EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

CWI Centrum Wiskunde & Informatica

Machine learning for closure models

Master Thesis

H.A. Melchers

Supervisors
prof.dr.ir. B. Koren
dr. V. Menkovski
prof.dr. D.T. Crommelin
dr.ir. B. Sanderse

June 20th 2022

Abstract

Many real-world physical processes, such as fluid flows and molecular dynamics, are understood well enough that their behaviour can be accurately translated into mathematical systems of equations, which can then be solved by a computer algorithm. This process forms the basis of the research field of Scientific Computing and although it is very successful, performing accurate simulations can be computationally expensive. As a result, techniques have been developed for Model Order Reduction (MOR), which aim to drastically reduce the complexity of such systems of equations while sacrificing only little accuracy compared to the original Full-Order Model (FOM). The resulting Reduced-Order Models (ROMs) often need to include a correction (or 'closure') term to account for the error that was introduced when performing the reduction. In recent years, Machine Learning (ML) has become a popular way to obtain such closure terms. However, the research on ML for closure terms differs from more general ML research in that relevant domain knowledge (i.e. applicable laws of physics or statistical observations) can be used to design the ML model. Moreover, ML closure models do not need to learn the entire dynamics of the underlying system but only the error between the real dynamics of the FOM and the approximate dynamics of the ROM.

In this thesis, several sets of experiments are performed that aim to assess the efficacy of simple ML closure models for a number of problems in the form of ordinary or partial differential equations, and to inform future uses of ML in ROM by comparing several ML architectures and training procedures. Even simple ML closure models are found to perform drastically better than models without closure term, while also outperforming 'pure' ML models that do not use prior knowledge as an approximation. Furthermore, models that are formulated to be continuous in time (as the underlying processes are) outperform models that are discrete in time, and models with domain knowledge embedded in their designs outperform models without such properties.

As for training procedures, several methods are compared and although one method clearly outperforms the others, the specific problem considered determines which ODE solvers are applicable, which in turn influences the suitability of different training procedures. Finally, some models are compared that allow for a memory effect, in which future states depend not only on the current state but also on past states. While models with memory effects have been found to perform well in other works, they do not outperform simpler memoryless models on the problem considered in this thesis.

Nevertheless, the value of ML for closure terms is clear since the accuracy of a numerical method can be improved significantly by supplementing the method with a relatively small neural network. However, more research should be done to examine the performance of such closure terms compared to purely numerical methods, preferably using more complex problems for testing. Finally, ML closure models with memory should be examined more critically to see how models can be obtained that do not suffer from overfitting.

Contents

1 Introduction 7
1.1 Model order reduction and closure terms . . . 7
1.1.1 Model order reduction . . . 7
1.1.2 Filters . . . 8
1.1.3 Model closures . . . 8
1.2 The Mori-Zwanzig formalism and memory effects . . . 9
1.3 Machine learning for closure terms . . . 10
1.4 Limitations of related work . . . 11
1.5 Contributions of this thesis . . . 11
1.6 Structure of this thesis . . . 11

2 Related work 13
2.1 Physics-informed Machine Learning . . . 13
2.2 Neural closure terms for reduced-order models . . . 13
2.3 Mori-Zwanzig and models with memory . . . 13
2.4 Auto-encoders and fully latent models . . . 14
2.5 Structure preservation . . . 14
2.6 Neural networks and ordinary differential equations . . . 15

3 Background on training procedures for neural ODEs 17
3.1 Adjoint sensitivity methods . . . 18
3.1.1 Backsolve adjoint method . . . 19
3.1.2 Interpolating adjoint method . . . 21
3.1.3 Quadrature adjoint method . . . 21
3.2 Forward-mode sensitivity . . . 22
3.3 Discrete optimisation for neural ODEs . . . 23
3.4 Algorithm comparison . . . 24
3.5 Notes on automatic differentiation . . . 24
3.6 Neural closure models for stiff ODEs . . . 25
3.7 Other methods for time series prediction . . . 26

4 Effect of including prior knowledge in ML models 27
4.1 Introduction . . . 27
4.2 Burgers' equation . . . 30
4.3 Data generation . . . 30
4.4 Experiments . . . 31
4.4.1 Neural network architecture . . . 31
4.4.2 Training procedure . . . 33
4.5 Results . . . 36
4.6 Conclusion . . . 38

5 Training procedures for neural ODEs **41**
5.1 Overview . . . 41
5.1.1 Derivative fitting . . . 41
5.1.2 Trajectory fitting . . . 43
5.2 The Kuramoto-Sivashinsky equation . . . 44
5.3 Data generation . . . 45
5.4 Evaluation of neural networks for chaotic systems . . . 45
5.5 Experiments . . . 46
5.5.1 Derivative fitting . . . 46
5.5.2 Derivative fitting with Lipschitz regularisation . . . 47
5.5.3 Optimise-then-discretise . . . 51
5.5.4 Discretise-then-optimise . . . 55
5.6 Results . . . 58
5.6.1 IMEX schemes . . . 58
5.6.2 Training methods . . . 58
5.7 Conclusions . . . 60

6 Extending ML models with memory **65**
6.1 Ways to include memory effects . . . 65
6.2 Discrete delay models . . . 66
6.3 ANODE models . . . 68
6.4 The Lorenz '96 model . . . 69
6.5 Data generation . . . 70
6.6 Experiments . . . 71
6.7 Results . . . 71
6.8 Conclusions . . . 76

7 Conclusions and recommendations **79**
7.1 Conclusions . . . 79
7.2 Future work . . . 80

Bibliography **83**

A Notation **89**

B Terminology **91**

C Abbreviations **93**

Chapter 1

Introduction

1.1 Model order reduction and closure terms

A major area of research in Scientific Computing is the numerical solution of Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs). Such equations appear in many different contexts including molecular dynamics, fluid flows, celestial mechanics, chemical reactions, quantum mechanics, and others. For many such differential equations, it is not possible to compute exact solutions and as such solutions are usually approximated using numerical methods. In particular, some types of PDEs usually require a discretisation step which transforms them into ODEs that approximate the original PDE. This is typically the case for PDEs of the form

$$\frac{\partial u}{\partial t}(x,t) = F\left(x, t, u, \left(\frac{\partial u}{\partial x_i}\right)_i, \left(\frac{\partial^2 u}{\partial x_i \partial x_j}\right)_{i,j}, \dots\right) \text{ for } x \in \mathcal{D}, t > 0, \tag{1.1}$$

where $\mathcal{D} \subseteq \mathbb{R}^{d_1}$ and $u : \mathcal{D} \times \mathbb{R}_{\geq 0} \to \mathbb{R}^{d_2}$.

Discretisation of a PDE yields an ODE of the form

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}, t), \tag{1.2}$$

where $\mathbf{u}(t)$ is an approximation to $u(x,t)$ and the function $f(\mathbf{u},t)$ approximates $F(\dots)$.

1.1.1 Model order reduction

The approximation of $u(x,t)$ by a vector $\mathbf{u}(t)$ can be done in different ways, but clearly the form of f depends on the way in which $\mathbf{u}$ approximates u. In general, the exact solution u of (1.1) (supplemented with suitable initial and boundary conditions) will contain behaviour that is not visible in the approximation $\mathbf{u}$. This may be sub-grid variations of u, its high-frequency components, or something else depending on the type of approximation used. As such, the solution u is considered to consist of two components: the **resolved component**, which can be reconstructed from $\mathbf{u}$, and the **unresolved component**, which cannot. Since in many discretisations, the resolved component consists of the large-scale behaviour of u and the unresolved component consists of the small-scale behaviour, the resolved and unresolved components are also referred to as resolved and unresolved scales. From here onward, the resolved component of the solution u will be written as $\overline{u}$ or in vector form as $\mathbf{u}$, and the unresolved component as $\widetilde{u}$. In all cases, it is assumed that there is a bijection between the space of resolved components and $\mathbb{R}^N$, so that the resolved component $\overline{u}$ can be written as a function of $\mathbf{u}$ and vice versa.

A common and simple way to approximate u by a vector is to define the components of $\mathbf{u}$ as approximations of the values of u at discrete points in $\mathcal{D}$, i.e. $\mathbf{u}_i \approx u(x_i)$. This is typical for a number of numerical methods, such as the method of lines, the finite element method, and the finite volume method. Then the resolved component $\overline{u}$ may be a piecewise linear or piecewise constant function with $\overline{u}(x_i) = \mathbf{u}_i$. The function f will approximate partial derivatives of u at the points x_i using finite differences, and use those to compute $\frac{\mathrm{d}u(x_i)}{\mathrm{d}t}$ for $i = 1, \dots, N$. Generally, achieving an accurate model using such a discretisation requires a very fine discretisation, i.e. one that yields an ODE over a very high-dimensional vector $\mathbf{u}(t)$. Such models are often referred to as **full-order models (FOMs)**.

Alternatively, $\overline{u}$ can be related through a set of basis functions as $\overline{u}(x) = \sum_{i=1}^{N} \mathbf{u}_i \varphi_i(x)$. For example, for **Proper Orthogonal Decomposition (POD)** methods, the function φ_1 is chosen to minimise the error of projecting exact solutions of the PDE onto the span of φ_1. Then, φ_2 is chosen to be orthogonal to φ_1 and to minimise error of projecting solutions onto the span of $\{\varphi_1, \varphi_2\}$, and so on. Proper orthogonal decomposition is one of many techniques in the field of **Model Order Reduction (MOR)**. An overview of many different MOR methods is given by Ahmed et al. [1].

1.1.2 Filters

The resolved and unresolved components of u can also be defined directly, as opposed to being defined through a discretisation. This typically happens when a PDE is **filtered**, for example by convolving the entire PDE with a kernel. The point of filtering a PDE is to obtain a new PDE that only describes the large-scale behaviour $\overline{u}$ of the original solution u, and ignores the small scales $\widetilde{u}$. Then, the large-scale behaviour $\overline{u}$ is known to be relatively smooth, and can therefore be approximated well by a discretisation onto a relatively coarse grid.

As an example, consider the non-linear Burgers' equation in one dimension:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) = 0. \tag{1.3}$$

When convolving (1.3) with a filter $G : \mathbb{R} \to \mathbb{R}$ (such as a Gaussian filter), one obtains

$$G * \frac{\partial u}{\partial t} + \frac{1}{2} G * \frac{\partial}{\partial x}\left(u^2\right) = 0,$$
$$\text{where } \left(G * \phi\right)(x) = \int_{\mathbb{R}} G(x-y)\phi(y)\mathrm{d}y.$$

Since convolutions commute with derivatives (under some regularity assumptions), this can be written as

$$\frac{\partial}{\partial t}\left(G * u\right) + \frac{1}{2}\frac{\partial}{\partial x}\left(G * u^2\right) = 0, \tag{1.4}$$

which is almost a PDE in the filtered variable $\overline{u} := G * u$. For brevity, define $\overline{\phi} := G * \phi$ for arbitrary x-dependent functions ϕ, so that (1.4) can be written as

$$\frac{\partial}{\partial t}\overline{u} + \frac{1}{2}\frac{\partial}{\partial x}\overline{u^2} = 0.$$

By adding the term $\frac{1}{2}\frac{\partial}{\partial x}\left(\overline{u}^2 - \overline{u^2}\right)$ on both sides, one obtains

$$\frac{\partial}{\partial t}\overline{u} + \frac{1}{2}\frac{\partial}{\partial x}\overline{u}^2 = \frac{1}{2}\frac{\partial}{\partial x}\left(\overline{u}^2 - \overline{u^2}\right).$$

Note that now, the left-hand side is a function of $\overline{u}$, but the right-hand side depends not only on $\overline{u}$ but also on $\widetilde{u}$ (since u^2 can equivalently be written as $(\overline{u} + \widetilde{u})^2$).

1.1.3 Model closures

Thus, regardless of how accurately the discretised system (1.2) is solved, its solution will generally only yield information about the resolved component of u. Unfortunately, in many PDEs there is an interaction between the resolved and unresolved components of u, meaning that the solution to (1.2) will not be very accurate. To remedy this, the ODE system must be modified by adding one or more terms to the right-hand side that model the effects of the unresolved component on the resolved component. Such terms are called **closure terms**, and must be approximated since the exact closure term, such as $\frac{1}{2}\frac{\partial}{\partial x}\left(\overline{u}^2 - \overline{u^2}\right)$ for Burgers' equation, cannot be computed exactly. A discretised PDE with closure term generally has the form

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \sum_{i=1}^{N_{\mathrm{CL}}} \mathrm{CL}_i(\mathbf{u}),$$

consisting of the discrete approximation f of the original differential operator as well as N_{CL} separate closure terms. Note that one could equivalently define a single closure term $\mathrm{CL}(\mathbf{u}) := \sum_{i=1}^{N_{\mathrm{CL}}} \mathrm{CL}_i(\mathbf{u})$. Discretisations with multiple closure terms are occasionally used when each closure term has a separate physical interpretation, but only cases with a single closure term will be considered in this work:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{CL}(\mathbf{u}). \tag{1.5}$$

A common occurrence of closure terms is in the Large Eddy Simulation (LES) method for incompressible fluid flows. Here, the Navier-Stokes equations for the momentum are filtered and discretised on a grid, and closure terms are added to represent the effect of small scales on the large-scale behaviour of the flow. Since the closure term represents the effects of solution features that are too small to be noticed on the grid, closure terms for LES are also called **subgrid-scale (SGS) models**. Note that in LES and most other PDE discretisations that contain closure terms, the closure term itself is a function of $\mathbf{u}$ so that the entire system is still a system of ODEs.

1.2 The Mori-Zwanzig formalism and memory effects

Related to closure problems is the **Mori-Zwanzig formalism**, named after the seminal works of Mori [53] and Zwanzig [86]. This formalism states that projection of an ODE onto a lower-dimensional subspace leads to memory and noise terms.

As a concrete example, consider the following linear ODE system on $\mathbb{R}^n$:

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{A}\mathbf{x}, \quad \mathbf{x}(0) = \mathbf{x}^{(0)}. \tag{1.6}$$

Suppose that, similar to a PDE, the solution $\mathbf{x}$ consists of a resolved component and an unresolved component. Splitting up $\mathbf{x}$ into into these two components, (1.6) can be written as

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y} &= \mathbf{B}\mathbf{y} + \mathbf{C}\mathbf{z} \\ \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z} &= \mathbf{D}\mathbf{y} + \mathbf{E}\mathbf{z}, \end{cases}, \qquad \text{where } \mathbf{A} = \begin{bmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{bmatrix}, \quad \mathbf{x} = \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix}.$$

The second ODE system can be integrated, yielding the solution

$$\mathbf{z}(t) = e^{t\mathbf{E}}\left(\mathbf{z}(0) + \int_0^t e^{-s\mathbf{E}}\mathbf{D}\mathbf{y}(s)\mathrm{d}s\right),$$

which can be substituted back into the equation for $\mathbf{y}_1$ to yield:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y} &= \mathbf{B}\mathbf{y} + \mathbf{C}e^{t\mathbf{E}}\left(\mathbf{z}(0) + \int_0^t e^{-s\mathbf{E}}\mathbf{D}\mathbf{y}(s)\mathrm{d}s\right) \\ &= \mathbf{B}\mathbf{y} + \int_0^t \underbrace{\mathbf{C}e^{(t-s)\mathbf{E}}\mathbf{D}\mathbf{y}(s)}_{\mathcal{M}(t-s,\mathbf{y}(s))}\mathrm{d}s + \underbrace{\mathbf{C}e^{t\mathbf{E}}\mathbf{z}(0)}_{\mathcal{N}(\mathbf{z}(0),t)}. \end{aligned}$$

More generally, for a non-linear ODE system $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = f(\mathbf{x})$, the state $\mathbf{x}$ can be split up into arbitrary 'resolved' and 'unresolved' components $\mathbf{y}, \mathbf{z}$, such that $\mathbf{y}(t)$ is the *exact* solution of:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \overline{f}(\mathbf{y}) + \int_0^t \mathcal{M}(\mathbf{y}(t-s), s)\mathrm{d}s + \mathcal{N}(\mathbf{z}(t=0), t), \tag{1.7}$$

for correctly chosen $\bar{f}, \mathcal{M}, \mathcal{N}$. Note that this is an ODE system with two extra terms:

- The term $\int_0^t \mathcal{M}(\mathbf{y}(t-s), s)\mathrm{d}s$ depends only on the resolved component $\mathbf{y}$. However, it depends on the entire history of $\mathbf{y}$ rather than its current value. Therefore, this term is known as the **memory term**.
- The term $\mathcal{N}(\mathbf{z}(t=0), t)$ depends on the unresolved component, but only its initial conditions. Since this component is typically not known but may follow some probability distribution, this term is known as the **noise term**.

The function $\overline{f}$ in the first term of the right-hand side of (1.7) represents the effect of the resolved scales on the dynamics of the resolved scales, and can be thought of as a 'truncation' of the original function f, similar to the linear case (1.6) where f corresponds to the matrix $\mathbf{A}$ and $\overline{f}$ to a submatrix $\mathbf{B}$ of $\mathbf{A}$. Importantly, (1.7) is not an approximation, but is equivalent to the original ODE. In theory, solving this ODE system exactly would result in an exact solution for the resolved component of $\mathbf{x}$. However, (1.7) is also not any easier to solve than the original ODE, since finding exact expressions for the functions $\mathcal{M}$ and $\mathcal{N}$ is generally not tractable (see Chorin et al. [10] for a more detailed derivation of the MZ formalism). Nevertheless, this formulation of the problem can serve as a different starting point for creating closure terms. Specifically, it shows that the solution to be obtained can be more accurate if the closure term is allowed to depend on the history of $\mathbf{y}$, rather than just its current value.

Note that while the two above examples are ODEs, in practice a similar reasoning can be used for PDEs. After all, one can accurately discretise a PDE such as (1.1) into an ODE of arbitrarily high dimension using the method of lines, after which the Mori-Zwanzig formalism allows for an ODE in far fewer variables which is still equivalent to the high-dimensional ODE, and therefore arbitrarily close to the original PDE.

1.3 Machine learning for closure terms

Regardless of whether or not the closure term contains a memory effect, finding accurate expressions for closure terms can be difficult. This is not surprising, given that the part of the behaviour that is easy to model is by definition included in the term $f(\mathbf{u})$. In some cases, notably LES, the closure term can be given a physical interpretation (see for example Sagaut [68] for an extensive overview). In general, however, analytical expressions for closure terms are not easy to find, as the extra terms introduced by the MZ formalism are generally not tractable. This has resulted in increased interest for the use of **Machine Learning** (ML) for modelling of closure terms. ML models for closure terms will be referred to as **neural closure models** and are a specific example of the field of **Scientific Machine Learning (SciML)**, which aims to combine scientific computing with machine learning approaches. An overview of goals and challenges in SciML is given by Baker et al. [3]. General problems in the field of neural closure modelling, and SciML in general, include:

- **What prior knowledge to include in the model:** if the underlying dynamics are continuous in time (usually as an ODE or PDE), the ML model can also be chosen to be continuous in time as well. ML models that use neural networks in the definition of an ODE system are called **Neural ODEs (NODEs)**. Such models preserve the ODE structure of the underlying problem, although models that are discrete in time may be simpler to create and train. Similarly, in neural closure modelling the inclusion of the 'approximate' derivative term $f(\mathbf{u})$ can be seen as inclusion of prior knowledge.

  For models that include memory effects derived from the Mori-Zwanzig formalism, there are multiple ways to include such memory effects which in turn affect the types of training procedures that can be used to train such a model.

- **How to enforce physical behaviour:** if the underlying dynamics satisfy certain physical laws, such as conservation laws, the ML model should preferably satisfy those same laws. However, enforcing such behaviour in the model can be difficult, depending on the complexity of the model and the form of law that is enforced. Models that preserve certain physical properties of the underlying model are referred to as **structure preserving**. Structure preservation is an important area of research in both the fields of machine learning and model order reduction.

- **How to train the resulting ML model:** whatever the specific structure of a machine learning model is, training the model can often be done in different ways. For example, when training any model to predict a time series, the length of the predictions made during training may affect the training speed, convergence rate, and accuracy of the model.

Different strategies for the above three problems lead to different ML models.

1.4 Limitations of related work

While there is a wide variety of work available that approaches closure modelling and related problems both from Scientific Computing and Machine Learning perspectives (see Chapter 2), a number of limitations affect the usefulness of this work for drawing more general conclusions:

- First, a significant portion of ML research for ODEs evaluates models on relatively small ODE systems such as the three-variable Lorenz system. These systems are much simpler than many real-world problems such as discretised PDEs and as such conclusions drawn from such models do not necessarily generalise to real-world problems.

- Second, existing ML research for neural ODEs considers problems where less domain knowledge is available than is the case for SciML applications. For example, a neural ODE for an image recognition task may be informed by translation and rotation invariances, but not by approximate ODE dynamics or conservation laws.

- Third, many related works consider a single ML model architecture, which is tested on a couple of data sets and typically compared to a much simpler 'baseline' model. As a result, related work rarely gives general insights regarding the effects of individual model features.

1.5 Contributions of this thesis

This thesis aims to investigate ways to address the three problems listed in Section 1.3 (including prior knowledge, enforcing physical behaviour, and training) in a way that overcomes the limitations of other works outlined above. This is done by comparing a variety of different models on non-trivial scientific computing problems. Firstly, the effect of including different kinds of prior knowledge is investigated. This is done by training a cartesian product of different ML models, each characterised by inclusion/exclusion of different kinds of prior knowledge, on the same data set.

Second, different training methods for continuous-time models are compared since training such models is less straightforward than training discrete models. A known theorem on ODEs is re-interpreted in a way that explains why some training methods may not be suitable since they may yield models that make highly inaccurate predictions despite achieving a low training error.

Finally, a number of different ML models are considered that include a memory effect, motivated by the Mori-Zwanzig formalism. However, these models do not significantly outperform models that do not include memory effects, indicating that for models with memory, a more sophisticated approach may be required in order to obtain accurate models.

1.6 Structure of this thesis

The remainder of this thesis is structured as follows. Chapter 2 lists related research in the field of Machine Learning for Scientific Computing problems. Chapter 3 lists and compares the main different training procedures that can be used for training Neural ODEs. In Chapters 4, 5, and 6, numerical experiments are performed that compare different model architectures and training procedures.

Chapter 4 investigates the effect of including different kinds of prior knowledge on the accuracy of ML models, addressing the first two challenges listed in Section 1.3. To address the third challenge, Chapter 5 empirically evaluates different training procedures by using them to train the same model on the same data set. Chapter 6 extends the previous chapters to models that include a memory effect.

Finally, Chapter 7 summarises the conclusions from previous chapters and makes recommendations regarding directions for future research.

Chapter 2

Related work

2.1 Physics-informed Machine Learning

The use of machine learning to find solutions to differential equations has been the subject of extensive research. In some cases, a neural network is trained to directly output the solution of a differential equation. For example, this is the approach taken by Lagaris et al. [45]. There, an ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}), \mathbf{u}(0) = \mathbf{u}^{(0)}$ is solved by training a neural network $\mathrm{NN}(\cdot;\vartheta)$ such that $\mathbf{u}(t) \approx \mathbf{u}^{(0)} + t \cdot \mathrm{NN}(\mathbf{u}^{(0)}, t; \vartheta)$. This form is chosen so that the initial conditions are satisfied by construction. Similarly, a PDE over a function $u(x), x \in [0, 1]$ with Dirichlet boundary conditions $u(0) = u(1) = 0$ can be trained as $u(x) = x \cdot (1 - x) \cdot \mathrm{NN}(x; \vartheta)$. Then, the neural network NN can be trained by minimising a loss function that uses the difference between the expected value of $\frac{\partial u}{\partial x}$ and its prediction $\frac{\partial}{\partial x}\left[x \cdot (1 - x) \cdot \mathrm{NN}(x; \vartheta)\right]$. Similarly, Raissi et al. [67] treat neural networks directly as functions that are trained to satisfy a given PDE. However, in that work boundary conditions are trained by including them in the loss function, rather than by enforcing them in the design of the neural network.

2.2 Neural closure terms for reduced-order models

The inclusion of prior knowledge in such models is typically done in the form of a neural closure model, so that the neural network only has to model the behaviour that is 'missed' by the prior knowledge. For example, Park and Choi [61] train neural networks to create the subgrid-scale model in LES, and a similar approach is taken by List et al. [48]. A similar but distinct problem is considered by San and Maulik [69]: there, Burgers' equation (1.3) is solved using a Model Order Reduction (MOR) technique called Proper Orthogonal Decomposition (POD), resulting in an approximate ODE for which the closure term is approximated by a neural network.

2.3 Mori-Zwanzig and models with memory

Supported by the Mori-Zwanzig formalism, Ma et al. [50] use Recurrent Neural Networks (RNNs) and Long Short-Term Memory models (LSTMs, see Hochreiter [29]) to model the closure term in a number of one-dimensional and two-dimensional PDEs., with good results. Wang et al. [80] use similar neural network architectures in the context of closure terms for POD models. Note that RNNs and LSTMs allow for a memory effect by updating a hidden (or **latent**) vector with each iteration, thereby allowing information to be remembered in between time steps. Another approach is to use a neural network which is given a finite history of the ODE/PDE solution as input, rather than just the most recent state. This approach is taken by Pawar et al. [63] and Fu et al. [22]. Both papers find that even a small amount of memory (i.e. a small number of past states being passed to the neural network) can significantly increase accuracy of the resulting model.

2.4 Auto-encoders and fully latent models

As mentioned, RNNs and LSTMs can incorporate memory effects by keeping a latent vector that can store memory over multiple time steps, in addition to the 'visible' state $\mathbf{u}(t)$ that approximates the ODE/PDE solution. An alternative approach is to forgo the visible state altogether, resulting in models that work completely in latent space. The resulting model typically consists of three ML models: an encoder/decoder pair $\mathrm{NN}_{\mathrm{enc}}, \mathrm{NN}_{\mathrm{dec}}$ that translates the visible state $\mathbf{u}(t)$ into the hidden state $\mathbf{h}(t)$ and back (i.e. $\mathrm{NN}_{\mathrm{enc}}$ and $\mathrm{NN}_{\mathrm{dec}}$ are approximately each other's inverse), and a third model that is used to perform time stepping over the vector $\mathbf{h}(t)$. The encode-decode pair is known as an **auto-encoder**, introduced by Kramer [43]. This technique is referred to as "non-linear principal component analysis", since in the special case that $\mathrm{NN}_{\mathrm{dec}}$ and $\mathrm{NN}_{\mathrm{dec}}$ are linear functions, the resulting auto-encoder can be found by (linear) principal component analysis. Wiewel et al. [82] use this approach to learn solutions to three-dimensional fluid flow problems. Erichson et al. [18] take a similar approach, but with the additional restriction that the update map $\mathbf{h}(t) \mapsto \mathbf{h}(t + \Delta t)$ is linear, allowing them to easily 'encourage' Lyapunov stability of the resulting iteration by considering the eigenvalues of this linear map in the loss function. At first sight, it may appear unlikely that non-linear dynamical systems can be turned into linear systems through a non-linear transformation. However, there are cases in which this can be done. For example, the **Cole-Hopf** transformation, named after the works of Cole [11] and Hopf [30], transforms the non-linear viscous Burgers equation into the linear heat equation:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) + \varepsilon\frac{\partial^2 u}{\partial x^2} \quad \Longleftrightarrow \quad \frac{\partial v}{\partial t} = \varepsilon\frac{\partial^2 v}{\partial x^2}, \\ \text{where } v(x,t) &= \exp\left[-\frac{1}{2\varepsilon}\int_0^x u(s,t), \mathrm{d}s\right], \qquad u(x,t) = -2\varepsilon\frac{\partial}{\partial x}\log v(x,t). \end{aligned}$$

Supported by this example, Gin et al. [23] train neural networks to find non-linear transformations that linearise PDEs, in the same way that the above transformation $u \mapsto v$ linearises Burgers' equation. A similar approach is taken by Lusch et al. [49].

An interesting variant on fully latent-space models is the approach called **reservoir computing**: here, the time stepping function $\mathbf{h}(t) \mapsto \mathbf{h}(t + \Delta t)$ is not trained, but simply chosen randomly, leaving only the auto-encoder to be trained. Vlachas et al. [79] and Pathak et al. [62] apply reservoir computing approaches to high-dimensional dynamical systems with good results.

2.5 Structure preservation

An overview of model order reduction techniques for fluid dynamics problems is given by Ahmed et al. [1]. Doing model order reduction in a structure preserving way is typically very problem-dependent. An example of structure preserving MOR for incompressible fluid flows is given by Sanderse [70]. Note that preserving stability, momentum conservation, and energy conservation requires specific choices for the data-driven decomposition, the spatial discretisation of the PDE, and the ODE solver used for the resulting ODE.

An overview of structure preservation for deep learning is given by Celledoni et al. [7]. For dynamical systems that have a Hamiltonian structure, preserving this structure in a machine learning model has been done with Symplectic Neural Networks (Jin et al. [34]), Hamiltonian Neural Networks (Greydanus et al. [25]) and Lagrangian Neural Networks (Cranmer et al. [13]).

Note that structure in a PDE can be in the form of conservation laws, but also in the form of symmetries such as translational or rotational invariance. Ling et al. [46] show two ways to enforce rotational invariance in a neural network: either by training the neural network on rotated copies of the training data, or by modifying the neural network to take inputs that are transformed to be rotationally invariant. The latter approach is found to be much more computationally efficient, and furthermore has the advantage that the rotational invariance is satisfied exactly (or at least up to numerical precision), rather than only being satisfied approximately as a result of the training procedure. The same strategy is used by Ling et al. [47] with positive results.

2.6 Neural networks and ordinary differential equations

The combination of neural networks and ODEs has been the subject of research even from a purely machine learning perspective. Since Neural ODEs (NODEs) were described by Chen et al. [9] as a continuous-time analogue of Residual Neural Networks (ResNets, see He et al. [27]), many variants and extensions have been proposed that include latent spaces (Augmented Neural ODEs or ANODEs, Dupont et al. [16]), control signals (Neural Controlled DEs, see Kidger et al. [41]), noise (Neural Stochastic Differential Equations, see Tzen and Raginsky [77]), and memory effects (Neural Delay Differential Equations, Zhuet al. [85]). The previously mentioned Hamiltonian and Lagrangian Neural Networks are also extensions of NODEs, and are themselves special cases of Second Order NODEs (SONODEs, see Norcliffe et al. [55]). However, a comparison study by Botev et al. [6] indicates that enforcing Hamiltonian or Lagrangian structure does not result in significantly improved accuracy over simpler neural ODEs.

Chapter 3

Background on training procedures for neural ODEs

As described in Chapter 2, one type of model that is of specific interest for machine learning applications is the Neural ODE, or NODE. Such models are also particularly interesting to consider for closure modelling, since in such applications the dynamics that the ML model aims to learn are often already written in the form of ODEs. However, while neural ODEs are an attractive way to incorporate machine learning elements into scientific computing, training such models is somewhat complicated. In this chapter, different methods of training NODEs are described, and their theoretical properties are compared. While this chapter only summarises existing work, knowledge of different training procedures and their properties is important since the performance of different methods (both in terms of computational efficiency and accuracy of the resulting model) varies between training methods.

For now, the focus will be on NODEs without any memory effect, meaning that the model can be expressed as an ODE system over the vector $\mathbf{u} \in \mathbb{R}^{N_x}$:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}; \vartheta), \tag{3.1}$$

where the function f combines the original ODE function (if present) with the neural closure term. This covers both 'pure' neural ODEs of the form $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u}; \vartheta)$, and neural closure models of the form $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f_{\text{original}}(\mathbf{u}) + \mathrm{NN}(\mathbf{u}; \vartheta)$. The vector $\vartheta \in \mathbb{R}^{N_\vartheta}$ contains the trainable parameters of the neural network. In order to train this neural ODE, i.e. to find a vector of parameters ϑ that minimises some loss function $\mathcal{L}$, it is usually required that there is a way to compute $\frac{\partial \mathcal{L}}{\partial \vartheta}$ so that some variant of gradient descent can be used for optimisation.

The loss function $\mathcal{L}$ can be chosen in multiple ways, and the choice of loss function determines the training procedure. Two common choices for loss functions are as follows.

- **Derivative fitting**. In derivative fitting, the output of the function f is directly compared to reference data. In this case, the training data consists of reference states $\mathbf{u}_{\text{ref}}$ and their time derivatives $\left[\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right]_{\text{ref}}$. Then, given N_p of such states and derivatives, as well as some loss function $\mathcal{L}$, such as a mean-square error, the training procedure aims to solve the following minimisation problem:

$$\min_{\vartheta} \sum_{i=1}^{N_p} \mathcal{L}\left(f(\mathbf{u}_{\text{ref},i}; \vartheta), \left[\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right]_{\text{ref},i} \right). \tag{3.2}$$

- **Trajectory fitting**. In trajectory fitting, also referred to as a solver-in-the-loop setup, (3.1) is solved using an ODE solver, and the resulting trajectory $\mathbf{u}_{\text{predict},i}(t)$ is compared to a reference trajectory using some loss function $\mathcal{L}$. For this training procedure, the training data consists of initial states $\mathbf{u}_{\text{ref},i}(0)$ and corresponding trajectories $\mathbf{u}_{\text{ref},i}(t)$, and the training procedure solves the following minimisation problem:

$$\min_{\vartheta} \sum_{i=1}^{N_p} \mathcal{L}\left(\mathbf{u}_{\text{predict},i}(\cdot), \mathbf{u}_{\text{ref},i}(\cdot)\right) \tag{3.3}$$

$$\text{where } \frac{\mathrm{d}\mathbf{u}_{\text{predict},i}}{\mathrm{d}t} = f\left(\mathbf{u}_{\text{predict};i}; \vartheta\right) \text{ and } \mathbf{u}_{\text{predict},i}(0) = \mathbf{u}_{\text{ref},i}(0). \tag{3.4}$$

Note that the loss function $\mathcal{L}$ compares two trajectories rather than two vectors. Thus, the objective function of (3.3) depends on the parameters ϑ through the ODE problem (3.4). The specific form of $\mathcal{L}$ can be arbitrary, but some examples are:

$$\begin{aligned}
\text{integral loss: } \mathcal{L}(\mathbf{u}_{\text{ref}}(\cdot), \mathbf{u}_{\text{predict}}(\cdot)) &= \int_0^T \|\mathbf{u}_{\text{ref}}(t) - \mathbf{u}_{\text{predict}}(t)\|^2 \, \mathrm{d}t \\
\text{summation loss: } \mathcal{L}(\mathbf{u}_{\text{ref}}(\cdot), \mathbf{u}_{\text{predict}}(\cdot)) &= \sum_{i=1}^{N_t} \|\mathbf{u}_{\text{ref}}(t_i) - \mathbf{u}_{\text{predict}}(t_i)\|^2 \\
\text{end-point loss: } \mathcal{L}(\mathbf{u}_{\text{ref}}(\cdot), \mathbf{u}_{\text{predict}}(\cdot)) &= \|\mathbf{u}_{\text{ref}}(T) - \mathbf{u}_{\text{predict}}(T)\|^2 .
\end{aligned}$$

In [50], these two strategies are referred to as direct training and coupled training, respectively.

If derivative fitting is used, the training procedure is relatively straightforward: given an input $\mathbf{u}_{\text{ref}}$ and desired output $\left[\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right]_{\text{ref}}$ from the training data, the gradient of the loss function can be computed easily by applying the chain rule. However, for trajectory fitting the gradient computation is not so simple, since it depends on the derivatives of the predicted state $\mathbf{u}(t_i)$ with respect to the parameters. Existing research has resulted in several known methods for performing the gradient computation step, some of which will be described here. A more extensive overview including detailed performance characteristics is given by Rackauckas et al. [66]. This section will use the same terminology as the above paper, which is also the terminology used by the `DifferentialEquations.jl` [65] software package. Note that differentiating the solution $\mathbf{u}(t)$ of an ODE, or some function of $\mathbf{u}(t)$, with respect to the parameters ϑ is a form of local sensitivity analysis, and as such methods for gradient computation are also referred to as sensitivity methods. Rackauckas et al. [66] compares five major sensitivity methods. The first three, called the **backsolve adjoint method**, the **interpolating adjoint method**, and the **quadrature adjoint method**, compute the gradients as the solution of a second ODE and will be described in Section 3.1. The fourth method, called the **forward-mode method**, instead augments the first ODE with additional variables to allow computing the gradients. This method is described in Section 3.2. The fifth method, called **discrete sensitivity analysis via automatic differentiation**, treats the ODE solver as a discrete process and differentiates through them in the same way as differentiation through layers of a neural network is done. This method is described in Section 3.3.

3.1 Adjoint sensitivity methods

The gradient computation method used by Chen et al. [9], as well as many variations on it, perform gradient computation using an **adjoint sensitivity method**, in which the gradient $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta}$ is computed as the solution of a second ODE. While the procedure given there assumes that the loss function depends only on the final state $\mathbf{u}(T)$, it can be extended to loss functions that depend on the entire predicted trajectory, either through an integral or a summation (for example by Massaroli et al. [51]). In this section, the emphasis will be on loss functions that are defined as a sum of expressions depending on the predicted and actual trajectories at discrete points in time, such as the example in (3.3):

$$\mathcal{L} = \sum_{i=1}^{N_t} L\left(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)\right).$$

The adjoint sensitivity methods all use the adjoint ODE [64] to compute gradients. This adjoint ODE involves the state vector $\mathbf{u}$, but also two time-dependent vectors $\mathbf{y}^\top \in \mathbb{R}^{N_x}, \mathbf{z}^\top \in \mathbb{R}^{N_\vartheta}$.

3.1.1 Backsolve adjoint method

The backsolve method is described in Algorithm 1.

Algorithm 1 Gradient computation with the backsolve adjoint method

```
1: procedure BACKSOLVE_ADJOINT($f, \vartheta, T, L, t_{1\cdots N_t}, \mathbf{u}_{\mathrm{ref}}(t_{0\cdots N_t})$)
2:     Solve $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u};\vartheta), \mathbf{u}(0) = \mathbf{u}_{\mathrm{ref}}(0)$ from $t = 0$ to $t = T$.
3:     Solve the adjoint ODE system
```

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u} = f(\mathbf{u};\vartheta), \quad \mathbf{u}(T) = \mathbf{u}(T), \tag{3.5a}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y}^\top = -\mathbf{y}^\top \frac{\partial}{\partial \mathbf{u}} f(\mathbf{u};\vartheta), \quad \mathbf{y}(T) = \mathbf{0}, \tag{3.5b}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z}^\top = -\mathbf{y}^\top \frac{\partial}{\partial \vartheta} f(\mathbf{u};\vartheta), \quad \mathbf{z}(T) = \mathbf{0}, \tag{3.5c}$$

```
       from $t = T$ to $t = 0$. At each $t = t_i$, update $\mathbf{y}(t) \leftarrow \mathbf{y}(t) + \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\mathrm{ref}}(t_i))$.
4:     Then $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta}$ is given by $\mathbf{z}(0)$.
```

There are a number of important things to note about this algorithm:

- First, although the vector $\mathbf{y}(t)$ is the solution of an ODE, it is also subject to discrete jumps at time points t_i. Since not all ODE solvers support such jumps, it may be necessary to perform some passes by solving each time interval $[t_i, t_{i+1}]$ separately. However, some ODE software libraries, including `DifferentialEquations.jl`, support callbacks [19] which can be used to run arbitrary code at the given time points t_i.

- Second, both the forward and adjoint ODEs can be solved by any ODE solvers and with any time step. Importantly, since only the values $\mathbf{u}(T)$ are required, the time step used for the forward ODE can be chosen to as large as the accuracy and stability requirements allow. For the adjoint ODE, the time step can still be mostly arbitrary although since the vector $\mathbf{y}(t)$ must be updated at each t_i, the time step cannot be greater than the time difference between subsequent t_i's.

- Third, this algorithm solves the original ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u};\vartheta)$ forward in time, and then again backwards in time together with $\mathbf{y}$ and $\mathbf{z}$ to compute the gradients. While the solution $\mathbf{u}(t)$ of the forward ODE could also be stored and re-used (see Section 3.1.2), the formulation as in Algorithm 1 has the advantage that the solution of the forward ODE does not need to be stored. As a result, the backsolve method requires $\mathcal{O}(N_x + N_\vartheta)$ memory regardless of the number of time steps taken to solve the ODE.

Unfortunately, the fact that the ODE for $\mathbf{u}$ must be solved in reverse makes this method unsuitable for many ODE problems that are discretised PDEs. This is because many PDEs are ill-posed when simulated backwards in time. For example, the diffusion equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ has the property that small irregularities in u are smoothed out as time progresses. When solving this PDE backwards, small irregularities will instead be amplified. As a result, small irregularities that exist in $u(T)$ due to rounding errors will often result in a solution $u(0)$ that is essentially meaningless, containing many spurious oscillations.

An example of this problem is shown in Figure 3.1. Here, the diffusion equation $\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}$ with $\nu = 0.001$ and boundary conditions $u(0,t) = -1, u(1,t) = 1$ is discretised into an ODE over a vector $\mathbf{u}(t) \in \mathbb{R}^{100}$ using the simple second-order finite-difference approximation $\frac{\partial^2 u}{\partial x^2}(x_i) \approx \frac{1}{\Delta x^2}(\mathbf{u}_{i-1} - 2\mathbf{u}_i + \mathbf{u}_{i+1})$. The ODE is solved forwards in time until $t = 10$, and then from $\mathbf{u}(10)$ backwards in time until $t = 9$. Both solutions were computed using the fourth-order, five-stage SSPRK method from Spiteri and Ruuth [73] (in their work, the algorithm is referred to as SSP$(5,4)$), with a time step of $\Delta t = 10^{-4}$. Notice that in the forward solution, the solution becomes smoother over time as sharp changes in $\mathbf{u}$ are dampened. In the reverse solution, however, these sharp changes are instead amplified yielding a meaningless solution that is dominated by spurious oscillations.

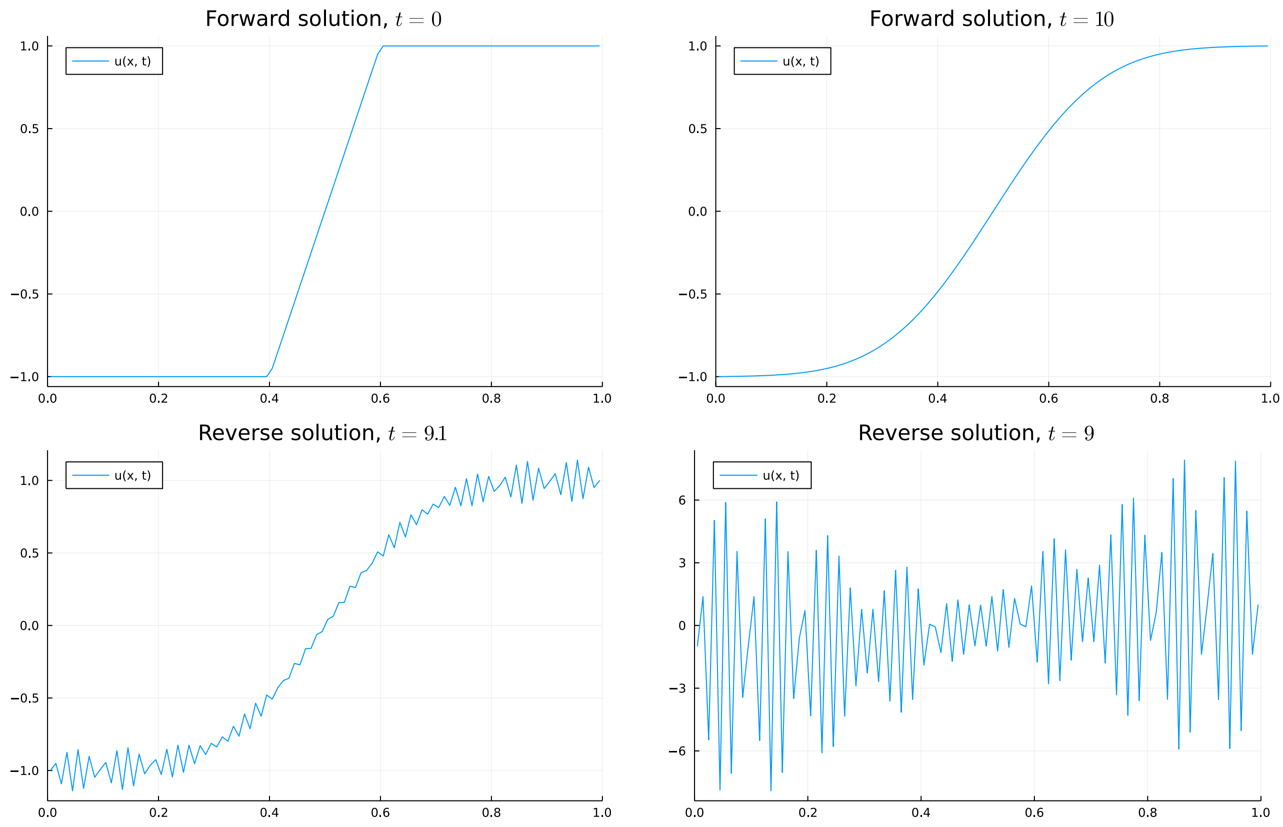

Figure 3.1: Four snapshots of solutions of the diffusion equation $\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}$ with $\nu = 0.001$, discretised into 100 finite volumes. Top left: the initial state $\mathbf{u}(0)$. Top right: the state $\mathbf{u}(t)$ at $t = 10$ obtained by solving the discretised PDE. Bottom left: the state at $t = 9.1$ by solving the ODE backwards from $t = 10$. Bottom right: the state at $t = 9$ by solving the ODE backwards from $t = 10$.

3.1.2 Interpolating adjoint method

The fact that reverse solutions can be unstable means that the backsolve method is not suitable for all types of ODEs. The interpolating adjoint method adapts the backsolve method to avoid computing reverse solutions of $\mathbf{u}(t)$ altogether (i.e. omitting (3.5a) from the adjoint ODE), instead using the solution from the forward pass. This is shown in Algorithm 2.

Algorithm 2 Gradient computation with the interpolating adjoint method

```
1: procedure INTERPOLATING_ADJOINT($f, \vartheta, T, L, t_{1\cdots N_t}, \mathbf{u}_{\mathrm{ref}}(t_{0\cdots N_t})$)
2:     Solve $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u};\vartheta), \mathbf{u}(0) = \mathbf{u}_{\mathrm{ref}}(0)$ from $t = 0$ to $t = T$.
3:     Solve the adjoint ODE system
           $$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y}^\top = -\mathbf{y}^\top \frac{\partial f}{\partial \mathbf{u}}(\mathbf{u}(t), \vartheta), \quad \mathbf{y}(T) = \mathbf{0}, \qquad (3.6a)$$
           $$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z}^\top = -\mathbf{y}^\top \frac{\partial f}{\partial \vartheta}(\mathbf{u}(t), \vartheta), \quad \mathbf{z}(T) = \mathbf{0}, \qquad (3.6b)$$
       from $t = T$ to $t = 0$. At each $t = t_i$, update $\mathbf{y}(t) \leftarrow \mathbf{y}(t) + \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\mathrm{ref}}(t_i))$.
4:     Then $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta}$ is given by $\mathbf{z}(0)$.
```

Note that since the forward solution $\mathbf{u}(t)$ is now re-used, the adjoint ODE system is actually linear: specifically, (3.6a) and (3.6b) can be written as

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} = -\mathbf{A}(t)\mathbf{y}, \quad \text{where} \quad \mathbf{A}(t) = \left[\tfrac{\mathrm{d}f}{\mathrm{d}\mathbf{u}}(\mathbf{u}(t);\vartheta) \quad \tfrac{\mathrm{d}f}{\mathrm{d}\vartheta}(\mathbf{u}(t);\vartheta)\right]^\top.$$

Under the assumption that the Jacobian of f (with respect to both $\mathbf{u}$ and ϑ) is bounded for all $t \in [0, T]$, the adjoint ODE system from Algorithm 2 is well-posed. As a result, the interpolating adjoint method is suitable for more ODEs (including discretisations of the diffusion equation such as the example in Figure 3.1), at the cost of higher memory usage due to the fact that the solution $\mathbf{u}(t)$ of the forward ODE must be stored. Note that many ODE solvers support dense output, in which additional data is kept so that values of $\mathbf{u}(t)$ in between time steps can be reconstructed using a higher-order (i.e. better than linear) interpolation. If only a linear interpolation of $\mathbf{u}(t)$ is available, more snapshots of the forward solution will need to be kept to offset the loss in accuracy due to the lower-order accuracy of this interpolation.

3.1.3 Quadrature adjoint method

The quadrature method adapts the interpolation method in a way that further increases memory usage but can be faster. This method utilises the fact that $\mathbf{z}$ does not appear in the right-hand side of (3.6a) or (3.6b). This means that it is possible to first compute the trajectory $\mathbf{y}(t)$ of the adjoint ODE by solving only (3.6a), after which $\mathbf{z}(0)$ can be computed as an integral (using e.g. Gauss quadrature), resulting in Algorithm 3. Computing the integral using a quadrature requires the solution $\mathbf{y}(t)$ at specific points t corresponding to the quadrature nodes. This can be done by choosing the time steps such that the ODE solution is computed at exactly these times, or by interpolating the solution from nearby snapshots.

While the quadrature method requires more memory since both the trajectories $\mathbf{u}(t)$ and $\mathbf{y}(t)$ must be kept in memory, this method can be faster in many cases due to the much smaller adjoint ODE (note that this ODE is now only over N_x variables, rather than $N_x + N_\vartheta$ for the interpolating adjoint method and $2N_x + N_\vartheta$ for the backsolve method).

Also, although the integral of equation (3.8) can in theory be computed by any algorithm for integrating functions, the fact that $\mathbf{y}(t)$ has discontinuities at each t_i means that it may be beneficial to split the integral up over the interval $(t_i, t_{i+1}), i = 1, 2, \ldots, N_t - 1$:

$$\int_0^T \mathbf{y}(t)^\top \frac{\partial}{\partial \vartheta} f(\mathbf{u}(t);\vartheta) = \sum_{i=1}^{N_t-1} \int_{t_i}^{t_{i+1}} \mathbf{y}(t)^\top \frac{\partial}{\partial \vartheta} f(\mathbf{u}(t);\vartheta). \qquad (3.9)$$

Algorithm 3 Gradient computation with the quadrature adjoint method

```
1: procedure QUADRATURE_ADJOINT($f, \vartheta, T, L, t_{1 \cdots N_t}, \mathbf{u}_{\text{ref}}(t_{0 \cdots N_t})$)
2:     Solve $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}; \vartheta), \mathbf{u}(0) = \mathbf{u}_{\text{ref}}(0)$ from $t = 0$ to $t = T$.
3:     Solve the adjoint ODE system
           $$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y}^\top = -\mathbf{y}^\top \frac{\partial f}{\partial \mathbf{u}}(\mathbf{u}(t), \vartheta), \quad \mathbf{y}(T) = \mathbf{0}, \qquad (3.7)$$
       from $t = T$ to $t = 0$. At each $t = t_i$, update $\mathbf{y}(t) \leftarrow \mathbf{y}(t) + \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i))$.
4:     Then $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta}$ is given by
           $$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta} = \int_0^T \mathbf{y}(t)^\top \frac{\partial}{\partial \vartheta} f(\mathbf{u}(t); \vartheta). \qquad (3.8)$$
```

3.2 Forward-mode sensitivity

While the above three methods all use the adjoint ODE, another possibility is to compute gradients using only a single 'forward' pass. For this, define

$$\mathbf{v}(t) = \sum_{i=1}^{N_t} \mathbf{1}_{t_i \le t} \frac{\partial L}{\partial \vartheta}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)),$$
$$\mathbf{W}(t) = \frac{\partial \mathbf{u}(t)}{\partial \vartheta}.$$

In words, $\mathbf{v} \in \mathbb{R}^{N_\vartheta}$ accumulates the gradient of the loss function with respect to ϑ as t increases, and $\mathbf{W} \in \mathbb{R}^{N_x \times N_\vartheta}$ stores the derivative of the current state $\mathbf{u}(t)$ with respect to the parameters ϑ. Then $\mathbf{W}(t)$ is the solution of another ODE, which can be solved simultaneously with $\mathbf{u}$. At each time point t_i, the vector $\mathbf{v}$ can be updated using the chain rule:

$$\begin{aligned}\frac{\partial L}{\partial \vartheta}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)) &= \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)) \frac{\partial \mathbf{u}(t_i)}{\partial \vartheta} \\ &= \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)) \mathbf{W}(t_i).\end{aligned}$$

This results in Algorithm 4.

Algorithm 4 Gradient computation with the forward method

```
1: procedure FORWARD_SENSITIVITY($f, \vartheta, T, L, t_{1 \cdots N_t}, \mathbf{u}_{\text{ref}}(t_{0 \cdots N_t})$)
2:     Set $\mathbf{v} \leftarrow \mathbf{0}$
3:     Solve the forward ODE system
           $$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u} = f(\mathbf{u}; \vartheta), \qquad \mathbf{u}(0) = \mathbf{u}_{\text{ref}}(0), \qquad (3.10a)$$
           $$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{W} = \frac{\partial f(\mathbf{u}; \vartheta)}{\partial \mathbf{u}}\mathbf{W} + \frac{\partial f}{\partial \vartheta}(\mathbf{u}, \vartheta), \qquad \mathbf{W}(0) = \mathcal{I}, \qquad (3.10b)$$
       from $t = 0$ to $t = T$. At each $t = t_i$, update $\mathbf{v} \leftarrow \mathbf{v} + \frac{\partial L}{\partial \mathbf{u}}(\mathbf{u}(t_i), \mathbf{u}_{\text{ref}}(t_i)) \mathbf{W}(t_i)$.
4:     Then $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\vartheta}$ is given by $\mathbf{v}$.
```

Note that similar to the adjoint ODE methods, this algorithm requires solving an ODE system with discrete jumps at set time points, which may need to be replaced by a sequence of ODEs over the individual time intervals (t_i, t_{i+1}) if such discrete jumps are not supported by the software library that is used.

Similar to the backsolve adjoint method, the forward sensitivity method does not require storing any trajectories in order to compute the gradients, making the total memory usage independent of the number of time steps taken by the ODE solver. Furthermore, since the forward sensitivity method only solves a forward ODE, it avoids the stability problems of the backsolve adjoint method. As a result, the forward sensitivity method is the only method described here that can compute gradients for irreversible ODEs with constant memory usage. The main disadvantage of this method, however, is that while the memory usage may be constant, the ODE system is much larger than for any of the adjoint methods due to the inclusion of the $N_x \times N_\vartheta$ matrix $\mathbf{W}$.

3.3 Discrete optimisation for neural ODEs

The above training methods all work either by solving an additional adjoint ODE system, or by augmenting the existing ODE in order to compute gradients. These strategies are also called “optimise-then-discretise”, since the gradient computation (as used in the optimisation procedure) is formulated as an ODE which is then discretised in time by an ODE solver. A problem with this approach is that the gradients are not computed exactly, since their computation is subject to the discretisation error of the adjoint ODE. In practice, this negatively affects the accuracy of the trained model since the loss function is never minimised exactly, even after training for arbitrarily long time. One solution is to decrease the step size of the ODE solver as training progresses, increasing the accuracy of the computed gradients but also slowing down the training procedure.

Another option is to compute gradients by directly differentiating through the ODE solver, also called “discretise-then-optimise”. In this setting, the function f is ‘embedded’ into an ODE solver, usually (although not necessarily) with a fixed time step Δt, resulting in a model that directly predicts $\mathbf{u}(t + \Delta t)$ from input $\mathbf{u}(t)$. Gradient computation is then done by back-propagating through the ODE solver instead of solving the adjoint ODE.

As mentioned by Kidger [40], the use of adjoint ODEs for training sacrifices some accuracy in the gradient, since the adjoint ODE is not solved exactly. The largest benefit of the adjoint ODE methods is typically listed as the lower memory usage, although this is only the case for the backsolve adjoint method. For neural closure models, however, the adjoint ODE methods have the additional advantage that they do not require the use of a differentiable solver. Back-propagating through an ODE solver may not be possible if the ODE solver is used as a ‘black-box’ algorithm. Furthermore, training neural closure models for stiff ODEs may require an implicit ODE solver, i.e. a solver that solves one or multiple linear on non-linear systems of equations in each time step. Back-propagating through such implicit solvers is possible as a result of the implicit function theorem and is of interest in the general Machine Learning space (see for example Kolter et al. [15] and Kawaguchi [38]). While back-propagating through implicit layers using the implicit function theorem is faster than back-propagating through the iterations of whatever numerical algorithm is used to compute the solution, it is still slower than explicit layers due to the need to solve linear systems of equations during back-propagation. Furthermore, differentiating through non-linear solves may not be possible if the non-linear solver used is a black-box algorithm for which no back-propagation procedure exists.

Nevertheless, the discretise-then-optimise approach has the potential to yield more accurate models than the optimise-then-discretise method, due to two factors:

- As mentioned, the optimise-then-discretise approach only yields approximations for the gradients, due to the temporal discretisation of the adjoint ODE. By contrast, discretise-then-optimise yields gradients that are correct up to perturbations due to rounding.
- Intuitively, when training with the discretise-then-optimise approach, the time step is usually fixed meaning that the error due to temporal discretisation of the (forward) ODE may be predictable, and therefore that the neural network may be able to correct for this error.

Onkel et al. [57] compare the two approaches to two problems, including a time series regression similar to the trajectory fitting problem described earlier in this section. Their findings indicate that training with discretise-then-optimise results in more computationally efficient training (fewer seconds per epoch), as well as faster convergence (fewer epochs required to reach a given level of accuracy).

Table 3.1: An overview showing the ODE sizes and memory requirements of the four sensitivity methods described above. Here, N_T is the number of time steps taken by the ODE solver in the forward or adjoint ODEs.

Algorithm	Forward ODE size	Adjoint ODE size	Memory footprint
Backsolve adjoint	N_x	$2N_x + N_\vartheta$	$2N_x + N_\vartheta$
Interpolating adjoint	N_x	$N_x + N_\vartheta$	$N_T N_x + N_\vartheta$
Quadrature adjoint	N_x	N_ϑ	$2N_T N_x + N_\vartheta$
Forward sensitivity	$N_x + N_x N_\vartheta$	-	$N_x + N_x N_\vartheta$
Discrete sensitivity	N_x	-	$N_T N_x$

3.4 Algorithm comparison

Table 3.1 shows how the five methods described above compare in their ODE sizes and total memory requirements. Note that N_T, the number of time steps taken by the ODE solver, is not known a priori but depends on the stiffness of the ODE, the length of the time interval $[0, T]$ over which the ODE solution is computed, and the ODE solver used. In the performance testing done by Rackauckas et al. [66], it is concluded that the forward sensitivity method is only competitive for small ODEs ($N_x + N_\vartheta < 100$), although in those cases even better performance can be achieved simply by differentiating through the ODE solver directly. For larger systems, the quadrature adjoint method is preferable for stiff ODEs, whereas for non-stiff ODEs the interpolating adjoint method performs better.

As for accuracy, the three adjoint methods as well as the forward sensitivity method are approximately equally accurate: all four methods introduce error due to the fact that the systems of ODEs are not solved exactly, and this error is expected to be approximately equal for different methods. The discrete sensitivity method is expected to yield more accurate gradients than the other four algorithms, due to the absence of temporal discretisation errors in the gradient computation. While not all methods are applicable in all situations, Chapter 5 will test the interpolating adjoint and discrete sensitivity methods in Sections 5.5.3 and 5.5.4, respectively.

3.5 Notes on automatic differentiation

Note that both derivative fitting and trajectory fitting require derivatives of several functions. Rather than specifying these derivatives manually, an increasingly popular method is to define them using **automatic differentiation** (AD), which has applications in machine learning and optimisation, but also in non-linear equation solvers, stiff ODE solvers, and other applications that involve derivatives of functions. For a survey of AD techniques and applications in machine learning, see [4]. While the use of AD is largely an implementation detail, the specific type of AD used can have a large effect on performance. Specifically, AD techniques broadly fall into two categories, called **forward-mode** [35] and **reverse-mode** [72].

- When computing a derivative $\mathbf{J} = \frac{\mathrm{d}}{\mathrm{d}\mathbf{u}} f(\mathbf{u})$ (whether this is a scalar, vector, gradient, or Jacobian), forward-mode AD works by computing the gradient $\nabla_{\mathbf{u}} r$ for each scalar sub-expression r that occurs in the function f. By composing these gradients using standard differentiation rules (product rule, chain rule, and so on), the derivative $\frac{\mathrm{d}f}{\mathrm{d}\mathbf{u}}$ can be computed. These gradients are computed in the same order as the sub-expressions themselves.

- In reverse-mode AD, derivatives are computed by computing $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}r}$ for each sub-expression r, where $\mathbf{y} = f(\mathbf{u})$. This is done by first computing $f(\mathbf{u})$ as usual, but storing the values of all intermediate expressions r (forward pass). Then, the derivatives $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}r}$ are computed in reverse order, eventually leading to the Jacobian $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{u}}$ (backward pass). This can be seen as a generalisation of back-propagation for multi-layer perceptrons. In fact, reverse-mode AD is also sometimes referred to as simply 'back-propagation'.

To differentiate a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the number of forward-mode AD operations required to compute the Jacobian is approximately n times the number of operations required to compute $f(\mathbf{u})$ itself. This is due to the fact that when differentiating, each operation on scalars r_1, r_2 that appears in f also involves operations on their gradients, which are vectors in $\mathbb{R}^n$. By contrast, reverse-mode AD requires approximately m times more work to differentiate than to just evaluate f, since each scalar sub-expression is now extended with the derivative of $\mathbf{y} \in \mathbb{R}^m$ with respect to this sub-expression. Additionally, reverse-mode AD requires more memory to differentiate, since the sub-expressions that occur in f must be saved in order to perform a backward pass.

As a result, when computing a Jacobian $\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}} f(\mathbf{u})$, which type of automatic differentiation is more efficient depends on the dimensions of the Jacobian, with forward-mode being more efficient if the Jacobian has far fewer columns than rows (meaning f has far more outputs than inputs), and reverse mode being more efficient if the opposite holds. In particular, for machine learning applications (and optimisation in general), the output is often a scalar loss function, meaning that reverse-mode AD will generally be much faster than forward-mode AD for such applications. However, computing the entire Jacobian is often not required, and one of the AD methods may be much more efficient than the other depending on the way the Jacobian is used.

- If the Jacobian is used in a matrix-vector product, forward-mode AD can be used much more efficiently, since
$$\mathbf{J}\mathbf{v} = \frac{\mathrm{d}}{\mathrm{d}t} f(\mathbf{u} + \mathbf{v}t).$$
In words, computing this Jacobian-vector product can be seen as computing the derivative of the function $\mathbb{R} \to \mathbb{R}^m, t \mapsto f(\mathbf{u} + \mathbf{v}t)$, which is a function with just one input meaning that forward-mode AD can be used to compute the entire matrix-vector product without forming the entire Jacobian.

- Conversely, if the Jacobian is used in a vector-matrix product, reverse-mode AD can be used much more efficiently, since
$$\mathbf{v}^\top \mathbf{J} = \frac{\mathrm{d}}{\mathrm{d}\mathbf{u}} \left(\mathbf{v}^\top f(\mathbf{u}) \right),$$
so computing a vector-Jacobian product is equivalent to computing the gradient of the function $\mathbb{R}^n \to \mathbb{R}, \mathbf{u} \mapsto \mathbf{v}^\top f(\mathbf{u})$. This is a function with just one output and as such, reverse-mode AD can compute this vector-Jacobian product without forming the entire Jacobian. This method is also sometimes called the (linearised) adjoint model, since the backward pass of reverse-mode AD is essentially a linear function $\mathbf{v} \mapsto \mathbf{J}^\top \mathbf{v}$, which is the adjoint of the Jacobian itself.

Importantly, such vector-Jacobian products appear in the adjoint sensitivity method (Algorithms 1, 2, and 3). As found by Rackauckas et al. [66], using reverse-mode AD to compute vector-Jacobian products results in better performance than computing the entire Jacobian and performing the vector-matrix multiplication, even if the Jacobian is computed using a highly-optimised manual implementation.

In Chapters 4, 5, and 6 of this thesis, numerical experiments perform reverse-mode AD using the auto-differentiation library Zygote [32]. Zygote defines the derivative of a function through the *pull-back*, which is a linear function that maps vectors $\mathbf{v}$ to $\mathbf{v}^\top \mathbf{J}$ using the reverse pass. These pull-backs can be composed, which makes it possible to compute gradients of nested functions $h(g(f(\mathbf{u})))$ without explicitly computing any Jacobians.

3.6 Neural closure models for stiff ODEs

While the above methods can be used to perform back-propagation for any neural ODE, there is a special case that deserves to be handled separately: the case of a stiff ODE, so that the right-hand side is defined as the sum of a function f_{stiff} and a closure term defined by a neural network.
$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f_{\text{stiff}}(\mathbf{u}(t)) + \mathrm{NN}(\mathbf{u}(t); \vartheta). \tag{3.11}$$

It is assumed that the function f_{stiff} is computationally cheap to evaluate, and that the neural network function $\mathrm{NN}(\cdot;\vartheta)$ is not stiff. In such cases, the neural ODE back-propagation may suffer from poor performance:

- If the ODE (3.11) is integrated using an explicit method, this method will require a comparatively small time step and therefore a large number of iterations, which implies a large number of evaluations of NN.
- By contrast, implicit methods that are able to handle stiff ODEs efficiently typically do so by computing the Jacobian $\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}}\left(f_{\text{stiff}}(\mathbf{u}(t)) + \mathrm{NN}(\mathbf{u}(t);\vartheta)\right)$ at least once per time step, which again is computationally expensive for large neural networks.

A possible solution to this poor performance is to solve the neural ODE using an implicit-explicit (IMEX) ODE solver (see pages 383 – 397 of Hundsdorfer and Verwer [31]): such solvers treat f_{stiff} and NN as separate functions, allowing them to be implicit in f_{stiff} but explicit in NN. The advantage of such methods is that due to being implicit in f_{stiff}, they can take larger time steps which reduces the required number of evaluations of NN. This is also the approach taken by Wang et al. [80]. The use of IMEX algorithms for neural closure models will be tested in Section 5.5.3.

3.7 Other methods for time series prediction

Apart from neural ODEs, using neural networks to predict time series arising from an ODE can also be done in other ways. One such way is the "Physics-Informed Neural Network" (PINN) approach suggested by Raissi et al. [67]. Here, rather than predicting the right-hand side of the ODE or some more general evolution operator, a neural network is trained to approximate the solution directly, i.e. $u(x,t) = \mathrm{NN}(x,t;\vartheta)$. The neural network can then be trained to satisfy the given PDE by minimising a loss function. Such models can be trained to directly satisfy the PDE, rather than being trained to match training data. Hence, PINNs can be trained based on just the relevant PDE, and do not require reference solutions to the PDE (hence the name physics-informed). For example, for the PDE $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ with boundary conditions $u(0,t) = 0, u(1,t) = 1$, the neural network is trained to minimise a loss function of the form

$$\begin{aligned}
&\lambda_1 \cdot \left\| \frac{\partial}{\partial t}\mathrm{NN}(x,t;\vartheta) - \frac{\partial^2}{\partial x^2}\mathrm{NN}(x,t;\vartheta) \right\|^2 \\
&+\lambda_2 \cdot \left[\|\mathrm{NN}(0,t;\vartheta)\|^2 + \|\mathrm{NN}(1,t;\vartheta) - 1\|^2 \right] \\
&+\lambda_3 \cdot \|\mathrm{NN}(x,t;\vartheta) - \mathbf{u}_{\text{ref}}(x,t)\|^2 .
\end{aligned}$$

In words, the loss function contains three terms that train the network to satisfy the PDE, to satisfy the boundary conditions, and to fit existing training data, respectively.

Chapter 4

Effect of including prior knowledge in ML models

4.1 Introduction

The goal of this section is to study how including prior knowledge about the training data in a machine learning model affects the performance of the resulting model. For this, consider a scalar PDE in one dimension with periodic boundary conditions, of the following form:

$$\frac{\partial u}{\partial t}(x,t) = \frac{\partial}{\partial x}F(u(x,t)) \text{ for } t>0, x\in(0,1), \tag{4.1}$$

$$u(x,0) = u_0(x) \text{ for } x\in[0,1), \tag{4.2}$$

$$u(0,t) = u(1,t) \text{ and } \frac{\partial}{\partial x}u(0,t) = \frac{\partial}{\partial x}u(1,t) \text{ for } t>0. \tag{4.3}$$

PDEs of this form often appear in fluid dynamics (although often over more than one variable), and are suitable for investigating the efficacy of different ML models. In this chapter, it is assumed that this PDE can be discretised into an ODE of the form $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u} = f(\mathbf{u})$ over the variables $\mathbf{u}\in\mathbb{R}^{N_x}$. The solution of this ODE approximates that of (4.1) but does not match it exactly. Then, a variety methods for constructing ML models are available.

Without any prior knowledge, the simplest model simply treats the neural network output as the next snapshot:

$$\mathbf{u}(t+\Delta t) = \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right). \tag{4.4}$$

Note that this model, as well as other models described later, iterate by feeding the model output from one step back into the model's input for the next step. Such models are typically called **autoregressive**.

Assuming that the underlying data is continuous in time, i.e. $\mathbf{u}(t+\Delta t)\approx\mathbf{u}(t)$, it may be beneficial to take an approach resembling ResNets (He et al. [27]), in which the neural network output is used as the difference between subsequent snapshots:

$$\mathbf{u}(t+\Delta t) = \mathbf{u}(t) + \Delta t\cdot\mathrm{NN}\left(\mathbf{u}(t);\vartheta\right). \tag{4.5}$$

The following four properties of the underlying PDE can be used to further inform the machine learning model (4.5):

i) **Approximate ODE definition**. The trajectories $\mathbf{u}(t)$ are approximated by the ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$, in the sense that solving $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$ is expected to yield solutions $\mathbf{u}(t)$ that approximate solutions to the original PDE (4.1). With this in mind, it may be advantageous to use the function $f(\mathbf{u})$ in the model in such a way that the neural network is used only to learn the error in this approximation, resulting in a neural closure model. In the discrete-time setting, this can be done by taking a forward Euler step combined with a neural network correction term:

$$\mathbf{u}(t+\Delta t) = \mathbf{u}(t) + \Delta t\cdot\left[f\left(\mathbf{u}(t)\right) + \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)\right].$$

Note that using forward Euler to take time steps results in limited accuracy and may be unstable. Using a higher-order ODE solver such as Runge-Kutta 4 may be preferable.

ii) **Continuity in time**. While the discretised trajectory $\mathbf{u}(t)$ is not exactly the solution of an ODE, it is still continuous in time. This property can be enforced in the model by choosing a Neural ODE (NODE) in which a neural network is used to approximate the right-hand side of an ODE:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u}; \vartheta).$$

By contrast, in discrete-time models, a neural network is used to predict a fixed time step, i.e. to approximate $\mathbf{u}(t + \Delta t)$ from $\mathbf{u}(t)$. However, note that using a NODE model allows for different ODE solvers as well as different training procedures as outlined in Chapter 3. Training the NODE with forward Euler as ODE solver and using a discretise-then-optimise training method, the resulting model is equivalent to the discrete model (4.5). However, formulating the model as a NODE allows using more accurate ODE solvers, as well as possibly more efficient training procedures.

iii) **Conservation of momentum**. The underlying PDE (4.1) satisfies conservation of momentum, meaning that the quantity $\int_0^1 u(x,t)\mathrm{d}x$ is constant. Analogously, many discretisations of the PDE satisfy that $\sum_{i=1}^{N_x} \mathbf{u}_i(t)$ is constant. The ML model (4.5) can be modified to conserve momentum by modifying the neural network output in such a way that the resulting vector entries sum to zero. Here, the way this is done is by taking differences of the output:

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \Delta t \cdot \Delta_{\mathrm{fwd}} \mathrm{NN}\left(\mathbf{u}(t); \vartheta\right).$$

Here, Δ_{fwd} is the linear operator that takes differences between subsequent entries of a vector:

$$\begin{aligned} (\Delta_{\mathrm{fwd}}\mathbf{v})_i &= \mathbf{v}_{i+1} - \mathbf{v}_i \text{ for } i = 1, 2, \ldots, N_x - 1, \\ (\Delta_{\mathrm{fwd}}\mathbf{v})_{N_x} &= \mathbf{v}_1 - \mathbf{v}_{N_x}. \end{aligned}$$

This model can be interpreted as using the neural network to predict the fluxes between grid cells, rather than the derivatives of the grid cells directly, i.e. $[\mathrm{NN}(\mathbf{u}(t); \vartheta)]_i \approx F(u(x_{i+1/2}, t))$. Instead of taking forward differences, backward or central differences could be used to enforce conservation of momentum. However, since this differencing is only done to the neural network output, there is no clear reason to favour one differencing scheme over another.

iv) **Translation invariance**. Finally, the PDE (4.1) is translation invariant, meaning that if $u(x,t)$ is a solution of (4.1), then so is $v(x,t) := u(x + r, t)$ for any $r \in \mathbb{R}$. Discretising this PDE then typically yields an ODE $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u} = f(\mathbf{u})$ that is also translation invariant, now with respect to discrete translations $\mathbf{u}_i \mapsto \mathbf{u}_{i+k}$ for fixed $k \in \mathbb{Z}$. In the ODE, this translation invariance can be enforced by using a Convolutional Neural Network (CNN) for the function $\mathrm{NN}(\cdot;\cdot)$. This is also done in existing work such as that by List et al. [48] and Beck et al. [5]. Convolutional Neural Networks are a common model choice for discretised PDEs, and require far fewer parameters than a non-convolutional network with the same hidden layers. As such, the performance difference between CNNs and dense NNs will not be investigated here, and all tested models will make use of CNNs.

Of course, PDEs can have more properties beyond the ones listed here. For example, many PDEs are either energy-conserving or dissipative, meaning that the energy $\int_0^1 u(x,t)^2\mathrm{d}x$ is constant or decreasing, respectively. PDEs in multiple dimensions may also satisfy additional invariances such as rotational invariance. However, enforcing such properties in the ML model is more complicated than the simple properties listed above, and as such the tests here will be limited to properties i-iii. Property iv will be used in all tested ML models.

Properties i-iii can be combined, yielding eight different models that each correspond to a different subset of properties enforced in the model. These models are shown in Table 4.1. Figures 4.1 and 4.2 visually depict some of the same model architectures.

Table 4.1: The eight different ML model types tested in Chapter 4, based on which prior knowledge is included or excluded.

	i)	ii)	iii)	Form
1				$\frac{1}{\Delta t}\left(\mathbf{u}(t+\Delta t)-\mathbf{u}(t)\right) = \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
2			✓	$\frac{1}{\Delta t}\left(\mathbf{u}(t+\Delta t)-\mathbf{u}(t)\right) = \Delta_{\mathrm{fwd}}\mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
3		✓		$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
4		✓	✓	$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \Delta_{\mathrm{fwd}}\mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
5	✓			$\frac{1}{\Delta t}\left(\mathbf{u}(t+\Delta t)-\mathbf{u}(t)\right) = f(\mathbf{u}(t)) + \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
6	✓		✓	$\frac{1}{\Delta t}\left(\mathbf{u}(t+\Delta t)-\mathbf{u}(t)\right) = f(\mathbf{u}(t)) + \Delta_{\mathrm{fwd}}\mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
7	✓	✓		$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}(t)) + \mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$
8	✓	✓	✓	$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}(t)) + \Delta_{\mathrm{fwd}}\mathrm{NN}\left(\mathbf{u}(t);\vartheta\right)$

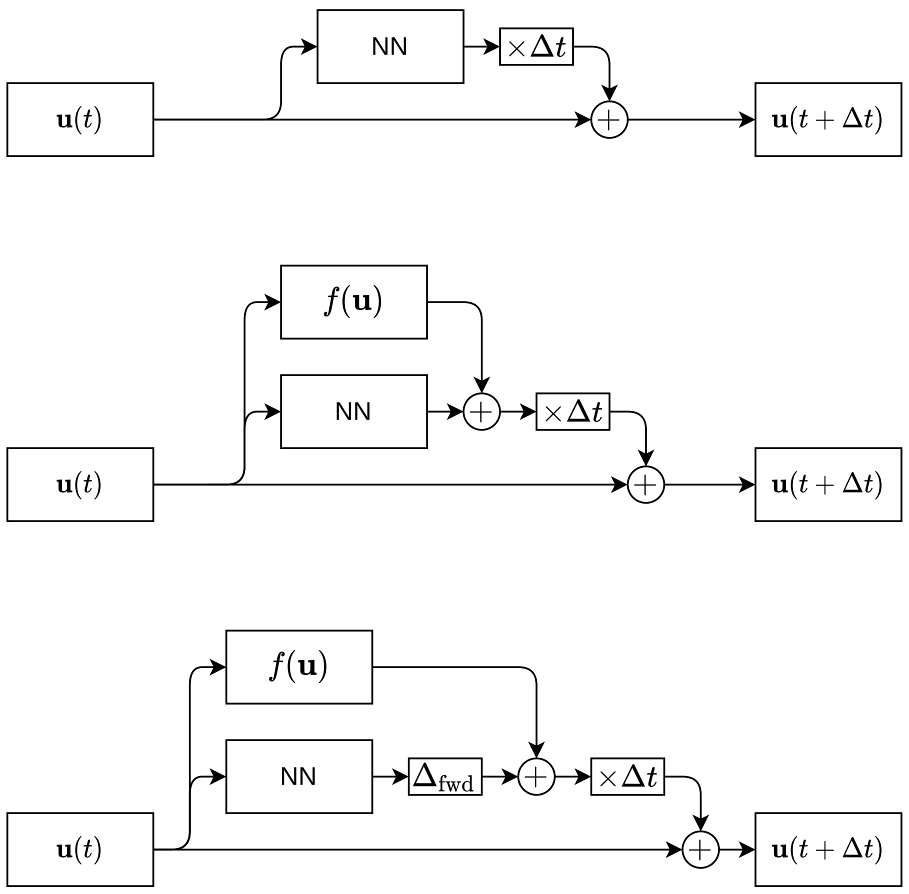

Figure 4.1: The architectures of models 1, 5, and 6.

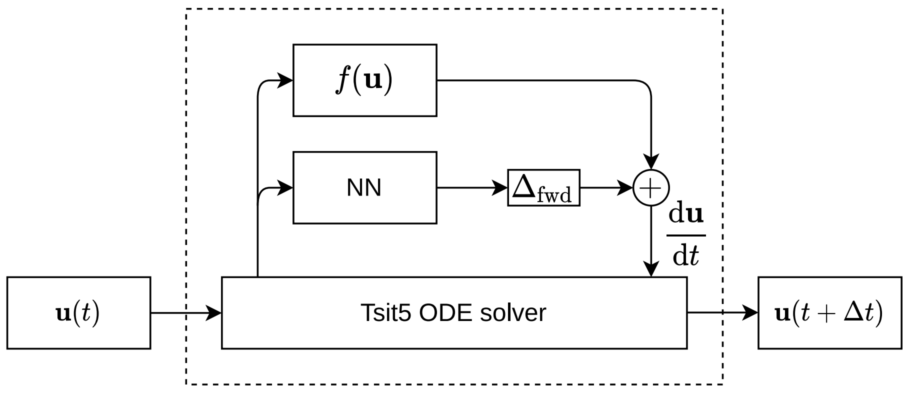

Figure 4.2: The architecture of model 8.

4.2 Burgers' equation

In order to test the effects of embedding the above properties into an ML model, a specific PDE of the form (4.1) must be chosen. Here, the PDE used is **Burgers' equation**. Burgers' equation is a PDE over one variable, the flow speed $u(x,t)$, as a function of space and time. Its general form is as follows:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} - \frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right), \tag{4.6}$$

where $\nu \geq 0$ is a constant. The PDE can be seen as a highly simplified one-dimensional version of a fluid flow problem, with a linear second-order diffusion term that models the effects of fluid viscosity, and a quadratic convection term that resembles the convection term in the Navier-Stokes equation. In this section, Burgers' equation will be used with periodic boundary conditions and a domain length of 1, i.e. $u(x+1,t) = u(x,t)$ for all x, t.

To numerically compute solutions to this equation, the PDE can be discretised in space using the finite volume method, where the function $u(x,t)$ is replaced by a time-dependent vector $\mathbf{u}(t) \in N_x$. Each component of $\mathbf{u}(t)$ represents the average value of u over a small subinterval, or cell, of the domain:

$$\mathbf{u}_i(t) \approx \frac{1}{\Delta x}\int_{(i-1)\Delta x}^{i\Delta x} u(x,t)\mathrm{d}x,$$

where $\Delta x = \frac{1}{N_x}$. Then, the behaviour of $\mathbf{u}(t)$ is approximated by an ODE of the form $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$, where the function f approximates the right-hand side of the PDE. This ODE can then be solved by standard methods, yielding an approximation for the solution $u(x,t)$. The error in this approximation is the result of two separate errors:

- The **spatial discretisation error**, which is introduced when the function $u(x,t)$ is replaced by a vector $\mathbf{u}(t)$. In this step, the PDE, which is continuous in time and space, is turned into an ODE of the form $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$, which is continuous in time but discrete in space.

  The magnitude of the spatial discretisation error depends on the grid size N_x as well as the chosen discretisation of the PDE. In this section, the function f is as given by Jameson [33], which is first-order accurate, meaning the error is proportional to the cell size $\Delta x = \frac{1}{N_x}$:

$$\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} = f(\mathbf{u})_i = \frac{\nu}{\Delta x^2}\left(\mathbf{u}_{i-1} - 2\mathbf{u}_i + \mathbf{u}_{i+1}\right) - \frac{1}{\Delta x}\left(\mathbf{f}_{i+1/2} - \mathbf{f}_{i-1/2}\right), \tag{4.7}$$

$$\text{where } \mathbf{f}_{i+1/2} = \frac{1}{6}\left(\mathbf{u}_i^2 + \mathbf{u}_i\mathbf{u}_{i+1} + \mathbf{u}_{i+1}^2\right) - \alpha_i\left(\mathbf{u}_{i+1} - \mathbf{u}_i\right), \tag{4.8}$$

$$\text{and } \alpha_{i+1/2} = \frac{1}{4}\left|\mathbf{u}_i + \mathbf{u}_{i+1}\right| - \frac{1}{12}\left(\mathbf{u}_{i+1} - \mathbf{u}_i\right). \tag{4.9}$$

- The **temporal discretisation error**, which is introduced when the ODE is solved. ODE solvers find solutions at specific points in time $t_0, t_1, \ldots$, thus approximating the continuous-time ODE by a discrete-time computation.

  The magnitude of the temporal discretisation error depends on the chosen ODE solver and the size of the time steps, Δt. In this section, the ODEs are solved using the `Tsit5` algorithm [76], which is the default ODE solver used by DifferentialEquations.jl [65]. `Tsit5` is a fourth-order, five-stage Runge-Kutta method with embedded error estimator.

4.3 Data generation

To create training data for the neural networks, 128 solutions are generated to Burgers' equation with $\nu = 0.0005$. The PDE is discretised into $N_x = 4096$ finite volumes and the initial states $\mathbf{u}(0)$ are randomly generated as the sum of random sine and cosine waves with wave numbers $1 \leq k \leq 10$:

$$\mathbf{u}_i(0) = \mathrm{Re}\left(\sum_{k=1}^{10} \hat{u}_k \exp(2\pi ik/N_x) + \sum_{k=1}^{10} \hat{u}_{-k} \exp(-2\pi ik/N_x)\right), \tag{4.10}$$

where $\hat{u}_{-10\cdots-1}$ and $\hat{u}_{1\ldots10}$ are randomly generated complex numbers with real and imaginary parts sampled from a unit Gaussian distribution. The resulting initial state is then multiplied by a scalar chosen such that $\max_i |\mathbf{u}_i(0)| = 2$. This way, the initial states are random but are all smooth and have zero net momentum.

The resulting ODEs are solved for $t \in [0, 0.5]$. Snapshots of the solution $\mathbf{u}(t)$ are saved at 2049 points in time, i.e. with a time of $\Delta t_h = 2^{-12}$ in between snapshots. Note that since `Tsit5` selects the time step automatically, the actual time steps used by the ODE solver may be smaller or larger than 2^{-12}.

In total, 128 solutions are obtained, each from a random initial state. Of these solutions, 96 are used for training and the remaining 32 are used for testing. The solutions are then down-sampled by a factor of 64 in both space and time. This way, training data is created to allow a neural network to work on the low-fidelity (downsampled) initial conditions, but to approximate the original high-fidelity solution. In the spatial dimension, the downsampling is performed by averaging the solution over chunks of 64 finite volumes. In the temporal dimension, the downsampling is performed by simply taking every 64th snapshot, leading to a new time step of $\Delta t_l = 2^{-6}$.

This process is illustrated in Figure 4.3. This figure also shows the result of solving the coarsely discretised ODE starting from the downsampled initial state $\mathbf{u}(0) \in \mathbb{R}^{64}$ (bottom left figure). Crucially, this is not equal to the downsampled fine-grid solution (top left). The downsampled fine-grid solution is by definition the most accurate low-fidelity approximation to the original high-fidelity data. The solution of the low-fidelity ODE introduces additional error (bottom right figure), and is therefore a less accurate approximation to the original data than the downsampled high-fidelity solution.

4.4 Experiments

4.4.1 Neural network architecture

To ensure that the experiments only test the effect of including prior knowledge, all nine models are trained using the same neural network architecture. However, since larger neural networks are generally expected to perform better than smaller neural networks, all nine models are trained twice: once using a small neural network, and again using a larger network.

The small neural network $\mathrm{NN}(\mathbf{u}(t); \vartheta)$ is defined as follows:

1. First, the vector $\mathbf{u} \in \mathbb{R}^{N_x}$ is extended with the component-wise square of $\mathbf{u}$, i.e. the matrix $\mathbf{U} \in \mathbb{R}^{N_x \times 2}$ where $\mathbf{U}_{i,1} = \mathbf{u}_i$ and $\mathbf{U}_{i,2} = \mathbf{u}_i^2$. This is done since the original PDE contains quadratic terms, and therefore the inclusion of quadratic terms in the inputs can help to create more accurate neural networks with fewer parameters. This is similar to the example in [54], where a neural ODE to fit a cubic function is given cubed values as inputs. However, the neural network here is also allowed to depend on the inputs $\mathbf{u}_i$ directly.

2. Then, the neural network uses two convolutional layers with filter width equal to 9. The first filter returns another $N_x \times 2$ matrix $\mathbf{V}$, defined as

$$\mathbf{V}_{ij} = \tanh\left(\sum_{k=-4}^{4}\sum_{l=1}^{2} \mathbf{A}_{l,k,j}\mathbf{U}_{i+k,l} + \mathbf{b}_j\right),$$

where the tensor $A \in \mathbb{R}^{9\times2\times2}$ and the vector $\mathbf{b} \in \mathbb{R}^2$ are formed from the parameters ϑ of the neural network. This layer therefore uses $9 \times 2 \times 2 + 2 = 38$ parameters. Since the PDE uses periodic boundary conditions, the vector $\mathbf{u}$ is seen as periodic (i.e. $\mathbf{u}_{k+N_x} = \mathbf{u}_{N_x}$) and the matrix $\mathbf{U}$ is seen as periodic in its first dimension. Note that this convolution operation corresponds to a convolution only in the first (spatial) dimension, and is also periodic in that dimension.

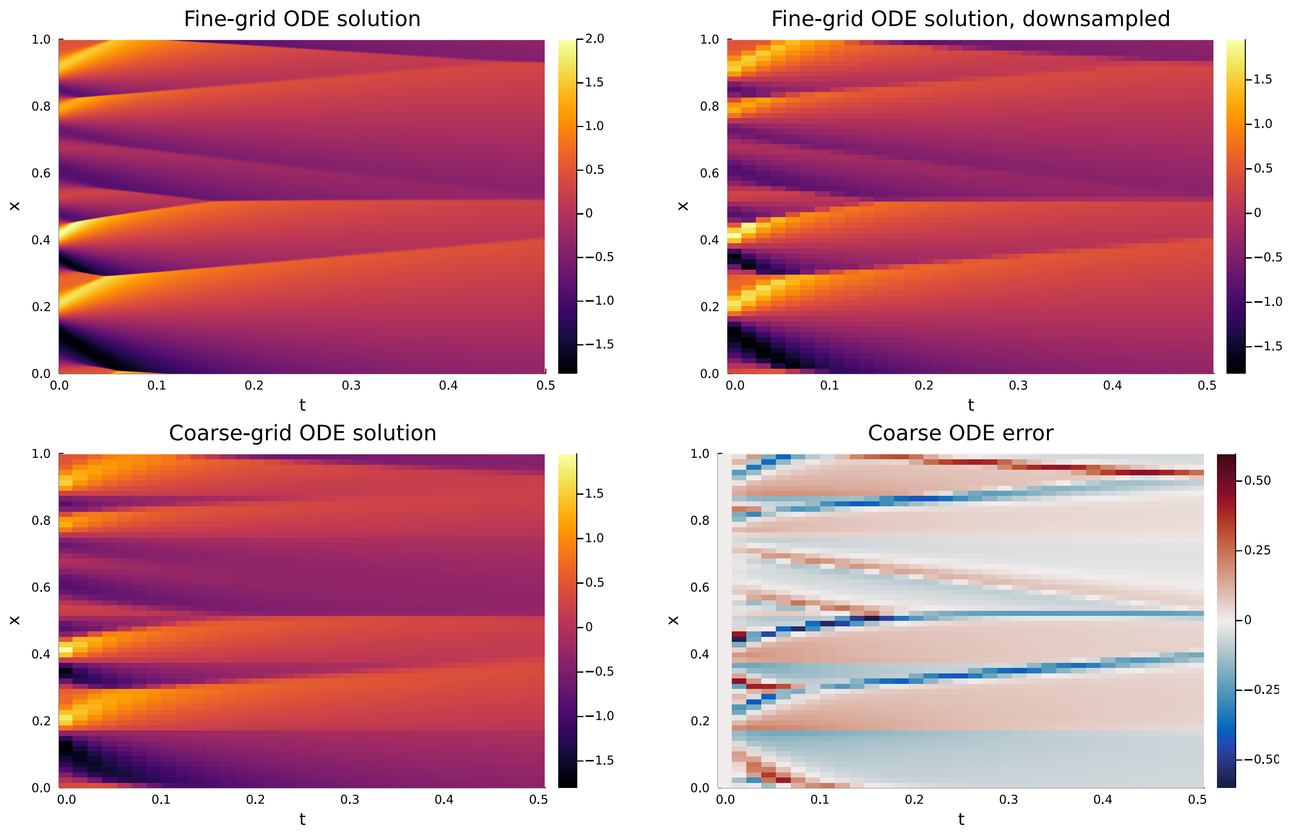

Figure 4.3: Top to bottom, left to right: (1) a numerical solution to Burgers' equation on the fine grid ($N_x = 4096$). (2) this same solution, downsampled to the coarse grid ($N_x = 64$). (3) the solution of the ODE on the coarse grid, starting from the downsampled initial condition. (4) the error between the downsampled fine-grid solution (2) and the coarse grid solution (3).

Table 4.2: A description of the small neural network structure used for the experiments on Burgers' equation.

Layer	Description	σ	Parameters
1	$\mathbf{u} \mapsto \left[(\mathbf{u}_i)_i \quad (\mathbf{u}_i^2)_i\right]$	-	0
2	9-wide conv, $2 \to 2$ channels	tanh	38 ($9 \times 2 \times 2$ weights and 2 biases)
3	9-wide conv, $2 \to 1$ channels	-	19 ($9 \times 2 \times 1$ weights and 1 bias)
Total			57

Table 4.3: A description of the large neural network structure used for the experiments on Burgers' equation.

Layer	Description	σ	Parameters
1	$\mathbf{u} \mapsto \left[(\mathbf{u}_i)_i \quad (\mathbf{u}_i^2)_i\right]$	-	0
2	5-wide conv, $2 \to 4$ channels	tanh	44 ($5 \times 2 \times 4$ weights and 4 biases)
3	5-wide conv, $4 \to 6$ channels	tanh	126 ($5 \times 4 \times 6$ weights and 6 biases)
4	5-wide conv, $6 \to 6$ channels	tanh	186 ($5 \times 6 \times 6$ weights and 6 biases)
5	5-wide conv, $6 \to 4$ channels	tanh	124 ($5 \times 6 \times 4$ weights and 4 biases)
6	5-wide conv, $4 \to 2$ channels	tanh	42 ($5 \times 4 \times 2$ weights and 2 biases)
7	5-wide conv, $2 \to 1$ channels	–	11 ($5 \times 2 \times 1$ weights and 1 bias)
8	$\mathbf{u} \mapsto \Delta_{\text{fwd}}\mathbf{u}$	–	0
Total			533

The second convolutional layer is similar to the first, but does not use a smoothing function and only has a single 'channel', meaning that the output is a single vector with the same dimension as $\mathbf{u}$ itself. Therefore, the second layer has a $9 \times 2 \times 1$ weight matrix and only a single scalar bias, leading to 19 parameters for this layer and $N_\vartheta = 57$ for the entire network.

The larger neural network uses a similar structure, but with more internal layers. The neural network structures are summarised in Tables 4.2 and 4.3. A schematic representation of the small neural network is shown in Figure 4.4. Note both neural networks, especially the small neural network, are very small compared to networks used in modern machine learning applications. This is intentional, and small neural networks were chosen since the Burgers equation is a relatively simple test problem. Specifically, Burgers' equation always yields relatively simple solutions containing travelling and dissipating waves. Any reasonably large neural network is assumed to be able to learn this behaviour, thus reducing the utility of including prior knowledge. The large neural network (which is still small by modern standards) is included in these experiments to test this assumption.

4.4.2 Training procedure

For the continuous models (neural ODEs), a comparison of training procedures has already been made in Chapter 3. Here, the neural ODEs are trained by trajectory fitting, using the interpolating adjoint method with the same `Tsit5` ODE solver that was used to generate the training data. For discrete models, there are two immediate ways to train neural networks on this data:

- **Single-step fitting**: here, the neural network is given a state $\mathbf{u}(t)$ as input, and returns a prediction for $\mathbf{u}(t + \Delta t)$ which is compared to the real value of $\mathbf{u}(t + \Delta t)$ from the training data.
- **Trajectory fitting**: here, the neural network is given an initial state $\mathbf{u}(0)$ and called $k > 1$ times to predict the next k snapshots, which can then be compared to the next k snapshots of the training data. For the experiments done here, k is chosen as 32 which is the number of snapshots in the training data. This means that the entire trajectories are predicted. Training by predicting multiple time steps is also referred to as **unrolling**.

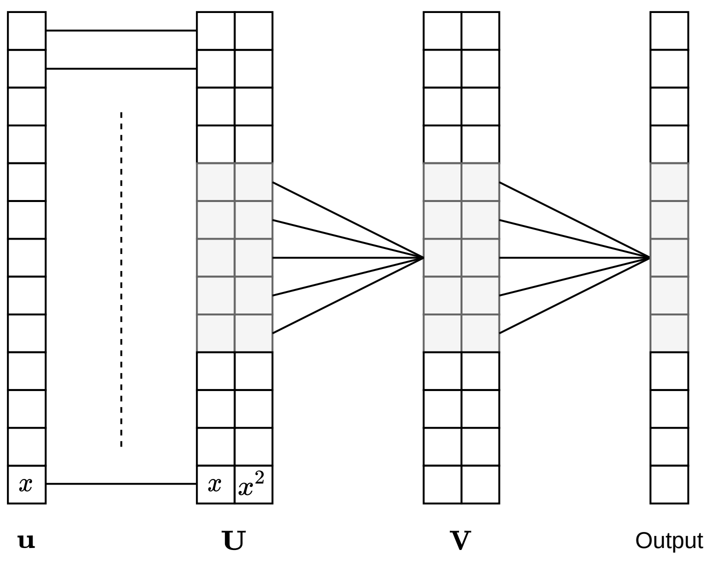

Figure 4.4: A depiction of the small CNN used in this chapter. For simplicity, the kernel widths are shown as 5 instead of 9 and the vector $\mathbf{u}$ is shown with a length of 15 instead of 32.

In either case, the actual and predicted data can be compared using a Mean Squared Error (MSE) loss function, which can then be used to perform back-propagation to update the parameters of the model.

Since the end goal is to obtain neural networks that accurately predict entire trajectories rather than single time steps, trajectory fitting appears to be a more logical choice. Indeed, in similar research, such as that done by List et al. [48], unrolling multiple time steps is found to be crucial in obtaining models that make accurate predictions. This is motivated by the following theorem:

Theorem 4.1. *Let $\mathbf{u}_{ref}(t_i), i = 0, 1, \ldots, N_t$ be a sequence of vectors in $\mathbb{R}^{N_x}$ where $t_i = i\Delta t$ and let $f : \mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$ be a function such that:*

$$a) \quad \|\mathbf{u}_{ref}(t_{i+1}) - f(\mathbf{u}_{ref}(t_i))\| \leq \varepsilon \text{ for all } i = 1, 2, \ldots, N_t,$$
$$b) \quad \|f(\mathbf{a}) - f(\mathbf{b})\| \leq C\,\|\mathbf{a} - \mathbf{b}\| \text{ for all } \mathbf{a}, \mathbf{b} \in \mathbb{R}^{N_x},$$

for fixed constants $C > 1, \varepsilon > 0$. Define the sequence $\mathbf{u}(t_{i+1}) = f(\mathbf{u}(t_i))$ with $\mathbf{u}(0) = \mathbf{u}_{ref}(0)$. Then the following error bound holds:

$$\|\mathbf{u}(t_k) - \mathbf{u}_{ref}(t_k)\| \leq \varepsilon \frac{C^k - 1}{C - 1}.$$

Proof. For arbitrary $k \geq 1$, the following holds:

$$\begin{aligned}\|\mathbf{u}(t_k) - \mathbf{u}_{\text{ref}}(t_k)\| &= \|f(\mathbf{u}(t_{k-1})) - \mathbf{u}_{\text{ref}}(t_k)\| \\ &\leq \|f(\mathbf{u}(t_{k-1})) - f(\mathbf{u}_{\text{ref}}(t_{k-1}))\| + \|f(\mathbf{u}_{\text{ref}}(t_{k-1})) - \mathbf{u}_{\text{ref}}(t_k)\| \\ &\leq C\,\|\mathbf{u}(t_{k-1}) - \mathbf{u}_{\text{ref}}(t_{k-1})\| + \varepsilon.\end{aligned}$$

Now, define $r_k = \|\mathbf{u}(t_k) - \mathbf{u}_{\text{ref}}(t_k)\| + \frac{\varepsilon}{C-1}$. Then $r_0 = \frac{\varepsilon}{C-1}$ and for $k \geq 1$:

$$\begin{aligned}r_k &= \|\mathbf{u}(t_k) - \mathbf{u}_{\text{ref}}(t_k)\| + \frac{\varepsilon}{C-1} \\ &\leq C\,\|\mathbf{u}(t_{k-1}) - \mathbf{u}_{\text{ref}}(t_{k-1})\| + \varepsilon + \frac{\varepsilon}{C-1} \\ &= C\left(r_{k-1} - \frac{\varepsilon}{C-1}\right) + \frac{C\varepsilon}{C-1} \\ &= Cr_{k-1}.\end{aligned}$$

As a result, $r_k \leq C^k r_0 = \frac{C^k \varepsilon}{C-1}$, so:

$$\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\| = r_k - \frac{\varepsilon}{C-1} \leq \varepsilon \frac{C^k - 1}{C-1}.$$

□

Note that the Lipschitz constant C is not always greater than one, although it is always positive. If $C = 1$, then proof can be adapted to show that $\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\| \leq \varepsilon k$. If $C < 1$ then f is a contraction and the sequence $\mathbf{u}(t_k)$ will converge to a fixed point.

The interpretation of this theorem is as follows: if f is a machine learning model trained to predict $\mathbf{u}(t + \Delta t)$ from $\mathbf{u}(t)$ and can do so with uniformly low error (at most ε for all training data), then this does not necessarily translate to accurate predictions for larger number of steps. In fact, the error after k steps can grow exponentially in k, where the exponential growth factor is given by C, the Lipschitz constant of f. During single-step fitting, only ε is minimised: as a result, as training progresses the model achieves progressively lower ε, but possibly at the cost of increased C. The result is that models trained for longer time may yield more accurate predictions in the very short term, but much worse in the long term.

Note that this result is likely not new: the fact that autoregressive models trained on single steps perform poorly has been known for at least a decade, and methods to mitigate this problem have been studied extensively. One such method is to add noise to the neural network's inputs (the vectors $\mathbf{u}_{\mathrm{ref}}(\cdot)$) during training, which trains the model to be less sensitive to small perturbations in its input (see for example Chapter 7.4 of Goodfellow et al. [24]). Since the input to the model is equal to the output of the previous step, this regularisation technique is therefore expected to decrease the rate at which the approximation error increases per step, meaning it has a similar effect to reducing C.

Another option is to simply use trajectory fitting, which directly trains the model to make accurate predictions over multiple time steps. However, trajectory fitting can perform poorly due to the **exploding gradients** problem: when the network is still untrained, the later time steps of the predicted trajectory will likely be much more sensitive to the network parameters (due to the repeated application of the network to predict later time steps), and the later time steps will likely be much less accurate. These two facts combined mean that the direction taken by the training algorithm (i.e. the update made to the network parameters, whether it be computed by gradient descent or a more advanced method) is mostly determined by the error in the tail of the predicted trajectory. The result is then that the network parameters may converge to a local minimum. Such local minima often correspond to neural networks that predict trajectories that quickly tend toward the long-term average of the solution, or, more generally, predicts trajectories that show significantly less variation than the actual trajectory.

An example of this phenomenon is shown in Figure 4.5. Here, a simple neural network consisting of two fully connected layers was trained to predict the trajectory $(\cos t, \sin t)$ of an oscillator using the time stepping method $\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \mathrm{NN}(\mathbf{u}(t); \vartheta)$. After training, the neural network essentially predicts a heavily damped oscillator instead, quickly converging to the long-term average value of $\mathbf{u}(t)$ which is $(0, 0)$.

The exploding gradient problem (as well as the similar vanishing gradient problem) is also present in the context of recurrent neural networks, as well as in very deep feed-forward neural networks. Since the formal identification of these problems by Hochreiter [28], many solutions to it have been proposed. These include updated training procedures such as **Truncated Back-Propagation Through Time (TBPTT)** [83] (note that this paper actually predates Hochreiter's work, and primarily suggests the use of TBPTT due to its greater computational efficiency), modified neural network structures such as LSTMs [29], or the use of different activation functions such as ReLU (Rectified Linear Unit). Here, exploding gradients are caused by the way the neural network is used to predict time series and is therefore unlikely to be mitigated by changing the neural network architecture. Instead, the exploding gradients problem is solved through the use of teacher forcing, which can be seen as an interpolation between single-step fitting and trajectory fitting. With teacher forcing, the network is still used to predict entire trajectories. However, after each step the predicted state is replaced by a linear interpolation between the predicted state and the actual state. This is a variant of the teacher

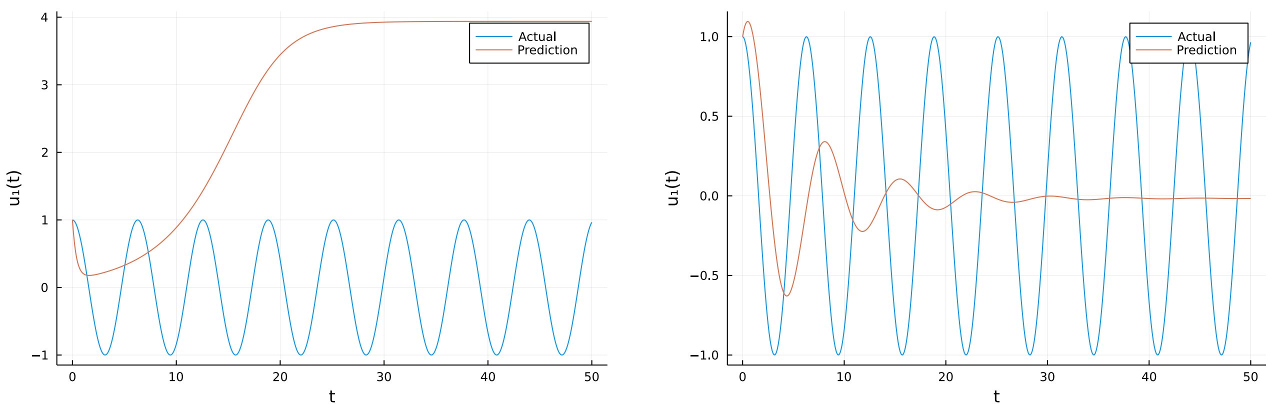

Figure 4.5: An example of unsuccessfully training a neural network to predict a simple trajectory. Left: the initial prediction before any training has taken place. The error is largest for large t. Right: the prediction after training has converged to a local minimum.

forcing introduced by Williams and Zipser [84], where the actual state is only partially known and only the known components of the output vector are modified after each iteration. In pseudocode, the algorithm to predict a trajectory using teacher forcing is as follows:

Algorithm 5 Trajectory prediction with teacher forcing

```
1: procedure PREDICT_TRAJECTORY(model, $\mathbf{u}(0), r_{\text{tf}}, (\mathbf{u}_{\text{ref}}(t_i), i = 1, 2, \ldots, N_t)$)
2:     for $i$ from 1 to $N_t$ do
3:         $\tilde{\mathbf{u}}(t_i) = \text{model}\left(\mathbf{u}(t_{i-1})\right)$
4:         $\mathbf{u}(t_i) = (1 - r_{\text{tf}}) \cdot \tilde{\mathbf{u}}(t_i) + r_{\text{tf}} \cdot \mathbf{u}_{\text{ref}}(t_i)$
5:     return $(\tilde{\mathbf{u}}(t_i), i = 1, 2, \ldots, N_t)$
```

Here, $\mathbf{u}(0)$ is the initial state and $\mathbf{u}_{\text{ref}}(t_i), i = 1, 2, \ldots, N_t$ is the real trajectory in the training data. The parameter r_{tf} controls the amount of teacher forcing that is done. Note that single-step fitting and trajectory fitting can be done with the same algorithm by setting $r_{\text{tf}} = 1$ or $r_{\text{tf}} = 0$, respectively.

For all models (discrete and continuous), after the trajectories $\mathbf{u}(t_i), i = 1, \ldots, N_t$ are computed, the loss value is computed as the mean squared error with a small regularisation term:

$$\text{Loss} = \frac{1}{N_x N_t} \sum_{i=1}^{N_t} \|\tilde{\mathbf{u}}(t_i) - \mathbf{u}_{\text{ref}}(t_i)\|^2 + \lambda \cdot \frac{1}{N_\vartheta} \|\vartheta\|^2, \text{ where } \lambda = 10^{-4}.$$

The training is done using the ADAM optimiser [42] with a learning rate $\eta = 0.001$, using a batch size of 8. All models are trained for 50000 epochs, where at epoch i the teacher forcing rate for discrete models is $r_{\text{tf}}(i) = \frac{1}{2} - \frac{1}{2}\tanh\left(\frac{i}{100} - \frac{5}{2}\right)$. A plot of this function for the first 750 epochs is shown in Figure 4.6. This teacher forcing rate is chosen so that the training effectively contains three stages: an initial stage in which the teacher forcing rate is close to 1 so that the models are effectively predicting single steps, followed by a stage in which the teacher forcing rate decreases and finally a long ‘tail’ in which it is close to zero so the neural network is trained to predict entire trajectories accurately. The number of epochs was chosen to be large enough that all models were approximately trained until convergence. A summary of the parameters used for training is shown in Table 4.4.

4.5 Results

After training, all models are evaluated on some testing data that was not used during training. The testing data consists of $N_p = 32$ trajectories which, like the training data, are sampled at an initial condition and $N_t = 32$ subsequent time points. For each of the evaluated models, initial conditions $\mathbf{u}_{\text{ref}}^{(j)}(0)$ for $j = 1, \ldots, N_p$ were given to the ML model. The model then uses these initial conditions

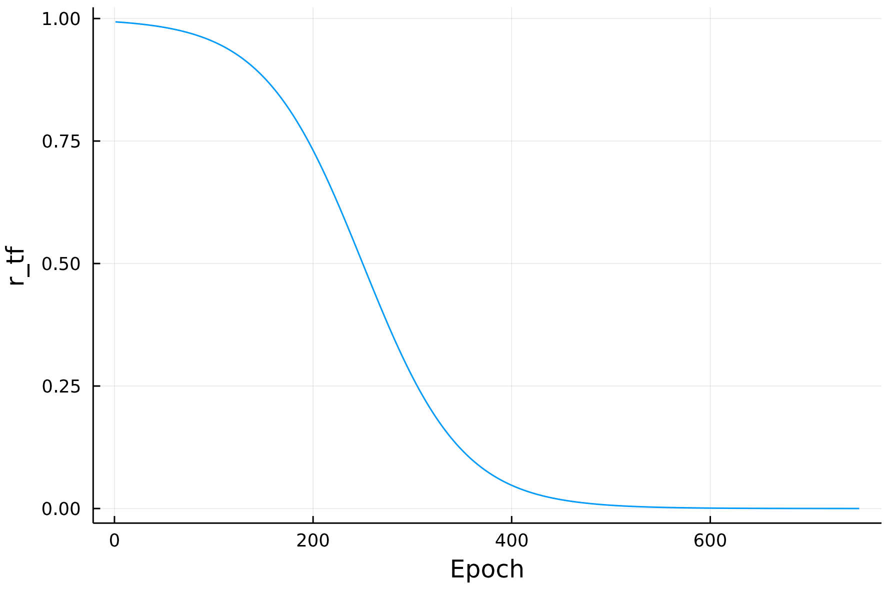

Figure 4.6: The teacher forcing rate as a function of the epoch number.

Table 4.4: The additional parameters of the training procedure from Chapter 4.

Property	Value
Training data	96 trajectories, 33 snapshots per trajectory
Optimiser	ADAM, learning rate $= 10^{-3}$
Batch size	8
Epochs	50000
Loss function	Mean-square error
Penalty term	$10^{-6} \cdot \frac{1}{N_\vartheta} \|\vartheta\|^2$
Testing data	32 trajectories, 33 snapshots per trajectory

Table 4.5: The Root-Mean-Square Error (RMSE) for each of the tested models on the 32 testing trajectories of the Burgers equation.

	i)	ii)	iii)	Testing error (RMSE)	
				Small NN	Large NN
1				0.0871	0.0134
2			✓	0.0431	0.0082
3		✓		0.0407	0.0123
4		✓	✓	0.0193	0.0074
5	✓			NaN	0.0066
6	✓		✓	NaN	0.0053
7	✓	✓		0.0287	0.0094
8	✓	✓	✓	0.0259	0.0050
(Coarse ODE solution)				0.104	
(Direct model (4.4))				0.0788	0.0807

to make predictions for the trajectories $\mathbf{u}^{(j)}_{\text{predict}}(t_i)$ for $j = 1, \dots, N_p$ and $i = 1, \dots, N_t$. These are then compared to the actual trajectories $\mathbf{u}^{(j)}_{\text{ref}}(t_i)$ by taking the root mean square, summing over all components of the vector $\mathbf{u}$, all time points t_i, and all 32 testing trajectories:

$$\text{RMSE} = \left[\frac{1}{N_x N_t N_p} \sum_{i=1}^{N_t} \sum_{j=1}^{N_p} \left\| \mathbf{u}^{(j)}_{\text{predict}}(t_i) - \mathbf{u}^{(j)}_{\text{ref}}(t_i) \right\|_2^2 \right]^{1/2} .$$

The RMS errors on the testing data for all models are shown in Table 4.5. Note that two of the models failed to train, since they yielded NaN values during training. Training these models was attempted five times, each time with different randomly initialised weights, but training failed all five times. The likely reason for this is that iterating a forward-Euler discretisation of an ODE can be unstable, meaning that the neural network has to 'learn' not only to correct for the error in the iteration, but even to stabilise it. This instability could be avoided by using a more stable iteration method as a base approximation, such as the midpoint rule or Runge-Kutta 4. However, doing so brings the resulting model much closer to neural ODEs, and such models will be evaluated in Chapter 5. Figure 4.7 shows an example of a reference trajectory, a model prediction, and the difference (prediction error) from the testing data. The prediction in this figure was made by the small NN with properties i, ii, and iii, which yields an RMSE of 0.0259 over all 32 testing trajectories.

4.6 Conclusion

Firstly, it must be noted that the RMSE values must be treated as approximate values: one effect of training with mini-batching is that the RMSE, both on the training and testing data sets, is expected to fluctuate during training, adding a small amount of 'noise' to the resulting RMSE. For example, this can be seen in that the larger NN, when used as a direct model, performs slightly worse than the smaller NN which is not expected to happen.

Nonetheless, a number of conclusions can still be drawn from these results.

- Firstly, enforcing momentum conservation (property iii) always improves performance of the neural network. While this is expected since the training data satisfies conservation of momentum exactly, enforcing this in the model makes a very large difference in some cases. This is somewhat surprising, since the last layer of both NN designs is linear, and momentum conservation is done by adding a linear layer Δ_{fwd} to the NN. This means that for each parameter vector ϑ_1 there is a parameter vector ϑ_2 such that $\Delta_{\text{fwd}}\text{NN}(\mathbf{u}; \vartheta_1) = \text{NN}(\mathbf{u}; \vartheta_2)$, i.e. each model is theoretically able to learn to satisfy conservation of momentum.

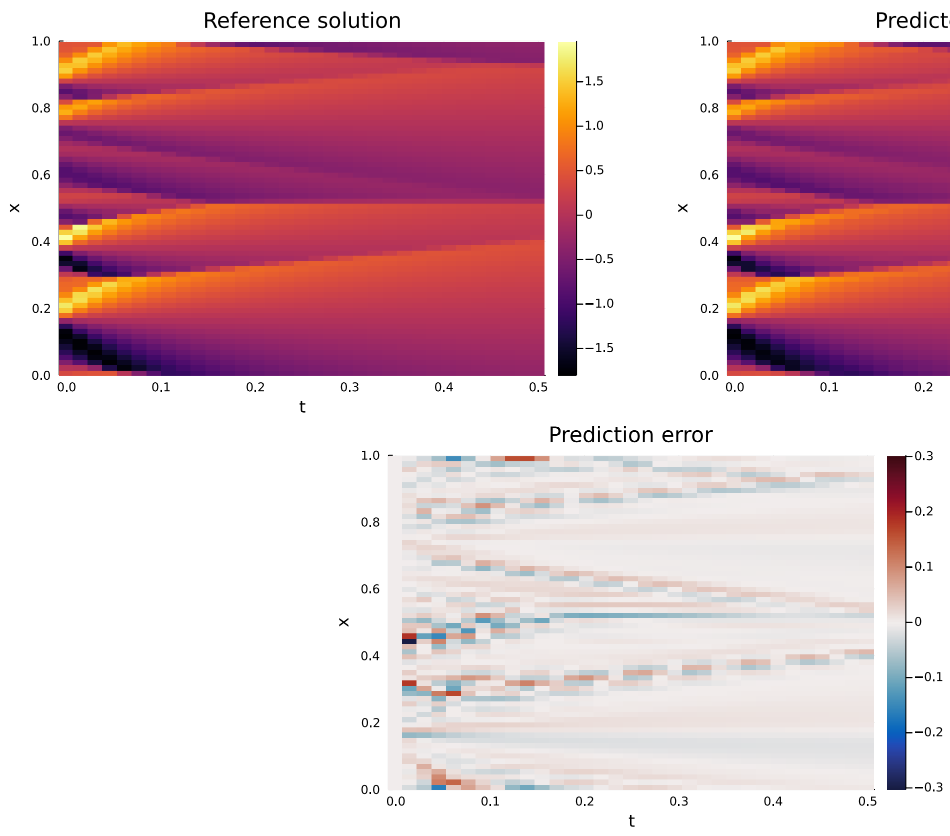

Figure 4.7: Top left: one reference solution to Burgers' equation from the testing data. Top right: the predicted solution made by the small NN with properties i, ii, and iii. Bottom: the difference between the predicted and actual trajectories.

- Secondly, for small NNs the inclusion of prior knowledge has a much greater positive effect than for large NNs. Most notably, for small NNs all time-continuous models (neural ODEs) outperform all time-discrete models, and neural ODEs that use the approximation $f(\mathbf{u})$ perform better on average than those that do not. For closure models, this can be explained by noting that the discrete models, which iterate $\mathbf{u}(t+\Delta t) = \mathbf{u}(t) + \Delta t f(\mathbf{u}(t)) + [\text{closure term}]$, introduce an additional error since the ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$ is only solved with forward Euler instead of a more accurate ODE solver. This extra error needs to be 'corrected' by the neural network, which is not the case for NODEs. However, this logic does not hold for the non-closure models.

  For large NNs, however, the differences are much smaller. The most likely explanation for this is that the larger NNs are simply able to learn much more complicated behaviour allowing them to achieve much lower error even when used in very simple models. Then, the inclusion of prior knowledge results in smaller benefits, as the accuracy of the resulting model becomes limited by the information lost when down-sampling the training data to a coarser grid. By contrast, the accuracy of small NNs may be limited due to underfitting, i.e. the small NNs are simply not large enough to learn the entire dynamics of Burgers' equation. As such, including prior knowledge may increase the accuracy of the model by reducing the 'amount of information' that the neural network needs to learn.

- Overall, including more prior information can significantly increase the accuracy of ML models, especially for models with relatively few parameters that suffer from underfitting when used without prior knowledge.

Note that models that are continuous in time and use the approximation $f(\mathbf{u})$ are essentially closure models, and these models also perform the best out of the models tested here. As such, closure models will be the subject of further research in the next section. Models that do not include the term $f(\mathbf{u})$, i.e. purely neural network-based models, will be referred to as **direct models** to distinguish them from **closure models**, which do make use of $f(\mathbf{u})$.

While the results obtained in this chapter are meaningful, one problem with the methodology is that different NODE-like models were trained using different methods from Chapter 3. Specifically, the four NODE models were trained with back-propagation using the interpolating adjoint method, whereas the discrete models, which can be seen as forward Euler discretisations of NODEs, were trained using the discrete optimisation method. Furthermore, Burgers' equation is still a relatively simple test case since solutions of this equation are very predictable, always resulting in travelling and dissipating waves. In particular, the long-term behaviour of the viscous Burgers equation is essentially trivial due to the dissipative effect of the diffusion term. These two limitations will be overcome in the next chapter, where different neural ODE training methods are considered on a more difficult test case with more complex long-term behaviour.

Chapter 5

Training procedures for neural ODEs

From the numerical results obtained using Burgers' equation, one can draw three conclusions:

- neural networks are able to increase the accuracy of numerical solutions to PDEs on coarse grids;
- using neural networks in the form of neural ODEs yields better results than using neural networks to advance the solution by a specific amount in time;
- using neural networks in the form of a closure term, i.e. correcting for the 'standard' discretised ODE, yields better results than predicting the entire time series using only the neural network.

However, the use of neural ODEs, and especially neural closure models, requires more complicated training procedures. In the previous experiments, neural ODEs were trained using the interpolating adjoint method, but as described in Chapter 3, other training methods are available. In this section, different training methods for neural ODEs will be compared by using multiple training methods to train the same neural network on the same training data. The goal is to find how these methods compare in terms of accuracy of the resulting model.

5.1 Overview

When training a neural ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u};\vartheta)$, there are two ways to train the neural network:

- **derivative fitting:** the neural network is trained to accurately replicate the derivatives of the system, e.g. using the loss function $\left\|\left(\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right)_{\mathrm{ref}} - \mathrm{NN}(\mathbf{u}_{\mathrm{ref}};\vartheta)\right\|^2$.
- **trajectory fitting:** the neural ODE is integrated yielding a trajectory $\mathbf{u}(t)$ which is compared to the reference trajectory, for example with a loss function such as

$$\mathrm{Loss} = \sum_{i=1}^{N_t} \|\mathbf{u}(t_i) - \mathbf{u}_{\mathrm{ref}}(t_i)\|^2, \text{ where } \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u};\vartheta).$$

  Trajectory fitting itself can be done in two different ways: training can be done using one of the adjoint ODE methods described in Section 3.1. Alternatively, the back-propagation step can be done by directly differentiating through the solver as described in Section 3.3.

5.1.1 Derivative fitting

The main advantage of derivative fitting is that in order to compute the gradient of the loss function with respect to ϑ, one only has to differentiate through the neural network itself. By contrast, in order to train by trajectory fitting one also has to differentiate through the function $f(\mathbf{u})$, and through the ODE solving process either by back-propagating through the ODE solver or using the adjoint sensitivity method as described by Chen et al. [9]. The most immediate disadvantage of derivative fitting is that the training data must consist of not just the values $\mathbf{u}$, but also their time derivatives $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}$. This data is not always available, for example in cases where the trajectories $\mathbf{u}(t)$ are obtained as measurements from a physical experiment. Fortunately, in the cases considered here the training data is obtained through a high-resolution numerical algorithm, meaning that the derivatives are available to be used for training. Nevertheless, training methods that only require snapshots of $\mathbf{u}(t)$ itself are preferable since they can be used in a wider range of applications.

However, another problem with derivative training is that a neural network that accurately approximates the derivatives may not yield accurate trajectories when the resulting neural ODE is solved. In practice, when NN is trained using derivative fitting, the resulting trajectory $\mathbf{u}(t)$ may diverge from the training data extremely quickly, even if the neural network approximates the derivatives with a very high accuracy.

This phenomenon is illustrated by Theorem 10.2 of Hairer et al. [26], which is referred to there as the 'fundamental lemma'. This theorem can be seen as the continuous equivalent to Theorem 4.1 and is repeated here:

Theorem 5.1. *Let $\mathbf{x}_{ref}(t), t \geq 0$ be given, and let $\mathbf{x}(t), t \geq 0$ be the solution of the ODE $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = f(t, \mathbf{x})$. If the following hold:*

$$\begin{aligned} a)\quad & \|\mathbf{x}_{ref}(0) - \mathbf{x}(0)\| \leq \varrho \\ b)\quad & \left\| \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}_{ref}(t) - f(t, \mathbf{x}_{ref}(t)) \right\| \leq \varepsilon \\ c)\quad & \|f(t, \mathbf{a}) - f(t, \mathbf{b})\| \leq C \|\mathbf{a} - \mathbf{b}\|, \end{aligned}$$

then the following error bound holds:

$$\|\mathbf{x}_{ref}(t) - \mathbf{x}(t)\| \leq \varrho e^{Ct} + \frac{\varepsilon}{C}\left(e^{Ct} - 1\right).$$

In the case that the initial condition is known exactly, it holds that $\varrho = 0$ so only the second term in the error bound is relevant. Important to note is that this theorem does not require that $\mathbf{x}_{\mathrm{ref}}(t)$ is itself the solution of an ODE. This is an important detail, since in our applications $\mathbf{x}_{\mathrm{ref}}(t)$ results from downsampling the solution of an ODE, and is therefore not necessarily the solution of an ODE itself.

The interpretation of Theorem 5.1 is as follows: suppose that a neural ODE is trained to approximate a trajectory $\mathbf{x}_{\mathrm{ref}}(t)$ as the solution of $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{x}; \vartheta)$. The result of the training procedure is a neural network that accurately predicts the time derivatives of the training data, i.e. $\mathrm{NN}(\mathbf{x}_{\mathrm{ref}}(t); \vartheta) \approx \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}_{\mathrm{ref}}(t)$. If this approximation has an error which is at most ε everywhere, so that $\left\|\mathrm{NN}(\mathbf{x}_{\mathrm{ref}}(t); \vartheta) - \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}_{\mathrm{ref}}(t)\right\| \leq \varepsilon$, and the initial state $\mathbf{x}_{\mathrm{ref}}(0)$ is known (i.e. $\varrho = 0$), then the error in the predicted trajectories at time t is bounded by $\frac{\varepsilon}{C}\left(e^{Ct} - 1\right)$, where C is the Lipschitz constant of the function $\mathbf{x} \mapsto \mathrm{NN}(\mathbf{x}; \vartheta)$. Note that expanding the exponential $e^{Ct} - 1 = Ct + \frac{1}{2}(Ct)^2 + \ldots$ shows that $\frac{\varepsilon}{C}\left(e^{Ct} - 1\right) \approx \varepsilon t$ for $Ct \ll 1$, meaning that if the derivative has an error of at most ε, then the trajectories $\mathbf{x}$ also have an error which is $\mathcal{O}(\varepsilon)$ for small t. Nevertheless, for larger t the difference between the actual and predicted trajectories may grow exponentially as e^{Ct}.

For a slightly more intuitive explanation of this result, note that the time derivative of the error can be split up as follows (this is again analogous to the discrete case of Theorem 4.1):

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\mathbf{x}_{\mathrm{ref}}(t) - \mathbf{x}(t)\right) = \left[\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}_{\mathrm{ref}}(t) - \mathrm{NN}(\mathbf{x}_{\mathrm{ref}}(t); \vartheta)\right] + \left[\mathrm{NN}(\mathbf{x}_{\mathrm{ref}}(t); \vartheta) - \mathrm{NN}(\mathbf{x}(t); \vartheta)\right].$$

The first of these two terms is at most ε, whereas the second is only bounded by $C\|\mathbf{x}_{\mathrm{ref}}(t) - \mathbf{x}(t)\|$. At $t = 0$, the second term is zero but the first term is approximately ε, causing a small but non-zero difference between the actual and predicted trajectories for small t. As $\|\mathbf{x}_{\mathrm{ref}}(t) - \mathbf{x}(t)\|$ increases, the second term becomes dominant and results in the exponential growth in error between the actual and predicted trajectories.

Furthermore, the exponential growth rate of the trajectory error $\mathbf{x}_{\mathrm{ref}}(t) - \mathbf{x}(t)$ is equal to the Lipschitz constant of the neural network, which may not decrease as the neural network is trained. As a result, training a neural network to minimise the error in the derivatives essentially aims to minimise ε while allowing $\mathrm{Lip}(\mathrm{NN}(\cdot; \vartheta))$ to become arbitrarily large unless large Lipschitz constants are specifically penalised by a regularisation term in the loss function. However, even with regularisation C will be positive meaning that the exponential error increase can not be avoided in the general case. Therefore, training the neural network this way can be expected to decrease the local error in the approximation, but at the cost of rapidly increasing global error.

This problem is encountered by Beck et al. [5]: they train neural networks to predict the closure term in three-dimensional fluid flows, and find that neural networks that predict this closure term with high accuracy nonetheless result in completely wrong predicted trajectories. See for example Fig. 10 in [5].

Interestingly, in Hairer et al. Theorem 5.1 is used in a completely different setting: there, $\mathbf{x}(t)$ is the exact solution of a given ODE, and $\mathbf{x}_{\text{ref}}(t)$ is the approximation to this solution obtained by an ODE solver such as forward Euler. Furthermore, the exponential upper bound is not interpreted as a problem that may need to be worked around, but rather as a positive result that guarantees a certain level of accuracy of the approximate ODE solution.

Similar to the discrete case, improving the long-term accuracy of neural ODEs may be possible by using derivative fitting in combination with a regularisation technique such as input perturbation, but another approach is to use a training strategy that does not suffer from the problem described above.

5.1.2 Trajectory fitting

In the case that derivative fitting is not sufficient for obtaining models that make accurate predictions, trajectory fitting may give better results. The main disadvantage of this method is that training on trajectories is computationally more expensive than training on just derivatives, resulting in slower training. This problem can be mitigated by adopting a **two-stage training** approach: first, the network is trained using derivative fitting. After a certain number of iterations or when some stopping criterion is met, the derivative fitting stops and the network is then trained using trajectory fitting. Importantly, if the derivative fitting step is only used to obtain a semi-accurate model which is then further refined by trajectory fitting, approximations for the derivatives $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}$ may be substituted for the real derivatives. For example, such approximations can be obtained through a finite difference approximation on the trajectory. This is similar to the two-stage training method described in Lecture 11 of the MIT course *Parallel Computing and Scientific Machine Learning* [60]. As mentioned there, approximations for the time derivative can even be created in the case that the trajectories are irregularly sampled, for example using a spline interpolation.

In a two-stage training context, the remaining problem is to choose for how long to train a model on derivative fitting. Training on derivative fitting should be done for long enough to reduce the required number of iterations for the slower trajectory fitting, but derivative fitting for too long may result in overfitting, in which the model continues to decrease the training error but starts to become worse at predicting trajectories. A simple but effective method for avoiding overfitting is to use **early stopping**, see for example Algorithm 7.1 of Goodfellow et al. [24]. In early stopping, the model is evaluated on some validation data (i.e. data not included in the loss function and back-propagation). When the accuracy on this data stops improving, the training procedure is stopped. Since many optimisers do not guarantee monotonic convergence, both the training and validation accuracy may fluctuate during training while still trending downward overall. To ensure that such fluctuations do not cause the training procedure to be stopped prematurely, early stopping is usually done with a **patience** parameter, which is the largest number of iterations allowed without improving the validation accuracy over the best accuracy achieved so far. In other words, then the best accuracy on the validation data set occurred more than `patience` epochs ago, the training is stopped and the parameters corresponding to the best validation accuracy are restored. In the context of two-stage training, early stopping can be implemented by using derivative fitting for training, but trajectory fitting for validation.

As mentioned, trajectory fitting itself can be done in two ways: back-propagation using the adjoint ODE, or back-propagation by differentiating through the ODE solver. These two methods are also referred to as optimise-then-discretise and discretise-then-optimise, respectively. The optimise-then-discretise method has the advantage of not requiring a differentiable solver, allowing a wider variety of ODE solvers to be used. However, the training procedure ignores errors introduced by the inexact ODE solver, leading to gradients that are inexact. This inexactness is absent for the discretise-then-optimise method, although this comes at the cost of requiring a differentiable solver. Note that a neural ODE embedded into an ODE solver with a fixed time step Δt can be interpreted as a discrete model that computes $\mathbf{u}(t + \Delta t)$ as a function of $\mathbf{u}(t)$ by performing one time step of the chosen ODE solver on the neural ODE.

An overview of the advantages and disadvantages is shown below.

- Derivative fitting
  - + Simple: only back-propagation through NN required.
  - − No long-term behaviour can be learned.
  - − Time-derivatives of $\mathbf{u}_{\text{ref}}(t)$ are required or must be approximated.
- Trajectory fitting with adjoint ODEs
  - + Long-term behaviour can be learned.
  - + Does not require differentiable ODE solvers.
  - − Length of trajectories used for training must be chosen.
  - − Gradient computation is inexact.
  - − Requires approximation $f(\mathbf{u})$ to be differentiable.
- Trajectory fitting with differentiable solvers
  - + Long-term behaviour can be learned.
  - + Gradients are exact.
  - − Length of trajectories used for training must be chosen.
  - − Requires differentiating through neural network, f, and ODE solver.
  - − Explicit ODE solvers are preferable for differentiation, but not always suitable.

5.2 The Kuramoto-Sivashinsky equation

Since Burgers' equation is relatively predictable (resulting in shock waves which then dissipate over time while moving through the domain), it is useful to consider a more challenging case. For this reason, experiments are performed using the Kuramoto-Sivashinsky equation, named after the two researchers who derived the equation in Kuramoto [44] and Sivashinsky [71]. This PDE is taken with periodic boundary conditions:

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4}, \tag{5.1}$$

$$u(x+L,t) = u(x,t). \tag{5.2}$$

In this PDE, the time-dependent behaviour of u is governed by three terms:

- A non-linear convection term $-\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right)$, the same as in Burgers' equation.
- A destabilising anti-diffusion term $-\frac{\partial^2 u}{\partial x^2}$. Note that this term appears on the right-hand with a minus sign, which is unusual for diffusion terms.
- A stabilising hyper-diffusion term $-\frac{\partial^4 u}{\partial x^4}$. Without this term, the equation would be ill-posed since the anti-diffusion term would cause solutions to blow up in a finite amount of time. After all, solving a PDE with negative diffusion is similar to solving a PDE with positive diffusion forwards in time, which is ill-posed as was illustrated in Figure 3.1.

Like Burgers' equation, the Kuramoto-Sivashinsky equation satisfies conservation of momentum. Since enforcing this conservation law in ML models was found to result in a small but consistent improvement in accuracy, all models considered in this section are chosen to conserve momentum by definition.

5.3 Data generation

Again, the function $u(x,t)$ is discretised as a time-dependent vector $\mathbf{u}(t)$. Since the non-linear convection term is the same as in Burgers' equation, it is discretised in the same way (see (4.7)) with the exception that the artificial diffusion term (given by the vector α in (4.8)) is no longer needed since the Kuramoto-Sivashinsky equation does not yield discontinuities. The linear diffusion and hyper-diffusion terms are discretised using simple 3-wide and 5-wide stencils, respectively, leading to the following discretisation for the three different terms:

$$\begin{aligned}
\left(\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right)\right)(\mathbf{x}_i) &\approx \frac{1}{\Delta x}\left(\mathbf{f}_{i+1/2}-\mathbf{f}_{i-1/2}\right), \\
\text{where } \mathbf{f}_{i+1/2} &= \frac{1}{6}\left(\mathbf{u}_i^2+\mathbf{u}_i\mathbf{u}_{i+1}+\mathbf{u}_{i+1}^2\right), \\
\frac{\partial^2 u}{\partial x^2}(\mathbf{x}_i) &\approx \frac{1}{\Delta x^2}\left(\mathbf{u}_{i-1}-2\mathbf{u}_i+\mathbf{u}_{i+1}\right), \\
\frac{\partial^4 u}{\partial x^4}(\mathbf{x}_i) &\approx \frac{1}{\Delta x^4}\left(\mathbf{u}_{i-2}-4\mathbf{u}_{i-1}+6\mathbf{u}_i-4\mathbf{u}_{i+1}+\mathbf{u}_{i+2}\right).
\end{aligned}$$

Since the PDE was chosen with periodic boundary conditions, the stencils shown here are also applied with periodic boundary conditions, i.e. $\mathbf{u}_{i+N_x}=\mathbf{u}_i$. Written out fully, the resulting ODE is:

$$\begin{aligned}
\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} = f(\mathbf{u})_i = &-\frac{1}{6\Delta x}\left(\mathbf{u}_{i+1}^2-\mathbf{u}_{i-1}^2+\mathbf{u}_i\left(\mathbf{u}_{i+1}-\mathbf{u}_{i-1}\right)\right) \\
&-\frac{1}{\Delta x^2}\left(\mathbf{u}_{i-1}-2\mathbf{u}_i+\mathbf{u}_{i+1}\right)-\frac{1}{\Delta x^4}\left(\mathbf{u}_{i-2}-4\mathbf{u}_{i-1}+6\mathbf{u}_i-4\mathbf{u}_{i+1}+\mathbf{u}_{i+2}\right). \quad (5.3)
\end{aligned}$$

The PDE is discretised with $N_x=1024$ finite volumes, and solved from $t=0$ to $t=256$ while saving a snapshot every $\Delta t=\frac{1}{4}$, yielding 1025 snapshots for each initial condition. The initial conditions are generated in the same way as for Burgers' equation, see (4.10). The fourth-order derivative in the PDE yields a highly stiff ODE system, for which the default `Tsit5` integrator is not suitable. Instead, the 3rd-order accurate stiff ODE solver `Rodas4P` [74] is used, following the recommendations from the DifferentialEquations.jl documentation [56]. Note that `Rodas4P` is a modified version of the `rodas` method created by Wanner et al. [81]. Ostermann et al. [58] shows that `rodas` and other Rosenbrock methods undergo a reduction in their order of accuracy when applied to discretised partial differential equations. The `Rodas4P` algorithm was constructed by modifying the coefficient set of `rodas` to mitigate this order reduction. The resulting solutions are downsampled to 128 finite volumes in space, and every other snapshot is discarded resulting in 513 snapshots with a time step of $\Delta t=\frac{1}{2}$ in between snapshots.

In total, 100 trajectories are computed in this manner. To avoid the effects of initial transients, the first 32 snapshots of the trajectories are not used for training. An example trajectory from the resulting training data is shown in Figure 5.1.

5.4 Evaluation of neural networks for chaotic systems

There is an important difference between Burgers' equation and the Kuramoto-Sivashinsky equation that requires some changes in the setup: the K-S equation produces chaotic behaviour, meaning that arbitrarily small differences in the state $\mathbf{u}(t)$ will eventually lead to a completely different solution. As a result, all methods of approximating this PDE are expected to diverge from the training data at some point, meaning that simply taking the RMSE between the predicted and actual trajectories is not a very useful metric for accuracy. Instead, the accuracy of a method can be evaluated using the Valid Prediction Time (VPT), which is the time until the error between the approximation and the training data exceeds some pre-defined threshold. Here, the VPT of a prediction $\mathbf{u}(t)$ to a real trajectory $\mathbf{u}_{\text{ref}}(t)$ is computed following the procedure used by Pathak et al. [62]: first, the average

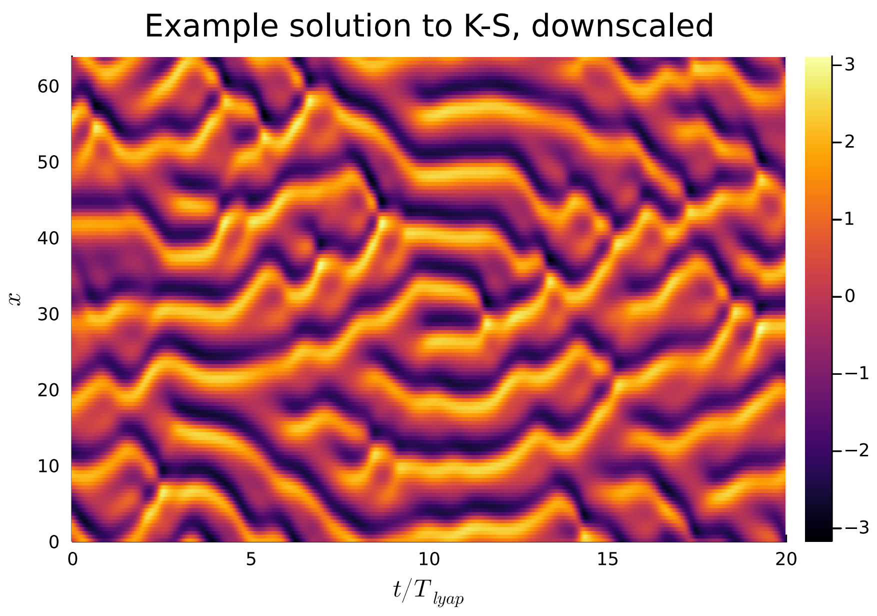

Figure 5.1: An example trajectory from the Kuramoto-Sivashinsky equation used for training models.

energy of the real trajectory is computed as

$$E_{\text{avg}} = \sqrt{\frac{1}{N_t}\sum_{i=1}^{N_t} \|\mathbf{u}_{\text{ref}}(t_i)\|^2}.$$

Then, the valid prediction time is

$$\text{VPT}(\mathbf{u}_{\text{ref}}, \mathbf{u}, t_{1,2,\ldots,N_t}) = \min\left\{t_i \mid \|\mathbf{u}(t_i) - \mathbf{u}_{\text{ref}}(t_i)\| \geq 0.4E_{\text{avg}}\right\}. \tag{5.4}$$

One property shared among many chaotic processes including the Kuramoto-Sivashinsky equation is that the difference between two solutions $\mathbf{u}_{\text{ref}}, \hat{u}$ grows approximately exponentially in time:

$$\|\mathbf{u}_{\text{ref}}(t) - \mathbf{u}(t)\| \approx e^{\lambda_{\max} t}\|\mathbf{u}_{\text{ref}}(0) - \mathbf{u}(0)\|. \tag{5.5}$$

In this equation, $\lambda_{\max}$ is the **Lyapunov exponent** of the ODE, which determines the growth rate of the error between two solutions. This can be generalised to a sequence of real eigenvalues: $\lambda_{\max} = \lambda_1 \geq \lambda_2 \geq \ldots$, but only the largest eigenvalue is of interest here. The Lyapunov exponent of (5.1) depends on the length L of the domain. Edson et al. [17] found the following approximation for the Lyapunov eigenvalues for varying L:

$$\lambda_i(L) \approx 0.093 - \frac{0.94}{L}(i - 0.39),\ i = 1, 2, \ldots$$
$$\implies \lambda_{\max}(L) \approx 0.093 - \frac{0.57}{L}.$$

The training data is created with $L = 64$, leading to a Lyapunov exponent of $\lambda_{\max} \approx 0.084$. The inverse of the Lyapunov exponent is the **Lyapunov time** T_{lyap}, which can be loosely interpreted as the time it takes for the error between to trajectories to grow by a factor e. Taking $L = 64$ yields $T_{\text{lyap}} \approx 12$.

Since the Lyapunov time represents the time scale on which trajectories diverge, it is common to express the valid prediction times of models as multiples of T_{lyap}.

5.5 Experiments

5.5.1 Derivative fitting

As described above, simply training a neural network to accurately predict derivatives may not guarantee accurate trajectories, and this can be mitigated by two-stage training, where training on derivatives

Table 5.1: The additional parameters of the training procedure from Section 5.5.1.

Property	Value
Training data	Trajectories $1-80$, snapshots $33-97$
Optimiser	ADAM, learning rate $= 10^{-3}$
Batch size	64
Epochs	1000
Penalty term	$10^{-4} \cdot \frac{1}{N_\vartheta} \|\vartheta\|^2$
Testing data	Trajectories $91-100$, snapshots $33-273$

is only the first of two training stages, with the second stage being done with trajectory fitting. In this case, it may be possible to use approximations for the derivatives obtained using finite differences, which would make this two-stage training approach suitable even for cases where the derivatives are not available.

In order to test whether the derivatives in the training data can be replaced by finite differences, two identical models are trained in the same way, on different training data:

- the first model is trained on the real time derivatives of the training data, i.e. the training data consisted of pairs $\left(\mathbf{u}_{\text{ref}}(t), \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}_{\text{ref}}(t)\right)$;
- for the second model, the derivatives are replaced by finite differences using the following 4th-order accurate discretisation:
$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}(t) \approx \frac{1}{12\Delta t}\left(\mathbf{u}(t-2\Delta t) - 8\mathbf{u}(t-\Delta t) + 8\mathbf{u}(t+\Delta t) - \mathbf{u}(t+2\Delta t)\right). \tag{5.6}$$

Unless otherwise specified, all models trained in this chapter use the 'large' neural network from Chapter 4, shown in Table 4.3. Parameters used in training are given in Table 5.1.

5.5.2 Derivative fitting with Lipschitz regularisation

As a result of Theorem 5.1, training a neural network to approximate the derivatives of a neural ODE will not result in good long-term accuracy, unless a penalty term is added to 'steer' the neural network towards smaller values of C. Note that for a function $g(\mathbf{x}) = \mathrm{NN}(\mathbf{x};\vartheta)$, one can define the Lipschitz constant as

$$\mathrm{Lip}(g) = \sup_{\substack{\mathbf{x},\mathbf{y}\in\mathbb{R}^n \\ \mathbf{x}\neq\mathbf{y}}} \frac{\|g(\mathbf{x}) - g(\mathbf{y})\|}{\|\mathbf{x}-\mathbf{y}\|}.$$

However, one can also use a different method of error analysis: suppose that t is relatively small, so that $\mathbf{x}_{\text{ref}}(t) \approx \mathbf{x}(t)$. Then:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\left(\mathbf{x}_{\text{ref}} - \mathbf{x}\right) &= f(\mathbf{x}_{\text{ref}}) - g(\mathbf{x}) \\ &= \left(f(\mathbf{x}_{\text{ref}}) - g(\mathbf{x}_{\text{ref}})\right) + \left(g(\mathbf{x}_{\text{ref}}) - g(\mathbf{x})\right) \\ &\approx \left(f(\mathbf{x}_{\text{ref}}) - g(\mathbf{x}_{\text{ref}})\right) + \frac{\mathrm{d}g(\mathbf{x})}{\mathrm{d}\mathbf{x}}\left(\mathbf{x}_{\text{ref}} - \mathbf{x}\right).\end{aligned}$$

Left-multiplying this equation by $(\mathbf{x}_{\text{ref}} - \mathbf{x})^T$ yields:

$$\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\|\mathbf{x}_{\text{ref}} - \mathbf{x}\|^2 \approx (\mathbf{x}_{\text{ref}} - \mathbf{x})^T\left(f(\mathbf{x}_{\text{ref}}) - g(\mathbf{x}_{\text{ref}})\right) + (\mathbf{x}_{\text{ref}} - \mathbf{x})^T\frac{\mathrm{d}g(\mathbf{x})}{\mathrm{d}\mathbf{x}}\left(\mathbf{x}_{\text{ref}} - \mathbf{x}\right).$$

The first term on the right-hand side again stems from the fact that g is only an approximation for the real derivative f, but the second term is now different from the one in Theorem 5.1. From this new approximation, it follows that the growth in error for small t can be decreased by limiting

$$\sup_{\mathbf{y}\in\mathbb{R}^n}\frac{\mathbf{y}^T\mathbf{J}\mathbf{y}}{\mathbf{y}^T\mathbf{y}}, \quad \text{where} \quad \mathbf{J} = \frac{\mathrm{d}g(\mathbf{x})}{\mathrm{d}\mathbf{x}}.$$

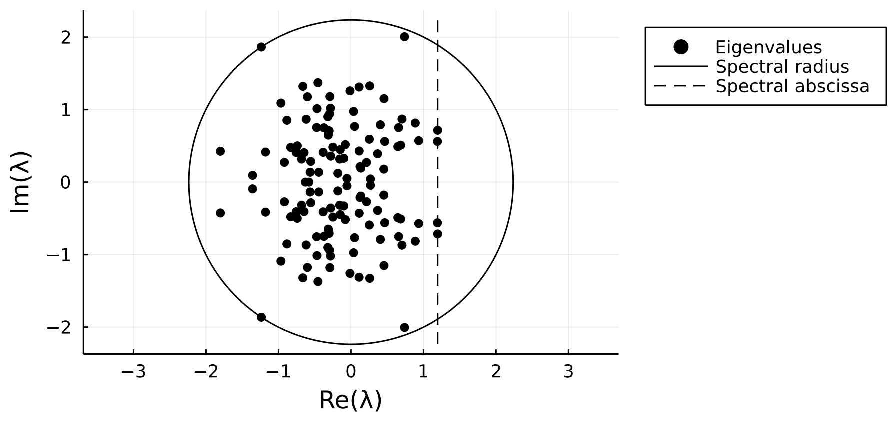

Figure 5.2: An illustration of the spectral radius and spectral abscissa for a given set of complex eigenvalues.

In fact, the Lipschitz constant $\mathrm{Lip}(g)$ can also be expressed in terms of the Jacobian. Federer [21] has shown that

$$\mathrm{Lip}(g) = \sup_{\mathbf{x}\in\mathbb{R}^n} \left\| \frac{\mathrm{d}}{\mathrm{d}\mathbf{x}} g(\mathbf{x}) \right\|,$$

where $\|\mathbf{J}\|$ is the operator norm of a matrix $\mathbf{J}$:

$$\|\mathbf{J}\| = \max_{\mathbf{0}\neq\mathbf{x}\in\mathbb{R}^{N_x}} \frac{\|\mathbf{J}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Note that as a result of this, it may be possible to penalise large values of C during training by adding a regularisation term based on $\left\|\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}} g(\mathbf{u})\right\|$ for each state vector $\mathbf{u}$. However, the operator norm is still difficult to compute numerically, and may instead be replaced by another matrix norm, such as the Frobenius norm $\|\mathbf{J}\|_F = \sqrt{\sum_{i,j=1}^n \mathbf{J}_{ij}^2}$. This is similar to the approach taken by Pan et al. [59], where it is indeed found that adding such a regularisation term can lead to better long-term prediction accuracy.

Note that in the case that the Jacobian is diagonalisable with an orthonormal basis of eigenvectors, both properties can be computed easily as a function of the eigenvalues λ_k of the Jacobian:

$$\sup_{\mathbf{0}\neq\mathbf{x}\in\mathbb{R}^{N_x}} \frac{\|\mathbf{J}\mathbf{x}\|}{\|\mathbf{x}\|} = \max_k |\lambda_k|, \qquad \sup_{\mathbf{0}\neq\mathbf{x}\in\mathbb{R}^{N_x}} \frac{\mathbf{x}^\top\mathbf{J}\mathbf{x}}{\mathbf{x}^\top\mathbf{x}} = \max_k \mathrm{Re}\left(\lambda_k\right).$$

The quantities $\max_k |\lambda_k|$ and $\max_k \mathrm{Re}(\lambda_k)$ are known as the **spectral radius** and **spectral abscissa** of the Jacobian $\mathbf{J}$, respectively. An example of how the spectral radius and abscissa depend on the eigenvalues λ_k is shown in Figure 5.2.

In general, adding a regularisation term based on the eigenvalues of the Jacobian of a neural network may be computationally expensive: for example, for the neural ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u};\vartheta)$ where $\mathbf{u}\in\mathbb{R}^{N_x}$, the Jacobian is an $N_x\times N_x$ matrix. One then needs to compute the eigenvalues of this matrix using some numerical algorithm, and the entire procedure must be differentiable in order to use a standard gradient-based optimiser. In particular, optimising a loss function that includes such a regularisation term involves the second partial derivatives $\frac{\mathrm{d}}{\mathrm{d}\vartheta}\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}}\mathrm{NN}(\mathbf{u};\vartheta)$, which not all auto-differentiation libraries are able to handle efficiently.

However, in the case that the network consists only of convolutional layers, one can make a number of simplifying assumptions that allow for approximating the eigenvalues of the Jacobian much more quickly. This is shown in the following theorem:

Theorem 5.2. *Let $NN_{lin} : \mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$ be a neural network consisting only of convolutional layers with periodic boundary conditions, without any bias vectors or smoothing functions. Then NN_{lin} is a linear operator whose eigenvalues are given by:*

$$\lambda_{1 \cdots N_x} = FFT(NN_{lin}(\mathbf{e}_1)),$$

where $\mathbf{e}_1$ is the vector $(1, 0, 0, \ldots, 0)^\top \in \mathbb{R}^{N_x}$.

Proof. For this proof, a discrete convolution with periodic boundary conditions will be written using a "$\star$":

$$(\mathbf{c} \star \mathbf{u})_i := \sum_{j=-K}^{K} \mathbf{c}_j \mathbf{u}_{i+j}, \quad \text{where } \mathbf{u}_{i+N_x} = \mathbf{u}_i.$$

Now, define the **cyclic right-shift matrix P**:

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & \ldots & 0 & 1 \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ldots & 0 & 0 \\ 0 & \ldots & 0 & 1 & 0 \end{bmatrix} \in \mathbb{R}^{N_x \times N_x}.$$

This matrix performs a cyclic right-shift, i.e. $(\mathbf{Pu})_i = \mathbf{u}_{i-1}$ for $1 < i \leq N_x$ and $(\mathbf{Pu})_1 = \mathbf{u}_{N_x}$. The eigenvectors and -values of $\mathbf{P}$ can be seen to be:

$$\mathbf{P}\mathbf{v}^{(k)} = \lambda_k \mathbf{v}^{(k)},$$

$$\text{where } \mathbf{v}^{(k)} = \begin{pmatrix} 1 \\ \omega^{k-1} \\ \omega^{2(k-1)} \\ \vdots \\ \omega^{(N_x-1)(k-1)} \end{pmatrix}, \quad \lambda_k = \omega^{-(k-1)}, \quad \omega = \exp\left(\frac{2\pi i}{N_x}\right).$$

As a result, matrix powers of $\mathbf{P}$ have the same eigenvectors and similar eigenvalues:

$$\mathbf{P}^m \mathbf{v}^{(k)} = \lambda_k \mathbf{v}^{(k)}, \quad \text{where } \lambda_k = \omega^{-(k-1)m}.$$

Now, note that since $\mathrm{NN}_{\mathrm{lin}}$ consists only of convolutional layers, it commutes with cyclic shifts:

$$\mathrm{NN}_{\mathrm{lin}}\left(\mathbf{P}^k \mathbf{u}\right) = \mathbf{P}^k \mathrm{NN}_{\mathrm{lin}}(\mathbf{u}), \quad \text{for all } \mathbf{u} \in \mathbb{R}^{N_x}, k \in \mathbb{Z}.$$

Combining this with the fact that $\mathrm{NN}_{\mathrm{lin}}$ is linear, it follows that $\mathrm{NN}_{\mathrm{lin}}$ is entirely defined by the output it gives for input $\mathbf{e}^{(1)}$:

$$\begin{aligned} \mathbf{u} &= \sum_{j=1}^{N_x} \mathbf{P}^j \left(\mathbf{u}_j \mathbf{e}^{(1)}\right) \\ \implies \mathrm{NN}_{\mathrm{lin}}(\mathbf{u}) &= \mathrm{NN}_{\mathrm{lin}}\left(\sum_{j=1}^{N_x} \mathbf{P}^j \left(\mathbf{u}_j \mathbf{e}^{(1)}\right)\right) \\ &= \sum_{j=1}^{N_x} \mathbf{u}_j \mathbf{P}^j \mathrm{NN}_{\mathrm{lin}}\left(\mathbf{e}^{(1)}\right). \end{aligned}$$

From this, it follows that the matrix representation of $\mathrm{NN_{lin}}$ can also be expressed entirely in terms of $\mathbf{P}$ and $\mathbf{v} := \mathrm{NN_{lin}}(\mathbf{e}^{(1)})$:

$$\begin{aligned}
&\mathrm{NN_{lin}}(\mathbf{u}) = \mathbf{Au},\\
&\text{where } \mathbf{A} = \begin{bmatrix}\mathbf{v} & \mathbf{Pv} & \mathbf{P}^2\mathbf{v} & \dots & \mathbf{P}^{N_x-1}\mathbf{v}\end{bmatrix}\\
&\qquad = \begin{bmatrix}
\mathbf{v}_1 & \mathbf{v}_{N_x} & \cdots & \mathbf{v}_3 & \mathbf{v}_2\\
\mathbf{v}_2 & \mathbf{v}_1 & \mathbf{v}_{N_x} & & \mathbf{v}_3\\
\vdots & \mathbf{v}_2 & \mathbf{v}_1 & \ddots & \vdots\\
\mathbf{v}_{N_x-1} & & \ddots & \ddots & \mathbf{v}_{N_x}\\
\mathbf{v}_{N_x} & \mathbf{v}_{N_x-1} & \cdots & \mathbf{v}_2 & \mathbf{v}_1
\end{bmatrix}\\
&\qquad = \sum_{j=1}^{N_x} \mathbf{v}_j \mathbf{P}^{j-1}.
\end{aligned}$$

Since $\mathbf{A}$ can be written as a linear combination of powers of $\mathbf{P}$, the eigenvectors of $\mathbf{A}$ are still $\mathbf{v}^{(k)}$ and the eigenvalues can be computed:

$$\begin{aligned}
&\mathbf{Av}^{(k)} = \lambda_k \mathbf{v}^{(k)},\\
&\text{where } \lambda_k = \sum_{j=1}^{N_x} \mathbf{v}_j \omega^{-(k-1)(j-1)}.
\end{aligned}$$

This last equality can be expressed as $\overline{\lambda_k} = \sum_{j=1}^{N_x} \mathbf{v}_j \omega^{(k-1)(j-1)}$: ω lies on the complex unit circle and therefore $\omega^{-1} = \overline{\omega}$. This means that the entire vector $\overline{\lambda_{1,2,\dots,N_x}}$ can be computed by simply taking the Fast Fourier Transform of the vector $\mathbf{v}$: $\lambda_{1\cdots N_x} = \overline{\mathrm{FFT}\left(\mathrm{NN_{lin}}(\mathbf{e}_1)\right)}$. However, even the complex conjugation can be omitted if only the eigenvalues are of interest: after all, since all the coefficients of the neural network are real, the matrix $\mathbf{A}$ is also real meaning that its eigenvalues $\lambda_{1,2,\dots,N_x}$ come in complex conjugate pairs. Therefore, omitting the complex conjugation only changes the order of the eigenvalues, which is not important if only the eigenvalues are used, and their corresponding eigenvectors are not. □

Thus, the eigenvalues of the weights-only model can be computed efficiently. Based on these eigenvalues, one can add a regularisation term that punishes large absolute values, or large real components. The advantages of this approach are that the computation of the eigenvalues can be done very efficiently, and does not depend on the input vector (unlike the eigenvalues of the Jacobian $\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}}\mathrm{NN}(\mathbf{u};\vartheta)$ of the original non-linear model). The main disadvantage is that the eigenvalues of the original model and of the weights-only model are not necessarily the same, and if this difference is significant the regularisation using the eigenvalues of the linearised model may not have a large effect on the eigenvalues of the actual model. Of course, the discrepancy between original and linearised eigenvalues may completely invalidate this approach to regularisation. Nevertheless, regularisation using the linearised eigenvalues is attractive due to the computational efficiency with which the linearised eigenvalues can be computed, making this regularisation technique worth investigating.

In total, six models are tested to evaluate the efficacy of these eigenvalue-based regularisation terms. The layer structure of the neural networks is shown in Table 5.2 and the details of the training procedure are shown in Table 5.3. Note that this is not the same layer structure as used in earlier experiments, since those experiments used a quadratic layer that is not compatible with Theorem 5.2. The final layer computes forward differences, which is compatible with this theorem since it can be thought of as a convolution with fixed parameters $\mathbf{c}_0 = -1, \mathbf{c}_1 = 1$. All six models have identical layer structure and the same initial parameters. Furthermore, all models are trained on the same data set in the same order. As such, the only differences between models are in the loss function, specifically the penalty term that is added.

- Models 1 and 2 are trained without regularisation based on eigenvalues: the first model uses no regularisation at all, and the second uses a simple sum-of-squares regularisation over all parameters.

Table 5.2: A description of the neural network structure used for the numerical experiments regarding eigenvalue-based regularisation on the K-S equation.

Layer	Description	σ	Parameters
1	5-wide conv, $1 \to 2$ channels	tanh	12 ($5 \times 1 \times 2$ weights and 2 biases)
2	5-wide conv, $2 \to 4$ channels	tanh	44 ($5 \times 2 \times 4$ weights and 4 biases)
3	5-wide conv, $4 \to 6$ channels	tanh	126 ($5 \times 4 \times 6$ weights and 6 biases)
4	5-wide conv, $6 \to 8$ channels	tanh	248 ($5 \times 6 \times 8$ weights and 8 biases)
5	5-wide conv, $8 \to 6$ channels	tanh	246 ($5 \times 8 \times 6$ weights and 6 biases)
6	5-wide conv, $6 \to 4$ channels	tanh	124 ($5 \times 6 \times 4$ weights and 4 biases)
7	5-wide conv, $4 \to 2$ channels	tanh	42 ($5 \times 4 \times 2$ weights and 2 biases)
8	5-wide conv, $2 \to 1$ channels	–	11 ($5 \times 2 \times 1$ weights and 1 biases)
8	$\mathbf{u} \mapsto \Delta_{\text{fwd}}\mathbf{u}$	–	0
Total			853

Table 5.3: The additional parameters of the training procedure.

Property	Value
Training data	Trajectories $1 - 80$, snapshots $33 - 97$
Optimiser	ADAM, learning rate $= 5 \cdot 10^{-4}$
Batch size	64
Epochs	4000
Testing data	Trajectories $91 - 100$, starting from snapshot 33

- Models 3 and 4 use regularisation terms based on the spectral radius and spectral abscissa, respectively. These two properties correspond closely to the notions of stiffness and instability in the neural ODE, but since they are both defined as maxima they are formally not differentiable everywhere.
- To avoid issues due to lack of smoothness in the regularisation, models 5 and 6 are trained with regularisation terms that penalise large absolute eigenvalues or large real components in a differentiable way.

After training on finite differences, the VPT on 10 testing trajectories was computed.

5.5.3 Optimise-then-discretise

Stiff ODEs and IMEX methods

When discretising the Kuramoto-Sivashinsky PDE (5.1) using the method of lines, the 4th derivative in x yields a term with a leading factor $\frac{1}{\Delta x^4}$ in the ODE (5.3). For fine discretisations, Δx is small resulting in a highly stiff ODE for which standard explicit methods are not suitable. Normally, one would therefore use an implicit method instead, such as the `Rodas4P` solver used to generate training data. However, such implicit methods typically require the computation of Jacobians of the right-hand side function, which in this case is computationally expensive if the neural network is large. Since the stiff ODE terms are contained in $f(\mathbf{u})$, the neural network term is not expected to be stiff and as such it is preferable to solve ODEs using a method that is implicit in $f(\cdot)$ but explicit in $\mathrm{NN}(\cdot;\vartheta)$. Such ODE solvers are called implicit-explicit (IMEX) methods.

The `DifferentialEquations.jl` software library contains implementations of a number of IMEX ODE solvers. After some benchmarking, good results were obtained using the `KenCarp47` algorithm created by Kennedy and Carpenter [39] (in their paper, this algorithm is referred to as `ARK4(3)7L[2]SA`$_1$). The `KenCarp47` algorithm combines an Explicit-first-stage Singly-Diagonally Implicit Runge-Kutta (ESDIRK) scheme with an explicit Runge-Kutta scheme. Both the implicit and explicit schemes are

7-stage 4th-order accurate with embedded 3rd-order methods for error estimation. Like most implicit ODE solvers, `KenCarp47` solves non-linear systems of equations in order to perform a time step. While typically Newton's method is used to solve such systems of equations, in this case it was found that Anderson's acceleration [2] yielded better performance, and also more accurate gradients. This is due to the following reasons:

- While Newton's method is typically very efficient, a single Newton iteration is often quadratic (or worse) in the size of the non-linear system, which is $N_B N_x$ when solving an ODE over N_x variables in a batch of N_B trajectories simultaneously. This results in a running time that scales with N_B^2, making Newton's method inefficient when running on a batch of multiple ODEs.
- It was found that because Newton's method is more stable than Anderson's acceleration, it results in larger time steps being taken when solving the forward ODE. While this is usually good, it also results in a less accurate interpolation for $\mathbf{u}(t)$ used in the adjoint ODE solution (Algorithm 2). This can be mitigated by forcing the time step to be smaller, but doing so also removes any performance benefit that `KenCarp47`+Newton's method had.

Note that all tests are done using the interpolating adjoint method (Algorithm 2), rather than the backsolve or quadrature adjoint methods (Algorithms 1 and 3). The backsolve method is not suitable for ODEs that are not well-posed when solved backwards in time, and the quadrature adjoint method was found to be less efficient than the interpolating adjoint method. The reason for this is that computing the terms $\mathbf{y}^\top \frac{\partial f}{\partial \mathbf{x}}$ and $\mathbf{y}^\top \frac{\partial f}{\partial \vartheta}$ as in (3.6a) and (3.6b) involves a lot of shared work, namely the back-propagation through all layers of the neural network. Thus, computing both derivatives simultaneously as in the interpolating method is more efficient than computing them in two separate passes as in the quadrature method.

Chaotic systems and error weight decay

A second problem encountered for the K-S equation is that its chaotic nature results in exploding gradients (see Section 4.4.2). In the case of chaotic systems (including the Kuramoto-Sivashinsky PDE and its discretisations), the exploding gradients problem can be derived theoretically: given the exponential growth rate in the difference between two initially similar trajectories:

$$\|\mathbf{u}_{\text{ref}}(t) - \mathbf{u}(t)\| \approx e^{\lambda_{\max} t} \|\mathbf{u}_{\text{ref}}(0) - \mathbf{u}(0)\|,$$

one can conclude that the Jacobian $\frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}\mathbf{u}(0)}$ should grow at approximately the same rate:

$$\left\| \frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}\mathbf{u}(0)} \right\|_2 \approx e^{\lambda_{\max} t}.$$

This can be checked numerically, since the $\frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}\mathbf{u}(0)}$ corresponds to the matrix $\mathbf{W}$ in algorithm 4, and can therefore be computed by solving the augmented ODE of (3.10a) and (3.10b). This was done for one initial condition $\mathbf{u}^{(0)}$ from the training data, resulting in the plot shown in Figure 5.3. This shows that the norm of the Jacobian indeed grows exponentially. Computing the slope on this graph between $t = 50$ and $t = 240$ yields the approximation

$$\left\| \frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}\mathbf{u}(0)} \right\|_2 \approx e^{0.093t},$$

and the exponential growth factor 0.093 is indeed close to the Lyapunov exponent $\lambda_{\max} \approx 0.084$.

The exponential increase in sensitivity also results in an exponential increase in the gradients. This can be seen by investigating the vectors $\mathbf{y}(t)$ and $\mathbf{z}(t)$ of the adjoint ODE ((3.6a) and (3.6b)). The norms of these vectors are shown in Figure 5.4. Note that since the adjoint ODE is solved for $t = T$ down to $t = 0$ (i.e. from right to left in the figure), they actually increase exponentially although the graph appears to show a decrease.

As claimed in Section 4.4.2, another effect of exploding gradients is that the total computed gradient $\frac{\mathrm{dLoss}}{\mathrm{d}\vartheta}$ depends mostly on the prediction error later in the time series. This claim can also

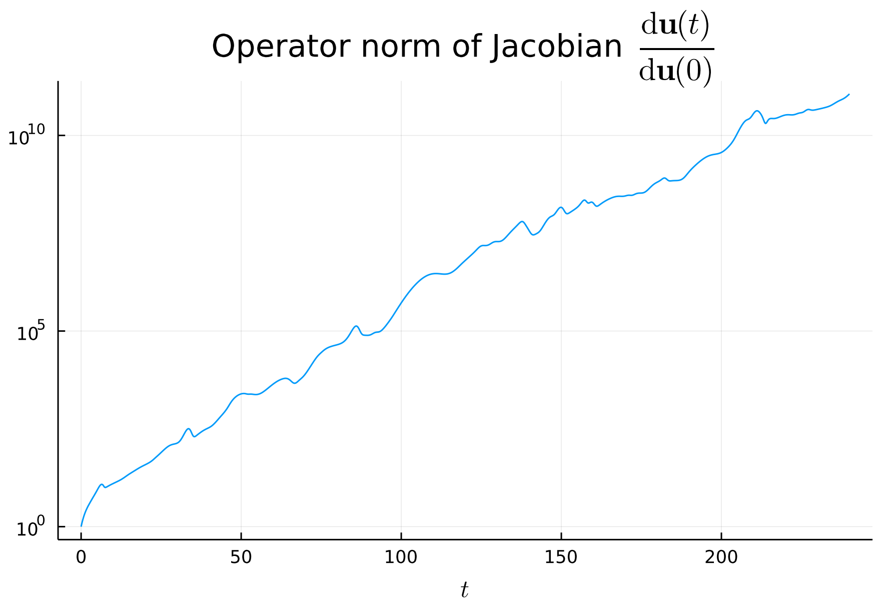

Figure 5.3: The operator norm of $\frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}\mathbf{u}(0)}$ as a function of t.

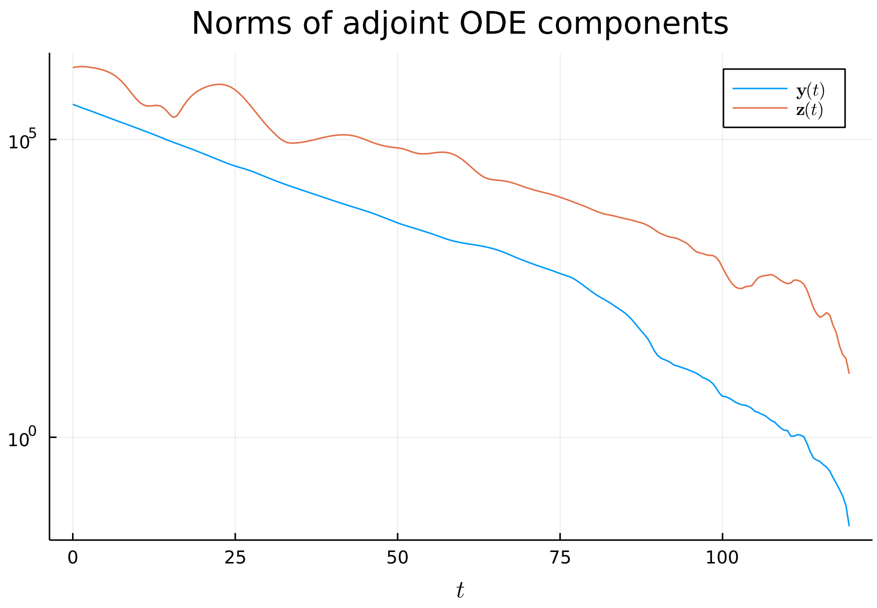

Figure 5.4: The norms of the vectors $\mathbf{y}(t), \mathbf{z}(t)$ of the adjoint ODE as a function of t.

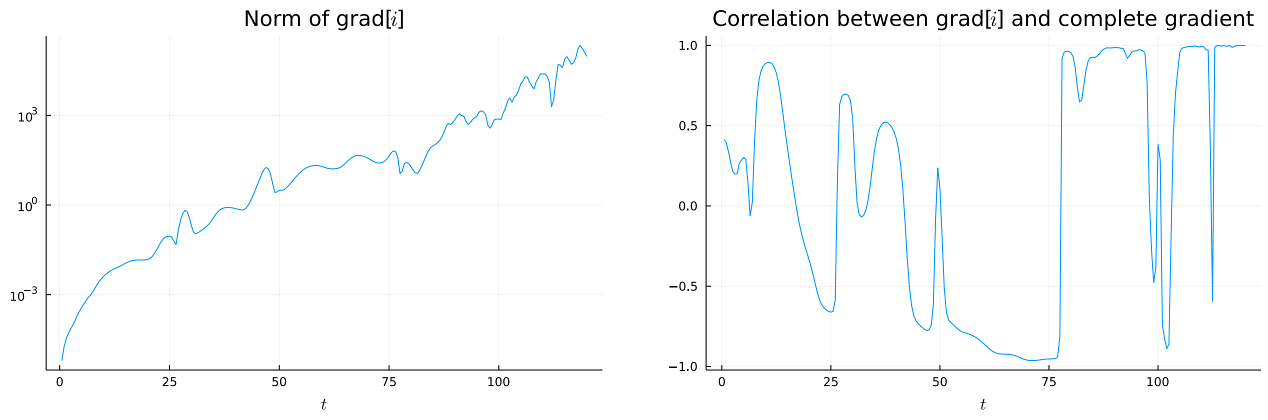

Figure 5.5: Left: the norm of the individual gradients grad_i as a function of time. As described, the gradients become exponentially larger as t increases. Right: the correlation of the individual gradients with the full gradient that is given to the optimisation algorithm.

be tested, by computing the solution of the adjoint ODE with Algorithm 2, but only performing the update to $\mathbf{y}(t)$ at one time point t_i. This way, gradients can be computed of the error at one time step with respect to the parameters:

$$\frac{\mathrm{dLoss}}{\mathrm{d}\vartheta} = \sum_{i=1}^{N_t} \mathrm{grad}_i \quad \text{where } \mathrm{grad}_i := \frac{\mathrm{d}}{\mathrm{d}\vartheta} L\left(\mathbf{x}(t_i), \mathbf{u}^{(i)}\right).$$

Figure 5.5 shows the norms of the individual gradients, as well as the correlations (as (5.7)) of the individual gradients with the complete gradient that is used for optimisation.

$$\mathrm{corr}(\mathbf{v}, \mathbf{w}) := \frac{(\mathbf{v}, \mathbf{w})}{\|\mathbf{v}\| \cdot \|\mathbf{w}\|}. \tag{5.7}$$

From this, the following becomes clear:

1. Indeed, the individual gradients of the error at time t with respect to ϑ grow exponentially in t.

2. As a result, the full gradient, which is simply the sum of individual gradients, is strongly correlated with the individual gradients for large t, and much less so for small t. Note that the correlation is still large for some smaller values of t, but this is likely just due to the chaotic nature of the ODE solution.

Due to the exploding gradients problem, training on long trajectories is not expected to work well without modifications. One alternative is to only train on relatively short trajectories. Another option is to train on long trajectories with a different loss function, that puts more 'weight' on the error for small t and less weight for large t. Since the Jacobian $\frac{\mathbf{du}(t)}{\mathbf{du}(0)}$ is known to grow as $e^{\lambda_{\max} t}$, a reasonable choice is to take a loss function such as:

$$\mathrm{Loss}_c = \frac{1}{Z} \sum_{i=1}^{N_t} \exp(-2c\lambda_{\max} t_i) \cdot \|\mathbf{u}(t_i) - \mathbf{u}_{\mathrm{ref}}(t_i)\|^2, \tag{5.8}$$

$$\text{where } Z = N_{\mathbf{x}} \sum_{i=1}^{N_t} \exp(-2c\lambda_{\max} t_i). \tag{5.9}$$

This loss function generalises the Mean Square Error by allowing the prediction error at time t_i to be weighted by a factor $\exp(-2c\lambda_{\max} t_i)$. Here, the constant c can be chosen arbitrarily, and taking $c = 0$ recovers the standard MSE. Note that the sum is over squared errors, which grow as $\exp(2\lambda_{\max} t)$, meaning that the reasonable choice according to the above theory is $c = 1$.

Note that the use of such an exponentially decaying loss function is not new: it is also mentioned as a strategy to avoid local minima in ODE parameter estimation in the documentation of the DiffEqFlux

Table 5.4: The training parameters used when training on short trajectories. Parameters that were different for training on long trajectories are shown in parentheses.

Property	Value
Training data	Trajectories 1 – 80, snapshots 33 – 57 (33 – 177)
Penalty term	None
Optimiser	ADAM, learning rate = 10^{-3} with gradient clipping to $r = 10^{-2}$
Batch size	10
Epochs	1000 (100)
Validation data	Trajectories 81 – 90, snapshots 33 – 129, evaluated every 10 epochs
Testing data	Trajectories 91 – 100, starting from snapshot 33

software library [75]. Furthermore, weighted MSE loss functions can be seen as a special case of minimising a **negative log-likelihood**. If, from the chaotic nature of the K-S equation, one makes the assumption that the individual components of the vector $\mathbf{u}(t)$ are distributed as a Gaussian distribution with mean $\mathbf{u}_{\text{ref}}(t)$ and standard deviation (proportional to) $\sigma_i := \exp(c\lambda_{\max}t)$, then the negative log-likelihood is given by:

$$\begin{aligned}
\text{NLL} &= -\log\left[\prod_{i=1}^{N_t}\prod_{j=1}^{N_x} p_t\left(\mathbf{u}_j(t_i) \mid \mathbf{u}_{\text{ref},j}(t_i)\right)\right] \\
&= -\log\left[\prod_{i=1}^{N_t}\prod_{j=1}^{N_x} \frac{1}{\sigma_i\sqrt{2\pi}}\exp\left(-\frac{\left[\mathbf{u}_j(t_i) - \mathbf{u}_{\text{ref},j}(t_i)\right]^2}{\sigma_i^2}\right)\right] \\
&= N_x\sum_{i=1}^{N_t}\log\left[\sigma_i\sqrt{2\pi}\right] + \sum_{i=1}^{N_t}\sum_{j=1}^{N_x}\frac{\left[\mathbf{u}_j(t_i) - \mathbf{u}_{\text{ref},j}(t_i)\right]^2}{\sigma_i^2} \\
&= N_x\sum_{i=1}^{N_t}\log\left[\sigma_i\sqrt{2\pi}\right] + \sum_{i=1}^{N_t}\frac{1}{\sigma_i^2}\sum_{j=1}^{N_x}\left[\mathbf{u}_j(t_i) - \mathbf{u}_{\text{ref},j}(t_i)\right]^2 \\
&= N_x\sum_{i=1}^{N_t}\log\left[\sigma_i\sqrt{2\pi}\right] + \sum_{i=1}^{N_t}\exp(-2c\lambda_{\max}t)\left\|\mathbf{u}(t_i) - \mathbf{u}_{\text{ref}}(t_i)\right\|^2,
\end{aligned}$$

which is equal to equation (5.8) up to a linear transformation, meaning that both loss functions result in equivalent optimisation problems. However, not only is the use of such a loss function necessary in this context, the underlying theory can here be used to find a good exponential decay rate, which would otherwise have to be found by trial-and-error.

Table 5.4 shows the training setup for the optimise-then-discretise training tests. In order to avoid very large gradients causing the training to fail, **gradient clipping** (5.10) is applied to the gradients before applying the optimiser. This way, gradients whose norm exceeds some constant r are scaled such that their norm is exactly r, thereby avoiding very large gradients while leaving small gradients unchanged.

$$\text{clipnorm}(\mathbf{v}, r) = \begin{cases}\mathbf{v} & \text{if } \|\mathbf{v}\| \le r, \\ r\frac{\mathbf{v}}{\|\mathbf{v}\|} & \text{if } \|\mathbf{v}\| > r.\end{cases} \tag{5.10}$$

5.5.4 Discretise-then-optimise

As mentioned in Section 3.3, training neural ODEs for time series regression can also be done without solving adjoint ODEs to compute gradients. Instead, gradients can be computed simply by back-propagating directly through the ODE solver. This approach is called “discretise-then-optimise”, referring to the fact that by embedding the neural network into an ODE solver with a fixed time step, the time-continuous ODE formulation is turned into a discrete formulation before training (i.e. weight optimisation) takes place.

The advantage of the discretise-then-optimise approach is that back-propagation through the ODE solver yields exact gradients of the computations that are performed, but this comes at the cost of requiring a differentiable ODE solver. While many standard ODE solvers such as explicit Runge-Kutta methods and explicit linear multi-step (Adams-Bashforth) methods are trivially differentiable, the same is not true for semi-explicit and implicit methods. Note that even semi-explicit methods that are explicit in the neural closure term, as described in Section 5.5.3, are not trivial to back-propagate through since even back-propagating through a single time step also requires computing gradients with respect to internal approximations for intermediate values of $\mathbf{u}(t)$. It is also important to note that back-propagating through implicit methods is possible, since the gradient of an implicitly defined function (i.e. a function that involves solving systems of equations) can be computed by the Implicit Function Theorem, as demonstrated by Kolter et al. [15]. Nevertheless, explicit ODE solvers are preferable over implicit methods whenever they are applicable, due to their simplicity and speed, as well as the property that back-propagation through explicit methods is comparatively easy. This poses a problem for the K-S equation, since discretising this PDE yields a stiff ODE, for which many explicit methods are not suitable:

- Most explicit methods, including Runge-Kutta methods and linear multi-step methods, only produce stable solutions to stiff ODEs when taking very small time steps, which negatively impacts performance.

- Some other explicit methods, such as Runge-Kutta-Chebyshev (RKC) methods (see pages $31-36$ of Wanner and Hairer [81] and pages $419-438$ of Hundsdorfer and Verwer [31]), are more stable than standard explicit methods thus allowing larger time steps, but such methods are often only first- or second-order accurate.

Finally, there is a class of 'exponential' time integration methods that assume that the stiff terms in the ODE are linear. Under that assumption, exponential time integrators compute the effect of those terms exactly while approximating the effect of the non-stiff non-linear terms using a standard Runge-Kutta method. Such methods can be used to solve the discretised K-S equation (5.3) since such ODEs are stiff mainly due to the anti-diffusion and hyper-diffusion terms, which are both linear. Exponential integrators of orders 2, 3, and 4 are derived by Cox and Matthews [12]. Their algorithms require coefficients that are difficult to compute numerically, but these issues were overcome by Kassam and Trefethen [37], resulting in `ETDRK4`, a numerically stable and efficient fourth-order time stepping method for ODEs with stiff linear terms and non-stiff non-linear terms. The resulting algorithm was found to perform very well on a variety of problems including the Kuramoto-Sivashinsky equation. As such, this algorithm can be used to take large time steps efficiently and differentiably, and is therefore suitable for training neural ODEs with a discretise-then-optimise approach.

To avoid performing costly matrix operations, the ODE is solved in the pseudo-spectral domain, meaning that the ODE is not over the variables $\mathbf{u}(t)$, but over their Discrete Fourier Transform $\hat{\mathbf{u}}(t) := \mathcal{F}\mathbf{u}(t)$. Transforming the K-S equation in this way yields the following ODE system:

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\mathbf{u}} = \left(\mathbf{\Lambda}^2 - \mathbf{\Lambda}^4\right)\hat{\mathbf{u}} - \frac{i}{2}\mathbf{\Lambda}\mathcal{F}\left(\left(\mathcal{F}^{-1}\hat{\mathbf{u}}\right)^2\right), \tag{5.11}$$

where $\mathbf{\Lambda}$ is a diagonal matrix $\mathbf{\Lambda} = \operatorname{diag}\left(\lambda_0, \lambda_1, \ldots, \lambda_{N_\mathbf{x}-1}\right)$ where

$$\lambda_k = \begin{cases} \frac{2\pi k}{L} & \text{for } 0 \leq k < \frac{N_\mathbf{x}}{2}, \\ 0 & \text{for } k = \frac{N_\mathbf{x}}{2}, \\ \frac{2\pi}{L}(k - N_x) & \text{for } \frac{N_\mathbf{x}}{2} < k \leq N_\mathbf{x} - 1. \end{cases}$$

Note that when adding a closure term to this ODE, it is natural to express the closure term in terms of $\hat{\mathbf{u}}$ as well. However, in order to be able to compare more directly to the previous models, the closure term will be given by the same neural network as used in earlier experiments, meaning that

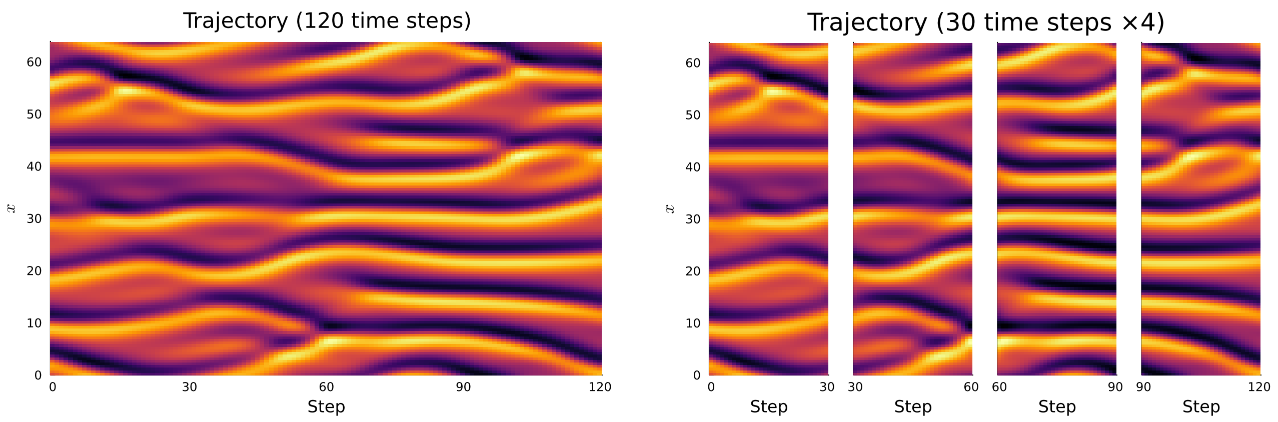

Figure 5.6: A visual representation of how the training data is split up into smaller trajectories. In this example, the trajectories consisting of an initial condition followed by 120 additional snapshots is split up into four trajectories, each consisting of an initial condition followed by 30 additional snapshots. The last snapshot of one trajectory is equal to the initial condition of the next trajectory.

Table 5.5: The additional parameters of the training procedure from Section 5.5.4.

Property	Value
Training data	Trajectories $1-80$, snapshots $33-153$, split up into smaller trajectories
Optimiser	ADAM, learning rate $= 10^{-3}$
Batch size	32
Epochs	5000
Penalty term	None
Testing data	Trajectories $91-100$, snapshots $33-273$

its input and output are in the physical domain. As such, the neural network term is preceded by an inverse Fourier transform and followed by a Fourier transform:

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\mathbf{u}} = \left(\mathbf{\Lambda}^2 - \mathbf{\Lambda}^4\right)\hat{\mathbf{u}} - \frac{i}{2}\mathbf{\Lambda}\mathcal{F}\left(\left(\mathcal{F}^{-1}\hat{\mathbf{u}}\right)^2\right) + \mathcal{F}\left(\mathrm{NN}\left(\mathcal{F}^{-1}\hat{\mathbf{u}};\vartheta\right)\right). \tag{5.12}$$

Also, note that the Fourier transform of a real vector $\mathbf{u}$ is a complex vector, and the inverse Fourier transform is again a complex vector. To avoid using complex numbers in the convection and closure terms, inverse Fourier transforms are always followed by converting the complex arrays to real arrays, i.e. by removing the imaginary components of the vector. This does not significantly affect the accuracy of the resulting model, since $\hat{\mathbf{u}}$ should always be such that $\mathcal{F}^{-1}\hat{\mathbf{u}}$ is real and in practice this is also the case up to rounding error.

While the new training method involves making discrete time steps, the internal neural network is still used to compute the closure term in the ODE. When training with differentiable solvers, one must choose the number of time steps that are predicted with the model. In their experiments with neural closure models for two-dimensional incompressible fluid flow problems, List et al. [48] refer to this as the number of unrolled steps N_{unroll}, and find that this has a significant effect on the stability of the resulting model. As such, a number of different models are trained by unrolling by a varying number of time steps. Specifically, 11 different models are trained with $N_{\text{unroll}} \in \{1, 2, 4, 8, 15, 30, 60, 90, 00, 110, 120\}$. The model with $N_{\text{unroll}} = 120$ is trained on snapshots $33-153$ of each of the 80 training trajectories. For the models with $N_{\text{unroll}} \in \{90, 100, 110\}$, the same 80 trajectories are truncated to the desired number of steps. For the remaining models, the training trajectories are split up into multiple shorter trajectories as shown in Figure 5.6. In this way, each training trajectory of 121 snapshots (an initial condition followed by 120 additional time steps) can be split up into 4 trajectories of 31 snapshots each, or 15 trajectories of 9 snapshots each, and so on.

While the method of training neural ODEs by back-propagating through a differentiable ODE solver allows for faster training and results in more accurate models, in theory it is tied to a specific

Table 5.6: An overview of the performance characteristics of different ODE solvers for solving the ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{NN}(\mathbf{u};\vartheta)$.

Batch size	ODE solver (+NL solver)	Pass	Time taken	Time steps	f calls	NN calls
	Tsit5	Forward	6.45s	4924	29559	29559
	KenCarp47+Newton	Forward	0.48s	38	3788	205
1	KenCarp47+Anderson	Forward	2.47s	1252	35474	7323
	Tsit5	Backward	46.56s	5022	30302	30302
	KenCarp47+Anderson	Backward	18.46s	1572	41499	8919
	Tsit5	Forward	15.98s	4926	29571	29571
	KenCarp47+Newton	Forward	15.08s	45	34118	255
10	KenCarp47+Anderson	Forward	6.57s	1362	35338	7531
	Tsit5	Backward	70.53s	5025	30308	30308
	KenCarp47+Anderson	Backward	30.31s	1490	39591	8388

Table 5.7: A comparison of the valid prediction times of the models trained on derivatives and finite differences.

Training data	Valid prediction time		
	Min	Avg	Max
Reference	1.17	1.93	3.00
Derivatives	1.79	3.42	4.88
Finite differences	2.17	3.06	4.25

ODE solver with a specific time step. This means that when taking the same neural network and embedding it into a different ODE solver, the performance is expected to deteriorate. To investigate the effect of changing the ODE solver and time step, one of the 11 trained models is tested with different combinations of ODE solvers and time steps. The tested ODE solvers include not only the `ETDRK4` method that is used for training, but also the lower order `ETDRK1-3` methods, described by Cox and Matthews [12]. These methods were used with a time step of $\Delta t = \frac{1}{2}$, as used for training, but also with $\Delta t = \frac{1}{8}, \frac{1}{4}, 1$.

5.6 Results

5.6.1 IMEX schemes

Table 5.6 shows the performance results of different ODE solvers for the neural closure model $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{NN}(\mathbf{u};\vartheta)$. The tests using the KenCarp47 ODE solver also mention the algorithm used for solving the internal non-linear systems of equations. Note that the IMEX methods still require a large number of time steps, indicating that the neural network term also contributes to the stiffness of the ODE. However, some fully implicit methods were tested as well, which took significantly longer than the IMEX methods. Implicit methods failed completely for the adjoint ODE, since such methods require computing second derivatives of the neural network term which is not supported by the used AD library. Even **Rosenbrock-W methods** (see pages 114 – 116 of [81]), which can produce ODE solutions using only finite-difference approximations to the Jacobian, were found to be much slower than `KenCarp47` due to the need to evaluate the neural network many more times per step in order to approximate the Jacobian.

5.6.2 Training methods

Table 5.7 shows the results of training a simple neural ODE on derivatives and finite difference approximations for those derivatives.

Table 5.8: The six different penalty terms that were tested, and the results of training with those penalty terms.

Model	Penalty term	Valid Prediction Time		
		Min	Avg	Max
1	None	3.29	4.29	5.71
2	$10^{-6} \cdot \frac{1}{N_\vartheta} \lVert\vartheta\rVert^2$	3.29	4.29	5.71
3	$10^{-6} \cdot \max_j \lvert\lambda_j\rvert$	3.21	3.72	4.33
4	$10^{-6} \cdot \max_j \Re(\lambda_j)$	1.96	3.14	4.17
5	$10^{-6} \cdot \sum_j \lvert\lambda_j\rvert^2$	2.12	3.54	5.25
6	$10^{-6} \cdot \sum_j \max(\Re(\lambda_j), 0)^2$	2.58	3.72	4.96

Table 5.9: The predicted and actual spectral radius and abscissa for each of the six models trained with varying regularisation terms.

Model	Spectral radius		Spectral abscissa	
	Predicted	Actual	Predicted	Actual
1	53.34	7.36	6.12	0.47
2	53.19	7.36	6.10	0.47
3	0.39	6.40	0.39	0.47
4	41.04	7.31	0.01	0.47
5	0.03	7.04	0.03	0.47
6	41.03	7.39	0.05	0.47

Table 5.8 shows the performance of models trained with different spectrum-based regularisation terms. From these results, it is clear that this approach to regularisation does not work. To see why, the exact spectral radius and abscissa of the models are computed by computing the eigenvalues of the Jacobian of each model on each vector $\mathbf{u}$ from the training data. The resulting spectral radius ρ and spectral abscissa η are compared to their approximations from the weights-only models. These results are shown in Table 5.9. Note that models 3 and 5, which were trained with a regularisation term penalising large absolute eigenvalues, indeed have a very small predicted spectral radius. The actual spectral radius is also somewhat smaller than that of the other models but not much. Similarly, models 4 and 6 were trained to have a small spectral abscissa, but the actual spectral abscissae are equal (up to two decimal places) to those of other models. Overall, Table 5.9 shows that using the weights-only neural network to approximate eigenvalues can drastically over- *and* under-estimate the actual eigenvalues. Thus, these methods are not effective at reducing the spectral radius or spectral abscissae of neural networks.

Table 5.10 shows the minimum, average, and maximum valid prediction times (VPTs) of neural ODEs with different training methods, and some example trajectories and predictions are shown in Figure 5.7. From this table, it can be seen that closure models indeed perform better than direct neural ODEs, since each closure model performs better than the direct model trained in the same way. Furthermore, performing trajectory fitting works best on short trajectories, where the resulting models slightly outperform models trained by derivative fitting. Training on long trajectories results in worse performance, even if the loss function is weighted to mitigate the exploding gradients problem. This is likely due to the fact that the adjoint ODE methods introduce an error in the gradients used during training. As is generally the case for ODE solutions, this error increases with the time interval over which the ODE is solved. Thus, training on longer trajectories introduces a greater error in the gradients, thus reducing the accuracy of the resulting model even if the error function is weighted to mitigate the exploding gradients problem.

Table 5.11 and Figure 5.8 show the minimum, average, and maximum VPTs of the 11 models

Table 5.10: An overview of the performance of many different tested models, sorted by training method and whether or not the model includes the term $f(\mathbf{u})$.

	Training method	Description	VPT		
			Min	Avg	Max
	-	Coarse ODE	1.17	1.93	3.00
Direct models	Derivative fitting	Last model	3.21	4.39	5.42
		Best model	2.54	4.30	5.92
	Trajectory fitting	Short trajectories	2.42	3.63	5.42
		Long trajectories	0.38	0.45	0.54
	Long trajectories, decaying error weights	$c = 0.5$	0.92	1.55	2.88
		$c = 1.0$	1.08	1.73	2.46
		$c = 1.5$	1.88	2.75	4.46
		$c = 2.0$	1.42	2.55	3.96
Closure models	Derivative fitting	Last model	3.67	5.26	6.54
		Best model	4.04	5.15	6.54
	Trajectory fitting	Short trajectories	4.08	5.84	8.29
		Long trajectories	2.38	3.38	4.67
	Long trajectories, decaying error weights	$c = 0.5$	2.42	4.20	5.38
		$c = 1.0$	2.96	4.38	6.29
		$c = 1.5$	3.29	4.58	5.88
		$c = 2.0$	2.71	4.29	5.75

trained using discretise-then-optimise on varying number of unrolling steps. It can be seen that models trained on 60 or more steps perform worse than models trained on fewer steps, although performance continues to improve after 1000 epochs.

After 5000 epochs, the model trained on 30 steps performed the best. This model is then re-used with different ODE solvers and time steps. Note that this model is not re-trained: the neural network is simply used to make new trajectory predictions in combination with different ODE solvers and time steps than those used during training. The average VPTs of the resulting models are shown in Table 5.12. The bold-faced entry for the combination $\left(\texttt{ETDRK4}, \Delta = \frac{1}{2}\right)$ is the configuration used for training, and unsurprisingly performs the best. Interestingly, while using 1st-order methods or large time steps gives very poor performance, the models with 3rd- and 4th-order methods and $\Delta t \in \left\{\frac{1}{8}, \frac{1}{4}\right\}$ yield VPTs that are similar to the VPTs of other models. This supports the hypothesis that the increased accuracy of the discretise-then-optimise strategy is due to the model's ability to correct for both the spatial and temporal discretisation error, rather than just the spatial discretisation error as is the case for derivative fitting or optimise-then-discretise.

Finally, the best result from each training method is shown in the table below:

Training method	Avg VPT
Finite differences	5.15
Optimise-then-discretise	
▷ Short trajectories	5.84
▷ Long trajectories	3.38
▷ Weighted error function	4.29
Discretise-then-optimise	**7.10**

5.7 Conclusions

From the above results, the following can be concluded:

1. As is the case for Burgers' equation, neural closure models (which include the approximation $f(\mathbf{u})$ for the time-derivative) outperform direct ML models (which do not use $f(\mathbf{u})$) on the K-S equation.

Table 5.11: The minimum, average, and maximum valid prediction times of each of the eleven models, after 1000 and 5000 epochs. The same data is shown visually in Figure 5.8.

Trajectory lengths	VPT					
	1000 epochs			5000 epochs		
	Min	Avg	Max	Min	Avg	Max
1	4.42	6.00	7.96	4.62	6.10	7.92
2	5.17	6.42	7.92	5.12	6.70	8.96
4	5.04	6.55	8.33	4.62	6.63	8.42
8	4.58	6.47	8.29	4.38	6.18	7.42
15	5.04	6.35	8.38	4.88	6.95	8.54
30	4.38	6.32	8.79	4.92	7.10	9.12
60	3.62	4.97	6.83	3.62	5.01	6.25
90	3.67	5.07	7.17	4.38	5.42	6.88
100	2.79	4.18	5.46	3.54	5.05	8.71
110	3.42	4.38	5.42	3.54	5.00	6.29
120	0.25	0.28	0.33	4.12	5.33	7.38

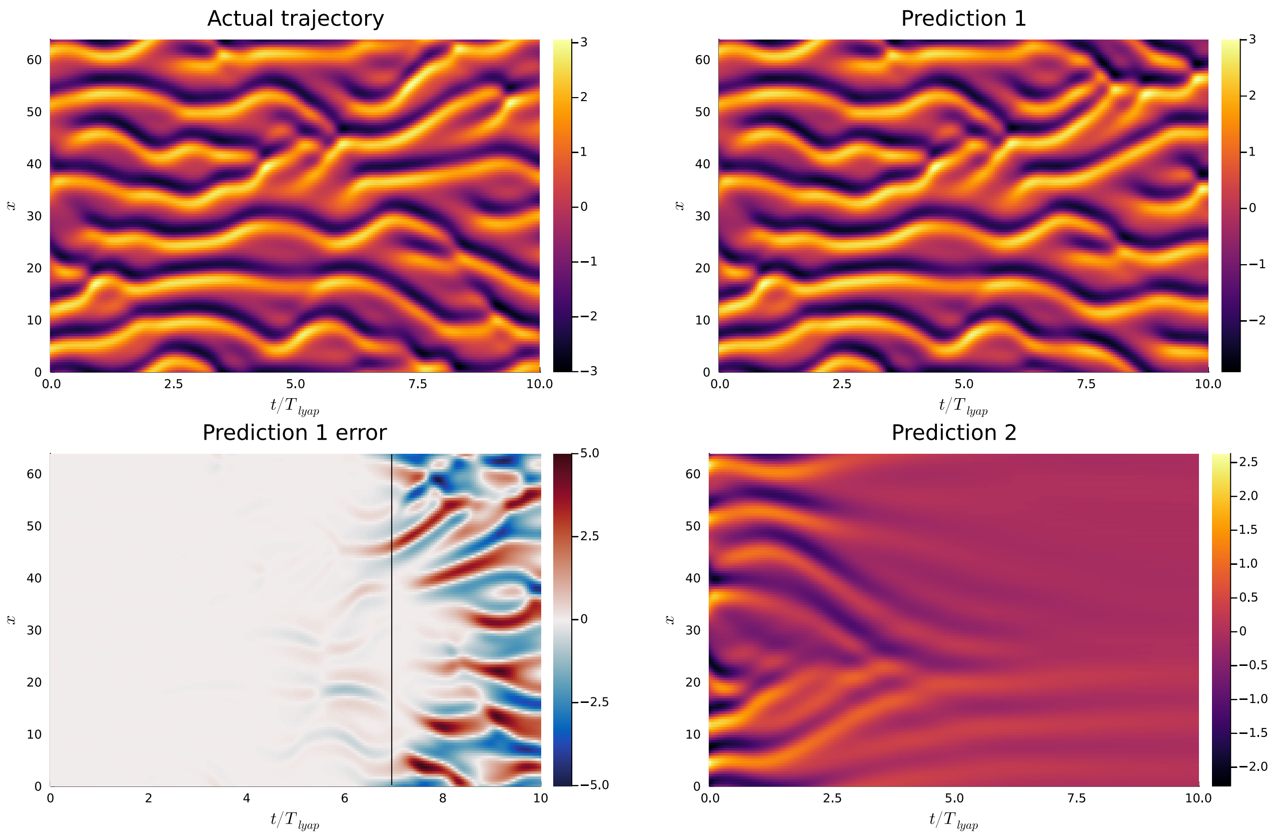

Figure 5.7: Top left: a trajectory of the K-S equation from the testing data. Top right: the prediction for this trajectory made by the closure model trained on short trajectories. Bottom left: the prediction error of the same model. The black vertical line indicates the valid prediction time for this trajectory. Bottom right: the prediction made by the direct model trained on long trajectories. Note how the trajectory is steered toward the long-term average, similar to the right plot in Figure 4.5.

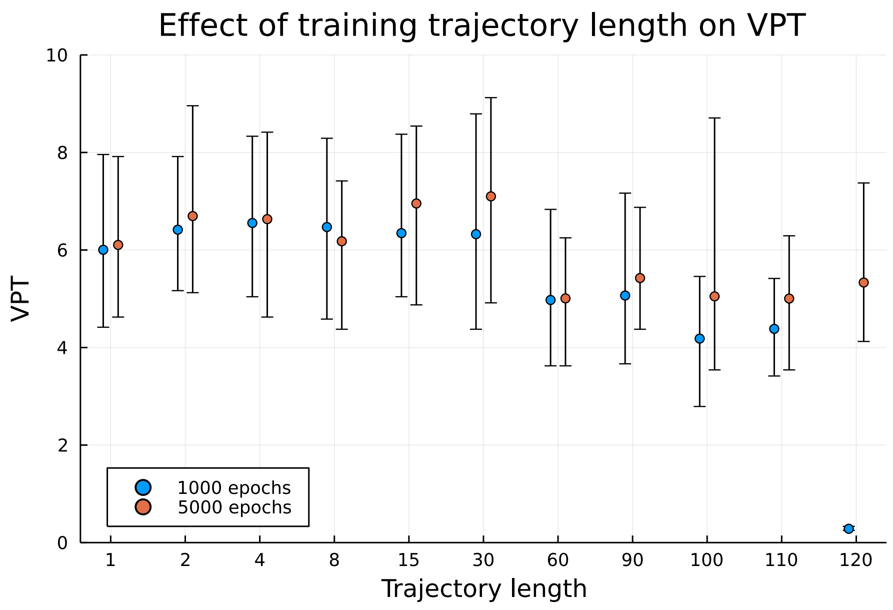

Figure 5.8: Chart showing the minimum, average, and maximum VPT across the 10 testing trajectories for models trained on the given trajectory length.

Table 5.12: The average valid prediction time over the 10 testing trajectories for various combinations of time steps and ODE solvers. The bold face entry corresponds to the combination of time step and solver that was used for training.

	ODE solver	$\Delta t = \frac{1}{8}$	$\Delta t = \frac{1}{4}$	$\Delta t = \frac{1}{2}$	$\Delta t = 1$
Without NN	ETDRK1	1.48	0.88	0.48	0.29
	ETDRK2	4.18	3.30	2.22	0.73
	ETDRK3	4.47	4.06	3.29	2.05
	ETDRK4	4.57	4.55	4.16	3.28
With NN	ETDRK1	1.59	1.04	0.52	0.15
	ETDRK2	4.89	3.66	2.41	0.50
	ETDRK3	5.23	5.58	3.50	1.10
	ETDRK4	5.08	5.20	**7.10**	1.81

2. Replacing time-derivatives by finite differences for derivative fitting can be done at the expense of only a small loss in accuracy. In the experiments done here, the model trained on finite differences instead of derivatives results in only slightly lower VPTs, although the long-term stability is more severely impacted. Nevertheless, when adopting a two-stage training approach, finite difference training can produce approximately trained models well.

3. For trajectory fitting the preferable approach is to use discretise-then-optimise strategies, rather than optimise-then-discretise. The reason for this is likely that the discretise-then-optimise approach can train models that correct for the error in the ODE solver, whereas the optimise-then-discretise approach cannot. However, discretise-then-optimise requires a differentiable solver, which may not always be easy to implement depending on the ODE.

   However, note that in the tests performed here, the spectral ODE solvers used in the discretise-then-optimise approach are also more accurate than the IMEX methods used for optimise-then-discretise, making it less clear how much of the difference in performance is due to the different training method, and how much is due to the different ODE solver. Unfortunately, this appears to be a fundamental issue that arises when comparing neural ODE training procedures. After all, making effective use of optimise-then-discretise training requires that the given testing ODE is either well-posed in reverse-time (for backsolve adjoint) or can be efficiently solved using an ODE solver that supports dense output (for interpolating/quadrature adjoint). Simultaneously, making effective use of discretise-then-optimise requires that there is an ODE solver that is differentiable, and therefore preferably explicit. While simple ODEs such as that of a non-linear pendulum, or the three-variable chaotic Lorenz system (sometimes referred to as L63, not to be confused with the Lorenz '96 model) satisfy all these properties, many real-world problems do not. Thus finding a test problem that is both non-trivial and suitable for testing a wide variety of training procedures is itself a challenge.

4. When training by trajectory fitting, training on relatively short trajectories is preferable to very long trajectories, both for optimise-then-discretise and discretise-then-optimise. For optimise-then-discretise, training on long trajectories results in less accurate models due to the exploding gradients problem. This can be mitigated by introducing a weighted error function, although the resulting training procedure still produces less accurate models than derivative fitting or discretise-then-optimise trajectory fitting due to the increased error in the gradient computation. Better results are obtained by simply training on shorter trajectories or even by derivative fitting. For discretise-then-optimise, decently accurate models can be obtained by training on long trajectories, although the convergence is much slower than when training on shorter trajectories.

Chapter 6

Extending ML models with memory

In the previous two sections, the training data and ML models were both 'Markovian', meaning that the future behaviour is is conditionally independent of past states given the present state. This assumption is visible in the resulting models, since in all cases the future behaviour, i.e. $\mathbf{u}(t + \Delta t)$ (for time-discrete models) or $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}$ (for neural ODEs) depends only on $\mathbf{u}(t)$ and not on $\mathbf{u}(t-\tau)$ for any $\tau > 0$. Such models can also be thought of as '**memoryless**', since all history of $\mathbf{u}$ is immediately forgotten.

For Burgers' equation and the Kuramoto-Sivashinsky equation, this approach worked reasonably well. However, purely memoryless models may in general not be able to predict trajectories as accurately as models with memory. This follows from the Mori-Zwanzig formalism (see Section 1.2), which shows how an ODE or PDE can be rewritten into a differential equation over a smaller number of variables, where the right-hand side depends on the entire history of the solution rather than just the current state.

In this section, different ways of including memory effects in machine learning models will be compared.

6.1 Ways to include memory effects

There are multiple ways to create a machine learning model that can take memory effects into account. Here, the term 'memory effect' is used to indicate that the behaviour of $\mathbf{u}(t)$ is not Markovian, i.e. that future states are not conditionally independent of past states given the present state. Two methods to create models with a memory effect are:

- **Models with earlier states as input.** One simple way is to create ML models in which the input consists of a number of earlier states of the solution, rather than just the most recent. In the discrete setting, this could be a model used in the following way:

$$\mathbf{u}\left(t+\Delta t\right) = \text{model}\left(\mathbf{u}\left(t\right), \mathbf{u}\left(t-\Delta t\right), \mathbf{u}\left(t-2\Delta t\right), \cdots, \mathbf{u}\left(t-N_h\Delta t\right); \vartheta\right), \tag{6.1}$$

  where N_h is a fixed integer describing how many past states are fed into the model in addition to the most recent state (i.e. taking $N_h = 0$ yields a memoryless model again). Models of this form are investigated by Fu et al. [22] and Pawar et al. [63], who find that such models can indeed create more accurate predictions than memoryless models for partially observed ODEs. They also conclude that, as expected, models with a longer memory (larger N_h) perform better than models with less memory.

- **Models with latent space.** A second approach to modelling memory effects is by augmenting the state vector $\mathbf{u}$ with a number of variables that are not compared to training data. Such variables are often called **latent variables**, and the vector space that contains them is called a **latent space**. Examples of discrete models with a latent space include Recurrent Neural Networks (RNNs) and Long-Short Term Memory networks (LSTMs [29]). Both of these models can generally be written as

$$\left(\mathbf{o}^{(t)}, \mathbf{h}^{(t)}\right) = \text{model}\left(\mathbf{i}^{(t)}, \mathbf{h}^{(t-1)}; \vartheta\right),$$

where $\mathbf{i}^{(1,2,\cdots)}$ is a sequence of inputs, $\mathbf{o}^{(1,2,\cdots)}$ is the sequence of outputs, and $\mathbf{h}^{(0,1,2,\cdots)}$ is a vector of latent variables. The memory effect in such models results from the fact that the output $\mathbf{o}^{(t)}$ is not a pure function of the corresponding input $\mathbf{i}^{(t)}$, but also depends on $\mathbf{h}^{(t-1)}$. The latent vector $\mathbf{h}^{(t-1)}$ depends on $\mathbf{i}^{(t-1)}$ and $\mathbf{h}^{(t-2)}$, which in turn depends on $\mathbf{i}^{(t-2)}$, and so on. As a result, $\mathbf{h}^{(t)}$ depends on the entire history $\mathbf{i}^{(1,2,\cdots,t)}$, thus giving rise to a memory effect. In the types of time series prediction considered here, there is no sequence of inputs, resulting in the following, slightly different formulation:

$$\left(\mathbf{u}\left(t+\Delta t\right), \mathbf{h}\left(t+\Delta t\right)\right) = \text{model}\left(\mathbf{u}(t), \mathbf{h}(t); \vartheta\right). \tag{6.2}$$

When training a model of this type, training data is available for $\mathbf{u}(t)$ but not for $\mathbf{h}(t)$. Note that as described in Section 2.4, including the 'visible' state $\mathbf{u}$ in the input and output of the model is not necessary: an alternative is to use a 'fully latent' model that iterates in the latent space $\mathbf{h}$ only, and uses other fixed or learned mappings between the latent vector and the visible state. However, such models generally do not allow for the inclusion of prior knowledge since nothing is known a priori about the behaviour of the latent vector $\mathbf{h}$. As such, fully latent models will not be considered here.

Both types of memory model listed above are discrete in time. However, both have continuous-time variants as well:

- A time-continuous model with history results in a **Neural Delay Differential Equation (DDE) [85]:**

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u} = \text{model}\left(\mathbf{u}(t), \mathbf{u}(t-\Delta t), \cdots, \mathbf{u}(t-N_h\Delta t); \vartheta\right). \tag{6.3}$$

- A time-continuous model with a latent space results in an **Augmented Neural ODE (ANODE) [16]:**

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{u}\\ \mathbf{h}\end{pmatrix} = \text{model}\left(\mathbf{u}, \mathbf{h}; \vartheta\right). \tag{6.4}$$

In all of the above model descriptions, the memory effect is a result of the inputs and outputs of the machine learning model, meaning that the function $\text{model}\left(\cdots; \vartheta\right)$ itself is still a pure function. As such, all these models can also be seen as applying a memoryless model to an augmented state, either $\mathbf{X}(t) := \begin{bmatrix}\mathbf{u}(t) & \mathbf{u}(t-\Delta t) & \cdots & \mathbf{u}(t-N_h\Delta t)\end{bmatrix}$ for history models, or $\mathbf{Y}(t) := \begin{bmatrix}\mathbf{u}(t) & \mathbf{h}(t)\end{bmatrix}$ for latent space models. In this chapter, the focus will be on time-discrete delay models (6.1) and time-continuous ANODE models (6.4). Figure 6.1 shows both model architectures. The specific features of these two models will be discussed in Sections 6.2 and 6.3. RNNs and LSTMs are not considered here since time-continuous models were already found to be preferable over discrete models, and neural DDEs will not be considered due to the increased complexity in their training procedure (see Section 6.2).

6.2 Discrete delay models

The first type of model that will be considered is the discrete delay model as in equation (6.1). Models of this form are relatively easy to train, since they can be seen as iterating a memoryless model over the state $\begin{bmatrix}\mathbf{u}(t) & \mathbf{u}(t-\Delta t) & \cdots & \mathbf{u}(t-N_h\Delta t)\end{bmatrix}$.

Note that as was the case in Chapter 4, including prior knowledge from an ODE or PDE into a discrete model must be done by discretising the underlying differential equation. In Chapter 4, this

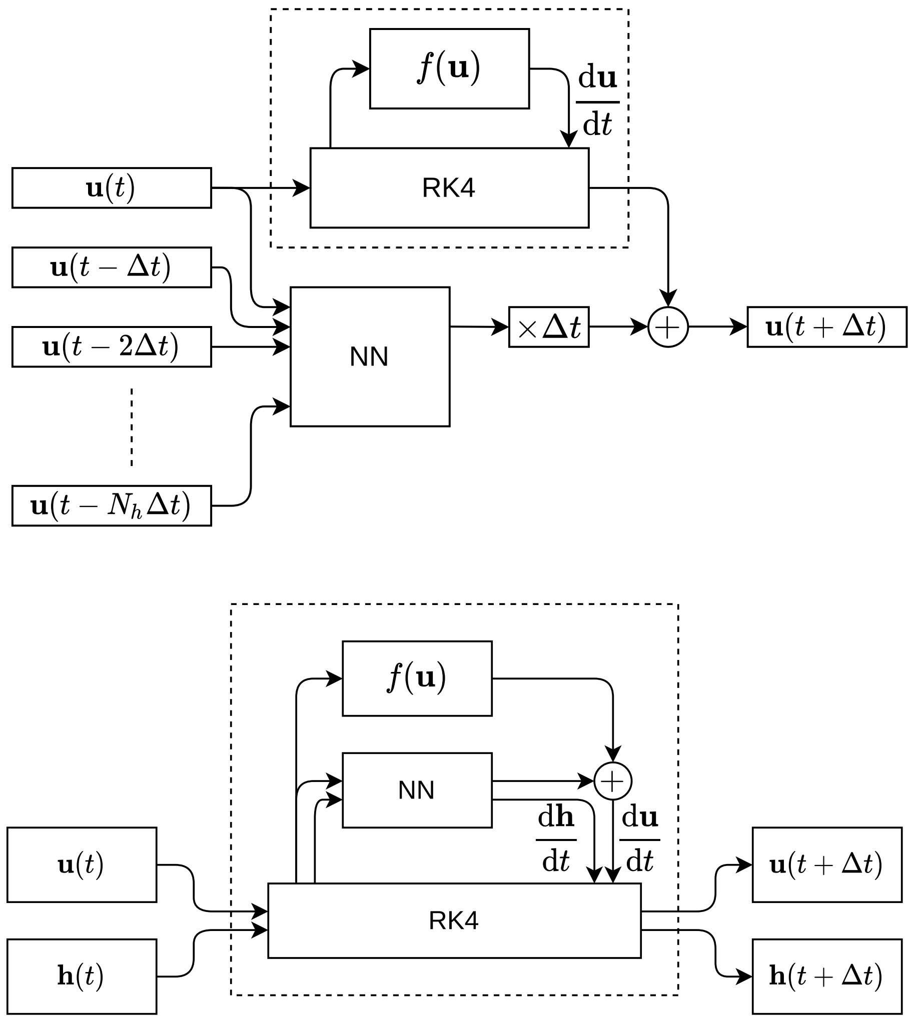

Figure 6.1: Schematic representations of the discrete delay model (top) and ANODE model (bottom) architectures that will be used in this Chapter.

was done by taking a forward Euler step using the approximate time-derivative $f(\mathbf{u})$, and adding a neural closure term.

$$\mathbf{u}(t+\Delta t) = \mathbf{u}(t) + \Delta t\left[f(\mathbf{u}(t)) + \mathrm{NN}(\mathbf{u}(t), \mathbf{u}(t-\Delta t), \cdots, \mathbf{u}(t-N_h\Delta t); \vartheta)\right].$$

As was the case for Burgers' equation, this model can be ineffective in some cases due to the fact that forward Euler may be unstable when using a large time step. A solution is to use a higher-order ODE solver to perform the 'prediction' term, while still using a forward Euler form for the neural network 'correction' term:

$$\mathbf{u}(t+\Delta t) = \texttt{stepRK4}(\mathbf{u}(t), f, \Delta t) + \Delta t \mathrm{NN}(\mathbf{u}(t), \mathbf{u}(t-\Delta t), \cdots, \mathbf{u}(t-N_h\Delta t); \vartheta).$$

While this form resembles a neural ODE to a greater extent than the forward Euler model, it is still fundamentally different. Importantly, the neural network term is still only used in a forward Euler way, and is therefore not included in the computations of the ODE solver. Including the NN term inside the Runge-Kutta 4 update would indeed result in a model that resembles a neural DDE. This is not done here since creating such a DDE model is significantly more complicated. Specifically, almost all ODE solvers with a reasonably high order of accuracy require evaluating the right-hand side function of the ODE at time points in between t and $t+\Delta t$. The Runge-Kutta 4 solver used here evaluates the RHS of the ODE on two different approximations for $\mathbf{u}(t+\frac{1}{2}\Delta t)$. In the neural network term, these evaluations would also require the inputs $\mathbf{u}(t-\frac{1}{2}\Delta t), \mathbf{u}(t-\frac{3}{2}\Delta t)$, and so on, which must therefore be approximated from the neighbouring snapshots of $\mathbf{u}(\cdot)$.

6.3 ANODE models

The second kind of model that will be tested in this chapter is the ANODE model as in equation (6.4). Compared to discrete delay models, ANODE models may be able to obtain better accuracy due to the fact that the entire system is written as an ODE, similar to the original dynamics which are also continuous in time. Furthermore, in these models the vector $\mathbf{h}(t)$ can be seen as a latent space representation of the unresolved component of the dynamics, meaning that ANODE models are a more 'direct' translation of the underlying problem into an ML model. Since closure models are of particular interest, some prior approximation $f(\mathbf{u})$ for the dynamics can be included in the model:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{u}\\ \mathbf{h}\end{pmatrix} = \begin{pmatrix}f(\mathbf{u})\\ \mathbf{0}\end{pmatrix} + \mathrm{NN}\left(\mathbf{u}, \mathbf{h}; \vartheta\right). \tag{6.5}$$

The use of a latent vector instead of a finite history means that ANODE models, like RNNs and LSTMs, can exhibit memory over arbitrarily long time spans, whereas the memory of delay models is limited to the number of previous states included in the model input. However, this also complicates the training procedure since $\mathbf{h}(0)$, the initial value of the latent vector, must be chosen in some way and this choice affects the predictions made by the model. This is done using **teacher forcing**, although this time teacher forcing is used in the same way as described by Williams and Zipser [84], whereas in Chapter 4 it was formulated differently. The teacher forcing is used as follows: first, the ANODE model $\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{u}\\ \mathbf{h}\end{pmatrix} = \mathrm{model}(\mathbf{u}, \mathbf{h})$ is discretised in time using an appropriate ODE solver, which is here again chosen as RK4:

$$\begin{pmatrix}\mathbf{u}(t+\Delta t)\\ \mathbf{h}(t+\Delta t)\end{pmatrix} = \texttt{stepRK4}\left(\begin{pmatrix}\mathbf{u}(t)\\ \mathbf{h}(t)\end{pmatrix}, \mathrm{model}, \Delta t\right) =: \mathrm{discretemodel}(\mathbf{u}(t), \mathbf{h}(t)).$$

Then, each time the model is used to predict a time series based on a finite history, the vector $\mathbf{h}(t)$ is initialised by running the model on that history, but substituting the reference snapshots $\mathbf{u}_{\mathrm{ref}}(t)$ for the vector $\mathbf{u}(t)$ instead of using those predicted by the model. This procedure is described in Algorithm 6. The notation $(\sim, \mathbf{h}(i\Delta t)) = \ldots$ is used to mean that the first output of the discrete model, which is the prediction for $\mathbf{u}(i\Delta t)$, is not used in further computations but is simply discarded. Instead, the exact state $\mathbf{u}(i\Delta t)$ is used in the next iteration of the loop.

Algorithm 6 Gradient computation with the forward method

```
1: procedure INITIAL_LATENT_STATE(discretemodel, $(\mathbf{u}(i\Delta t), i = 0, 1, \ldots, N_t)$)
2:     $\mathbf{h}(0) = \mathbf{0}$
3:     for $i$ from 1 to $N_t$ do
4:         $(\sim, \mathbf{h}(i\Delta t)) = \text{discretemodel}\,(\mathbf{u}((i-1)\Delta t), \mathbf{h}((i-1)\Delta t))$.
5:     return $\mathbf{h}(t)$
```

While this algorithm can be used to initialise **h**, it is not obvious for how many steps this teacher forcing should be done in order to ensure that **h** is 'correctly' initialised. Note that in the limit as N_t, the number of teacher forcing steps, approaches infinity, it is expected that the sensitivity with respect to the initial condition, here $\mathbf{h}(0) = \mathbf{0}$, goes to zero. As such, one way to initialise is to perform Algorithm 6 with multiple different (possibly random) choices for $\mathbf{h}(0)$ in parallel, and iterating the teacher forcing loop until the variance between the different results for $\mathbf{h}(t)$ drops below some small threshold. However, this is impractical since it requires multiple parallel teacher forcing procedures, and since the number of teacher forcing steps performed is variable and could be arbitrarily large.

Instead, a novel technique is used in which the ANODE model (6.5) is regularised by adding a decay rate to the latent state **h**:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{u}\\ \mathbf{h}\end{pmatrix} = \begin{pmatrix}f(\mathbf{u})\\ -\lambda\mathbf{h}\end{pmatrix} + \mathrm{NN}(\mathbf{u}, \mathbf{h}; \vartheta), \tag{6.6}$$

where λ is a positive constant. This decay rate has the effect that the sensitivity of $\mathbf{h}(t)$ with respect to $\mathbf{h}(t-\tau)$ decreases exponentially as $\exp(-\lambda\tau)$. As a result, the number of teacher forcing steps N_t can be chosen in advance so that $\tau = N_t\Delta t$ is sufficiently small.

While for discrete delay models the 'amount' of memory included can be varied by changing N_h, the number of past states given to the model, for ANODE models this is done by changing the size of the latent vector **h**.

6.4 The Lorenz '96 model

To test the efficacy of different models with memory effects, it is important to choose training data that requires such a memory effect to yield good predictions. As was seen from previous experiments, neither Burgers' equation nor the Kuramoto-Sivashinsky equation require a memory effect to be modelled accurately, meaning that a different equation is required. Here, the chosen equation is the **Lorenz '96 (L96) model**:

$$\frac{\mathrm{d}x_k}{\mathrm{d}t} = x_{k-1}\left(x_{k+1} - x_{k-2}\right) - x_k + f + b_k, \tag{6.7a}$$

$$\frac{\mathrm{d}y_{j,k}}{\mathrm{d}t} = \frac{1}{\varepsilon}\left[y_{j+1,k}\left(y_{j-1,k} - y_{j+2,k}\right) - y_{j,k} + h_y x_k\right], \tag{6.7b}$$

$$\text{where } b_k := \frac{h_x}{J}\sum_{j=1}^{J} y_{j,k}. \tag{6.7c}$$

The variables x_k and $y_{j,k}$ are periodic:

$$x_k = x_{k+K}, \quad y_{j,k} = y_{j,k+K}, \quad y_{j+J,k} = y_{j,k+1}. \tag{6.7d}$$

Essentially, $x_{1\cdots K}$ and $y_{1\cdots J,1\cdots K}$ are coarse and fine discretisations of the same periodic domain. Machine learning models will be trained to predict trajectories consisting only of the coarse variables x_k, meaning that the term b_k, which depends on the fine variables $y_{j,k}$, will be approximated by a neural closure term.

The parameters of the model are:

- $f \geq 0$ is a **forcing term** that applies to all elements x_k.

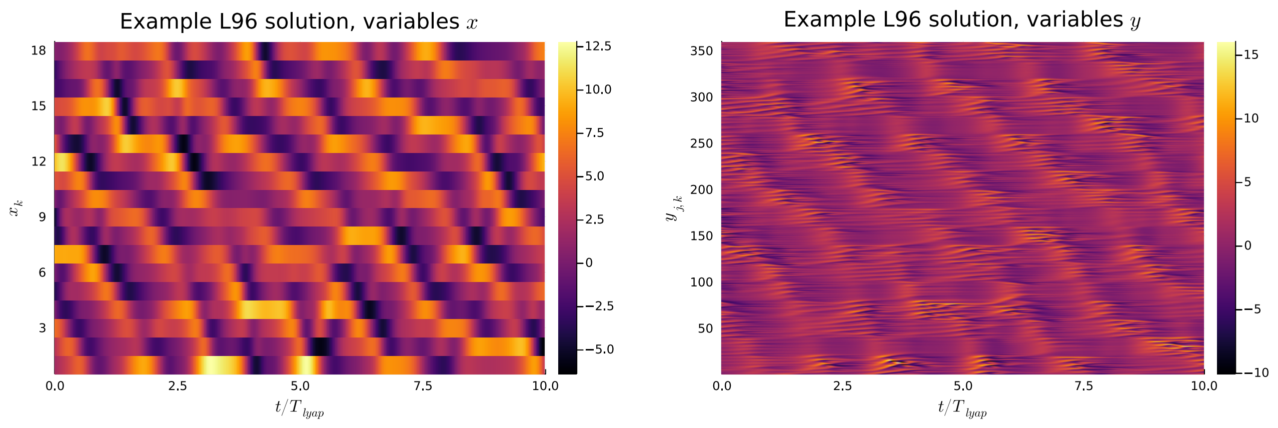

Figure 6.2: An example solution of the Lorenz '96 model with the parameter settings as in (6.8).

- ε is the **separation of time scales**: note that the characteristic time scale of $y_{j,k}$ is approximately ε.
- h_x and h_y are the effects of x_k and $y_{j,k}$ on each other.

For the experiments done here, the parameter settings are the same as those taken by Crommelin and Vanden-Eijnden [14]:

$$(K, J, f, \varepsilon, h_x, h_y) = \left(18, 20, 10, \frac{1}{2}, -1, 1\right). \tag{6.8}$$

An example solution of the Lorenz '96 model with these parameter settings is shown in Figure 6.2. Note that while ε represents the amount of separation of time scales, taking $\varepsilon \ll 1$ is actually not necessarily the best choice for these experiments: for such extreme time scale separation, other efficient algorithms for computing solutions to the model are available, see for example the work by Fatkullin et al. [20].

Similar to the K-S equation, the L96 equation with these parameters is chaotic meaning models will be evaluated in terms of their Valid Prediction Time (equation (5.4)). With the chosen parameter settings, the Lyapunov exponent is empirically estimated as $\lambda_{\max} \approx 1.25$ and therefore $T_{\text{lyap}} \approx 0.8$, which is approximately in accordance with the results of Karimi et al. [36].

6.5 Data generation

Using the parameter settings from equation (6.8), 100 trajectories are generated. Each trajectory is computed by solving the ODE system (6.7) with the ODE solver `Vern7`, a 7th-order 10-stage explicit Runge-Kutta method by Verner [78], with a time step $\Delta t = 0.01$ for $t = 0$ to $t = 25.6$. For each trajectory, the variables x_k and $y_{j,k}$ are initialised randomly according to independent unit Gaussian distributions (i.e. with zero mean and unit variance). The training data is obtained by discarding the variables $y_{j,k}$ and down-sampling the ODE solution by a factor 4, i.e. taking snapshots every $\Delta t = 0.04$. To ensure that the initial transients due to the random initialisation of the variables don't affect the training, the training data starts at $t = 1.6$.

The reason for the higher-order ODE solver than in previous methods is that since the Lorenz '96 model is an ODE system rather than a PDE, the accuracy is only limited by the ODE solver and time step. Thus it is worthwhile to use a high-order accurate method and small time step to obtain near-exact solutions, whereas for PDEs this would require a prohibitively fine spatial discretisation. Indeed, the obtained trajectories are essentially exact: solving the same ODE system with the same initial conditions using both `Vern7` and `RK4` shows that the two methods yield approximately the same solution for 20 Lyapunov times, showing that the error in the generation of the training data is negligible. However, neural ODEs for this problem will use the lower-order `RK4` method, which is computationally more efficient due to the lower number of stages while still being accurate enough that the prediction errors of ML models is due to the inexactness of the ML model itself rather than the ODE solver.

6.6 Experiments

In order to test the efficacy of the described model architectures, a total of nine models are trained including four discrete delay models, four ANODE models, and one memoryless closure model (NODE) used as a baseline. All models use convolutional neural networks internally, due to the translation invariance and periodicity of the L96 ODE system. All CNNs use convolution kernels of width 5.

The NODE model is integrated using Runge-Kutta 4 with fixed time step $\Delta t = 0.04$ and trained by back-propagating through the ODE solver. The discrete delay models are chosen with $N_h \in \{1, 3, 7, 15\}$, i.e. with $2, 4, 8, 16$ input states. The input states are treated as separate 'channels' in the CNN, meaning the neural network layers are convolutional over the space coordinate but not over the time coordinate. The four ANODE models are created with the same values of N_h. The layer structure of the resulting neural networks is summarised in Table 6.1. The memory decay parameter λ in the ANODE models is chosen as 3.0, and the initial latent vector is computed using Algorithm 6 with a warmup period of $N_t = 40$ states. This way, the sensitivity with respect to $\mathbf{h}(0)$ is expected to be approximately $\exp(-3.0 \times 0.04 \times 40) \approx 0.0082$, which should be small enough to lead to noticeably improved accuracy with respect to the memoryless baseline model.

To train the models, 80 of the 100 generated trajectories are split up into chunks of 40 subsequent snapshots, similar to Figure 5.6. Now, however, the models are trained on pairs of subsequent chunks, where the first pair is used for initialising the model and the second chunk is compared with the model prediction. For example, to train an ANODE model it is first initialised on snapshots $1 - 40$ of some trajectory, then used to predict 40 additional snapshots which are then compared to reference snapshots $41 - 80$ of that same trajectory. The same is done with snapshots $41 - 80$ for warmup and $81 - 120$ for predicting, and so on. A similar process is done for the testing data, although there the 'prediction' chunks are longer so that the valid prediction time of the trained models can be measured. For discrete delay models, the training is done in the same way except of course that of the 'warmup' data, only the last $N_h + 1$ snapshots are used and these directly form the input for the first model prediction. Other parameters of the training procedure are shown in Table 6.2. Note that the training is done for a smaller number of epochs, but this training duration is found to be sufficient.

6.7 Results

During training, all models are found to overfit relatively quickly, meaning that the models continue to improve their accuracy on the training data, but without improving on testing data. Figure 6.3 shows this phenomenon for the second ANODE model.

Table 6.3 shows the valid prediction times (expressed in Lyapunov times) of the four discrete delay models, the four ANODE models, and the two reference models. The results are perhaps unexpected, since:

- the inclusion of memory effects has a positive but small on the VPTs;
- inclusion of more memory, i.e. including more past states as input or using a bigger latent space, does not significantly improve the VPTs;
- unlike tests on previous equations, no models are able to achieve consistent prediction performance, i.e. the minimum VPT is much lower than was the case for models on the K-S equation.

Figure 6.4 shows the RMSE over all testing trajectories as a function of time. From these figures, it can be seen that for all models, the prediction error immediately increases, with a similar rate of increase near $t = 0$ for all models. This means that the models with memory are not able to predict the closure term b_k more accurately than the memoryless model.

Several possible reasons exist why these models do not perform as expected.

Table 6.1: The layer structure of the CNNs used in the models trained in Chapter 6.

Model type	N_h	Convolution channels	Total params
ODE without closure model	-	-	0
Memoryless closure model	0	$1 \to 2 \to 4 \to 8 \to 8 \to 4 \to 2 \to 1$	769
Discrete delay	1	$2 \to 4 \to 8 \to 8 \to 4 \to 2 \to 1$	757
	3	$4 \to 16 \to 12 \to 8 \to 4 \to 2 \to 1$	2013
	7	$8 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$	4753
	15	$16 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$	18897
ANODE	1	$2 \to 4 \to 8 \to 8 \to 4 \to 2$	746
	3	$4 \to 16 \to 16 \to 12 \to 8 \to 4$	3256
	7	$8 \to 32 \to 32 \to 16 \to 12 \to 8$	10500
	15	$16 \to 64 \to 64 \to 32 \to 24 \to 16$	41800

Table 6.2: The additional parameters of the training procedure from Chapter 6.

Property	Value
Training data	Trajectories $1 - 80$, snapshots $1 - 600$, split into smaller trajectories
Validation data	Trajectories $81 - 90$, snapshots $1 - 80$
Optimiser	ADAM, learning rate $= 10^{-3}$
Batch size	16
Epochs	500
Patience	100 epochs
Penalty term	None
Testing data	Trajectories $91 - 100$, snapshots $1 - 600$, split into smaller trajectories

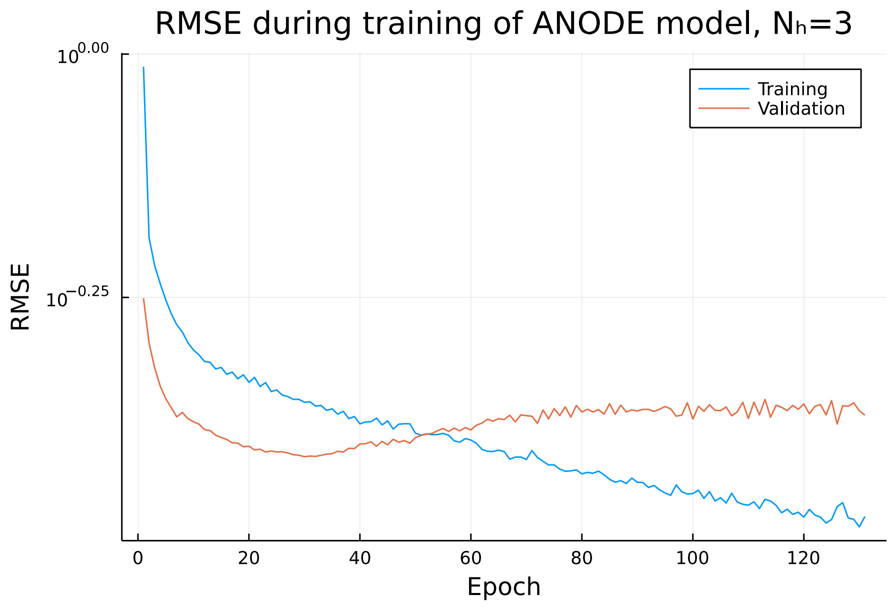

Figure 6.3: The loss curve of the second ANODE model. Note how as training progresses, the error on the training data continues to decrease whereas the error on validation data does not.

Table 6.3: The valid prediction time of all trained models on the testing data.

Model type	N_h	VPT		
		Min	Avg	Max
ODE without closure model	-	0.85	1.31	2.85
Memoryless closure model	-	1.10	3.20	10.70
Discrete delay	1	1.20	3.43	10.60
	3	1.45	3.27	8.85
	7	1.30	3.59	10.00
	15	1.15	3.42	9.15
ANODE	1	1.25	3.45	8.60
	3	1.45	3.72	9.50
	7	1.45	3.63	9.50
	15	1.15	3.54	9.10

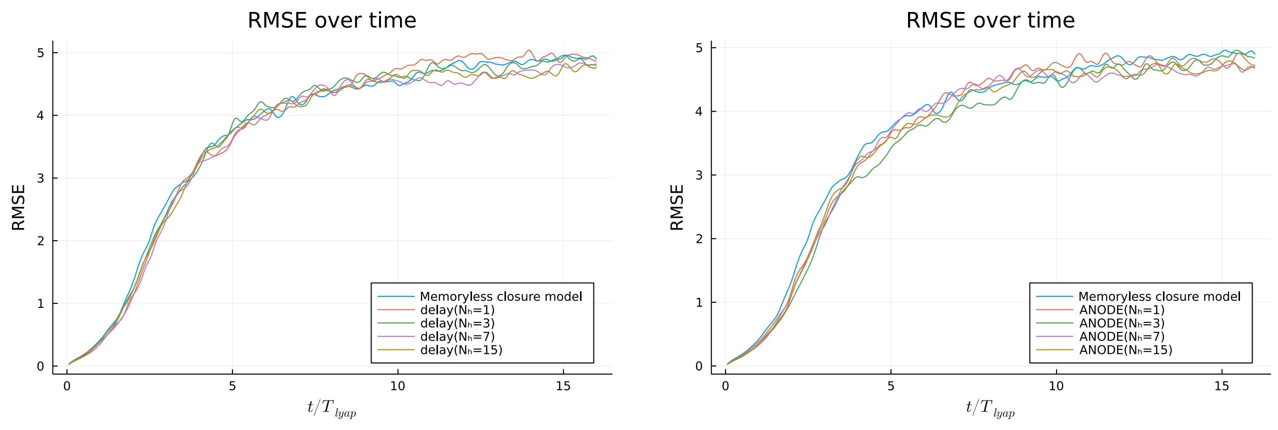

Figure 6.4: The Root-Mean-Square Error (RMSE) over time of the discrete delay models (left) and ANODE models (right). Both plots show the RMSE over time of the memoryless closure model for reference.

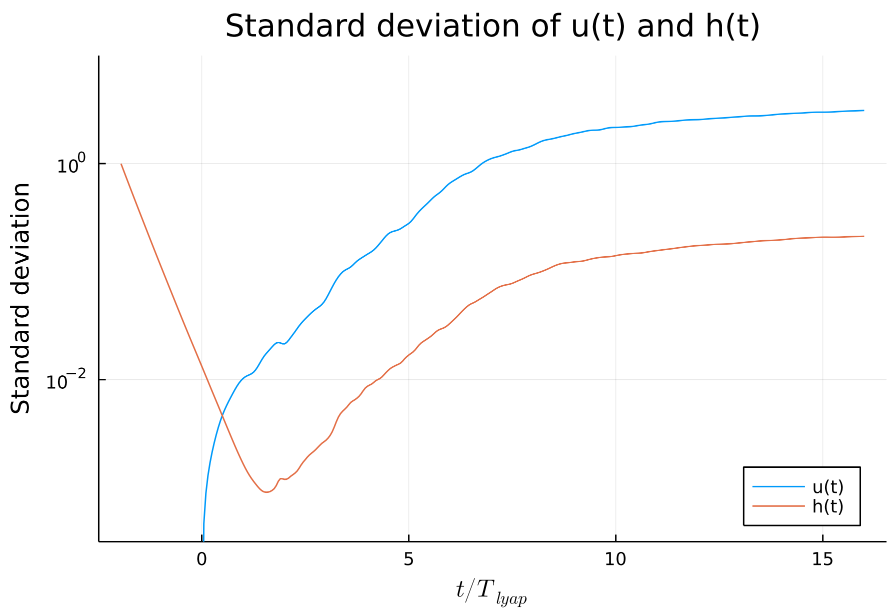

Figure 6.5: The standard deviation in $\mathbf{u}(t)$ and $\mathbf{h}(t)$ as a result of different choices for $\mathbf{h}(0)$ for the second ANODE model.

Firstly, a common reason for poor model performance is overfitting. However, this was avoided by using the early stopping algorithm. Note that in the early stopping algorithm, the final model parameters are not those after the last epoch, but those with the best accuracy on the validation set. For example, in Figure 6.3 the best accuracy on the validation set was achieved after 30 epochs. After not achieving better performance for another 100 epochs, the training was stopped having trained for 130 epochs in total. The model was then tested with parameters after 30, not 130, epochs of training. This way the training procedure is stopped when overfitting begins to appear and indeed, none of the models were found to perform significantly better on the training data meaning that the final models do not suffer from overfitting.

Secondly, for ANODE models it was assumed that the warmup period is long enough to ensure that the initial value for $\mathbf{h}(0)$ does not significantly affect the model predictions. To check that this is the case, the second ANODE model is used on the testing trajectories (Algorithm 6 followed by solving (6.6)), this time with $\mathbf{h}(0)$ not initialised to zero, but randomly according to a standard Gaussian distribution. This is done $N_s = 100$ times, resulting in 100 different predicted trajectories $\mathbf{u}^{(i)}(t), \mathbf{h}^{(i)}(t)$. Based on these 100 samples, the 'mean trajectories' $\bar{\mathbf{u}}(t)$ and $\bar{\mathbf{h}}(t)$ are computed, followed by the standard deviations of the trajectories:

$$\bar{\mathbf{u}}(t) = \frac{1}{N_s} \sum_{i=1}^{N_s} \mathbf{u}^{(i)}(t),$$

$$\text{stddev } \mathbf{u}(t) = \sqrt{\frac{1}{N_s N_x} \sum_{i=1}^{N_s} \left\| \mathbf{u}^{(i)}(t) - \bar{\mathbf{u}}(t) \right\|_2^2},$$

and similarly for $\mathbf{h}$. Figure 6.5 shows the standard deviation of these predictions for the second ANODE model. As can be seen from the figure, during the warmup period (indicated as $t/T_{lyap} < 0$ in the figure), the variance in $\mathbf{h}(t)$ decreases exponentially in t. Then, during prediction (positive t/T_{lyap}), the variance in $\mathbf{h}(t)$ actually continues to decrease for a short amount of time, but the differences in $\mathbf{h}(t)$ across different runs cause differences in the resulting predicted trajectories $\mathbf{u}(t)$. Since the Lorenz '96 model is chaotic, it is expected that different initialisations of $\mathbf{h}(0)$ cause different predicted trajectories. However, the standard deviation in $\mathbf{u}(t)$ actually stays much smaller than the error between $\mathbf{u}$ and $\mathbf{u}_{\text{ref}}(t)$. For example, after 5 Lyapunov times, the standard deviation in $\mathbf{u}(t)$ is roughly 0.30, but the prediction error at that point (see Figure 6.4) is more than $10\times$ larger. This shows that while the initialisation of $\mathbf{h}(0)$ does have an effect on the resulting prediction, this effect

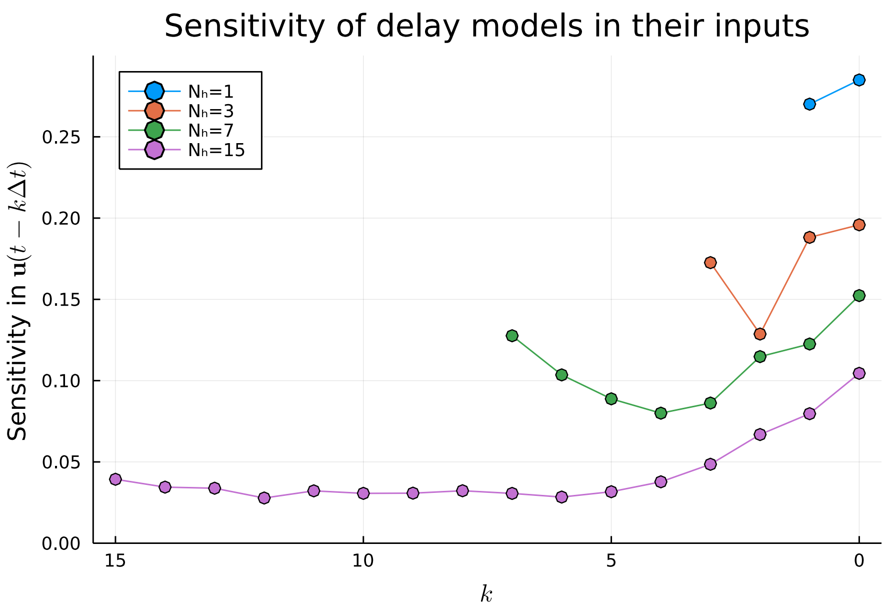

Figure 6.6: Average sensitivity of the four discrete delay models with respect to their different inputs.

is not significant enough to explain the prediction error of the model, meaning that eliminating or mitigating this source of error (by extending the warmup period) will not result in significantly more accurate predictions.

Thirdly, it is possible that the models 'learn' to behave as memoryless models, essentially ignoring the earlier states $\mathbf{u}(t-k\Delta t)$ or latent vector $\mathbf{h}(t)$ for delay models and ANODE models, respectively. This can be checked by evaluating the sensitivity of the models' outputs with respect to their inputs. For delay models, this is done by taking N_h+1 subsequent states $\mathbf{u}(t-k\Delta t), k=0,1,\ldots,N_h$ from the predicted trajectories, perturbing one of the states $\mathbf{u}(t-k\Delta t)$ by a small amount, and measuring the difference in the resulting output $\mathbf{u}(t+\Delta t)$. This process is described in pseudocode in Algorithm 7. Analogously, the sensitivity of ANODE models can be estimated by perturbing either $\mathbf{u}(t)$ or $\mathbf{h}(t)$, and measuring the difference in model output. Importantly, for ANODE models only the sensitivity of the $\mathbf{u}$-component of the model's output is measured with respect to the model's inputs. After all, if changing the value of $\mathbf{h}(t)$ changes the model's output for $\mathbf{h}(t+\Delta t)$ but not for $\mathbf{u}(t+\Delta t)$, then the model still produces effectively memoryless behaviour in $\mathbf{u}(t)$.

Algorithm 7 Sensitivity analysis for discrete delay models

```
1: procedure DISCRETE_DELAY_SENSITIVITY(model, $[\mathbf{u}^{(i)}(t)\ldots]$)
2:     for 10000 samples do
3:         $\mathbf{X}_{\text{ref}} \leftarrow [\mathbf{u}(t), \mathbf{u}(t-\Delta t), \ldots, \mathbf{u}(t-N_h\Delta t)]$, randomly sampled from given data
4:         Compute $\mathbf{Y}_{\text{ref}} = \text{model}(\mathbf{X}_{\text{ref}}; \vartheta)$
5:         for each $k = 0, 1, \ldots, N_h$ do
6:             $\tilde{\mathbf{X}}^{(k)} \leftarrow \mathbf{X}_{\text{ref}}+$ a small perturbation to $\mathbf{u}(t-k\Delta t)$
7:             $\tilde{\mathbf{Y}}^{(k)} \leftarrow \text{model}(\tilde{\mathbf{X}}^{(k)}; \vartheta)$
8:             Compute sensitivity $s_k = \frac{\text{rms}(\tilde{\mathbf{Y}}^{(k)}-\mathbf{Y}_{\text{ref}})}{\text{rms}(\tilde{\mathbf{X}}^{(k)}-\mathbf{X}_{\text{ref}})}$
9:     return the averages of $s_k$ for each $k$
```

The sensitivities of the discrete models are shown in Figure 6.6, and the sensitivities of the ANODE models are given in Table 6.4. As can be seen from these data, the discrete delay models are sensitive to changes in the input other than the most recent state $\mathbf{u}(t)$, meaning that the models do not learn to behave as memoryless models. The ANODE models, however, do appear to behave as memoryless models: the ANODE models' output for $\mathbf{u}(t+\Delta t)$ depends heavily on $\mathbf{u}(t)$ but is almost completely independent of $\mathbf{h}(t)$. As such, these models do indeed behave almost as memoryless models, which may explain their poor accuracy. However, it is not clear what causes this behaviour or how it can be

Table 6.4: Average sensitivity of the four trained ANODE models with respect to inputs $\mathbf{u}$ and $\mathbf{h}$.

N_h	Sensitivity in $\mathbf{u}$	Sensitivity in $\mathbf{h}$
1	0.9792	0.0239
3	0.9789	0.0158
7	0.9796	0.0087
15	0.9747	0.0046

avoided. One possibility is that the warmup strategy for $\mathbf{h}$ (Algorithm 6) is not sufficiently accurate, causing the model to ignore the latent space in making predictions. The accuracy of the warmup strategy can be improved by increasing the warmup period N_t, the memory decay rate λ, or both. However, this explanation is not entirely satisfactory: as shown in Figure 6.5, the standard deviation in $\mathbf{h}(t)$ at the end of the warmup period is small enough that one would expect the model to at least be capable of making better short-term predictions. Even this is not the case, though, as indicated by the short-term error in Figure 6.4, which is similar to the short-term error of other methods.

6.8 Conclusions

From the above results, it appears that training models with memory is not as straightforward as training memoryless models. In particular, as seen in the loss curve of Figure 6.3, even small models are prone to overfitting when memory effects are included. For reference, the ANODE model whose loss curve is shown in that figure has 3256 parameters in total, while the training data consists of 865440 numbers in total. This is particularly relevant for the work by Fu et al. [22], who keep the number of training data between 5 and 10 times the number of model parameters, arguing that this is sufficient to avoid overfitting. Wiewel et al. [82] use a similar argument. Both papers evaluate models only on training data, since they assume that keeping the models small is sufficient to avoid models that overfit. However, this assumption does not appear to hold for models with memory effects.

Thus, in order to train models that learn memory effects while also generalising to unseen data, more advanced training techniques may be necessary. For example, the training in this chapter was done with early stopping, and while this successfully avoided overfitting it also yielded poor accuracy on the testing data set. Furthermore, since discrete delay models and ANODE models perform about equally well no, recommendations can be made about which model type is more suitable for including memory effects. Thus, a different strategy to avoid overfitting may be required to obtain better results, such as dropout regularisation or input perturbation.

Another possibility is to use more advanced models to avoid overfitting, rather than the relatively straightforward CNNs used here. For example, discrete delay models use neural networks that are convolutional in space, but dense in time. It may be useful to instead consider models that are convolutional in both space and time. For ANODE models it may also be useful to consider models that are convolutional in the visible state $\mathbf{u}$ but dense in $\mathbf{h}$. With such techniques being used, it may be possible to effectively train models with more memory, i.e. larger values of N_h, without overfitting becoming a problem. LSTMs may also be a more suitable model type for these problems. However, including prior knowledge into an LSTM model is not as trivial as it is for ANODE models or discrete delay models due to the relatively complex internal structure of an LSTM cell. The same holds for fully latent models. Fully latent models may nevertheless work well on the L96 system as found by Chattopadhyay et al. [8], who test three different models on a similar ODE system as (6.7), although with a third 'layer' of variables $z_{i,j,k}$. There, the models tested include a reservoir computing model, an LSTM model, and a memoryless NN model. Of these three models, the reservoir computing model yields the best performance while having the smallest number of trainable parameters.

Finally, it is useful to note that the Lorenz '96 model is also a relatively difficult case for machine learning models to learn. Particularly, the choice $\varepsilon = 0.5$ resulting in only a weak separation of time scales may be an important cause of the poor model performance. For reference, Fu et al. [22] only consider simple linear ODE systems and a small non-linear ODE system with separation of time scales:

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}t}x_1 &= -x_2 - x_3, \\ \frac{\mathrm{d}}{\mathrm{d}t}x_2 &= x_1 + \frac{1}{5}x_2, \\ \frac{\mathrm{d}}{\mathrm{d}t}x_3 &= \frac{1}{5} + y - 5x_3, \\ \frac{\mathrm{d}}{\mathrm{d}t}y &= \frac{1}{\varepsilon}\left(x_1x_2 - y\right), \end{cases}$$

where the ‘fast’ variable y is omitted from the training data, so that a model with memory effect is required to accurately model the effect of $y(t)$ on $x_3(t)$. However, the aforementioned paper considers this ODE system with $\varepsilon = 0.01$, i.e. a very strong separation of time scales. As mentioned earlier, very small values for ε in the Lorenz ’96 model are easier to deal with, see for example Fatkullin et al. [20]. As a result, the positive results obtained by Fu et al. may not generalise to non-trivial test problems.

Chapter 7

Conclusions and recommendations

7.1 Conclusions

For this thesis, many different neural networks have been trained to perform regression tasks based on one-dimensional PDEs, or ODEs of which not all variables are resolved. These neural networks were either used to predict the full dynamics of the underlying equation, or only as a closure term. The aim of this work was to find how the inclusion of prior knowledge into a machine learning model affects the model's accuracy, how neural ODEs for such problems are best trained, and how memory effects can be included in such models.

As for the inclusion of prior knowledge, three types of prior knowledge were considered: approximate ODE definitions, continuity in time, and conservation of momentum. It was found that inclusion of any of these types of prior knowledge generally leads to an improvement in the accuracy of the resulting model, provided that the neural network used is not too large. For Burgers' equation, it was found that a very small neural closure model with just 57 parameters yields roughly $3-4\times$ lower error than either the coarsely discretised ODE without neural closure term, or the neural network without inclusion of prior knowledge. For larger NNs, the effect of including prior knowledge has a much smaller effect, since larger NNs are able to learn the underlying dynamics more accurately even without prior knowledge included.

As for training procedures for neural ODEs, the discretise-then-optimise approach (i.e. gradient computation by back-propagation through the ODE solver directly) was found to be preferable to the optimise-then-discretise approach (i.e. gradient computation by solving an adjoint ODE). This comparison was not entirely fair, since the discretise-then-optimise approach was also tested using a more accurate discretisation of the original PDE. As such, it is not clear how well this result generalises to other problems. Nevertheless, training with differentiable solvers has shown to be preferable over training with adjoint ODEs for more general ML tasks (see for example Onken and Ruthotto [57]) which is in line with the results obtained in this work.

A significant drawback of the adjoint ODE methods is that their gradient computation is inexact, limiting the accuracy of the resulting training procedure unless the tolerances for the ODE solver are decreased, which in turn comes with a hefty performance penalty. Furthermore, this gradient error increases with the length of the trajectories used for training, meaning that training this way would require carefully choosing the trajectory length to ensure that the model learns long-term behaviour without sacrificing too much accuracy in the gradients. Additionally, while performance differences between training methods were not investigated in this thesis, the theoretical performance advantage of adjoint ODE methods were found not to apply to the problems considered in this thesis. This is due to two reasons:

- The reduced memory usage of adjoint ODE training was not observed since this advantage is only present when the backsolve adjoint algorithm can be used, which could not be used since the problems considered are not backward-stable.
- The speed advantage due to not having to differentiate through the ODE solver also did not materialise, since the differentiation is replaced by solving an adjoint ODE, which is compu-

tationally even more expensive. Specifically, if the forward ODE is solved using an implicit or implicit-explicit ODE solver (i.e. solving non-linear systems of equations for each time step), then back-propagation using discretise-then-optimise requires solving linear systems of equations to compute gradients which are generally computationally easier to solve than non-linear systems. Back-propagation using the adjoint method, however, applies the same ODE solver again which must solve non-linear equations for each time step.

As a result, the adjoint ODE training procedures are only useful in two scenarios. Firstly, if the neural ODEs are solved using an external software library for which auto-differentiation is not available, manually computing gradients using the adjoint ODE method requires less programming work than manually writing back-propagation code for the ODE solver. Secondly, if the training procedure is heavily memory-constrained and the relevant ODE is backward-stable, the backsolve adjoint algorithm can be used which requires less memory to work.

The simplest training method, derivative fitting, was found to perform reasonably well for the Kuramoto-Sivashinsky equation. While training by trajectory fitting with differentiable solvers was still found to perform better, a two-stage approach where derivative fitting is used to provide an approximate model which is then refined by trajectory fitting, may result in a speed-up in training.

Finally, different ways to model memory effects in neural closure models were tested on the L96 ODE system. The resulting models were largely unsuccessful in the sense that they did not significantly outperform much simpler models. The exact reasons for this are not entirely clear, although it was found that the ANODE models do not appear to use the information in the latent space, hence they essentially act as memoryless models. It is possible that using more advanced regularisation techniques to avoid overfitting may allow training the models for longer times, and may result in better accuracy of the models, as well as more significant differences between models.

7.2 Future work

Based on the results of this thesis, there are a number of directions in which further research can be done.

Firstly, in order to draw stronger conclusions regarding different neural ODE training methods, it would be beneficial to test all these methods on the same underlying problem, using the same ODE solver, such that the training method is the only difference between experiments. Finding such problems is non-trivial, since many real-world ODEs and PDEs have properties that make certain training methods unsuitable. For example, the fact that the K-S equation is stiff makes most explicit ODE solvers methods unsuitable, and the fact that it is not backward-stable means that the backsolve adjoint training method cannot be applied. While comparing training methods fairly is possible on simpler ODE systems (a popular example being the chaotic three-variable Lorenz system), such systems are in turn less representative of complex real-world problems.

Second, the problem that training on single time steps (for discrete models) or on derivatives (for continuous models) leads to ML models with poor forecasting abilities, as encountered by Beck et al. [5] and by List et al. [48], was not encountered for the K-S equation. Some crude regularisation techniques were compared, but none of them resulted in an improvement in accuracy, primarily since the model trained with derivative fitting without regularisation was already reasonably accurate. Since derivative fitting and single-step fitting are computationally efficient training strategies, making them work on a wider variety of problems is a relevant problem for future research. However, this would require a different equation for generating training data (for example a two-dimensional fluid flow) where derivative fitting without regularisation is not effective.

Third, this thesis made no attempt to compare computational performance of different ML models or training methods. The reason for this is that for one-dimensional PDEs, computing high-accuracy solutions using a purely numerical method on a fine grid is computationally feasible. For both Burgers' equation and the K-S equation, the dynamics are approximated by a function $f(\mathbf{u})$ that is very cheap to compute, whereas the NNs with convolutional layers are significantly more expensive to compute. As a result, for PDEs in one dimension enhancing the accuracy of a model by adding a neural network closure term is generally less effective (in terms of speed and accuracy) than simply taking a finer discretisation of the PDE. Furthermore, all equations tested are one-dimensional with periodic boundary conditions, meaning that convolutional neural networks for such problems are efficiently implemented using FFTs (see Mathieu et al. [52]). Implementing CNNs this way is a prerequisite for making meaningful comparisons between training procedures. More generally, performance comparisons are only useful if all methods are near-optimal implementations, which includes FFT-based CNNs but also more general techniques such as efficient GPU acceleration. Such techniques were not used here, making performance comparisons essentially meaningless.

Related to performance is the possibility of 'hybrid' ODE models. In Chapter 4, it was found that NODEs discretised using forward Euler did not always train properly due to stability issues, but could achieve good accuracy if they did converge. This makes it appealing to consider models in which the approximate dynamics are integrated using a high-order ODE solver, in which a neural closure term is simply added to the result, for example, using Runge-Kutta 4:

$$\mathbf{u}(t+\Delta t) = \texttt{steprk4}(\mathbf{u}(t), f, \Delta t) + \Delta t \cdot \mathrm{NN}(\mathbf{u}(t); \vartheta).$$

Such models may be computationally more efficient since the neural network term is only evaluated once per time step as opposed to four times per time step if the NN term is also integrated using RK4. However, further research needs to be done to see if this does not come at the cost of reduced stability or accuracy of the resulting model.

Bibliography

[1] Shady E Ahmed et al. "On closures for reduced order models—A spectrum of first-principle to machine-learned avenues". In: *Physics of Fluids* 33.9 (2021), p. 091301.

[2] Donald G Anderson. "Iterative procedures for nonlinear integral equations". In: *Journal of the ACM (JACM)* 12.4 (1965), pp. 547–560.

[3] Nathan Baker et al. *Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence*. Tech. rep. USDOE Office of Science (SC), Washington, DC (United States), 2019.

[4] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *Journal of Machine Learning Research* 18 (2018), pp. 1–43.

[5] Andrea D. Beck, David G. Flad, and Claus-Dieter Munz. "Deep Neural Networks for Data-Driven Turbulence Models". In: *J. Comput. Phys.* 398 (2019).

[6] Aleksandar Botev et al. "Which priors matter? Benchmarking models for learning latent dynamics". In: *arXiv preprint arXiv:2111.05458* (2021).

[7] Elena Celledoni et al. "Structure-preserving deep learning". In: *European Journal of Applied Mathematics* 32.5 (2021), pp. 888–936.

[8] Ashesh Chattopadhyay, Pedram Hassanzadeh, and Devika Subramanian. "Data-driven predictions of a multiscale Lorenz 96 chaotic system using machine-learning methods: reservoir computing, artificial neural network, and long short-term memory network". In: *Nonlinear Processes in Geophysics* 27.3 (2020), pp. 373–389.

[9] Ricky TQ Chen et al. "Neural ordinary differential equations". In: *Advances in neural information processing systems* 31 (2018).

[10] Alexandre Joel Chorin and Ole H Hald. *Stochastic Tools in Mathematics and Science*. Vol. 1. Springer, 2009, pp. 145–150.

[11] Julian D Cole. "On a quasi-linear parabolic equation occurring in aerodynamics". In: *Quarterly of Applied Mathematics* 9.3 (1951), pp. 225–236.

[12] Steven M Cox and Paul C Matthews. "Exponential time differencing for stiff systems". In: *Journal of Computational Physics* 176.2 (2002), pp. 430–455.

[13] Miles Cranmer et al. "Lagrangian Neural Networks". In: *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2020.

[14] Daan Crommelin and Eric Vanden-Eijnden. "Subgrid-scale parameterization with conditional Markov chains". In: *Journal of the Atmospheric Sciences* 65.8 (2008), pp. 2661–2675.

[15] *Deep Implicit Layers - Neural ODEs, Deep Equilibirum Models, and Beyond.* URL: `http://implicit-layers-tutorial.org/` (visited on 03/25/2022).

[16] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. "Augmented neural ODEs". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 3140–3150.

[17] Russell A Edson et al. "Lyapunov exponents of the Kuramoto–Sivashinsky PDE". In: *The ANZIAM Journal* 61.3 (2019), pp. 270–285.

[18] N Benjamin Erichson, Michael Muehlebach, and Michael W Mahoney. "Physics-informed autoencoders for Lyapunov-stable fluid flow prediction". In: *arXiv preprint arXiv:1905.10866* (2019).

[19] *Event handling and callback functions > DiscreteCallback examples.* URL: `https://diffeq.sciml.ai/v7.1/features/callback_functions/#Example-1:-Interventions-at-Preset-Times` (visited on 02/21/2022).

[20] Ibrahim Fatkullin and Eric Vanden-Eijnden. "A computational strategy for multiscale systems with applications to Lorenz 96 model". In: *Journal of Computational Physics* 200.2 (2004), pp. 605–638.

[21] Herbert Federer. *Geometric Measure Theory.* Springer, 2014, p. 64.

[22] Xiaohan Fu, Lo-Bin Chang, and Dongbin Xiu. "Learning reduced systems via deep neural networks with memory". In: *Journal of Machine Learning for Modeling and Computing* 1.2 (2020), pp. 97–118.

[23] Craig Gin et al. "Deep learning models for global coordinate transformations that linearise PDEs". In: *European Journal of Applied Mathematics* 32.3 (2021), pp. 515–539.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[25] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. "Hamiltonian neural networks". In: *Advances in Neural Information Processing Systems* 32 (2019).

[26] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* 2nd ed. Springer-Verlag, 1993, 1993.

[27] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2016, pp. 770–778.

[28] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". 1991.

[29] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[30] Eberhard Hopf. "The partial differential equation $u_t + uu_x = \mu u_{xx}$". In: *Communications on Pure and Applied Mathematics* 3.3 (1950), pp. 201–230. DOI: `https://doi.org/10.1002/cpa.3160030302`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160030302`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160030302`.

[31] Willem H Hundsdorfer and Jan G Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations.* Vol. 33. Springer, 2003, pp. 383–393.

[32] Michael Innes. "Don't unroll adjoint: Differentiating ssa-form programs". In: *arXiv preprint arXiv:1810.07951* (2018).

[33] Antony Jameson. "Energy estimates for nonlinear conservation laws with applications to solutions of the Burgers equation and one-dimensional viscous flow in a shock tube by central difference schemes". In: *Proceedings of the 18th AIAA Computational Fluid Dynamics Conference.* 2007.

[34] Pengzhan Jin et al. "SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems". In: *Neural Networks* 132 (2020), pp. 166–179.

[35] Johann Joss. "Algorithmisches Differenzieren". PhD thesis. ETH Zürich, 1976.

[36] Alireza Karimi and Mark R Paul. "Extensive chaos in the Lorenz-96 model". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 20.4 (2010), p. 043105.

[37] Aly-Khan Kassam and Lloyd N Trefethen. "Fourth-order time-stepping for stiff PDEs". In: *SIAM Journal on Scientific Computing* 26.4 (2005), pp. 1214–1233.

[38] Kenji Kawaguchi. "On the Theory of Implicit Deep Learning: Global Convergence with Implicit Layers". In: *International Conference on Learning Representations (ICLR).* 2021.

[39] Christopher A Kennedy and Mark H Carpenter. "Higher-order additive Runge–Kutta schemes for ordinary differential equations". In: *Applied Numerical Mathematics* 136 (2019), pp. 183–205.

[40] Patrick Kidger. "On Neural Differential Equations". PhD thesis. University of Oxford, 2022.

[41] Patrick Kidger et al. “Neural controlled differential equations for irregular time series”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6696–6707.

[42] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[43] Mark A Kramer. “Nonlinear principal component analysis using autoassociative neural networks”. In: *AIChE Journal* 37.2 (1991), pp. 233–243.

[44] Yoshiki Kuramoto. “Diffusion-induced chaos in reaction systems”. In: *Progress of Theoretical Physics Supplement* 64 (1978), pp. 346–367.

[45] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. “Artificial neural networks for solving ordinary and partial differential equations”. In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000.

[46] Julia Ling, Reese Jones, and Jeremy Templeton. “Machine learning strategies for systems with invariance properties”. In: *Journal of Computational Physics* 318 (2016), pp. 22–35.

[47] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance”. In: *Journal of Fluid Mechanics* 807 (2016), pp. 155–166.

[48] Björn List, Li-Wei Chen, and Nils Thuerey. “Learned turbulence modelling with differentiable fluid solvers”. In: *arXiv preprint arXiv:2202.06988* (2022).

[49] Bethany Lusch, J Nathan Kutz, and Steven L Brunton. “Deep learning for universal linear embeddings of nonlinear dynamics”. In: *Nature communications* 9.1 (2018), pp. 1–10.

[50] Chao Ma, Jianchun Wang, et al. “Model reduction with memory and the machine learning of dynamical systems”. In: *arXiv preprint arXiv:1808.04258* (2018).

[51] Stefano Massaroli et al. “Dissecting neural odes”. In: *arXiv preprint arXiv:2002.08071* (2020).

[52] Michael Mathieu, Mikael Henaff, and Yann LeCun. “Fast training of convolutional networks through FFTs”. In: *Proceedings of the 2nd International Conference on Learning Representations, ICLR 2014*. 2014.

[53] Hazime Mori. “Transport, collective motion, and Brownian motion”. In: *Progress of Theoretical Physics* 33.3 (1965), pp. 423–455.

[54] *Neural Ordinary Differential Equations with sciml_train*. URL: `https://diffeqflux.sciml.ai/v1.47/examples/neural_ode_sciml/` (visited on 06/16/2022).

[55] Alexander Norcliffe et al. “On second order behaviour in augmented neural odes”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 5911–5921.

[56] *ODE Solvers > Recommended Methods > Stiff Problems*. URL: `https://diffeq.sciml.ai/v7.1/solvers/ode_solve/#Stiff-Problems` (visited on 01/18/2022).

[57] Derek Onken and Lars Ruthotto. “Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows”. In: *arXiv preprint arXiv:2005.13420* (2020).

[58] Alexander Ostermann and Michel Roche. “Rosenbrock methods for partial differential equations and fractional orders of convergence”. In: *SIAM Journal on Numerical Analysis* 30.4 (1993), pp. 1084–1098.

[59] Shaowu Pan and Karthik Duraisamy. “Long-time predictive modeling of nonlinear dynamical systems using neural networks”. In: *Complexity* 2018 (2018).

[60] *Parallel Computing and Scientific Machine Learning (SciML): Methods and Applications*. URL: `https://book.sciml.ai/` (visited on 03/09/2022).

[61] Jonghwan Park and Haecheon Choi. “Toward neural-network-based large eddy simulation: application to turbulent channel flow”. In: *Journal of Fluid Mechanics* 914 (2021). A16.

[62] Jaideep Pathak et al. “Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach”. In: *Physical Review Letters* 120.2 (2018), p. 024102.

[63] Suraj Pawar et al. "A deep learning enabler for nonintrusive reduced order modeling of fluid flows". In: *Physics of Fluids* 31.8 (2019), p. 085101.

[64] L.S. Pontryagin et al. *The Mathematical Theory of Optimal Processes.* John Wiley & Sons, 1962.

[65] Christopher Rackauckas and Qing Nie. "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia". In: *Journal of Open Research Software* 5.1 (2017). 5(1):15.

[66] Christopher Rackauckas et al. "A comparison of automatic differentiation and Continuous sensitivity analysis for derivatives of differential equation solutions". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)* (2021), pp. 1–9.

[67] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[68] Pierre Sagaut. *Large Eddy Simulation for Incompressible Flows: an Introduction.* Springer Science & Business Media, 2006.

[69] Omer San and Romit Maulik. "Neural network closures for nonlinear model order reduction". In: *Advances in Computational Mathematics* 44.6 (2018), pp. 1717–1750.

[70] Benjamin Sanderse. "Non-linearly stable reduced-order models for incompressible flow with energy-conserving finite volume methods". In: *Journal of Computational Physics* 421 (2020), p. 109736.

[71] Gregory I Sivashinsky. "Nonlinear analysis of hydrodynamic instability in laminar flames—I. Derivation of basic equations". In: *Acta astronautica* 4.11 (1977), pp. 1177–1206.

[72] Bert Speelpenning. "Compiling fast partial derivatives of functions given by algorithms". PhD thesis. University of Illinois at Urbana-Champaign, 1980.

[73] Raymond J Spiteri and Steven J Ruuth. "A new class of optimal high-order strong-stability-preserving time discretization methods". In: *SIAM Journal on Numerical Analysis* 40.2 (2002), pp. 469–491.

[74] Gerd Steinebach. "Order-reduction of ROW-methods for DAEs and method of lines applications". In: *Preprint/Technische Hochschule Darmstadt, Fachbereich Mathematik* 1741 (1995).

[75] *Strategies to Avoid Local Minima.* URL: `https://diffeqflux.sciml.ai/v1.47/examples/local_minima/` (visited on 03/04/2022).

[76] Ch Tsitouras. "Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2 (2011), pp. 770–775.

[77] Belinda Tzen and Maxim Raginsky. "Neural stochastic differential equations: Deep latent Gaussian models in the diffusion limit". In: *arXiv preprint arXiv:1905.09883* (2019).

[78] James H Verner. "Numerically optimal Runge–Kutta pairs with interpolants". In: *Numerical Algorithms* 53.2-3 (2010), pp. 383–396.

[79] Pantelis R Vlachas et al. "Backpropagation algorithms and reservoir computing in recurrent neural networks for the forecasting of complex spatiotemporal dynamics". In: *Neural Networks* 126 (2020), pp. 191–217.

[80] Qian Wang, Nicolò Ripamonti, and Jan S Hesthaven. "Recurrent neural network closure of parametric POD-Galerkin reduced-order models based on the Mori-Zwanzig formalism". In: *Journal of Computational Physics* 410 (2020), p. 109402.

[81] Gerhard Wanner and Ernst Hairer. *Solving Ordinary Differential Equations II.* Springer Berlin Heidelberg, 1996.

[82] Steffen Wiewel, Moritz Becher, and Nils Thuerey. "Latent space physics: Towards learning the temporal evolution of fluid flow". In: *Computer Graphics Forum.* Vol. 38. 2. Wiley Online Library. 2019, pp. 71–82.

[83] Ronald J Williams and Jing Peng. “An efficient gradient-based algorithm for on-line training of recurrent network trajectories”. In: *Neural Computation* 2.4 (1990), pp. 490–501.

[84] Ronald J Williams and David Zipser. “A learning algorithm for continually running fully recurrent neural networks”. In: *Neural Computation* 1.2 (1989), pp. 270–280.

[85] Qunxi Zhu, Yao Guo, and Wei Lin. “Neural delay differential equations”. In: *arXiv preprint arXiv:2102.10801* (2021).

[86] Robert Zwanzig. “Memory effects in irreversible thermodynamics”. In: *Physical Review* 124.4 (1961), p. 983.

Appendix A

Notation

$\mathbb{R}^n$	The space of vectors of length n.
$\mathbb{R}^{n \times m}$	The space of matrices with n rows and m columns.
$\mathbb{R}^{n_1 \times \cdots \times n_k}$	The space of tensors of shape $(n_1, \ldots, n_k)$.
$\mathbf{a}, \mathbf{b}, \ldots$	Vectors.
$\mathbf{a}_i$	The i^{th} component of vector $\mathbf{a}$.
$\mathbf{a}^{(i)}$	The i^{th} vector in a finite or infinite sequence of vectors $\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \ldots$.
$\|\mathbf{a}\|$ where $\mathbf{a} \in \mathbb{R}^n$	Vector 2-norm (Euclidean norm) $\sqrt{\sum_{i=1}^n \lvert\mathbf{a}_i\rvert^2}$.
$\|\mathbf{a}\|_p$ where $\mathbf{a} \in \mathbb{R}^n$	Vector p-norm $\left(\sum_{i=1}^n \lvert\mathbf{a}_i\rvert^p\right)^{1/p}$.
$\mathbf{A}, \mathbf{B}, \ldots$	Matrices.
$\|\mathbf{A}\|$ where $\mathbf{A} \in \mathbb{R}^{n \times m}$	Matrix operator norm $\sup_{\mathbf{x} \in \mathbb{R}^n} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\mathbf{x} \in \mathbb{R}^n, \|x\|=1} \|\mathbf{A}\mathbf{x}\|$.
$\|\mathbf{A}\|_F$ where $\mathbf{A} \in \mathbb{R}^{n \times m}$	Matrix Frobenius norm $\sqrt{\sum_{i=1}^n \sum_{j=1}^m \lvert\mathbf{A}_{ij}\rvert^2}$.
$a \mapsto b$	Anonymous function definition, equivalent to "f where $f(a) = b$"
$A \to B, a \mapsto b$	Equivalent to "f where $f(a) = b \in B$ for $a \in A$".

Appendix B

Terminology

Trajectory: Time-dependent solution $\mathbf{u}(t)$ of an ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \ldots$. This can be either as a function $\mathbf{u}(t)$ or a sequence of states $\mathbf{u}^{(i)} = \mathbf{u}(t_i)$.
Snapshot: Value of a trajectory $\mathbf{u}(t)$ at a specific point in time.
Derivative: Unless otherwise specified: time-derivative of a the trajectory of an ODE (i.e. right-hand side of that same ODE), or finite-difference approximation thereof.
Training data: Input/output pairs where the difference between the predicted and actual output is used to train the parameters of a neural network.
Validation data: Input/output pairs where the difference between the predicted and actual output is not used to train parameters, but *is* used to inform the training procedure in another way, such as early stopping.
Testing data: Input/output pairs that are used to test the accuracy of a neural network after training has been completed.

Appendix C

Abbreviations

AD	Automatic differentiation
ADAM	Adaptive moment estimation (stochastic optimisation algorithm)
ANODE	Augmented Neural ODE
CNN	Convolutional neural network
ESDIRK	Explicit-first-stage singly diagonally implicit Runge-Kutta
FFT	Fast Fourier Transform
IMEX	Implicit-explicit
LSTM	Long short-term memory
ML	Machine Learning
MSE	Mean-square error
NN	Neural network
NODE	Neural Ordinary Differential Equation
ODE	Ordinary differential equation
PDE	Partial differential equation
PINN	Physics-informed neural network
RK4	Runge-Kutta 4, 4^{th}-order accurate ODE solver
RMS	Root-mean-square
RMSE	Root-mean-square error
RNN	Recurrent neural network
SciML	Scientific Machine Learning
SSPRK	Strong-stability preserving Runge-Kutta
VPT	Valid prediction time