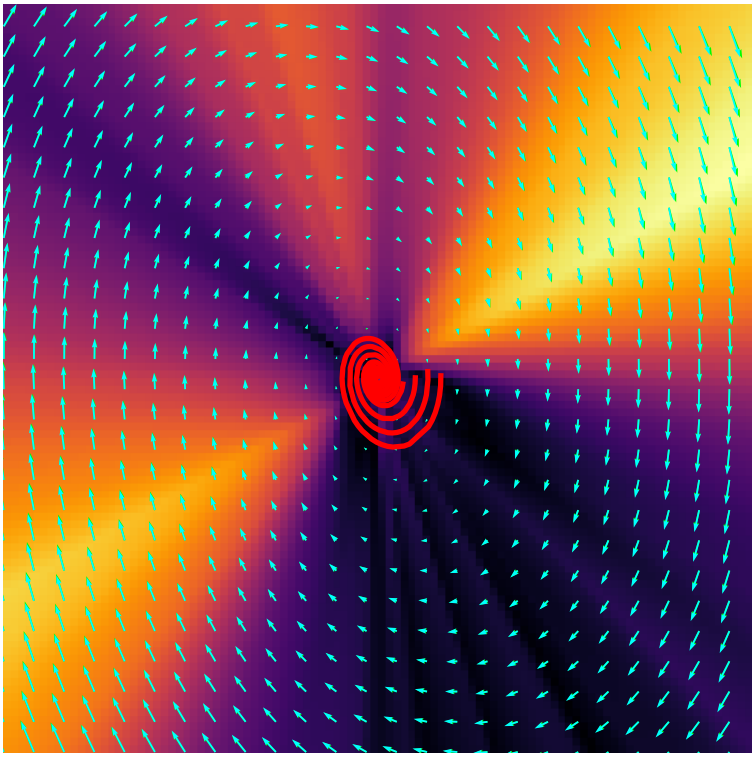


Modelling Dynamical Systems Using Neural Ordinary Differential Equations

Learning ordinary differential equations from data using neural networks

Master's thesis in Complex Adaptive Systems

DANIEL KARLSSON & OLLE SVANSTRÖM

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Modelling Dynamical Systems Using Neural Ordinary Differential Equations

Learning ordinary differential equations from data using neural networks

Daniel Karlsson & Olle Svanström

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2019

Modelling Dynamical Systems Using Neural Ordinary Differential Equations
Learning ordinary differential equations from data using neural networks
Daniel Karlsson & Olle Svanström

Supervisors: Michaël Ughetto & Laura Masaracchia, Semcon

Examiner: Mats Granath, Department of Physics

Master's Thesis 2019
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Heatmap of the relative error between the ordinary differential equation describing a damped harmonic oscillator and the time derivative approximated by ODENet. The red lines are the training data as phase space trajectories of the oscillator for different initial values. The cyan arrows are field lines for the time derivative learned by ODENet, almost perfectly overlapping corresponding green arrows for the harmonic oscillator ordinary differential equations. Darker areas represent low-error regions, while bright colors correspond to a higher error. The error is lower closer to the training data and in the directions parallell to the major axis of the elliptical training data.

Typeset in LaTeX
Gothenburg, Sweden 2019

Modelling Dynamical Systems Using Neural Ordinary Differential Equations
Learning ordinary differential equations from data using neural networks
Daniel Karlsson & Olle Svanström
Department of Physics
Chalmers University of Technology

Abstract

Modelling of dynamical systems is an important problem in many fields of science. In this thesis we explore a data-driven approach to learn dynamical systems from data governed by ordinary differential equations using Neural Ordinary Differential Equations (ODENet). ODENet is a recently introduced family of artificial neural network architectures that parameterize the derivative of the input data with a neural network block. The output of the full architecture is computed using any numerical differential equation solver. We evaluate the modelling capabilities of ODENet on four datasets synthesized from dynamical systems governed by ordinary differential equations. We extract a closed-form expression for the derivative parameterized by ODENet with two different methods: a least squares regression approach and linear genetic programming. To evaluate ODENet the derivatives learned by the network were compared to the true ordinary differential equations used to synthesize the data. We found that ODENet learns a parameterization of the underlying ordinary differential equation governing the data that is valid in a region surrounding the training data. From this region a closed-form expression that was close to the true system could be extracted for both linear and non-linear ODEs.

Keywords: Artificial neural networks, ordinary differential equations, time-series prediction, dynamical systems, deep learning, linear genetic programming.

Acknowledgements

We want to express our deepest thanks to our supervisors Laura and Michaël for all their help and guidance throughout the project. We would also like to thank all the other nice people we have met at Semcon for making us feel very welcome during our work.

Finally, we would like to give our thanks to our thesis worker colleagues Martin and Annie. Without them this thesis would have been finished faster, but we would not have had as much fun along the way.

Olle Svanström Daniel Karlsson, Gothenburg, June 2019

Contents

List of Figures

1
Introduction

1.1 Background

The world is ever-changing due to the passage of time and different phenomena interacting with and affecting each other. A part of these changes in systems are predictable and can be modelled. Due to their repeated change over time such systems are referred to as dynamical systems. Dynamical systems occur in a wide array of scientific subjects in fields such as physics, chemistry, biology, and engineering making modelling of dynamical systems a task of scientific importance. For continuous dynamical systems the time-evolution can in many cases be described by differential equations. In this thesis we will only focus on systems described by ordinary differential equations (ODEs).

Learning dynamical systems from data is generally a difficult task. Traditionally, modelling dynamical systems from data has been performed as a parameter estimation problem. This requires construction of a prior model and then fitting its parameters to the available data. A bad initial model means that the learned model will not perfectly capture the dynamics of the system. Therefore, domain-specific knowledge and insights into the available data is necessary to produce an appropriate model.

To reduce the need for domain knowledge and to automate modelling of dynamical systems from data, there has been work into creating model-free methods that only rely on the available data. One approach that have started to be explored in recent years is the utilization of artificial neural networks and deep learning for data-driven modelling of dynamical systems.

Artificial neural networks (ANNs), and more specifically deep learning, have gained much popularity in recent years. Deep learning is today used in a wide range of fields: in entertainment to recommend shows that we would like to watch [1], in medicine for drug discovery and medical imaging [2], in engineering for computer vision in self-driving cars [3], and in fundamental physics [4], just to name a few fields and applications. This widespread usage is in large due to major improvements to computational power and the quantities of data being collected, as deep learning algorithms require large amounts of data to be trained on. There is a variety of network structures suitable for different applications. Some of the most common variants include Recurrent Neural Networks (RNNs) for time-series prediction and natural language processing, and Convolutional Neural Networks (CNNs) for image recognition [5].

A Residual neural network (ResNet) is a relatively new type of network archi-

tecture constructed to reduce problems inherent with very deep networks [6, 7, 8]. ResNet utilizes *skip connections* to learn residuals, small iterative changes to the input from an identity mapping, rather than a full mapping from the input to the output. This is illustrated in figure 1.1.

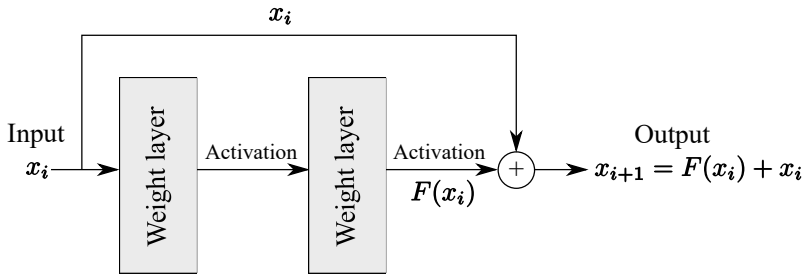


Figure 1.1: The skip connection in a residual neural network block. The network learns the change to the input $F(x_i)$, the residual, needed to acquire the desired output $x_{i+1} = x_i + F(x_i)$ rather than learning the entire mapping $x_{i+1} = \tilde{F}(x_i)$ directly.

1.1.1 Neural Ordinary Differential Equations

Recently, Chen et al. introduced a new family of ANNs called Neural Ordinary Differential Equations (ODENet) [9], based on the notion that the skip connections in ResNet can be seen as a realization of Euler's method for numerically solving ODEs. Instead of letting the network learn the residuals between fixed points as in ResNet, ODENet parameterizes the local derivative of the input data with a neural network block. They then let an ODE solver be responsible for the steps taken from an input x_i at time t_i to an output x_{i+1} at time t_{i+1}, with the derivative at each point fed to the ODE solver to take each step. This is illustrated in figure 1.2. A strength of ODENet compared to ResNet lies in the choice of the ODE solver. By building ODENet with an ODE solver with an adaptive step length it can step between any two time points. This makes it possible for ODENet to handle time-series with irregular measurements and also trade off accuracy for speed by taking larger steps.

Chen et al. introduced three different applications for ODENet. They proposed that it can be used as a drop-in replacement for a ResNet block, has some advantages when computing normalizing flows [10], and to model and make predictions of time-series. The latter is the focus of this thesis, in which we explore how ODENet and its time-series capabilities can be used to model and learn dynamical systems from data without prior knowledge of the underlying dynamics.

What allows ODENet to be used for dynamical systems modelling is the fact that the ANN block of ODENet learns the local derivative $\frac{\mathrm{d}x}{\mathrm{d}t}(t_i)$ at input point $x(t_i)$ necessary for the ODE solver to step to the output $x(t_{i+1})$, as pictured in figure 1.2. This means that the underlying ODE of the data is directly encoded in the ANN block during training. After training, ODENet can take any point $x(t)$ as input to predict the next point $x(t + \Delta t)$ after time Δt by taking steps in the ODE solver, based on the information about the derivative encoded in the ANN block. This means that ODENet is predicting the dynamics of the unknown system. Since

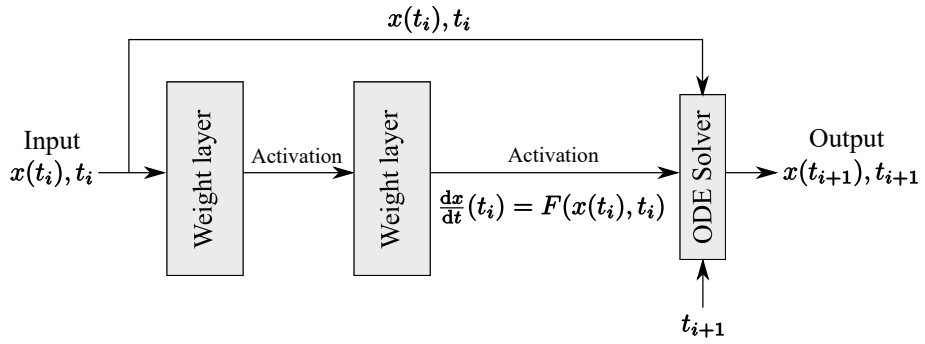


Figure 1.2: The structure of ODENet with an ANN block composed of two layers connected to an ODE solver. In comparison to ResNet instead of learning the residual from the input x_i to the output x_{i+1}, the ANN block learns the time derivative for the input at time t_i. The derivative is then used by an ODE solver to step to the output x_{i+1} at time t_{i+1}.

the dynamical system is described by a corresponding ODE, finding the analytical dynamics amounts to extracting the information about the local derivative encoded in the network.

1.2 Aim and scope

The aim of this thesis is to examine ODENet as a tool for modelling dynamical systems from data and evaluate how well the network approximates the true underlying dynamics of the system. We also explore two methods for extracting analytical closed-form expressions of the ODE from the learned network: least squares regression which require making some prior guesses about the system, and linear genetic programming [11] which is almost completely model-free but with some inherent randomness.

In addition to evaluating the network performance the aim of this thesis is to evaluate how to optimally train ODENet as well as construct the network structure, finding suitable parameters for training and the network architecture. We will not be focusing on optimizing the absolute training time or optimizing the algorithms for hardware, i.e. training the model on the GPU or the CPU. We will evaluate our findings in terms of the resulting network performance as a modelling tool for learning ODEs from data.

1.3 Related Work

Even though the ODENet architecture by Chen et al. is novel, the connection between neural network layers and differential equations have been observed previously. Among others Lu et al. [12], and Haber and Ruthotto [13] have previously worked on constructing new classes of neural networks inspired by stepping methods from ODE solvers and their connection to ResNet layers. Ruthotto and Haber [14] similarly published an interpretation of deep residual CNNs as nonlinear systems

of partial differential equations (PDEs) and constructed a class of neural networks from this motivation. These works have primarily focused on improving the ResNet architecture by taking inspiration from ODEs and PDEs, and not on the learning of the actual dynamical systems.

There has also been other recent work on extending traditional modelling of dynamical systems with tools from machine learning to learn behaviour from data. Some previous papers have exploited statistical machine learning methods, as opposed to deep learning. Raissi et al. [15, 16] explored different methods for learning from data for predicting dynamics governed by PDEs. They developed a method of learning PDEs by means of Gaussian processes and putting a Gaussian prior on the dynamics that was to be learned, taking steps between states according to well-known numerical ODE solvers. Raissi et al. [17, 18] instead used a neural network based approach, with the dynamics discretized by a multistep ODE solver scheme. The networks in these works have been *physics-informed*, meaning that they have knowledge of the dynamical system that is to be learned except for a few scalar parameters. The focus of their method is on fitting the unknown scalar parameters of their already constructed model to data.

The task of learning unknown dynamics have been explored by Long et al. [19] with PDE-Net. PDE-Net intends to learn PDE dynamics with a network architecture inspired by the Euler ODE solver stepping scheme, the only assumption being the maximum order of the PDE. A similar idea is proposed by Raissi et al. [20], using a stepping scheme inspired by ODE solvers from the linear multistep family. In comparison to ODENet both networks rely on data sampled at regular short time intervals.

There has also been similar work done following the release of ODENet, as dynamical systems modelling using deep learning is an ongoing topic. During the work on this thesis Ayed et al. [21] published a paper regarding the same problem of forecasting dynamical systems from data following an unknown ODE. They followed the same approach used in ODENet by approximating the unknown ODE with a neural network, in their case a ResNet module. Ayed et al. utilized Euler's forward method as a set ODE solver while ODENet allows any ODE solver to be used. The fixed ODE solver allowed Ayed et al. to make use of the fact that all steps in the solver are known, which simplifies training at the cost of flexibility in the ODE solver. Furthermore, by using a fixed time-step solver they are bound to a single time-step during training and therefore lack the capability of dealing with training data sampled at irregular time intervals compared to ODENet.

2

Theory

The chapter presents theory underlying the methods explored in the thesis. We start by giving a brief overview of ordinary differential equations and their occurrences in science. We then present the differential equations that are used as training data in the thesis, followed by a section on standard numerical methods for solving ordinary differential equations. We treat ANNs starting with a short summary of general concepts for feedforward networks leading to a description of ODENet and its applications for modelling dynamical systems. The last section describes least squares regression and linear genetic programming, the two methods used to extract an analytical ODE from a trained ODENet.

2.1 Ordinary differential equations

Ordinary differential equations are used to describe the dynamics of a changing system. Dynamical systems can be found in chemistry (e.g. rate of change of the components in chemical reactions [22]), biology (e.g. population dynamics [23], disease spreading [24]), and physics (e.g. laws of motion, harmonic oscillators). In general, ODEs describe the change of a variable $y(x)$ with respect to some independent variable x. In our case, we will only deal with systems describing the rate of change of $y(t)$ with respect to time t.

An explicit ODE of the nth order describing the rate of change of a variable y with respect to time t as an independent variable is on the form

$$y^{(n)} = F(t, y, y', y'', ..., y^{(n-1)}). \tag{2.1}$$

Here F is a function of the independent variable t, the dependent variable $y(t)$ and its derivatives with respect to t, $y^{(i)} = \frac{\mathrm{d}^i y}{\mathrm{d}t^i}$ for $i = 1...n$. A higher order ODE, with derivatives of order greater than one, can always be rewritten as a system of first-order ODEs by redefining the higher order differentials as new variables [25, Chapter I.1].

The special case of a system of D linear first order ODEs can be written on matrix form as

$$\mathbf{y}' = \mathbf{A}\mathbf{y} + \mathbf{b}, \tag{2.2}$$

with the $D \times D$ coefficient matrix $\mathbf{A}$ and the $D \times 1$ constant vector $\mathbf{b}$.

2.1.1 Initial value problems (IVPs)

A solution $y(t) = G(t)$ to a first-order ODE is only uniquely defined up to an unknown integration constant. Similarly, an nth order ODE gives rise to n arbitrary integration constants. Therefore, a full solution requires knowledge of an initial condition $(t_0, \mathbf{y}_0)$ that can be used to determine the constants. An ODE together with a given initial condition is known as an initial value problem (IVP). A solution $\mathbf{y}(t)$ to the IVP is a function satisfying both the ODE and the initial condition $\mathbf{y}(t_0) = \mathbf{y}_0$.

2.2 Examined ODEs

The datasets used in the thesis were synthesized from four different systems governed by ODE equations: parabolic motion, harmonic oscillators, Lotka-Volterra predator-prey equations, and a model for seismic activity on a building with multiple floors. The following sections briefly describes the mathematics behind these problems.

2.2.1 Parabolic motion

For the movement of a projectile in two-dimensional space with coordinates $(x(t), y(t))$, Newtons second law of motion gives rise to a simple system of decoupled second-order ODEs

$$\begin{aligned} x' &= v_x \\ y' &= v_y \\ x'' &= 0 \\ y'' &= -g \end{aligned} \tag{2.3}$$

along with a set of initial conditions $\{x(0), y(0), v_x(0), v_y(0)\}$. Here, v_x and v_y are the horizontal and vertical velocity measured in $\mathrm{m\,s^{-1}}$, respectively. If the initial horizontal velocity $v_x(0) = 0\,\mathrm{m\,s^{-1}}$ the system is described by the vertical motion from y' and y''.

In the above case the projectile is moving in a vacuum. To account for air resistance a dampening term proportional to the square of the velocity, but opposite in direction, is added. This leads to a non-linear second-order ODE system

$$\begin{aligned} x' &= v_x \\ y' &= v_y \\ x'' &= -\frac{k}{m} x'|x'| \\ y'' &= -\frac{k}{m} y'|y'| - g \end{aligned} \tag{2.4}$$

Here k $[\mathrm{kg\,m^{-1}}]$ is the drag proportionality constant and m the mass of the projectile in kg. An example of the three cases described are illustrated in figure 2.1.

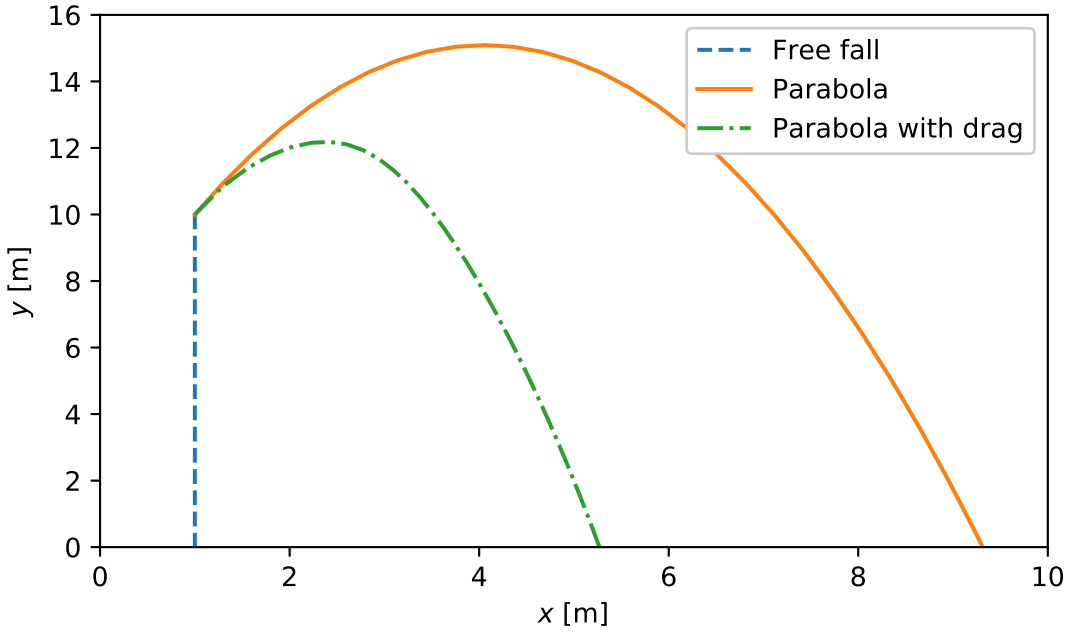


Figure 2.1: Examples of three types of parabolic trajectories all starting from the same initial position in Euclidean space. The acceleration due to gravity is $g = 9.82\,\mathrm{m\,s^{-2}}$. The blue dashed trajectory is an object with zero initial velocity both vertically and horizontally. The other trajectories share the same initial velocity. For the orange solid line there is no acceleration in the horizontal direction, and only gravity acting vertically. For the green dash-dotted line a drag term proportional to the square of the velocity is added. This slows the object down both vertically and horizontally, making the parabola shorter and lowering its maximum.

2.2.2 Harmonic oscillator

Harmonic oscillator equations occur as analogous systems in different subjects, for example mass-spring systems and pendulums in classical mechanics and RLC electronic circuits. These are equivalent in the sense that the ODEs describing the dynamics are on the same form and can be treated by the same mathematical framework.

The simplest case of a harmonic oscillator consists of a weight attached to a spring. When disturbed from its equilibrium position the spring exerts a restoring force $\mathbf{F}_s$ [N] proportional to the displacement x measured in m given by Hooke's law $\mathbf{F}_s = -kx$, with k [N/m] being the spring constant. For a weight of mass m in kg Newton's second law gives rise to the ODE

$$\begin{aligned} x' &= v_x \\ x'' &= -\frac{k}{m}x, \end{aligned} \tag{2.5}$$

with v_x being the velocity measured in $\mathrm{m\,s^{-1}}$.

Adding friction dampens movements in the system, giving rise to a force in the direction opposite to the velocity $\mathbf{F}_c = -c\frac{\mathrm{d}x}{\mathrm{d}t}$ [N], with dampening coefficient c [$\mathrm{kg\,s^{-1}}$].

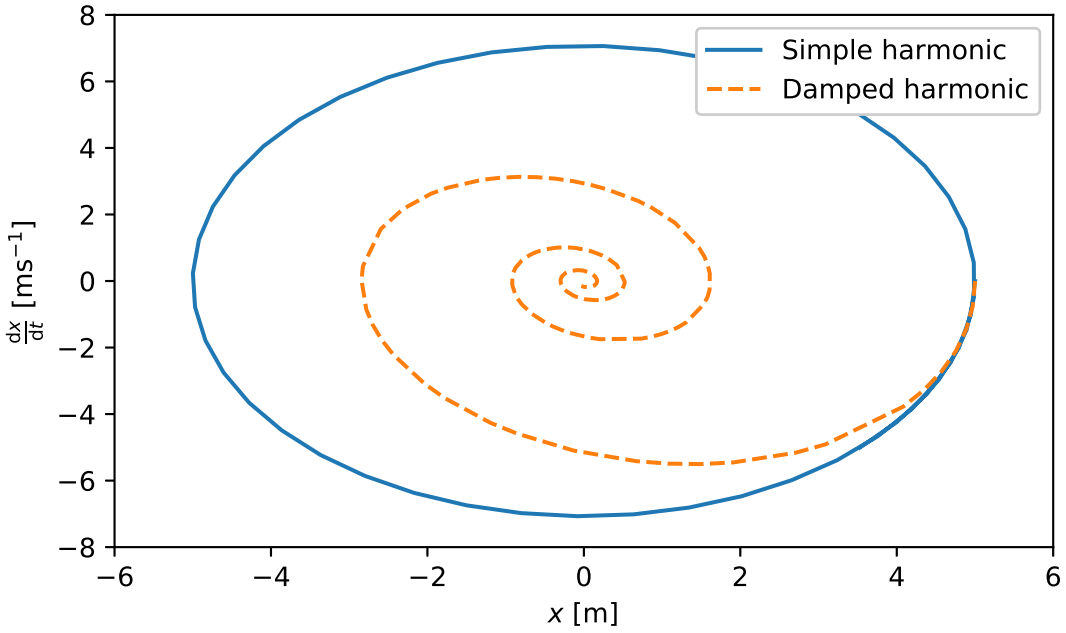


Figure 2.2: Example of two types of harmonic oscillators in phase space. The solid blue line is a harmonic oscillator. The dashed orange line is the trajectory of a damped oscillator starting with the same initial value as the simple. Due to the dampening the phase portrait is an asymptotically stable spiral eventually reaching the fixed point in the origin.

The ODE for a damped oscillator is then

$$\begin{aligned} x' &= v_x \\ x'' &= -\frac{k}{m}x - \frac{c}{m}x'. \end{aligned} \tag{2.6}$$

2.2.3 Lotka-Volterra predator-prey equations

The Lotka–Volterra equations are commonly used to describe the continuous interactions between a predator population $P(t)$ and a prey population $N(t)$ with undefined units. The predators feed on the prey, growing in numbers while reducing the amount of prey. When there is too little prey the predators compete for food making their population size decrease. Few predators means that the prey can thrive. This interaction is described by the coupled ODE system

$$\begin{aligned} N' &= N(\alpha - \beta P) \\ P' &= P(\gamma N - \epsilon), \end{aligned} \tag{2.7}$$

where α, β, γ, and ϵ are positive constants [26]. An example is shown in figure 2.3.

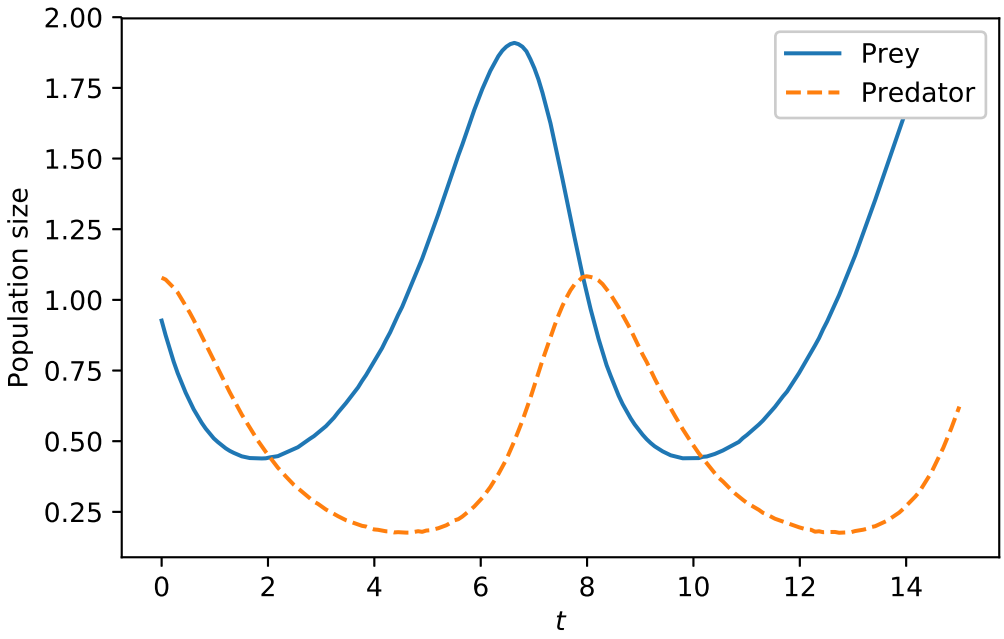


Figure 2.3: The dynamics over time between two population following the Lotka-Volterra equations, with the population size and time t measured in arbitrary units. The populations periodically grow and decrease with the peak of orange dashed line representing predators always following a peak in the blue line depicting the prey population. The values of the parameters in this example were $a = \frac{2}{3}$, $b = \frac{4}{3}$, $c = 1$, $d = 1$.

2.2.4 Earthquake Effects on Buildings

A model for an earthquake's effects on buildings is described in [27]. The model describes each floor i of a multistory building as a point mass m_i measured in kg located at the floor's center of mass. The position of the floor x_i is the distance in metres from the equilibrium position $x_i = 0\,\mathrm{m}$. Each floor is acted upon by a restoring force according to Hooke's law with Hooke's constants k_i [$\mathrm{N\,m^{-1}}$]. Dampening effect are ignored in the system. The earthquake is modelled as a driving oscillation $F(t) = F_0 \cos(\omega t)$ affecting each floor.

We assume that all floors are equal with mass m and Hooke's constant k. For a two-story building this leads to the system of ODEs

$$\begin{aligned} x_1' &= v_{x_1} \\ x_2' &= v_{x_2} \\ x_1'' &= \frac{k}{m}(-2x_1 + x_2) - F_0\omega^2 \cos(\omega t) \\ x_2'' &= \frac{k}{m}(x_1 - x_2) - F_0\omega^2 \cos(\omega t), \end{aligned} \tag{2.8}$$

where v_{x_1} and v_{x_2} are the velocities measured in ms^{-1} for the first and second floor,

respectively. An example of the movements of a two-floor building is shown in figure 2.4

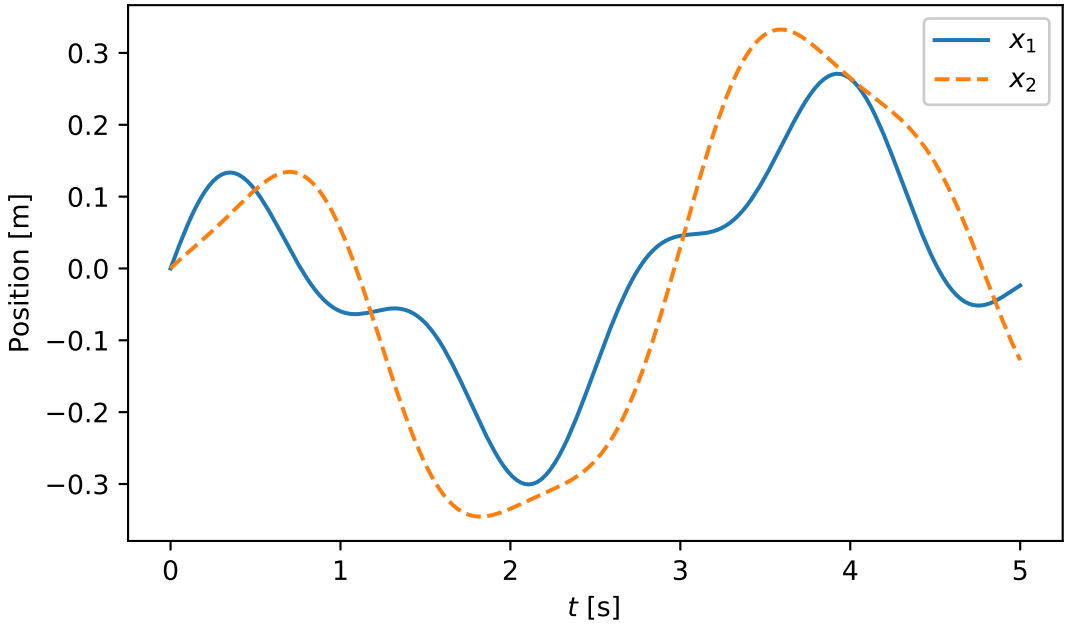


Figure 2.4: Example of movement over time for a two-floor building affected by an earthquake. The solid blue line describes the position of the first floor x_1, while the dashed orange line is the position of the second floor x_2.

2.3 Numerical solutions to IVPs

Since ODEs and IVPs are such common occurrences in industry and science there is a natural need for computational methods to solve them. One of the most simple and well known is Euler's method which, starting from an initial condition $y(t_0) = y_0$, takes linear steps along the ODE $y'(t) = F(t, y(t))$ with a fixed time-step h. This means an update is

$$y_{i+1} = y_i + hF(t_i, y_i), \tag{2.9}$$

with time updates as $t_{i+1} = t_i + h$. One can prove that the global error estimate is proportional to the size of the chosen time step h. Therefore, Euler's method is referred to as a method of order one [25, Chapter I.7].

Euler's method is an explicit method, meaning that a later state is calculated from the current state $y_{i+1} = G(y_i)$. There are also implicit methods, where a subsequent state y_{i+1} is calculated by solving an equation dependent on both the subsequent and the current, and sometimes also previous, states $H(y_{i+1}, y_i, y_{i-1}) = 0$. Most methods for numerically solving IVPs are made for systems of first-order ODEs, since all higher-order ODEs can be rewritten as such. Besides the notion of explicit or implicit methods, the first-order solvers can generally be divided into two main groups: the Runge-Kutta family, and linear multistep methods [25].

Runge-Kutta solvers are based on the idea that instead of going from y_i to y_{i+1} directly, a set of intermediate steps are taken at fractions of the step size h and added in a weighted average. As the name implies multistep methods also use many steps, but in contrast to Runge-Kutta they use a number of previous whole steps. For example, calculating y_{i+3} from the 3 previous steps y_{i+2}, y_{i+1},, and y_i [25, Chapter II, Chapter III].

Figure 2.5 shows the difference in stepping for Euler's method compared to Runge-Kutta solvers and multistep methods, all with the same time step size from t_n to t_{n+1}. Euler's method takes a single step in the direction of the local derivative. Runge-Kutta takes several intermediate shorter steps in the time interval, and from these constructs a final long step. The multistep method takes a single step on the interval t_n and t_{n+1}, but this step is dependent on previous steps taken by the algorithm.

2.3.1 ODE solvers

In this thesis, two algorithms are used to numerically solve IVPs: the Dormand-Prince method (or Runge-Kutta (4,5)), and an adaptive Adams-Moulton-Bashforth method that belongs to the linear multistep class. The described methods are implemented by Chen et al, the authors of the original ODENet [9], and can be found in their GitHub repository [28].

2.3.1.1 Dormand-Prince method

The Dormand-Prince (dopri5) method is a fourth order Runge-Kutta method, meaning that the global error grows as h^4, where h is the stepsize. The local error at each step is proportional to h^5. Since the error is dependent on the step size a smaller step size leads to more accurate results, but at the cost of many more computations. It would be time consuming to manually choose an appropriate step size to optimize this trade-off at every point. Hence, the method uses an algorithm to automatically adapt the step size at each step. The step size adjustment is controlled by two user-supplied parameters, relative error tolerance *Rtol* and absolute error tolerance *Atol*, that makes it possible to still influence the speed/accuracy trade-off. For details we refer to a more comprehensive text on the subject, for example [25, Chapter II].

2.3.1.2 Adaptive order Adams-Moulton-Bashforth

The other algorithm belongs to the linear multistep family and and is an adaptive order Adams-Moulton-Bashforth method. This method is implicit as opposed to the explicit Dormand-Prince method. Implicit methods tend to perform better than explicit methods on certain problems, such as stiff ODEs. A stiff ODE includes terms that vary rapidly, forcing the solver to take extremely small time-steps even though the exact solution might be relatively smooth. This leads to a very large increase of the time required to solve the ODE. Solutions with certain solvers, such as the explicit Dormand-Prince solver, can take a very long time while for example the implicit Adaptive-order Adams might be better [29].

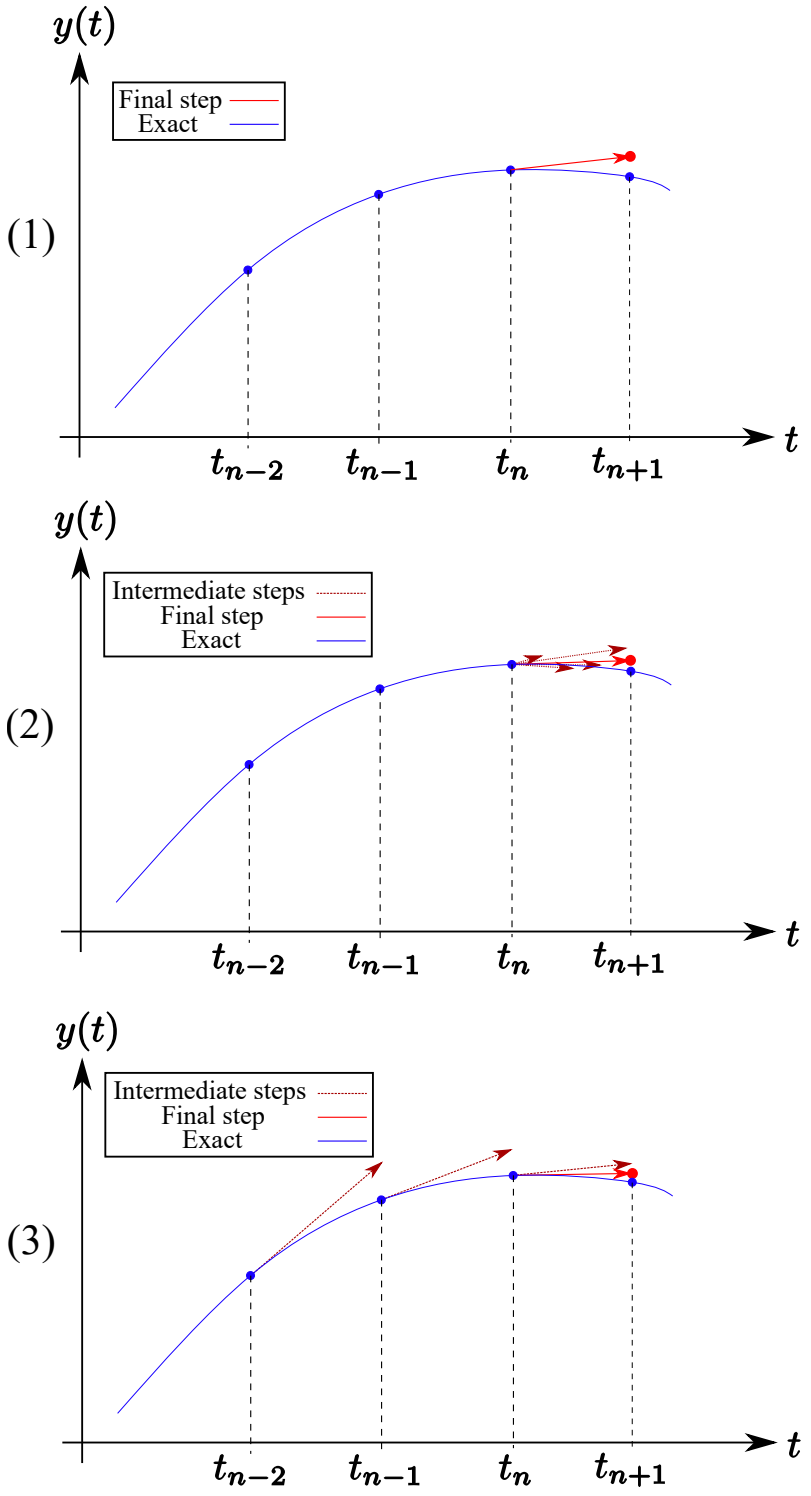


Figure 2.5: Illustration of stepping schemes for different ODE solvers taking a step from time t_n to t_{n+1}. Case (1) is Euler's method that takes a single step along the local derivative. Case (2) is a Runge-Kutta method constructing its step from a weighted average of several shorter intermediate steps. Case (3) is a multistep method. The single step from t_n to t_{n+1} is dependent on the previous steps.

The method also makes use of an adaptive step size with the same *Rtol* and *Atol* parameters as described in the previous paragraph. For details on the choice of order and step size, see [25, Chapters III.7].

2.4 Feedforward artificial neural networks

Originally inspired by neurons and connections in biological brains, the fundamental building blocks for ANNs are layers of interconnected computational units. We will refer to these computational units as neurons throughout this work. An ANN is composed of layers of neurons, with each layer connected to the next with a set of weights $W^{[n]}$ between neurons in different layers. In a fully-connected layer, all neurons in layer n are connected to all neurons in layer $n+1$ with weight matrix $\mathbf{W}^{[n]}$. This is illustrated in figure 2.6. The input to each layer $n+1$ is the output from the previous layer $\mathbf{z}^{[n]}$. The output is calculated by computing a weighted sum of the inputs to each neuron and running the sum through an activation function $a(x)$, determining how much the neurons fire

$$\mathbf{z}^{[n+1]} = a(\mathbf{W}^{[n]}\mathbf{z}^{[n]}). \tag{2.10}$$

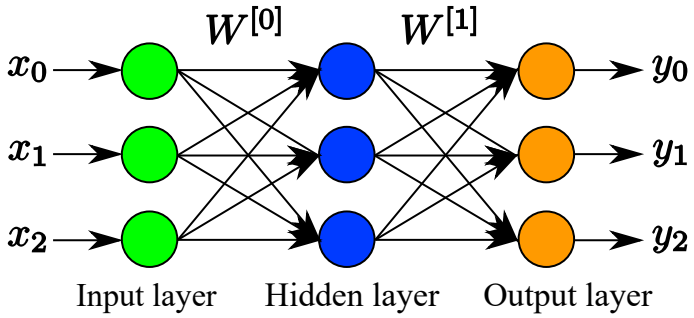


Figure 2.6: A visual representation of a simple fully connected feedforward artificial neural network. Each layer n of the network is connected to the next layer $n+1$ with a set of weights $W^{[n]}$.

2.4.1 Activation functions

Activation functions introduce non-linearity to the mapping learned by the network. This is important since the network otherwise would only be able to learn an affine transformation [30]. There are different kinds of activation functions, appropriate for different kinds of networks and training data. One of the most used in the Rectified Linear Unit (ReLU) activation function

$$a(x) = \max(0, x), \tag{2.11}$$

with a sharp cutoff for negative inputs [5, Chapter 6]. Sometimes, it is preferable to not fully eliminate all negative values. In those situations, an alternative is Leaky ReLU defined as

$$a(x) = \begin{cases} x & \text{if } x > 0 \\ hx & \text{otherwise,} \end{cases} \tag{2.12}$$

with h as a small positive constant. These activation functions are shown in figure 2.7.

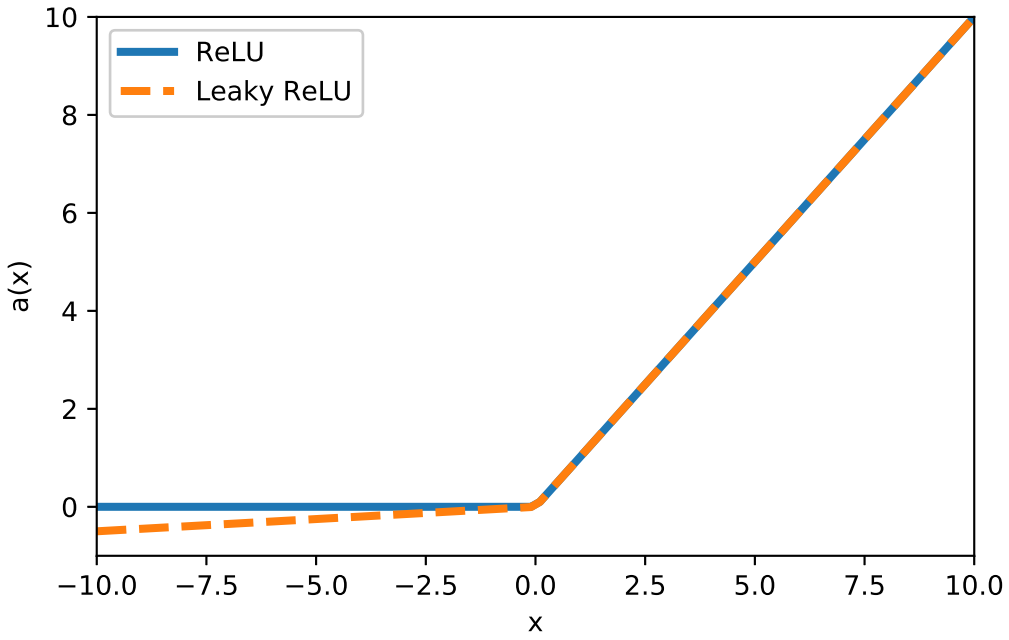


Figure 2.7: Two regularly used activation functions $a(x)$ for neural networks. The solid blue line is ReLU, and the orange dashed line is Leaky ReLU with $h = 0.05$. Both functions produce a linear response for inputs above zero, but cuts inputs below zero. Leaky ReLU permits 'leakage' of small negative values.

2.4.2 Calculating the loss

The output of the final layer is the output of the whole neural network

$$\hat{\mathbf{y}} = \mathbf{z}^{[N]} = a(\mathbf{W}^{[N]}\mathbf{z}^{[N-1]}) \tag{2.13}$$

It can be the value of a single neuron, or many neurons. In supervised learning we have both inputs to the network $\mathbf{x}$ and the appropriate output $\mathbf{y}$. The error on the approximation from the network is the loss $L(\mathbf{y}, \hat{\mathbf{y}})$ calculated between the target output $\mathbf{y}$ and the output from the network $\hat{\mathbf{y}}$. The loss is the objective function that the network should minimize.

The choice of loss function is dependent on the objective that the neural network should solve. In this paper we use the Mean Squared Error (MSE) over all training outputs y_i and corresponding outputs

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{m}\sum_i (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2, \tag{2.14}$$

which is suitable for regression problems [31].

2.4.3 Backpropagation

To minimize the loss the weights in the network need to be updated. This is done using gradient descent, which says that the weights should be updated in the direction of the negative gradient of the loss function with respect to the weights. The weights $\mathbf{W}^{[n]}$ in layer n are updated as

$$\mathbf{W}^{[n]} = \mathbf{W}^{[n]} - \alpha \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{W}^{[n]}}, \tag{2.15}$$

with the learning rate α as a small positive parameter [5].

The update to the last layer $\mathbf{W}^{[N]}$ in a network is calculated with a single application of the chain rule to equations 2.14 and 2.13. Since the network layers are connected the gradient of the loss with respect to the previous layers $\mathbf{W}^{[N-1]}$ depend on the gradient with respect to $\mathbf{W}^{[N]}$. Therefore, all updates can be calculated with the repeated application of the chain rule. This propagates the loss backwards through the network, and is referred to as backpropagation. Figure 2.8 pictures the flow of backpropagation in a small network.

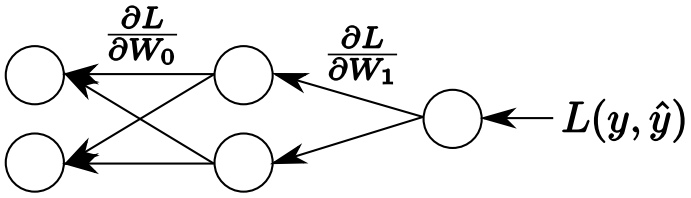


Figure 2.8: Motivation for backpropagation. Due to the layered network structure, the gradient of the loss with respect to the first layer is dependent on the gradient with respect to the last layer. The loss can therefore be seen as propagating backwards through the network structure with the repeated application of the chain rule.

2.5 ODENet

The feedforward process for a single ResNet block, described in section 1.1, is

$$\mathbf{x}_{t+1} = \mathbf{x}_t + hf(\mathbf{x}_t, \theta_t), \quad h = 1 \tag{2.16}$$

where $\mathbf{x}_t$ is the input to the current block, $f(\mathbf{x}_t, \theta_t)$ is a function parametrized by θ_t and $\mathbf{x}_{t+1}$ is the layer output [6], as seen in figure 1.1. This can be seen as a single step of forward Euler discretization (see section 2.3)

$$y_{i+1} = y_i + hF(t_i, y_i) \tag{2.17}$$

of the ODE $\frac{\mathrm{d}y_i(t)}{\mathrm{d}t} = F(t_i, y_i)$ [32].

The notion of Neural Ordinary Differential Equations (ODENet) was introduced by Chen et al [9] based on the observation that when the number of layers n in the

network is increased and the step size h is decreased, in the limit $n \to \infty, h \to 0$ the continuous derivative can be parametrized by a neural network

$$\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = f(\mathbf{y}(t), t, \theta). \tag{2.18}$$

Given an initial condition $\mathbf{y}(0)$ the output from ODENet $\mathbf{y}(T)$ can be specified as the solution to the ODE at time T. The output can be computed with any desired accuracy using any numerical ODE solver as

$$\mathbf{y}(T) = \mathbf{y}(0) + \int_0^T \frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t}\,\mathrm{d}t = \mathbf{y}(0) + \int_0^T f(\mathbf{y}(t), t, \theta)\,\mathrm{d}t. \tag{2.19}$$

An overview of ODENet is shown in figure 1.2. Each step taken by the ODE solver can be seen as a layer output in ODENet, similar to how the step through each block in ResNet can be seen as a step according to Euler's method for solving differential equations. Hence, ODENet can be seen as a continuous-depth network, where at runtime the ODE solver decides how many layers, representing steps in the ODE solver, are required to forward pass through in order to get the output with the desired accuracy. Similar to numerically solving differential equation taking many small steps in the approximation increases the accuracy compared to taking one large step.

2.5.1 Backpropagation using the adjoint sensitivity method

The main difficulty of using a neural network parametrization of an ODE is the backpropagation through the ODE solver. Even though backpropagating through the ODE solver steps is straightforward, the memory cost is high and additional numerical errors occur. In order to perform backpropagation the gradient of the loss function with respect to all parameters must be computed. As different ODE solver may take a varying number of steps when numerically integrating between two time points, a general method for computing the gradient of the loss at each intermediate step is required. Chen et al. introduce a solution to this problem using the adjoint sensitivity method [33, 34] which can be used regardless the choice of ODE solver and with constant memory consumption. This section gives an overview of how the adjoint sensitivity method for backpropagation works. A proof of this method is given by Chen et al. in appendix B of their paper [9].

The adjoint method works by constructing the so called adjoint state $\mathbf{a}(t) = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}(t)}$ where L is the loss function and $\mathbf{y}(t)$ is the output after each step taken by the ODE solver, which follows the differential equation

$$\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = f(\mathbf{y}(t), t, \theta). \tag{2.20}$$

Here $f(\mathbf{y}(t), t, \theta)$ is a parametrization of the time-derivative and θ are the parameters. It can be shown that the adjoint state follows the differential equation

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)\frac{\partial f(\mathbf{y}(t), t, \theta)}{\partial \mathbf{y}(t)}. \tag{2.21}$$

For a proof of this, see appendix B.1 in [9].

We can then solve the differential equation backwards in time, i.e. from the final output time t_N to the starting time t_0, similar to regular backpropagation. Hence we acquire the gradients with respect to the hidden state at any time as

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}(t_N)} = \mathbf{a}(t_N) \tag{2.22}$$

and

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}(t_0)} = \mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t}\,\mathrm{d}t = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{y}(t), t, \theta)}{\partial \mathbf{y}(t)}\,\mathrm{d}t. \tag{2.23}$$

When using multiple steps in an ODE solver we can simply integrate backwards in time between each timestep taken in order to get the gradient at all the different time steps, i.e. from t_N to t_{N-1}, then from t_{N-1} to t_{N-2} and so on, and summing up the gradients after each solved step.

When backpropagating we also need the gradients with respect to t_0, t_N and θ. In order to do this, we view t and θ as states with constant time derivatives and set

$$\frac{\partial t(t)}{\partial t} = 1, \quad \frac{\partial \theta(t)}{\partial t} = \mathbf{0}. \tag{2.24}$$

These can be combined with $\mathbf{y}(t)$ to form the augmented state

$$f_{\text{aug}}([\mathbf{y}, \theta, t]) = \frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\mathbf{y}\\ \theta \\ t\end{bmatrix} := \begin{bmatrix} f(\mathbf{y}(t), t, \theta) \\ \mathbf{0} \\ 1 \end{bmatrix} \tag{2.25}$$

and the augmented adjoint state

$$\mathbf{a}_{\text{aug}}(t) = \begin{bmatrix}\mathbf{a}\\ \mathbf{a}_\theta \\ \mathbf{a}_t\end{bmatrix}(t), \quad \mathbf{a}_\theta(t) = \frac{\mathrm{d}L}{\mathrm{d}\theta(t)}, \quad \mathbf{a}_t(t) = \frac{\mathrm{d}L}{\mathrm{d}t(t)}. \tag{2.26}$$

We then obtain the ODE

$$\frac{\mathrm{d}\mathbf{a}_{\text{aug}}(t)}{\mathrm{d}t} = -\begin{bmatrix}\mathbf{a}(t) & \mathbf{a}_\theta(t) & \mathbf{a}_t(t)\end{bmatrix}\frac{\partial f_{\text{aug}}(t)}{\partial[\mathbf{y}, \theta, t]} \tag{2.27}$$

where $\frac{\partial f_{\text{aug}}(t)}{\partial[\mathbf{y},\theta,t]}$ is the Jacobian of the augmented state

$$\frac{\partial f_{\text{aug}}(t)}{\partial[\mathbf{y}, \theta, t]} = \begin{bmatrix}\frac{\partial f}{\partial \mathbf{y}} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}(t). \tag{2.28}$$

The first element of equation 2.27 is as expected what is shown in equation 2.21. The gradient with respect to the model parameters $\frac{\mathrm{d}L}{\mathrm{d}\theta}$ can be acquired by setting $\mathbf{a}_\theta(t_N) = \mathbf{0}$ and integrating the second element of equation 2.27 backwards in time. We then acquire

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \mathbf{a}_\theta(t_0) = -\int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{y}(t), t, \theta)}{\partial \theta}\,\mathrm{d}t. \tag{2.29}$$

Finally we acquire the gradients with respect to t_0 and t_N as

$$\frac{\mathrm{d}L}{\mathrm{d}t_N} = \mathbf{a}_t(t_N) = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}(t_N)}\frac{\mathrm{d}\mathbf{y}(t_N)}{\mathrm{d}t_N} = \mathbf{a}(t_N)f(\mathbf{y}(t_N), t_N, \theta) \tag{2.30}$$

and

$$\frac{\mathrm{d}L}{\mathrm{d}t_0} = \mathbf{a}_t(t_0) = \mathbf{a}_t(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{y}(t), t, \theta)}{\partial t}\,\mathrm{d}t. \tag{2.31}$$

Hence we can acquire all the gradients needed to backpropagate through the ODE solver by solving the ODE for the augmented state backwards in time. If several steps are taken between the times t_0 and t_N, the gradients can simply be calculated by solving the augmented ODE backwards between each time step in the same way as for the adjoint state, i.e. from t_N to t_{N-1}, then from t_{N-1} to t_{N-2} and so on, and summing up the gradients after each solved step.

2.5.2 Using ODENet to model dynamical systems

Since ODENet approximates the derivative of a system, it can be used as a tool for modelling dynamical systems. Given a time series data set described by an ODE, one can learn the underlying dynamics of the system by integrating the model between any two data points in the data set and then backpropagating to update the model parameters. As the model is continuously defined, it can be integrated between any two time points, eliminating the common problem of modelling data with irregular time steps. With a black box ODE solver, the approximated ODE can be integrated both forward and backward for any arbitrarily long time step given an initial condition $\mathbf{y}(0)$.

Chen et al. propose a model for latent-variable time series prediction using a recognition RNN and training the model as a variational autoencoder. This is a useful tool for time series prediction, but in this thesis we want to investigate using ODENet to directly learn the dynamics of the data set. The main reason behind this choice was to be able to investigate the actual learned dynamics of the neural network. The methods used to achieve this are further explained in chapter 3.

2.6 Finding analytical solutions to ODEs

The ANN block of ODENet numerically encodes the local derivative of the system on which it has been trained. This means that we get a numerical value for the local derivative at any point. Since the dynamical system is described by a corresponding ODE, finding an analytical expression for it is equivalent to extracting a closed-form expression for the derivative $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}(t) = F(\mathbf{x}(t), t)$ parameterized by the network. Since we know the numerical inputs $\mathbf{y}$ with corresponding output $\mathbf{y}'$ finding the closed-form expression can be seen as a function fitting problem. We rely on two different methods for finding these expressions: least squares approximation and linear genetic programming.

2.6.1 Least squares approximation

Least squares approximation is a method used to find the best approximate solution to an over-determined system. An equation system $A\mathbf{x} = \mathbf{b}$, where $A \in \mathcal{R}^{m \times n}$, is over-determined if $m > n$. It is generally not possible to find a vector $\mathbf{x}$ which exactly solve these equation systems. The least squares solution is the vector $\mathbf{x}$ which minimizes the residual vector $\mathbf{r} = A\mathbf{x} - \mathbf{b}$. Hence the least squares solution is defined as

$$\arg\min_{\mathbf{x}} ||A\mathbf{x} - \mathbf{b}||_2. \tag{2.32}$$

The most common ways of solving this problem are using QR decomposition or SVD decomposition [35].

2.6.2 Linear genetic programming

Linear genetic programming (LGP) is a stochastic evolutionary algorithm used for evolving programs consisting of multiple simple instructions. Figure 2.9 illustrates how these programs work. The general idea behind LGP is to initialize a "population" of programs and then letting this population "evolve" over time, evaluating how well the programs perform on the specified task at each time step by decoding the program and calculating its error. The process of decoding an individual is shown in figure 2.10. The programs then evolve each iteration using techniques such as crossover, the process of swapping parts of the program in two individuals, and mutations, randomly changing one or several instructions in an individual. A more in-depth description of LGP can be found in [11].

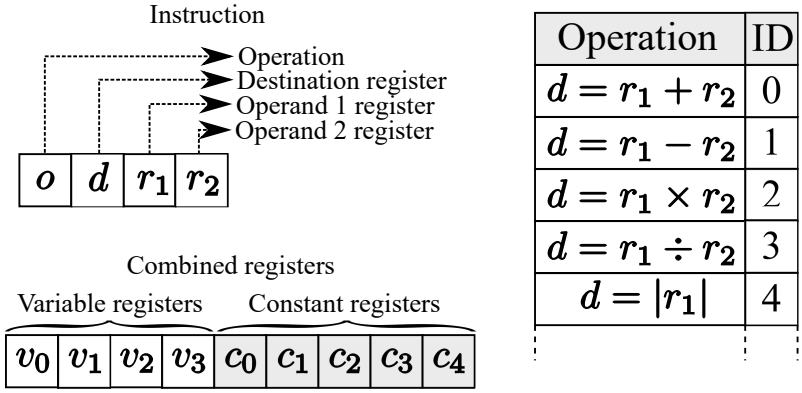


Figure 2.9: The main components of the LGP algorithm for trying to find the function $y = f(x_0, x_1, \dots)$. Each individual contains a set of instructions, which each are comprised of 4 integers. The first integer in each instruction contains the ID of the operation to perform, the second integer contains the destination register and the final two integers contain the registers on which to perform the operation. There are a set number of variable registers in which the output of an operation can be saved, and a set number of constant registers which may not be changed. When initializing the decoding each variable $x_0, x_1, \dots$ are put in the variable registers, and the remaining variable registers are set to zero. The number of operations which can be performed is decided before running the algorithm, and the operations are not bound to the ones shown in this figure. The output can be selected to be any variable register as long as the same register is used for all individuals and throughout all generations.

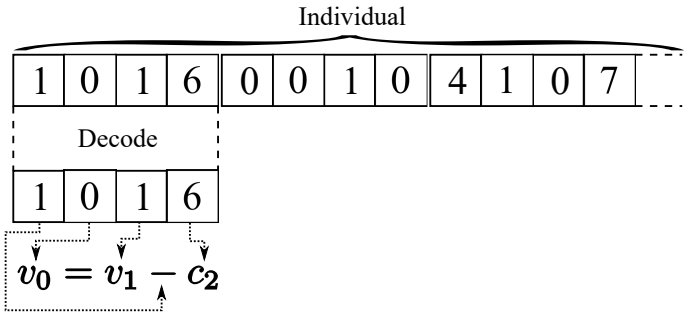


Figure 2.10: An example of decoding the first instruction in an individual. The first integer in the instruction contains the operation to be performed, the second integer contains the destination register and the final two operations contain the operand registers. The registers as well as operations for this example are shown in figure 2.9. Note that the destination register always must be a variable register, as the constant registers are not allowed to be changed. Each instruction is decoded in order from left to right.

3
Methods

The following chapter describes the architecture of ODENet along with how it was trained, and how the the model was evaluated. The first section outlines how we synthesized the datasets used in the thesis. Following that is a description of the network architecture showing how training was performed and the corresponding split of data into training and validation sets. We also talk about design choices such as batch size and weight initialization. The last two sections treat the extraction of a closed-form expression of the learned ODE from the trained network, and how the performance of our model was evaluated.

All methods were implemented in the Python programming language using the PyTorch machine learning library [36].

3.1 Data

The datasets used for training ODENet were composed of M different time series with states $\mathbf{y}(t)$ at time t from initial time t_0 to final time t_T, on the form $Y_m = \{\mathbf{y}(t_0^{[m]}), \mathbf{y}(t_1^{[m]}), ..., \mathbf{y}(t_{T_m}^{[m]})\}$. Both initial and final times, and the time steps between points varied between time series. The time series were represented as a set of M trajectories

$$\mathcal{M} = \{v_1(\mathbf{y}(t), t), ..., v_N(\mathbf{y}(t), t)\}, \tag{3.1}$$

with the evolution of the trajectories described by the same shared global dynamics that we were trying to learn. Therefore, each trajectory $v_m(\mathbf{y}(t), t)$ was determined by a local initial condition $\mathbf{y}(t_0^{[m]})$ at time $t_0^{[m]}$ under the global dynamics. All datasets were synthesized.

3.1.1 Dataset synthetization

Each dataset was synthesized by solving initial value problems using an ODE solver. This is illustrated in figure 3.1. We set up the true ODE and chose a set of initial values that determine the trajectories. Initial values were selected from a range for each variable, and either sampled uniformly at random or spaced linearly. The solver was then used to solve the ODE starting from the initial condition at predetermined sample times forming a time series. The time series were configured by choosing a time span forward, and the number of equally spaced time steps that the trajectory was to be be sampled at. After a trajectory had been generated, we had the possibility to add noise sampled from a Gaussian distribution with a chosen mean and

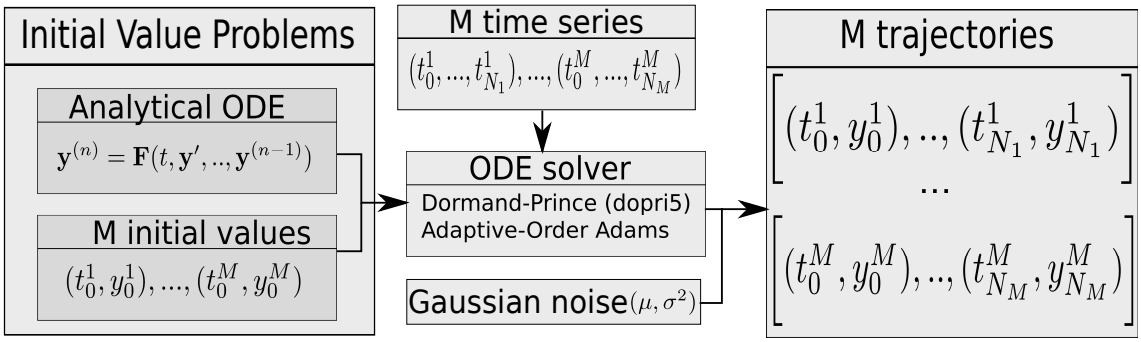


Figure 3.1: Flowchart depicting the method with which data was synthesized. An initial value problem is constructed from a known ODE that the network should try to model, along with M initial values (t_0^i, y_0^i). The initial value problem is then solved by an ODE solver, in our case the Dormand-Prince method or Adaptive order Adams, at points given by a time series for each initial value. The output from the ODE solver is a set of M trajectories, to which we can add noise sampled from a chosen Gaussian distribution.

variance. Lastly, we could remove randomly chosen points from the dataset to make the time steps between points unequal.

For second-order ODEs, such as parabolic motion, we saved both the coordinates (zeroth order derivative), and velocities (first order derivative) at each time step. In principle, the velocity could be approximated from the positional coordinates and the time, but in a experimental setup velocity would presumably be recorded along with the position. Therefore, we chose to assume that the instantaneous velocity would be available at each time step. An alternative would have been to let the network try to learn the dynamics only from positional coordinates along with the time step, but this is assumed to be a more difficult problem and lead to worse approximations.

3.2 ODENet

The full ODENet architecture was composed of two parts. The first was a standard fully connected feedforward neural network, followed by a numerical ODE solver. Figure 3.2 outlines the architecture. These three design parameters: number of hidden layers, number of neurons, and type of activation function were varied to test how they affected performance. We tested different sizes for the network, between one to five hidden layers, and with between 10 to 50 neurons in each layer. The default architecture had three hidden layers with 50 neurons in each layer, with Leaky ReLU activation functions. This network structure was chosen as it performed well on each of the examined problems and as the change in computational time did not increase significantly compared to the smaller network structures.

All layers were connected with Leaky ReLU activation functions. The neural network takes the state $\mathbf{y}_i$ as input and outputs an approximation of the time derivative $\mathbf{y}_i'$. This was then used in an ODE solver to approximate the next state $\hat{\mathbf{y}}_{i+1}$. We used the ODE solvers implemented for GPU support by the authors of the original

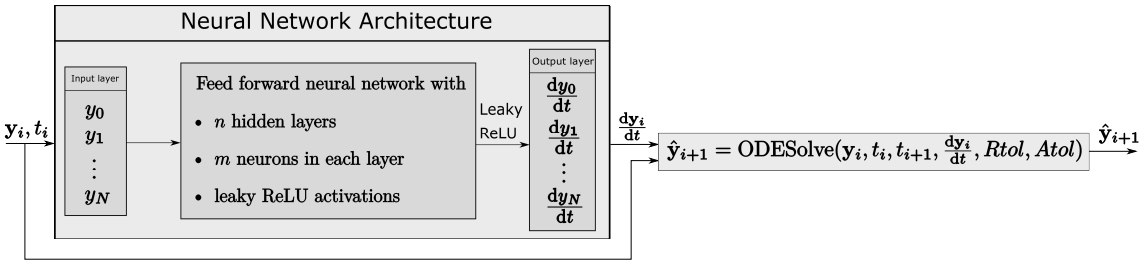


Figure 3.2: Illustration of the ODENet architecture used in the thesis. The time derivative is approximated by a regular feed-forward network with Leaky ReLU as activation function between layers. The input to the neural network was a state $\mathbf{y}_i$, with the corresponding network output being the approximated local time derivative $\mathbf{y}'_i$. Both the input and the approximated derivative was then fed to an ODE solver to get the approximated next state $\hat{\mathbf{y}}_{i+1}$

paper, published in their Github repository [9, 28].

The default solver was the Dormand–Prince (dopri5) method with the alternative Adaptive-order implicit Adams (adams), both described in section 2.3. There were also other fixed-step methods that could be used, but the fixed step size may be a disadvantage for many problems. The choice of solver could be important, depending on the problem. For example, so-called stiff ODEs tend to be a problem for the Dormand-Prince method. Solutions with certain solvers, such as the explicit Dormand-Prince solver, can take a very long time while for example the implicit Adaptive-order Adams might be better, see section 2.3.1.

3.2.1 Training and validation

During training the data was split into two sets, a training set and a validation set. The training set was used to update the weights in the network while the validation set was used for monitoring overfitting and how well the training generalised.

Training was performed by randomly selecting a point in the training set and predicting the next point forward its time series, corresponding to a step in the ODE solver with the derivative approximated by the neural network. From the training point $\mathbf{y}_i$ at time t_i we evaluated the trajectory forward to time t_{i+1} with the ODE solver to get the prediction for the next point $\hat{\mathbf{y}}(t)$. This was then compared to the true target point $\mathbf{y}(t)$. The loss was calculated as the mean squared error (MSE) between the true point $\mathbf{y}(t)$ and the predicted point $\hat{\mathbf{y}}(t)$. We utilized batching, with a default batch size of 10 samples per batch. The batch size was chosen due to the relatively low amount of training data and the initial observation that large batch sizes lead to the network loss oscillating during training. It is plausible that different problems benefit from different batch sizes. The loss over a batch of size B was calculated as

$$L(\mathbf{y}(t), \hat{\mathbf{y}}(t)) = \frac{1}{B} \sum_{i=1}^{B} \left(\mathbf{y}_i - \hat{\mathbf{y}}_i \right)^2. \tag{3.2}$$

Since the prediction was the next point in a series we could see the edges between points as the real members of the training and validation sets instead of the points

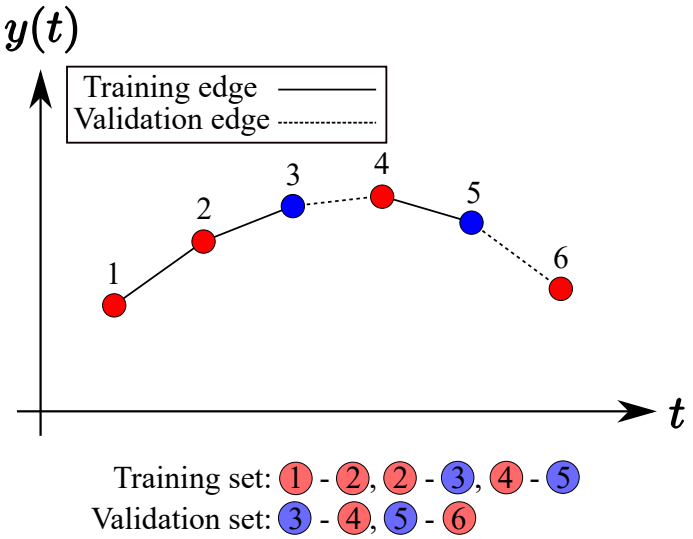


Figure 3.3: Illustrating the split of data into a training set and validation set. In the figure the red points are training points and the blue are validation points. Since the loss for a training point $\mathbf{y}_i$ is calculated as the MSE between the true next point $\mathbf{y}_{i+1}$ and the predicted $\hat{\mathbf{y}}_{i+1}$ training can be seen as being performed on edges between points, and not the actual points. This means that points can belong to both the training set and validation set, but either as an end point or a starting point. The actual step between points is only used once, either in training or in validation.

themselves. This is shown in figure 3.3. Each point can belong to both the training set and validation set at the same time, but only either as an end point or a starting point for a step. The subset of data points not chosen for training was used for validation. The validation loss was calculated in the same way as for training, but for all points after each epoch instead of in batches.

3.2.1.1 Weight initialization

For normal neural networks proper initialization of weights can speed up training performance and avoid exploding or vanishing gradients [8]. In the case of ODENet a bad approximation of the time derivative from the feedforward network could lead to instability in the ODE solver. If the approximation changes rapidly even though the true derivative is smooth, the solver will be forced to take extremely small steps emulating what happens in the case of a stiff ODE. This will lead to each training step taking a very long time. Hence, proper initialization is important.

For initialization we used Glorot initialization [8]. This leads to the weights being initialized sampled from a small Gaussian distribution centered around zero. The motivation was that this could reduced the chance of dramatic variations in the output until the network has had time to train and stabilize the approximated derivative.

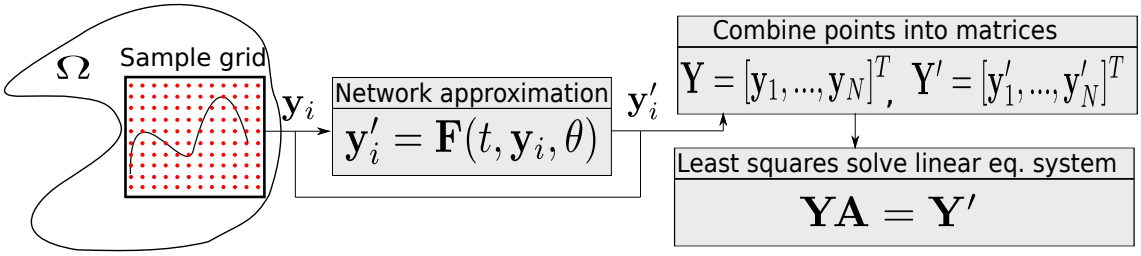


Figure 3.4: Flowchart depicting the method used to solve for a linear coefficient matrix $\mathbf{A}$. Points are sampled in a uniformly spaced grid around the area containing training data. These points $\mathbf{y}_{i}{}_{i=1}^{N}$ are fed through the network to approximate the local value of the time derivative $\mathbf{y}'_i$. The pairs of points and derivatives are combined as rows in a respective matrix and set up as an equation system with $\mathbf{Y}$ as input, $\mathbf{Y}'$ as output and the linear coefficient matrix as unknown $\mathbf{A}$. We then use the method of least squares to solve for $\mathbf{A}$.

3.3 Extraction of ODE coefficients

From a fully trained model we wanted to extract an analytical expression for the ODE governing the training data. We explored two different methods for this: a least squares approach and using linear genetic programming. Both methods have certain strengths and weaknesses. The least squares approach is appropriate for ODEs assumed to be linear, while the linear genetic programming method can find non-linear expression but is not guaranteed to converge to an optimal fit in a reasonable time frame.

3.3.1 Least squares approximation of ODE coefficients

When a model has finished training the network should have learned the dynamics in the training region and approximates the first-order system of ODEs $\mathbf{f}$ mapping the input $\mathbf{y}$ to its derivative

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(\mathbf{y}(t), t, \theta). \tag{3.3}$$

This means that for any point in the input space we could get a pointwise approximation of the ODE from the corresponding output for the feedforward network. The learned ODE could then be visualized by plotting the vector value of f for chosen inputs. This captured the dynamics, but only in terms of a numerical value for each point. But, by utilizing the pointwise knowledge we were able to extract an approximated close-form function assuming that the ODE was linear.

For a nth order order system of ODEs the vector $\mathbf{y}$ contains all variables and their derivatives of order $(n-1)$ in the nth order ODE rewritten as a system of couples first-order ODEs, see section 2.1, with their corresponding derivatives in $\mathbf{y\prime}$. If the ODE system is linear it can be written on matrix form as $\mathbf{y}' = \mathbf{Ay} + \mathbf{b}$. Here, learning the dynamics amounts to extracting the square weight matrix $\mathbf{A}$ and constant vector $\mathbf{b}$ from the local approximations.

To learn $\mathbf{A}$ and $\mathbf{b}$ we created a grid of initial points $\{\mathbf{y}\}_{i=1}^{N}$ in the region of interest, around the area where we had training data. These points were fed as input to the

network which outputted the approximation of the derivatives $\{\mathbf{y}'\}_{i=1}^{N}$. This lead to an overdetermined system of equations, with many more equations than unknowns. Such a system can be solved using least squares approximation. To account for the constant terms we concatenated the vector of constants $\mathbf{b}$ as a last column in $\mathbf{A}$ and added a corresponding row of ones in the input matrix

$$\begin{bmatrix} a_{11} & a_{12} & . & . & a_{1D} & b_1 \\ a_{21} & a_{22} & . & . & a_{2D} & b_2 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{D1} & a_{D2} & . & . & a_{DD} & b_D \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & . & . & \mathbf{y}_N \\ 1 & 1 & . & . & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{y'}_1 & \mathbf{y'}_2 & . & . & \mathbf{y'}_N \end{bmatrix} \tag{3.4}$$

$$\iff \tilde{\mathbf{A}}\mathbf{Y} = \mathbf{Y}'.$$

This equation system was then solved using the least squares method and yielded the coefficients in the $\tilde{\mathbf{A}}$ matrix. This method is pictured in figure 3.4. Since there was no guarantee that the network generalized the dynamics globally we also tried the above approach but only evaluated on points in the actual training data, instead of points randomly sampled in space sometimes far away from the training region. The reasoning was that training points would be where the network had learned the most, which would make our method similar to a regular regression function fitting problem. A potential risk was the increased effect of eventual overfitting.

A downside of the the above method was that it assumed that the underlying ODE was linear and did not, for example, contain any terms raised to a power. We tested a possible approach to extend the method to non-linear ODEs by assuming that some non-linearities exists and appending $\mathbf{y}$ and $\mathbf{y\prime}$ with dummy variables calculated from the existing inputs. For instance, if $\mathbf{y} = (x, x')$ and we assumed that the ODE could contain inputs raised to up to the power of three the new input would have been $\mathbf{y} = (x, x', x^2, (x')^2, x^3, (x')^3)$. A major drawback of this approach is that the dimensionality of the optimization problem problem quickly increases and that there is no easy way to know when the correct system has been found.

3.3.2 Linear genetic programming approximation of coefficients

Linear genetic programming was implemented and used to approximate the coefficients of the learned ODE. Initially the gradient $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}(t)$ is calculated on the entire dataset that the ODENet model has been trained on. Once the gradient is calculated the algorithm tries to find the mapping $f(\mathbf{x}(t), t)$ that approximates the gradient acquired from ODENet. The LGP algorithm is described in section 2.6.2.

One of the motivations for using LGP rather than the least squares method is that non-linearities and powers can be computed without introducing dummy variables. As each program in LGP consists of a set of instructions non-linearities such as $x_0 \times x_1$ can be performed applying multiplication on the two registers containing x_0 and x_1. Similarly, powers can be achieved by performing multiplication on the same variable twice.

The major drawbacks of the LGP algorithm comes from its stochastic nature. Due to the stochasticity there are no guarantees of convergence when function fitting with

LGP. This also causes the drawback of having in some cases a very long runtime. There are however many parameters that can be adjusted in order to achieve faster convergence such as the mutation probability, which controls how often to mutate an individual, and the crossover probability, which controls how often to perform crossover on two individuals. Additionally, the target error can be changed, and hence it is possible to trade off accuracy for run time.

3.4 Evaluation

There were a few different methods that could be used to evaluate how well a model had learned the underlying ODE of the training data. Since we trained on artificial data we had knowledge of the true ODE, which we could use to compare to the output of ODENet. One way this could be done was to generate two trajectories from the same initial value, one using the trained ODENet and the other from the true ODE. The trajectories were then compared.

Another approach utilized only the output from the feedforward part of ODENet, corresponding to the pointwise time derivative. As in section 3.3.1 we evaluated the feedforward network for a grid of points in input space to get the approximated time derivative vector $f(t, \mathbf{y}, \theta)$ at each point, equivelent to the method described in figure 3.4. The same points were then fed through the ODE function used to synthesize the data, which gave us the true time derivative $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}$. By comparing the network output to the true value of the ODE function we got an error value for each point. This allowed us to construct a statistical error metric both on actual training data points, points inside the region that training data resides in (but not the actual points), as well as points outside of this region. In this way we evaluated how well the model learned the general field far away from the training region, in comparison to how well it performs close to the training data. The error was calculated as the relative error

$$\frac{|\mathbf{v}_{\text{true}} - \mathbf{v}_{\text{approx}}|}{|\mathbf{v}_{\text{true}}|} = \frac{|\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} - f(t, \mathbf{y}, \theta)|}{|\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}|} \tag{3.5}$$

We chose the relative error as opposed to the absolute error due to the magnitude of the derivatives varying greatly both in the input space of a single setting, and between different settings. For example, in the case of an harmonic oscillator centered around the origin in phase space the rotation is larger further from the origin. This means deviations that occur at points far from the origin would have an absolute error much larger than deviations close to the origin, even if the relative error is the same. This approach also has the advantage of not relying on the ODE solver, hence removing the error due to the numerical integration.

4
Results and Discussion

In the following chapter we present our results and discuss our findings. The chapter is divided into sections based on the physical problem that is being modelled. Following the problem specific results are a more general discussion about parameter exploration and extending the model for time-dependent ODEs. In the chapter we will report the closed-form ODEs extracted from the trained network as linear combinations of all variables in the system. The coefficients will be shown numerically with implicit dimensions, but they are not dimensionless constants.

4.1 Parabolic motion

For the case of parabolic motion we evaluated ODENet on three different subproblems: an object with no horizontal motion, projectile motion in two dimensions, and projectile motion in two dimensions with a drag term added to model air resistance. Training data consisted of a number of trajectories starting from different initial conditions evaluated for a set time span, or until the trajectory reached the 0 m ground level at which the trajectory was terminated. This means that trajectories could contain different number of points, depending on how far above the ground level the trajectory started. For all simulations the acceleration due to gravity was set to $g = 9.82\,\mathrm{m\,s^{-2}}$.

4.1.1 One-dimensional falling motion

In the one-dimensional case of free fall we trained ODENet on a dataset composed of seven trajectories. The initial heights y were sampled at uniform from 0 m to 10 m, all with zero initial velocity $y' = 0\,\mathrm{m}$. Each trajectory was sampled at 100 points over three seconds of free fall. The ODE governing the system was

$$\begin{aligned} y' &= y' \\ y'' &= -9.82. \end{aligned} \tag{4.1}$$

The input to ODENet was the height y and velocity y' for each training point. The network was trained for 40 epochs using a batch size of 10 points per batch.

Figure 4.1 shows the trajectories in red along with the field produced by the ODE. The green arrows represent the true ODE while the cyan arrows are the ODE approximated by the network. The underlying heatmap measures the relative error between the true and the approximated ODE, as given in 3.4. The network has learned the dynamics well in a region surrounding the training data, but diverges

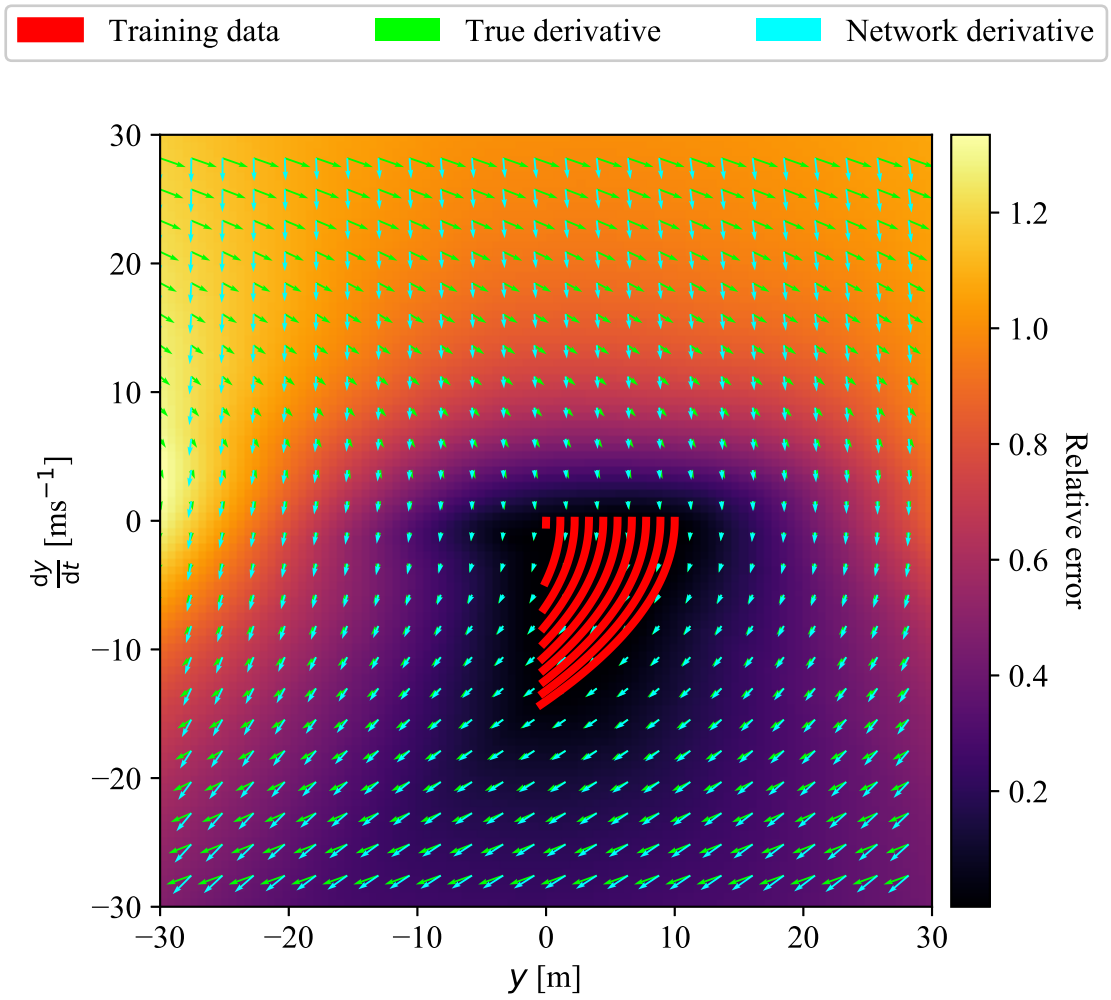


Figure 4.1: The error is low close to the training data, and higher for positive velocities than for negative.

from the true ODE further away. For positive velocities, corresponding to an object being thrown upwards, the error is large. This means that the network has failed to generalize the dynamics, presumably due to the fact that there was no training data in the positive velocity region.

To verify this we trained a new network using training data with nonzero initial velocities sampled in the interval $5\,\mathrm{m\,s^{-1}}$ to $15\,\mathrm{m\,s^{-1}}$, producing the corresponding heatmap in figure 4.2. It is evident that the larger training region produced a larger region in which the network approximates the gradient well. Although, the problem with bad approximations in the positive velocity region was still apparent.

For the first network presented in figure 4.1 we used the least squares method described in section 3.3.1 to extract the coefficients of the linear ODE equation to

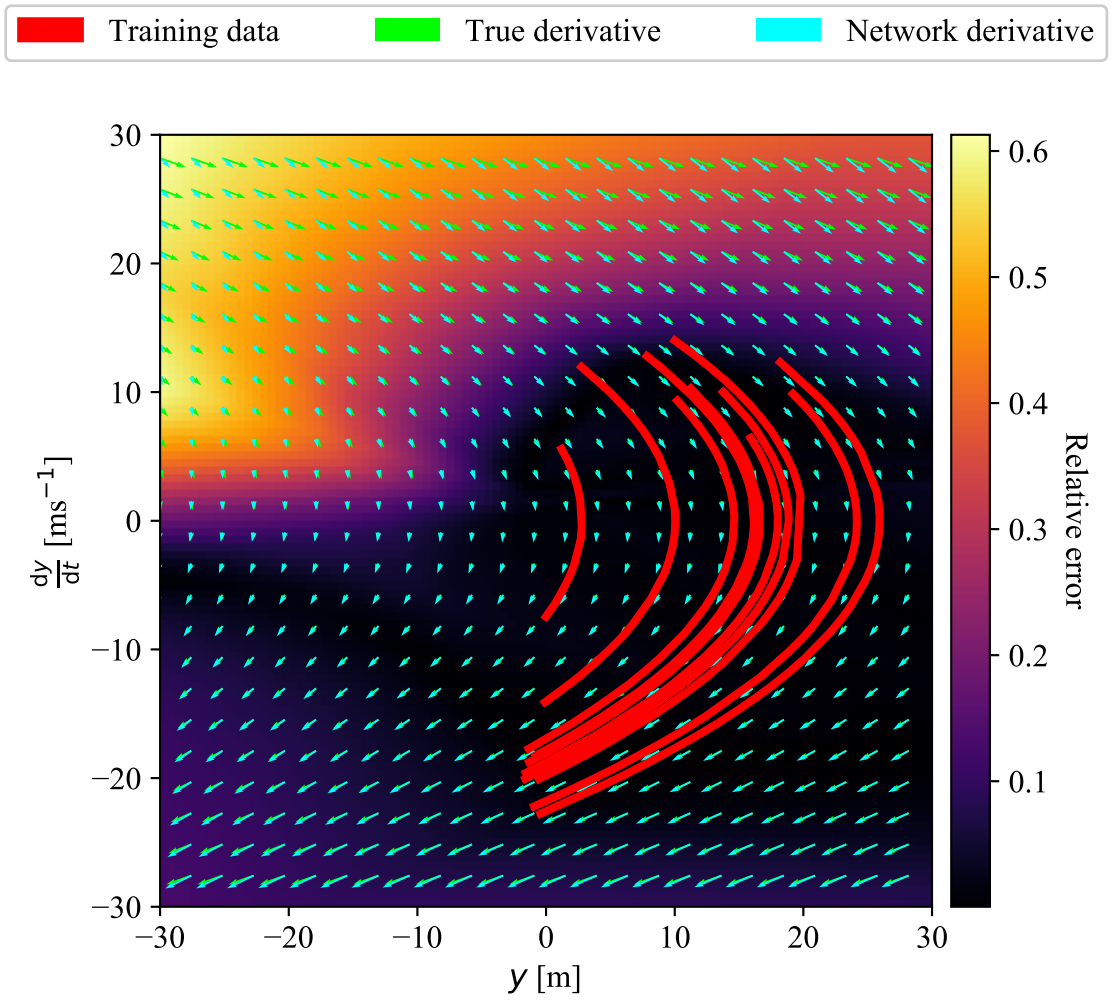


Figure 4.2: Heatmap of the relative error between the ODE describing an object in free fall and the corresponding time derivative approximated by ODENet. The red lines are the extended training data as phase space trajectories starting from different heights and with positive velocities. The cyan and green arrows are field lines for the time derivative learned by ODENet and the true ODE, respectively. The larger dataset produces a larger low-error region, corresponding to a darker background.

get the system of ODEs

$$\begin{aligned} y' &= 0.0009y + 0.9963y' - 0.0097 \\ y'' &= -0.0014y + 0.0033y' - 9.8108, \end{aligned} \tag{4.2}$$

rounded to four decimals, which was close to the analytical system.

This result used a computational region spanned by 0 m to 10 m in height and $-5\,\mathrm{m\,s^{-1}}$ to $0\,\mathrm{m\,s^{-1}}$ in velocity, with 100 point along each axis. This is inside the black colored low-error region in figure 4.1. We repeated the same procedure, but

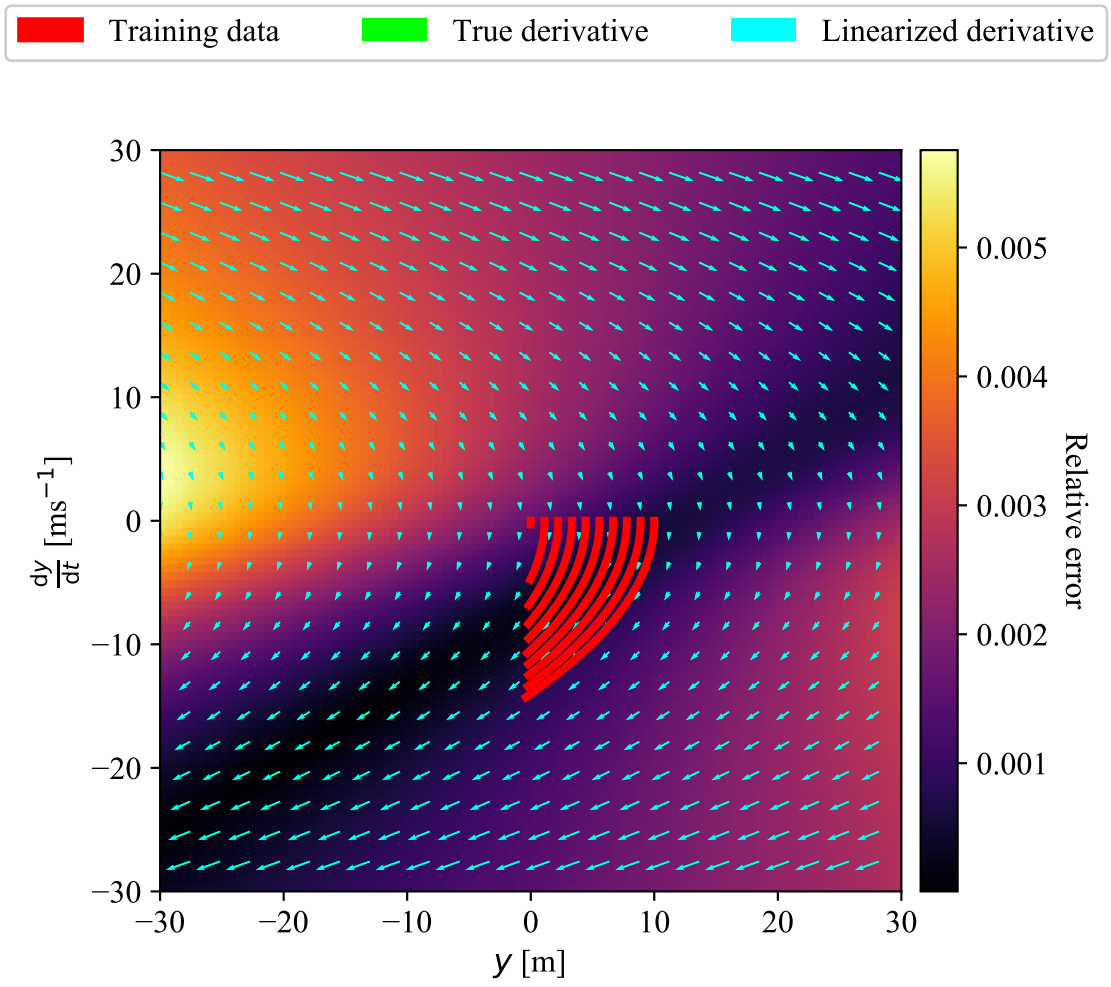


Figure 4.3: Heatmap of the relative error between the ODE describing an object in free fall and the least squares linear approximation extracted from ODENet. The red lines are the extended training data as phase space trajectories starting from different heights and with positive velocities. The cyan and green arrows are field lines for the linear approximation from ODENet and the true ODE, respectively. The magnitude of the error is low globally.

instead sampled in the region bounded by 0 m to 120 m in height and $-50\,\mathrm{m\,s^{-1}}$ to $0\,\mathrm{m\,s^{-1}}$ in velocity. This included the training region, but extended much further in both variables. The results were, rounded to four decimals,

$$\begin{aligned} y' &= 0.0109y + 0.8471y' - 1.6536 \\ y'' &= -0.3528y + 0.1661y' - 8.0959. \end{aligned} \tag{4.3}$$

This diverged from the true system. The least squares approach is sensitive to the choice of region in which it is applied. This is apparent from figures 4.1 and 4.2

where the network is able to generalize in a neighbourhood surrounding the training region.

To try to minimize the impact of how the region for least squares was chosen we decided to primarily make use of only the actual training points when trying to find the linear ODE. The results were similar to when the method was performed in a region surrounding the training data

$$\begin{aligned} y' &= 0.0010y + 0.9981y' - 0.0155 \\ y'' &= -0.0007y + 0.0009y' - 9.8163. \end{aligned} \tag{4.4}$$

A corresponding heatmap comparison of the linearized ODE in the above equation and the true ODE is given in figure 4.3. There are 100 samples per dimension. In comparison to the network approximation of the ODE in 4.1 the linearized ODE is consistent globally. So, even though the network has not managed to correctly learn the underlying dynamics of the ODE everywhere it could be used to compute a better approximation. Histograms corresponding to the relative errors plotted in the heatmaps are presented in figure 4.4. Both panels seem to contain the same peak close to zero. Larger errors are removed by the linearized model, meaning that it produces a more accurate global representation of the underlying system. Although, the linearized model appears to not produce the small tail towards zero as seen for the network approximation. This is probably explained by the points with the best approximations from the network being averaged out by the points with worse approximations in the least squares method.

The approach based on only training data eliminates the need to manually choose a region and utilizes that the network should produce a better fit the closer to the training data it gets. Potential downsides would include risks from overfitting and cases where the amount of training data is very small, although in such cases one could argue that the network would not be properly trained anyway.

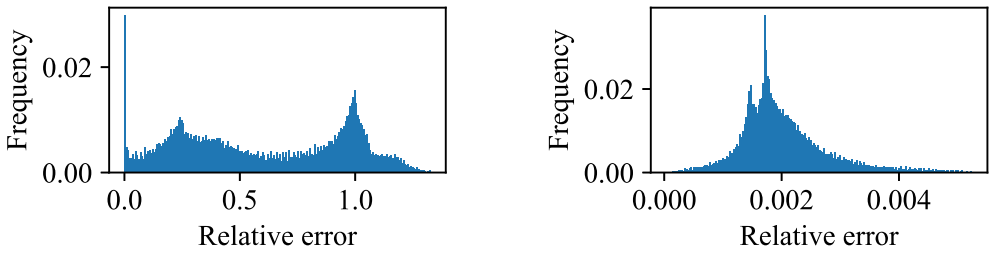


Figure 4.4: Histograms of the relative error of the true ODE for free fall compared to: the derivative learned by the network (left panel), and the extracted linearized ODE using the least squares method on the training data (right panel). Both panels contain the same peak close to zero, but the linearized model removes larger errors.

Even though the underlying dynamics is a simple linear system we tested using LGP to find the analytical solution to the problem. When using LGP we found the

system

$$\begin{aligned} y' &= y' \\ y'' &= -9.8333 \end{aligned} \tag{4.5}$$

rounded to four decimals. The ODE is close to the true equation and the ODE from the least squares method.

4.1.2 Two-dimensional parabola

In the case of two-dimensional parabolic motion training trajectories were still generated starting at different heights y from 0 m to 10 m and zero horizontal displacement $x = 0$, but with the addition of non-zero horizontal and vertical velocities (x', y'). The initial velocities were randomly sampled in the range $-3\,\mathrm{m\,s^{-1}}$ to $3\,\mathrm{m\,s^{-1}}$ horizontally and $0\,\mathrm{m\,s^{-1}}$ to $10\,\mathrm{m\,s^{-1}}$ vertically. Data was generated according to the system

$$\begin{aligned} x' &= x' \\ y' &= y' \\ x'' &= 0 \\ y'' &= -9.82. \end{aligned} \tag{4.6}$$

The inputs to ODENet were the position and velocity (x, y, x', y') and the network was trained for 40 epochs with a batch size of 10.

As in the previous section we used the least squares approach to extract a linear ODE expression from the derivative learned by the network. First, we used a computational grid $x \in [-5, 5]$ m, $y \in]0, 10]$ m, $x' \in (-3, 3)$ ms^{-1}, and $y' \in [-10, 0]$ ms^{-1} with 100 samples along each axis. Although this was inside the training region the resulting system did not match the original ODE

$$\begin{aligned} x' &= 0.2814x + 0.0497y + 0.5709x' - 0.0296y' - 0.6304 \\ y' &= -0.0419x + 0.0728y - 0.1307x' + 0.9577y' - 0.6733 \\ x'' &= -0.0091x + 0.0055y - 0.0027x' - 0.0013y' - 0.0250 \\ y'' &= 0.8962x - 8.444y + 1.1193x' + 0.4077y' - 4.2743. \end{aligned} \tag{4.7}$$

This might be an effect of the sampling size not being large enough. Since the problem is four-dimensional the least squares methods quickly grows in complexity and size a large increase in the number of sampling points would be impractical. Furthermore, it might not even make a difference if the result is due to the training data not being sufficient for the network to learn a generalization in the computational grid used.

Instead, we used the approach to apply the least squares method only on points in the training data where we assumed the network had learned the dynamics. This

produced the following system

$$\begin{aligned} x' &= 0.0022x - 0.0007y + 0.9917x' + 0.0007y' - 0.0093 \\ y' &= -0.0016x + 0.0001y - 0.0008x' + 0.9931y' - 0.0041 \\ x'' &= -0.0004x + 0.0002y - 0.0002x' - 0.0002y' - 0.0009 \\ y'' &= 0.0005x - 0.0013y + 0.0019x' + 0.0009y' - 9.8252, \end{aligned} \tag{4.8}$$

matching the real ODE rather closely.

Figure 4.5 shows the relative error between the true underlying ODE and the approximation from the network, as well as the extracted least squares approximation. The derivatives were compared in a grid bounded by $[-30, 30])$ in all dimensions with 100 sample points per dimension. The ODE from the least squares approach seems to generalize much better than the network approximation. The tail of the histogram for the network approximation continues after the point pictured, but with a very low frequency.

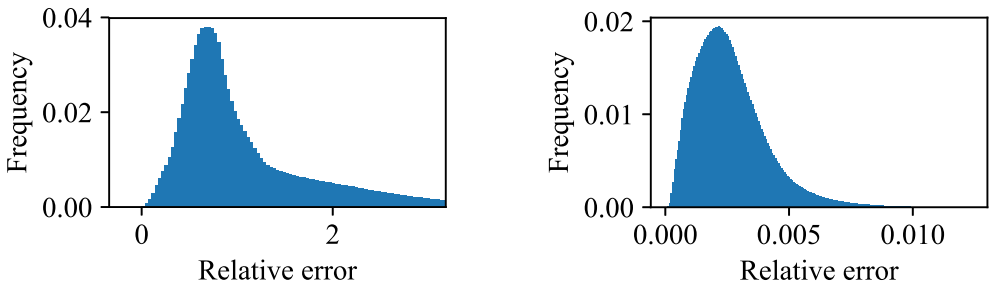


Figure 4.5: Histograms of the relative error of the true ODE for parabolic motion compared to: the derivative learned by the network (left panel), and the extracted linearized ODE using the least squares method on the training data (right panel). Both histograms have similar shape, but the linearized model is squeezed close to zero and does not contain the tail of high magnitude errors that the network approximation exhibits.

We also used LGP to extract a closed-form ODE from the network, resulting in

$$\begin{aligned} x' &= x' \\ y' &= y' \\ x'' &= 0.0033x \\ y'' &= -9.8100 \end{aligned} \tag{4.9}$$

rounded to four decimals. The method finds the relevant variables, with only slight deviations for the accelerations x'' and y''. For the horizontal acceleration x'' LGP includes a term proportional to x. The magnitude of x in the data is smaller than 10, meaning that the error for the data is relatively small. While this could lead to problems when generalizing for very large values of x, this could presumably be removed by forcing a stricter error tolerance in the LGP algorithm.

4.1.3 Two-dimensional parabola with air-resistance

To evaluate our method for non-linear ODEs we added a square drag term to the parabolic motion system resulting in

$$\begin{aligned} x' &= x' \\ y' &= y' \\ x'' &= -0.3448x'|x'| \\ y'' &= -0.3448y'|y'| - 9.82. \end{aligned} \tag{4.10}$$

This was used to generate training data with the same initial conditions as in the previous section. The extra drag extra term also needs two parameters, the drag constant k and the mass m. These were set to $k = 0.05\,\mathrm{kg\,m^{-1}}$ and $m = 0.145\,\mathrm{kg}$, resulting in the proportionality constant $\frac{k}{m} \approx 0.3448\,\mathrm{m}$.

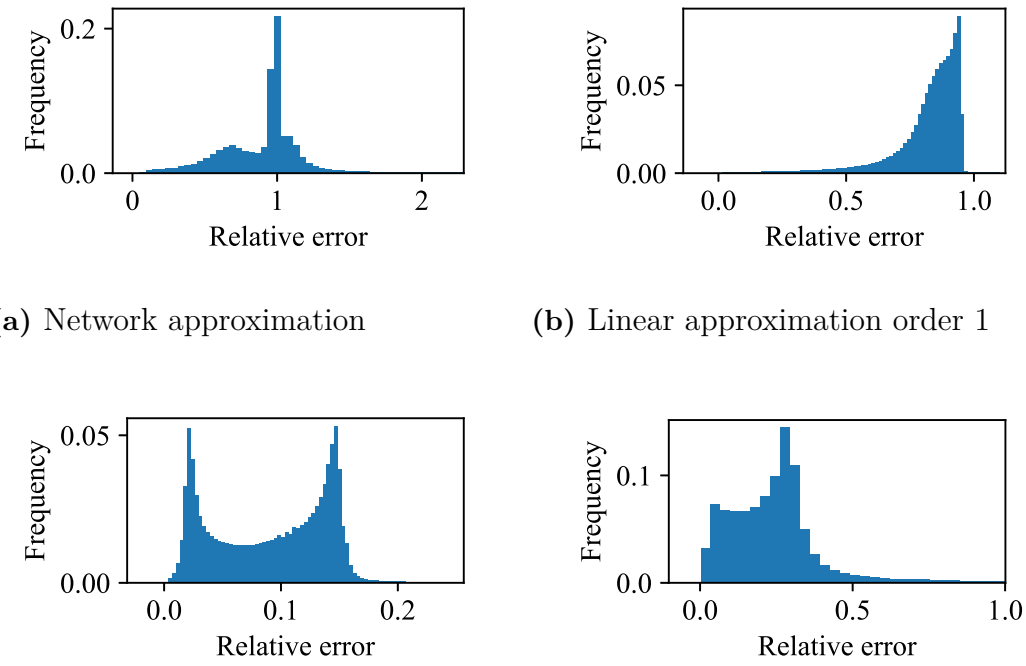
Since the drag term is non-linear, proportional to the square of velocity, the usual method to extract the coefficients does not work, see section 3.3.1. By extending the linear system, adding the square of each variable with the sign preserved as dummy variables, the method can be used. The resulting system rounded to four decimals was

$$\begin{aligned} x' =& 0.0001x - 0.0051y + 0.9883x' - 0.0018y' + 0.0002x|x| \\ &+0.0004y|y| + 0.0075x'|x'| + 0.0004y'|y'| + 0.0094 \\ y' =& 0.0044x - 0.0024y + 0.0179x' + 1.0027y' - 0.0002x|x| \\ &-0.0000y|x| - 0.0094x'|x'| + 0.0007y'|y'| + 0.0042 \\ x'' =& -0.0158x - 0.0086y - 0.0604x' - 0.0092y' + 0.0034x|x| \\ &+0.0012y|x| - 0.2987x'|x'| - 0.0009y'|y'| - 0.0096 \\ y'' =& 0.0063x + 0.0041y - 0.0615x' - 0.0358y' + 0.0002x|x| \\ &-0.0015y|x| + 0.027x'|x'| - 0.3359y'|y'| - 9.7886 \end{aligned} \tag{4.11}$$

This is similar to the true ODE, but with a lot of noise.

By adding dummy variables to the equation the solution becomes dependent on the choice of the dummy variables. A comparison of the the method with variables raised to the power of one, two and three is shown in figure 4.6. The figure shows histograms of the corresponding relative error between the approximations and the true ODE. We see that model with terms of order two, the correct order, produces the error distribution closest to zero. The model with order one performs worse than the model of order three. This is presumably due to the fact that the latter also contains the correct terms of order two, while the former only contains the linearized approximation. Including an order too much would probably result in a better approximation than having the maximum order too low, but at the cost of increased computational complexity of the least square problem.

LGP were better at eliminating the unused variables than the least squares fit in equation 4.11. It found the correct nonlinearity in the expression for x'', but could not find it for y'' meaning it never converged under the set error limit. The reason for this is unknown. The difference between the two nonlinear expressions is only a constant, which the LGP method should be able to find. Different approaches to



(a) Network approximation

(b) Linear approximation order 1

(c) Linear approximation order 2

(d) Linear approximation order 3

Figure 4.6: Histograms of the relative error of the true ODE for parabolic motion with air resistance compared to: (a) the derivative learned by the network, and the extracted linearized ODE using the least squares method on the training data with dummy variables of powers up to order (b) one, (c) two, and (d) three. Order two is the correct number, and also produces the histogram with the smallest errors. Including more then the correct orders, three, produces smaller errors than including fewer orders, one.

alleviate this includes further testing of different parameter setups or using other more advanced LGP implementations. The final system was

$$\begin{aligned} x' &= x' \\ y' &= y' \\ x'' &= -\frac{1}{3}x'|x'| \\ y'' &= \text{NULL}. \end{aligned} \tag{4.12}$$

The best approximation for y'' that was found was

$$y'' = -3 * (y' - \cos(y' - |x'| + 0.1)) * \cos(1) - 10.3348708389442. \tag{4.13}$$

4.2 Harmonic oscillator

We evaluated on two different harmonic oscillator systems: a simple harmonic oscillator and a damped harmonic oscillator. In both cases data was generated modelling an object of mass $m = 1\,\text{kg}$ on a spring with spring constant $k = 2\,\text{kg}\,\text{s}^{-2}$.

4.2.1 Simple harmonic oscillator

In the simple harmonic oscillator case only one trajectory of data was generated. The choice of only generating one trajectory was made as this problem is highly symmetric, to evaluate generalizing over a large area using few data points. The initial values were set to $x(0) = 5\,\mathrm{m}$ and $x'(0) = 0\,\mathrm{m\,s^{-1}}$. The resulting ODE is then

$$\begin{aligned} x' &= x' \\ x'' &= -2x. \end{aligned} \tag{4.14}$$

The inputs to the network were the position of the oscillator $x(t)$ and its velocity $x'(t)$.

We used the least squares linearization method to find the coefficients of the ODE. The resulting ODE was

$$\begin{aligned} x' &= -0.0005x + 0.9998x' + 0.0013 \\ x'' &= -2.0002x + 0.0003x' - 0.0013 \end{aligned} \tag{4.15}$$

when rounded to four decimals.

A heatmap showing the relative error between the network approximation and the true ODE is shown in figure 4.7. As seen in the figure, the network generalizes well outside of the training area, with the exception of the origin. A possible reason for this is that the input to the network at the origin is too small, causing the gradient to disappear, hence giving a bad approximation of the true dynamics. Another possibility is that the origin is the point in which the rotation of the vector field changes. As the network has not been trained on any data in this area, this transition may be poorly approximated by the network, hence causing the error.

This problem does not occur for the linearized approximation of the ODE, as it is calculated on the training data, at which the network behaves well. Hence the linearized approximation generalizes well over the entire extended area.

The two histograms in figure 4.8 show the relative error distributions of the network approximation and the linearized ODE. As seen in the figures, the relative error of the network approximation is rather small. However, the linearized approximation of the ODE still improves the approximation, and its relative error is virtually nonexistent.

When using LGP we found the correct system

$$\begin{aligned} x' &= x' \\ x'' &= -2x. \end{aligned} \tag{4.16}$$

This is exactly the true ODE describing the system.

4.2.2 Damped harmonic oscillator

Five trajectories were synthesized for damped harmonic oscillator. The initial values in the first dimension, $x(0)$, were set to $1, 2, \ldots, 5$ for each trajectory respectively.

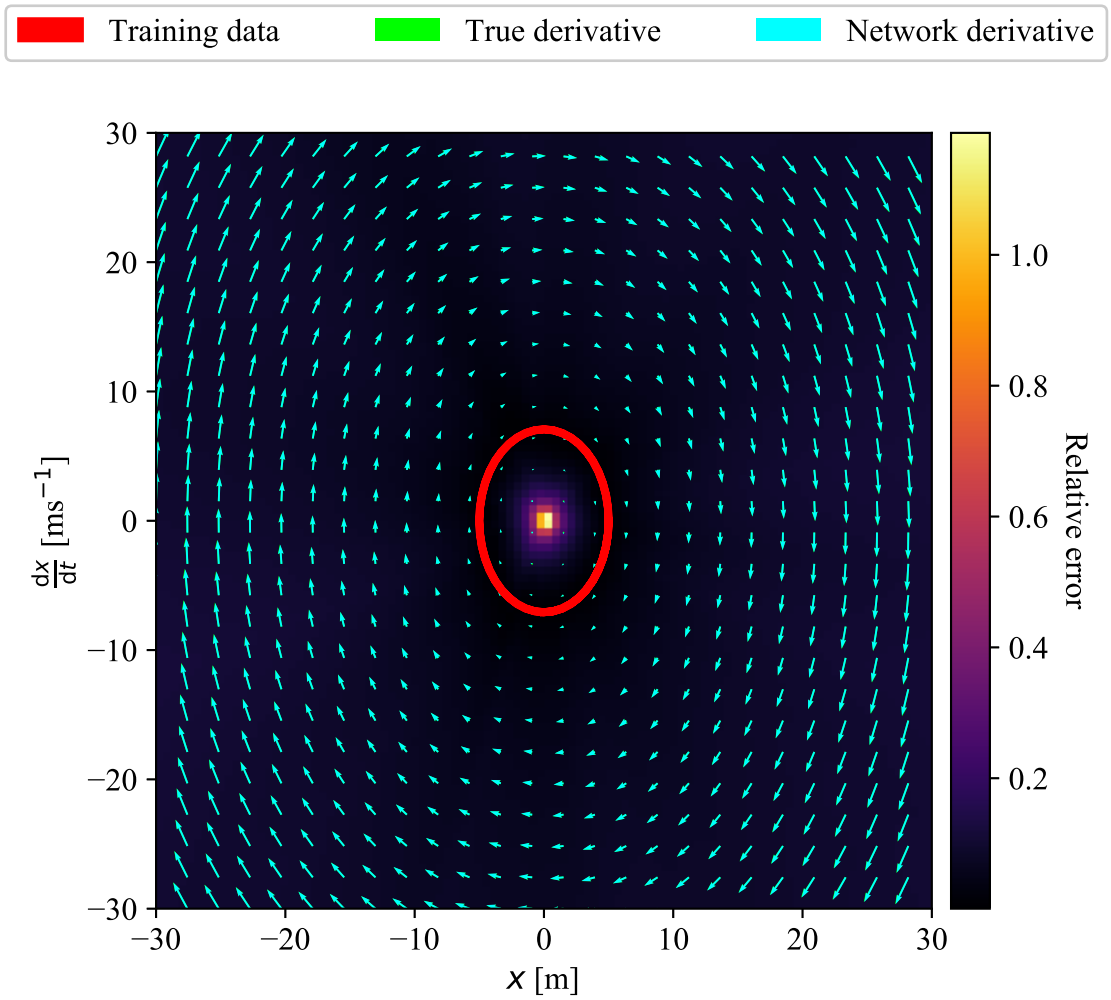


Figure 4.7: Heatmap of the relative error between the ODE describing a simple harmonic oscillator and the ODENet approximation of the same system. The red line is the training data as a phase space trajectory. The cyan and green arrows are field lines for the approximation from ODENet and the true ODE, respectively. The magnitude of the error is low globally, except from in the origin.

For the second dimension, $x'(0)$, the initial values were set to $-1, -\frac{1}{2}, 0, \frac{1}{2}, 1$. The added dampening constant was set to $c = \frac{1}{2}\,\mathrm{kg\,s^{-1}}$. This results in the ODE

$$\begin{aligned} x' &= x' \\ x'' &= -2x - \frac{1}{2}x'. \end{aligned} \tag{4.17}$$

The input to ODENet was $x(t)$ and $x'(t)$, as in the case of the simple harmonic oscillator.

Figure 4.9 shows a heatmap of the relative error between the ODENet approximation of the system and the true ODE. As seen in the figure, the network error is

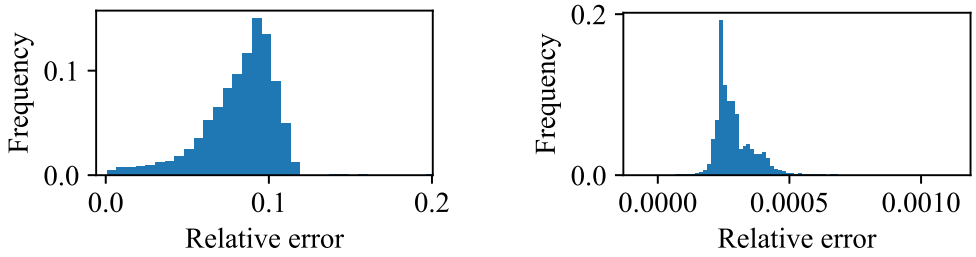


Figure 4.8: Error histograms for the simple harmonic oscillator. The left panel shows the error distribution of the network approximation of the ODE. The right panel shows the error distribution of the linearized ODE approximated on the training data using the least squares method.

relatively low. The heatmap also shows that the error is larger along the minor axis compared to the error along the major axis of the training data. This is likely due to the data being more concentrated in the regions parallel to the minor axis due to the slower rotation along these points. This means that the network has been trained on more densely packed data, and subsequently more data in general, along these axes which results in a better fit on this data.

When linearizing the network approximation using the least squares method at the training data points, the ODE found was

$$\begin{aligned} x' &= -0.0064x + 0.9955x' - 0.0089 \\ x'' &= -1.9941x - 0.5044x' - 0.0068. \end{aligned} \tag{4.18}$$

As seen, this is close to the true ODE describing the system.

Figure 4.10 shows two histograms of the error distributions of the ODENet approximation and linearized approximation respectively. As in the case of the simple harmonic oscillator, the error distribution for the network approximation is relatively small, and the error for the linearized system is close to zero, indicating a good generalization over the entire extended area.

When using LGP we found the system

$$\begin{aligned} x' &= x' \\ x'' &= -2x - 0.5333x' \end{aligned} \tag{4.19}$$

when rounding to four decimals. This is close to the true ODE which is shown in equation 4.17. To get a better, or even exact, function fit the error threshold in the LGP algorithm could be decreased.

4.3 Lotka-Volterra predator-prey equations

For the Lotka-Volterra equations two trajectories of training data were synthesized. The initial states $N(0), P(0)$ were sampled uniformly random in the range $(0, 1)$.

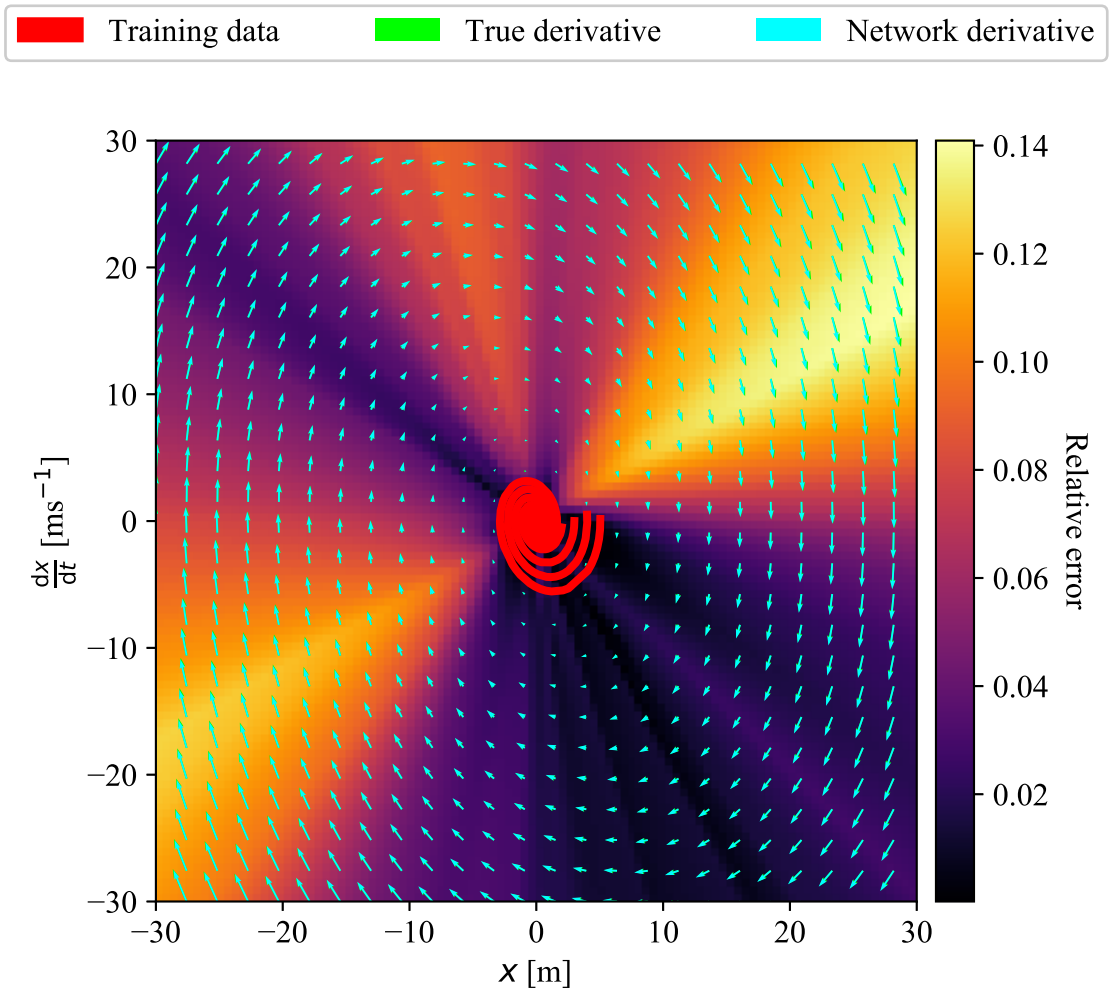


Figure 4.9: Heatmap of the relative error between the ODE describing a damped harmonic oscillator and the ODENet approximation of the same system. The red lines are the training data as a phase space trajectories. The cyan and green arrows are field lines for the approximation from ODENet and the true ODE, respectively. As seen in the figure, the relative error is lower in the direction of the major axis of the data. This is likely due to the higher concentration of data points parallel to the minor axis, as the rotation in this direction is slower than the rotation parallel to the major axis.

The constants in equation 2.7 were chosen as $\alpha = \frac{2}{3}$, $\beta = \frac{4}{3}$, $\gamma = 1$ and $\epsilon = -1$. The resulting ODE describing the dynamics of the system is then

$$\begin{aligned} N' &= N\left(\frac{2}{3} - \frac{4}{3}P\right) \\ P' &= P(N-1). \end{aligned} \tag{4.20}$$

We trained ODENet for 40 epochs on data composed of two different trajectories

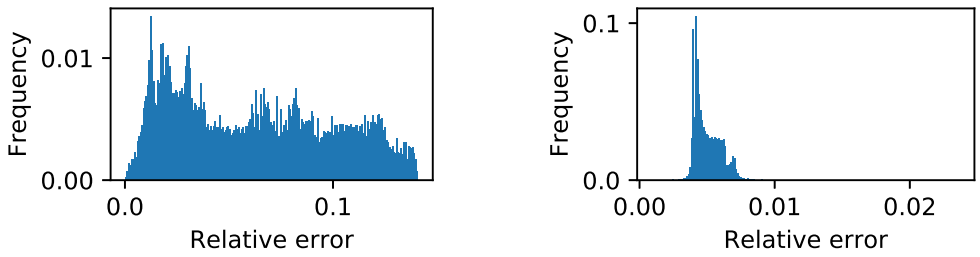


Figure 4.10: Two histograms showing the relative error distributions for the damped harmonic oscillator. The left histogram shows the error distribution of the neural network approximation. The right histogram shows the error distribution of the least squares approximation.

from different initial conditions. Since the system is nonlinear with mixed terms the least squares method can not be applied, at least not without first assuming such nonlinearities. Therefore, we first used LGP, which found the system

$$\begin{aligned} N' &= N(0.6667 - 1.3333P) \\ P' &= |P|(N - 1) \end{aligned} \tag{4.21}$$

when rounding to four decimals. This is the correct system, except for the absolute value operator which has no effect since all values are in the first quadrant and hence already positive. The LGP implementation could be extended to look for equivalent expressions for the data, e.g. by removing the absolute value operator if the data is in the first quadrant.

4.4 Learning time dependencies

All ODEs examined previously has been time independent, with no explicit occurence of time in the system. To learn time dependent behaviour, such as a driving function $F(t)$ for an harmonic oscillator, we had to extend the network. There are different ways to accomplish this. We tried to extend the system of ODEs by adding a time variable following a constant differential equation $\frac{\mathrm{d}t}{\mathrm{d}t} = 1$ and feeding the time as an extra input to ODENet along with the variables in the system. We evaluated the approach on a model for the effects of an earthquake on buildings.

4.4.1 Earthquake Effects on Buildings

Training data was composed of three different trajectories for a building with two floors, sampled over 0 s to 5 s. Both floor started in the equilibrium position $x_1(0) = x_2(0) = 0\,\mathrm{m}$. The initial velocities were sampled at random in the interval $-1\,\mathrm{m\,s^{-1}}$ to $1\,\mathrm{m\,s^{-1}}$, which might not be physically sound. The parameters of equation 2.8 was set to $k = 10\,000\,\mathrm{N\,m^{-1}}$, $m = 1000\,\mathrm{kg}$, $\omega = 0.5\,\mathrm{s^{-1}}$ and $F_0 = 2\,\mathrm{m}$ resulting in

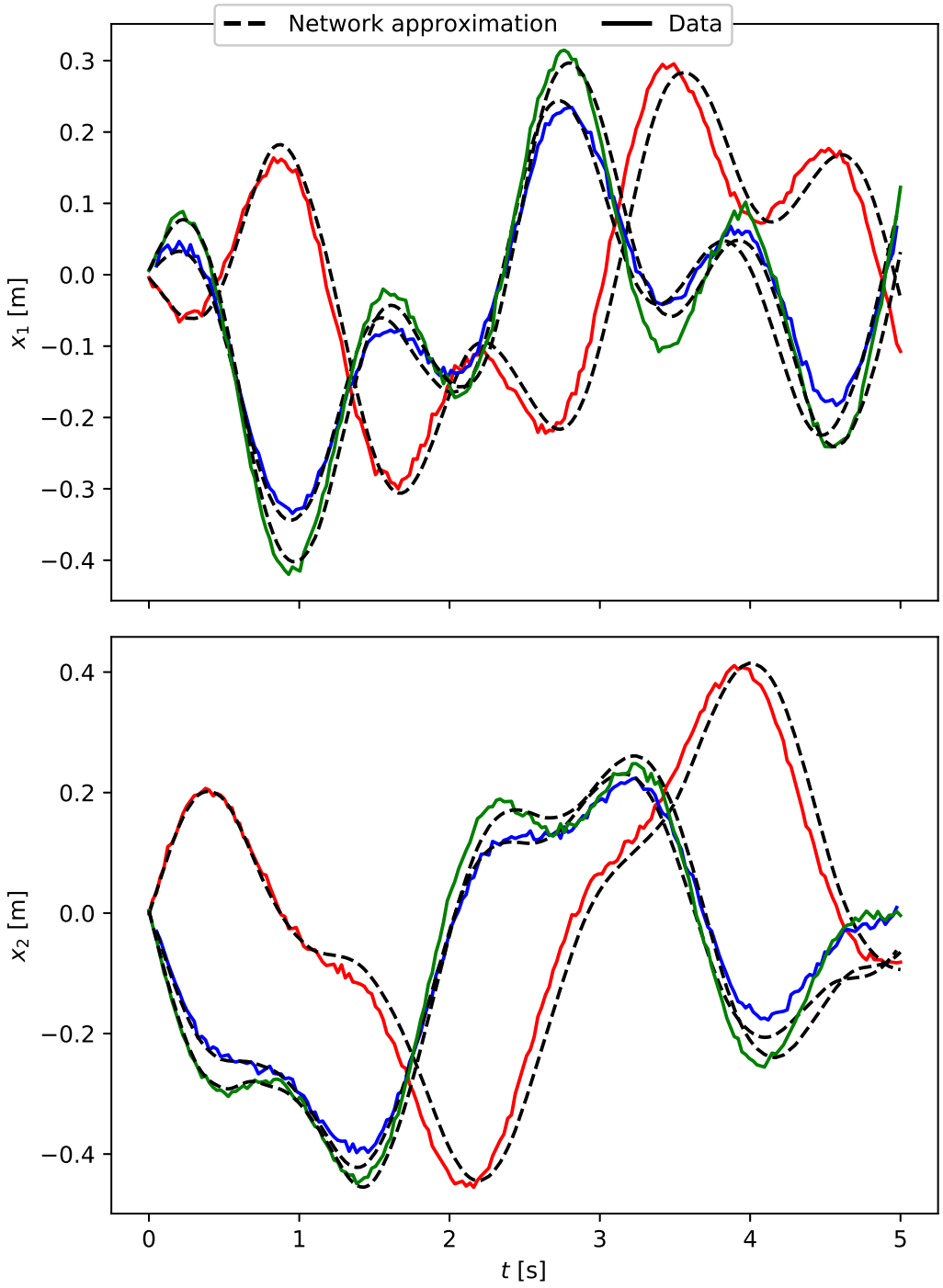


Figure 4.11: The movement over time of a two-story building affected by an earthquake. The upper panel is the position of the first floor x_1, and the lower panel the position of the second floor x_2. Each colored solid line correspond to a single training trajectory. The dashed black lines are trajectories predicted by the ODE learned by the network. The predictions start from the same initial conditions and are sampled at the same time steps as the training trajectories.

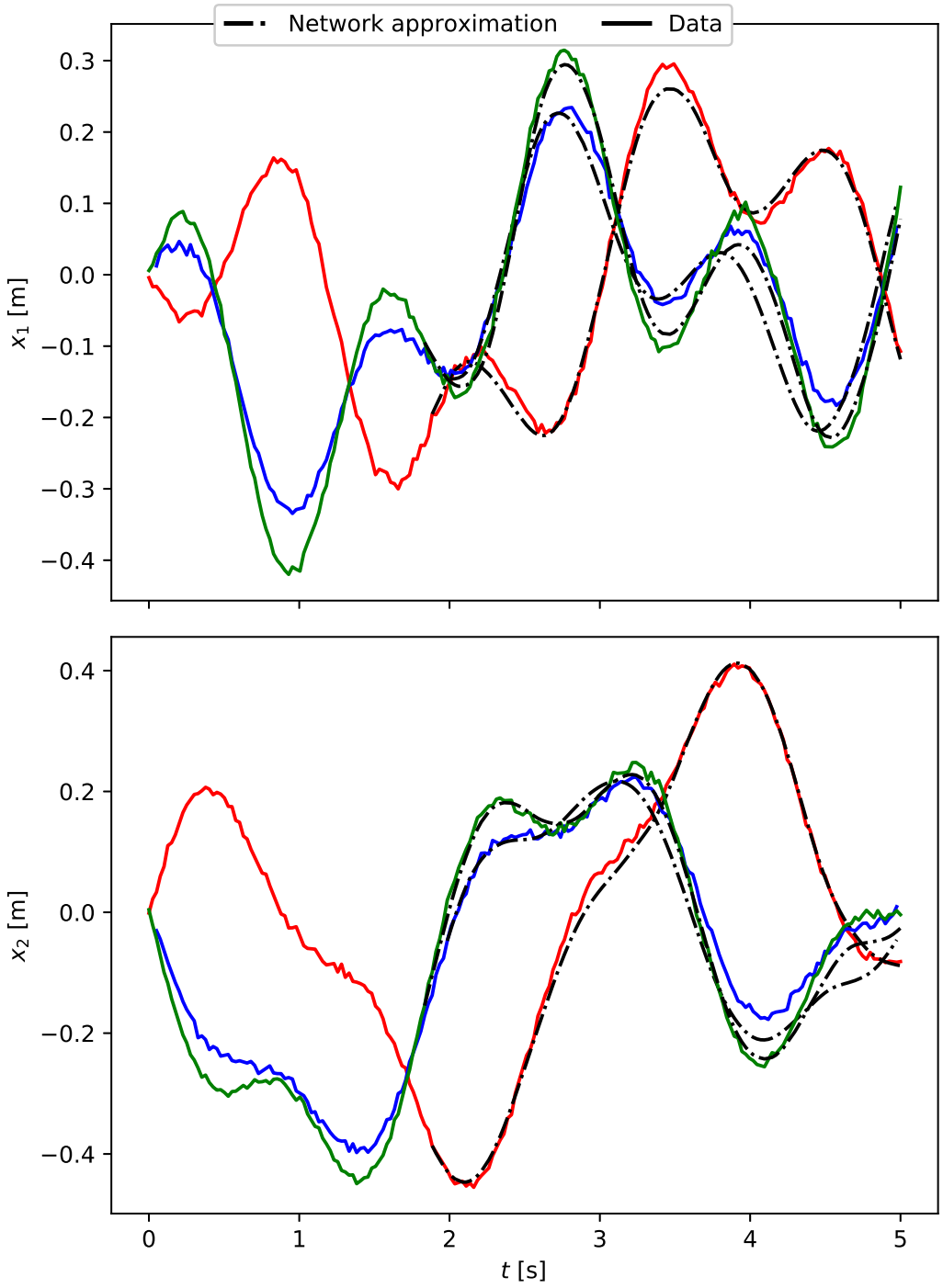


Figure 4.12: The movement over time of a two-story building affected by an earthquake. The upper panel is the position of the first floor x_1, and the lower panel the position of the second floor x_2. Each colored solid line correspond to a single training trajectory. The dashed black lines are trajectories predicted by the ODE learned by the network. The predictions start from points along the training trajectories after about 1.9 s, and are then sampled at the same time steps as the original trajectories.

the system

$$\begin{aligned} x_1' &= v_{x_1} \\ x_2' &= v_{x_2} \\ x_1'' &= 10(-2x_1 + x_2) - \frac{1}{2}\cos(\frac{t}{2}) \\ x_2'' &= 10(x_1 - x_2) - \frac{1}{2}\cos(\frac{t}{2}). \end{aligned} \tag{4.22}$$

We trained the network for 50 epochs. Figure 4.11 shows the position of the two floors over time for the three training trajectories along with the corresponding predicted trajectories from an ODE solver using the derivative approximated by the trained network. The previous approach using the error between the approximated and true ODE could not be used, since we did not produce any closed-form expressions. Since we know that the accuracy of the network approximation is dependent on the measurement region, a single histogram of the error between the true ODE and the network approximation does not give a quantitative measure.

The predicted trajectories are sampled at the same time steps as the training trajectories. Both the training trajectories and the predicted trajectories have the same initial conditions. The initial fit is close to the true trajectories. Over time the deviation grows larger, which is due to the prediction at subsequent steps carrying the error from previous steps. Despite this the prediction is similar to the true data implying that the network has learned a valid approximation for the underlying ODE of the data, at least close to the training data. Figure 4.12 shows plots of trajectories calculated using the same ODENet model but starting from a later time point. This plot shows that the time-offset which can be seen when calculating the trajectories on the entire dataset has decreased, validating that the offset seen in figure 4.11 is indeed due to error propagation when calculating the predicted trajectories. This means that the network is capable to modelling systems directly dependent on time.

4.5 Discussion

When training the models on the datasets, the batching method used was to select batches of subsequent point pairs and training the model on the edges between those points, as described in section 3.2.1. There are however several different ways that the batching can be done. One of these methods is to first select a random starting point and then selecting m subsequent points and training on the edges connecting these points. Another possible method is to view each trajectory as a batch and training the model on each trajectory entirely at once. Both of these methods were implemented in this thesis, but were not used when training the final models, due to giving worse results than the single-pair batching, as well as increasing the computational time of the training.

One of the major issues encountered while training the models comes from irregular occurrences when the gradient approximated by ODENet behaves like a stiff ODE. One attempt to solve this issue was to normalize the training data in order to cause a smoother training curve. This was implemented, but ultimately not used, as

the underlying dynamics of the system is changed when the dataset is normalized. The change to the ODEs describing the system is not trivial, and hence the transformation from the learned dynamics to the true dynamics is not straight forward. Instead, in order to minimize the issues caused by ODENet behaving like a stiff ODE, LeakyReLU was chosen as the activation function for all network layers and weight decay was used.

The choice of using LeakyReLU instead of ReLU as activation function was made to avoid the gradient approximation being zero in a large area of the input space. As $\text{ReLU}(x) = 0$ for all $x \leq 0$, the risks of the gradient being zero is increased compared to using LeakyReLU. When using adaptive step-size ODE solvers, we observed that the step size could converge to zero, causing the number of iterations needed between two time points to increase drastically. This in turn causes the ODE solvers to iterate infinitely or results in underflow for the stepsize.

Weight decay was added as a means to avoid the gradient approximation becoming too large. When the ODE solver takes a step with a too large gradient, it causes a numerical overflow. Weight decay restricts the weights in ODENet from growing uncontrollably, hence reducing the risk of exploding gradients.

A general problem when training was that datasets with a large number of trajectories close to each other in some regions lead to the network not being able to distinguish between different trajectories. The learned dynamics looked similar to an average between the trajectories. The solution to this was to actually remove data by eliminating some trajectories from the training set. This improved how the network generalized the learned dynamics. This phenomenon of more data causing a worse approximation is counter-intuitive, as the general case in deep learning is that a larger dataset typically corresponds to a more accurate network output.

4.6 Future work

To be able to reliably use ODENet as a modelling tool for dynamical systems there are areas left to explore. We did not focus on the computational performance of the implementation of ODENet. Due to the way the ODE solver was implemented we were only able to parallelize computations on the GPU if they were for time steps of the same length. Further research should be conducted on the topic of extending the ODE solvers to also be able to run different time steps in parallel to speed up training. It is also necessary to perform measurements of the training time and how it is affected by different problem settings and network parameters.

A strength of ODENet is that any ODE solver can be used to produce the output. Modern ODE solvers with adaptive step sizes can make a trade-off between accuracy and speed, and can be controlled by the user by changing two parameters governing the error tolerances. Future work could treat the effect of varying these parameters in different problem settings to find an optimal setting achieve fast training without sacrificing accuracy.

Another topic is how to extract a closed-form ODE from the derivative approximated by the neural network. This includes both further work on the two methods we used, as well as exploring other possibilities. The least squares approach could for example be replaced by a more advanced regression algorithm. Linear genetic

programming has several parameters that can be tuned and should be examined for each different problem separately.

We started working on extending the network structure to allow for training on time dependent ODEs by simply adding time as an extra dimension in the network input. Due to the non-conclusive results there is need of further work. This could include the use of other types of network structures trained in conjunction with ODENet to account for the time dependence.

There is also the direction of using the ODENet on more advanced synthesized problems or on data gathered from actual measurements of real systems. This might give further insight into the feasibility of actually using ODENet in a production setting.

5
Conclusion

The aim of this thesis was to examine the use of ODENet as a modelling tool for learning dynamical systems governed by ODEs directly from data. We found that ODENet was able to parameterize the underlying ODE and produce an accurate approximation of the local derivative in a region surrounding the training data. Further away from the training data the parameterized derivative diverges from the true equation. This means the network does not generalize globally in general. For problems with symmetry the learned dynamics exhibit a greater degree of generalization. Closed-form expressions for the derivative extracted from a region close to the training data, where the network parameterization is accurate, are close to the analytical ordinary differential equations and hence valid globally. Linear genetic programming produces a more accurate closed-form expression than the least squares approach for nonlinear ordinary differential equations, and similar results for linear problems. Convergence is not guaranteed for linear genetic programming, and is generally slower than least squares. In conclusion ODENet is a valid tool for modelling dynamical systems and can accurately learn a parameterization for an ordinary differential equation from data. Although the network parameterization is generally not accurate globally, it can be used to extract a global closed-form expression for the underlying dynamical system.

Bibliography

[1] Carlos A. Gomez-Uribe and Neil Hunt. "The Netflix Recommender System: Algorithms, Business Value, and Innovation". In: *ACM Trans. Manage. Inf. Syst.* 6.4 (Dec. 2015), 13:1–13:19. ISSN: 2158-656X. DOI: 10.1145/2843948. URL: http://doi.acm.org/10.1145/2843948.

[2] Travers Ching et al. "Opportunities and obstacles for deep learning in biology and medicine". In: *bioRxiv* (2018). DOI: 10.1101/142760. eprint: https://www.biorxiv.org/content/early/2018/01/19/142760.full.pdf. URL: https://www.biorxiv.org/content/early/2018/01/19/142760.

[3] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[4] Gregor Kasieczka et al. "Deep-learning top taggers or the end of QCD?" In: *Journal of High Energy Physics* 2017.5 (May 2017), p. 6. ISSN: 1029-8479. DOI: 10.1007/JHEP05(2017)006. URL: https://doi.org/10.1007/JHEP05(2017)006.

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

[6] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[7] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227. DOI: 10.1109/72.279181.

[8] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[9] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366 (2018). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366.

[10] Danilo Jimenez Rezende and Shakir Mohamed. "Variational Inference with Normalizing Flows". In: *arXiv e-prints*, arXiv:1505.05770 (May 2015), arXiv:1505.05770. arXiv: 1505.05770 [stat.ML].

[11] M. Wahde. *Biologically Inspired Optimization Methods: An Introduction*. WIT Press, 2008, pp. 35–78. ISBN: 9781845641481.

[12] Yiping Lu et al. "Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations". In: *CoRR* abs/1710.10121 (2017). arXiv: 1710.10121. URL: http://arxiv.org/abs/1710.10121.

[13] Eldad Haber and Lars Ruthotto. "Stable Architectures for Deep Neural Networks". In: *CoRR* abs/1705.03341 (2017). arXiv: 1705.03341. URL: http://arxiv.org/abs/1705.03341.

[14] Lars Ruthotto and Eldad Haber. "Deep Neural Networks motivated by Partial Differential Equations". In: *CoRR* abs/1804.04272 (2018). arXiv: 1804.04272. URL: http://arxiv.org/abs/1804.04272.

[15] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Numerical Gaussian processes for time-dependent and non-linear partial differential equations". In: *arXiv preprint arXiv:1703.10230* (2017).

[16] Maziar Raissi and George E. Karniadakis. "Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations". In: *CoRR* abs/1708.00588 (2017). arXiv: 1708.00588. URL: http://arxiv.org/abs/1708.00588.

[17] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations". In: *CoRR* abs/1711.10561 (2017). arXiv: 1711.10561. URL: http://arxiv.org/abs/1711.10561.

[18] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. "Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations". In: *CoRR* abs/1711.10566 (2017). arXiv: 1711.10566. URL: http://arxiv.org/abs/1711.10566.

[19] Zichao Long et al. "Pde-net: Learning pdes from data". In: *arXiv preprint arXiv:1710.09668* (2017).

[20] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Multistep neural networks for data-driven discovery of nonlinear dynamical systems". In: *arXiv preprint arXiv:1801.01236* (2018).

[21] Ibrahim Ayed et al. "Learning Dynamical Systems from Partial Observations". In: *CoRR* abs/1902.11136 (2019). arXiv: 1902.11136. URL: http://arxiv.org/abs/1902.11136.

[22] Luca Cardelli. "From Processes to ODEs by Chemistry". In: *Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008*. Ed. by Giorgio Ausiello et al. Boston, MA: Springer US, 2008, pp. 261–281. ISBN: 978-0-387-09680-3.

[23] Y. Takeuchi et al. "Evolution of predator–prey systems described by a Lotka–Volterra equation under random environment". In: *Journal of Mathematical Analysis and Applications* 323.2 (2006), pp. 938–957. ISSN: 0022-247X. DOI: https://doi.org/10.1016/j.jmaa.2005.11.009. URL: http://www.sciencedirect.com/science/article/pii/S0022247X0501142X.

[24] Herbert W. Hethcote. "The Mathematics of Infectious Diseases". In: *SIAM Rev.* 42.4 (Dec. 2000), pp. 599–653. ISSN: 0036-1445. DOI: 10.1137/S0036144500371907. URL: http://dx.doi.org/10.1137/S0036144500371907.

[25] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems*. Berlin, Heidelberg: Springer-Verlag, 1993. ISBN: 0-387-56670-8.

[26] J.D. Murray. *Mathematical Biology: I. An Introduction*. Interdisciplinary Applied Mathematics. Springer New York, 2011. Chap. 3, pp. 79–83. ISBN: 9780387952239. URL: https://books.google.se/books?id=4WbpP90Gk1YC.

[27] Brian Winkel. *Gustafson-2250 Systems Differential Equations Applications*. Sept. 2017. URL: https://www.simiode.org/resources/3892.

[28] Tian Qi Chen et al. *Differentiable ODE solvers with full GPU support and O(1)-memory backpropagation.* https://github.com/rtqichen/torchdiffeq. 2018.

[29] George D Byrne and Alan C Hindmarsh. "Stiff ODE solvers: A review of current and coming attractions". In: *Journal of Computational physics* 70.1 (1987), pp. 1–62.

[30] Randall Balestriero and Richard G. Baraniuk. "From Hard to Soft: Understanding Deep Network Nonlinearities via Vector Quantization and Statistical Inference". In: *CoRR* abs/1810.09274 (2018). arXiv: 1810.09274. URL: http://arxiv.org/abs/1810.09274.

[31] Léon Bottou, Frank E Curtis, and Jorge Nocedal. "Optimization methods for large-scale machine learning". In: *Siam Review* 60.2 (2018), pp. 223–311.

[32] Yiping Lu et al. "Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations". In: *CoRR* abs/1710.10121 (2017). arXiv: 1710.10121. URL: http://arxiv.org/abs/1710.10121.

[33] L.S. Pontryagin. *Mathematical Theory of Optimal Processes*. Classics of Soviet Mathematics. Taylor & Francis, 1987. ISBN: 9782881240775. URL: https://books.google.se/books?id=kwzq0F4cBVAC.

[34] Ronald M Errico. "What is an adjoint model?" In: *Bulletin of the American Meteorological Society* 78.11 (1997), pp. 2577–2592.

[35] I. Gustafsson and K. Holmåker. *Linjär Algebra och Numerisk Analys*. Gothenburg, Sweden: Chalmers University of Technology, 2013.

[36] Adam Paszke et al. "Automatic Differentiation in PyTorch". In: *NIPS Autodiff Workshop*. 2017.