \leq \frac{1}{\delta_n}.$$

This means that for any given n,

$$\mathcal{L}(\theta_n) - \mathcal{L}(\hat{\theta}) \leq \frac{2L\|\theta_0 - \hat{\theta}\|^2}{n-1}, \tag{16}$$

which means that for any arbitrarily small $\epsilon > 0$, we can find N sufficiently large, such that $\forall n > N$, $|\mathcal{L}(\theta_n) - \mathcal{L}(\hat{\theta})| < \epsilon$. This concludes the proof. □

A more detailed proof, as well as further theorems for convergence rates involving learning rates η, can be found in [8].

B Comparison of ODE Solvers

In this appendix, we provide the Julia code used to compare several numerical ODE solvers, namely the `Euler()`, `RK4()` and `Vern9()` methods. The file containing all the code below can be found in the project's GitHub repository at `ODE_Solvers.ipynb`. The initial value problem evaluated here was $\dot{z} = \cos(t)z, z(0) = 1$ in the interval $[0.0, 10.0]$, which has the trivial solution $z(t) = e^{\sin t}$. We first import relevant packages and define the problem in Julia:

```
using DifferentialEquations, Plots
function ode_problem!(dz, z, p, t)
    dz[1] = cos(t) * z[1]
end
u0 = [1.0]
T = 10.0
tspan = (0.0, T)
prob = ODEProblem(ode_problem!, u0, tspan)
```

We then generate the solutions for each method by discretising the interval and applying the relevant method:

```
times = 0:0.1:T
euler = solve(prob, Euler(), tstops=times)
rk4 = solve(prob, RK4(), tstops=times)
vern9 = solve(prob, Vern9(), tstops=times)
```

The additional code below is used to generate Figure 8, comparing the actual solution with the approximations provided by the numerical ODE solvers. We note that `rk4` and `vern9` produce outputs that do not give the solution at the specified output points (namely `times`), requiring an extra step each to discretise:

```
actualsol = exp.(sin.(times))
plot(times, actualsol, marker=:+, label="Solution",
    title="Numerical Approximations of Solutions to ODE")
plot!(euler, label="Euler")
rk4_discrete = [rk4(time)[1] for time in times]
plot!(times, rk4_discrete, marker=:"circle", markersize=:2, label="RK4")
vern9_discrete = [vern9(time)[1] for time in times]
plot!(times, vern9_discrete, label="Vern9")
```

C A Simple Example of the Adjoint Method

In this appendix, we provide a simple example and try to apply the adjoint method to find the desired gradients. We aim to create a model where, given an input x, the output is xe. We initialise our neural ODE with $f(z(t), \theta, t) = \theta z$, or $\dot{z} = \theta z$ over the timespan $[t_0, t_1]$, where θ, t_0, t_1 are constants. We can fix $t_0 = 0$, and then t_1 and θ are trainable parameters. Suppose that the model has one evaluation point at time t_1. Given an input x, the output will be $xe^{\theta t_1}$. So, the model will perfectly align with our goal if $\theta t_1 = 1$.

After setting up the model, we provide one training data as $z(t_0)$. We can then solve the initial value problem and get $z(t) = z(t_0)e^{\theta(t-t_0)}$, then output $z(t_1) = z(t_0)e^{\theta t_1}$. We also provide the expected output value $c := z(t_0)e$. To use the adjoint method, we first calculate the loss function. For simplicity, we choose the mean square error (MSE) as the loss function. Thus, we have

$$\mathcal{L} = \frac{1}{2}\left(c - z(t_0)e^{\theta t_1}\right)^2.$$

Here we add $\frac{1}{2}$ to simplify the derivatives. For the adjoint method, we need to calculate the derivative for a_{aug} and its derivative $\frac{\mathrm{d}a_{\text{aug}}}{\mathrm{d}t}$. To make it clear, in this example, instead of solving an IVP for a_{aug} in one call, we solve three IVPs with respect to a, a_θ, and a_t numerically, providing an idea of how this method actually works.

For a, we know

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -a\frac{\partial f(z(t), \theta, t)}{\partial z(t)} = -a\theta, \qquad a^{\text{initial}} = z(t_1) - c.$$

Solving this IVP, we get

$$a(t) = (z(t_1) - c)e^{-\theta(t-t_1)}.$$

Next, for a_θ,

$$\frac{\mathrm{d}a_\theta}{\mathrm{d}t} = -a\frac{\partial f(z(t), \theta, t)}{\partial \theta} = -az, \qquad a_\theta^{\text{initial}} = 0.$$

So we have

$$\begin{aligned}\frac{\mathrm{d}a_\theta}{\mathrm{d}t} &= -a(t)z(t)\\ &= -(z(t_1) - c)e^{-\theta(t-t_1)}z(t_0)e^{\theta t}\\ &= -(z(t_1) - c)z(t_0)e^{\theta t_1}.\end{aligned}$$

Here, $c_\theta := -\frac{\mathrm{d}a_\theta}{\mathrm{d}t}$ is a constant. Hence, $a_\theta(t_0) = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = t_1 c_\theta$, and this is the gradient we need for updating θ.

For a_t,

$$\frac{\mathrm{d}a_t}{\mathrm{d}t} = -a\frac{\partial f(z(t), \theta, t)}{\partial t} = 0,$$

$$\begin{aligned}-\left.\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}\right|_{t=t_1} = a_t^{\text{initial}} &= -a(t_1)f(z(t_1), \theta, t)\\ &= -a(t_1)z(t_1)\theta\\ &= -(z(t_1) - c)z(t_1)\theta.\end{aligned}$$

So we can obtain the required gradients to update the time parameter $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t_1} = (z(t_1) - c)z(t_1)\theta$.

After obtaining these gradients, we can use them in the gradient descent to adjust the parameters in the model. Additionally, by setting some random values for θ and t_1, we can see that the gradients we get always point in the right direction to make $t_1\theta$ closer to 1.

D Data Preprocessing of Financial Time Series

In this appendix, we provide the Julia code used to preprocess the financial time series. The file containing all the code below can be found in the project's GitHub repository at `Neural_CDE.ipynb`.

We first read the data into the compiler.

```
using DataFrames, CSV, Plots
amzn = CSV.read(".../AMZN.csv", DataFrame)
```

We plot the original data to obtain a visualisation of the discrete time series data.

```
p1 = plot(amzn."Date", amzn."Adj Close", title="Original Time Series
Data", legend=:topright)
```

Having observed the pattern of the data, we find that the trend of the daily adjusted closing price of Amazon is overall increasing. Thus, the best strategy to scale the data is to use min-max scaling as explained previously.

```
using Statistics
function min_max_scaling(column)
    min_val = minimum(column)
    max_val = maximum(column)
    return (column .- min_val) ./ (max_val .- min_val)
end
amzn_norm = copy(amzn)
for col in names(amzn_norm)
    if eltype(amzn_norm[!, col]) <: Number
        amzn_norm[!, col] = min_max_scaling(amzn_norm[!, col])
    end
end
```

We then have to calculate the relative dates as defined in Section 5.5.2. The first step is to extract the sorted dates and adjusted close values.

```
using Dates
dates = amzn_norm."Date"
print(dates)
adj_close = amzn_norm."Adj Close"
```

Next, we transform the dates into the desired format "yyyy/mm/dd".

```
output_date_format = DateFormat("yyyy/mm/dd")
function parse_mixed_date(date_str)
    date_parsed = nothing
    try
        date_parsed = Date(date_str, DateFormat("m/d/yyyy"))
    catch
    end
    if isnothing(date_parsed)
        try
            date_parsed = Date(date_str, DateFormat("m/dd/yyyy"))
```

```
        catch
        end
    end
    return date_parsed
end
formatted_dates = [Dates.format(parse_mixed_date(date),
    output_date_format) for date in dates]
```

Finally, we can calculate the relative dates.

```
Formatted_dates = Date.(formatted_dates, output_date_format)
init_date = Formatted_dates[1]
rel_dates = []
for x in Formatted_dates
    x = x - init_date
    push!(rel_dates, Dates.value(x))
end
```

The last step is to create a cubic spline using the processed time series data.

```
using DataInterpolations
time_series = collect(zip(rel_dates, adj_close))
cubic_spline_t = DataInterpolations.CubicSpline(rel_dates, rel_dates)
cubic_spline_p = DataInterpolations.CubicSpline(adj_close, rel_dates)
function cubic_spline(t)
    return (cubic_spline_t(t), cubic_spline_p(t))
end
```