FACULTY OF SCIENCE AND TECHNOLOGY

MASTER THESIS

Study programme / specialisation:
Master of Science in Computational Engineering

The spring semester, 2022

Open

Author: Jonathan Perilla Arias

.....
(signature author)

Course coordinator: Steinar Evje

Supervisor(s): Steinar Evje

Thesis title: Influence of Hyperparameters of Neural Ordinary Differential Equations in Their Ability to Model Dynamic Systems Governed by ODEs

Credits (ECTS): 30

Keywords: Neural Ordinary Differential Equations, NODEs, Neural ODEs, hyperparameters of NODEs

Pages: 56

+ appendix: 13

Stavanger, June 15 /2022

Influence of Hyperparameters of Neural Ordinary Differential Equations in Their Ability to Model Dynamic Systems Governed by ODEs

Author
Jonathan Perilla Arias

Supervisor
Steinar Evje

A thesis presented for the degree of
Master of Science
in
Computational Engineering

Department of Energy Resources
University of Stavanger
Norway
2022

Abstract

In this thesis the Neural Ordinary Differential Equations (NODEs) are studied in their ability to model dynamic systems governed by ODEs. NODEs are a new type of artificial neural network that uses a feed-forward artificial neural network as the source of gradient to construct a continuous trajectory. Although there are several investigations showing NODEs extraordinary ability to model time series, no comprehensive study of the influence of its hyperparameters on its performance has been conducted. In this investigation the objective was to evaluate the influence of some of the NODEs' hyperparameters on the NODEs capabilities of modeling. Special focus was set on the evaluation of the influence of the gradient computation algorithm used, because it determines to a great extent the speed of the training session. Three gradient computation algorithms were analyzed, including a novel method proposed in this thesis; this new approach is based on a modification of the adjoint sensitivity method.

In order to reach these aims, an implementation of NODEs was created using object-oriented programming in the Matlab suite. Then, a group of ODE systems was used to generate several trajectories that were used to train a collection of NODEs that had a different set of hyperparameters. The trained NODEs were used to approximate a set of new trajectories generated by the same systems of ODEs, and the error in the trajectories was used to quantify the influence of the hyperparameters.

The results indicated that the hyperparameters have a big impact on the performance of the NODEs in modeling dynamic systems. Some characteristics of the data to model can give a hint in potential initial hyperparameters, but the evidence showed that many tests need to be done in order to get the optimal hyperparameters. In this regard, the new method proposed for gradient calculation showed potential, because it was ten times faster than the other methods analyzed; that in effect could allow a broader set of hyperparameters to be tested when facing a modeling problem.

To my wife and daughter that always stand by me.

Acknowledgments

I would like to express my gratitude to my supervisor Steinar Evje for his continuous support and guidance in this thesis. I would also like to thank my friend and teacher Bryan Thalhammer for his invaluable feedback.

Contents

List of Figures

Chapter 1

Introduction

A dynamic system is a system in which its next state is defined by its current state and a rule of change. Modelling dynamic systems in order to be able to predict their state is of key importance, because it allows the manipulation of variables to produce a desired behaviour. Data-based models and more specifically artificial neural networks (ANNs) offer a great advantage: they do not require any assumptions about the underlying relationships between the input-output data. Although artificial neural networks (ANN) have revolutionized the artificial intelligence field, most of the successful implementations are based on ideas that were presented some decades ago that do not adapt well to continuous dynamic systems. Recently, a new ANN architecture, the neural ordinary differential equation (NODE) was introduced; this architecture was a breakthrough because it was able to produce continuous outputs and to be trained with irregular spaced samples. This research aims to investigate the ability of NODEs in modeling dynamic systems and to quantify the effect of their hyperparameters in the accuracy of the model obtained.

This chapter will provide an introduction to the study by first discussing the basic concepts behind ANNs and the legacy ANNs structures that first attempted to model time series. Then it shows the current ideas, with the main topic being Neural ODEs. Finally the knowledge gap, the research aims and the objectives of the research are presented.

1.1 Previous Work in the Topic

1.1.1 Feed-forward ANN

ANNs are structures inspired by the brain in the way that they are composed of neurons or nodes and links between them. Each node stores information (a numerical value) and passes that information to other nodes through links that are characterised by its strength or weight. These structures are designed to adapt themselves to generate a desired combination of input-output vectors; they achieve this by adjusting the weights of the links between nodes.

The most basic ANN structure is the feed forward topology, in a feed-forward ANN the

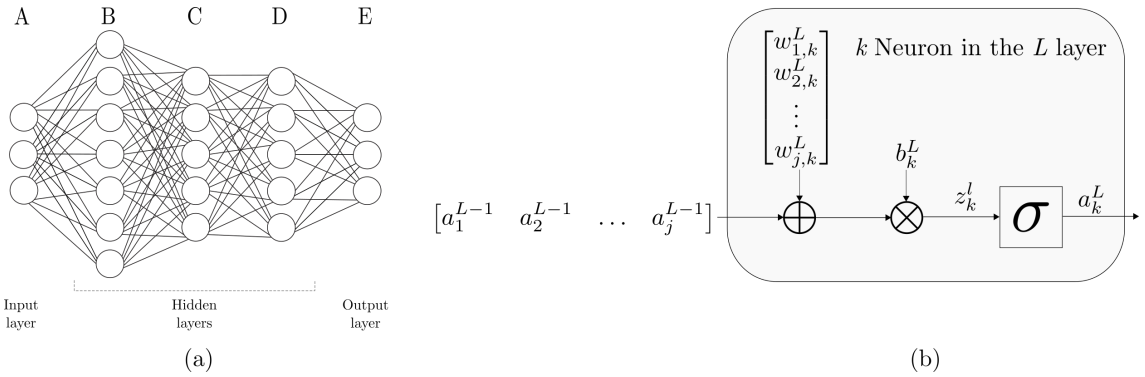


Figure 1.1: Feed-Forward ANN, (a) Feed-forward ANN example, (b) Single neuron structure.

information only moves forward from the input layer to the hidden layers and then to the output layer; there are no loops in the network (Figure 1.1a). Each of the layers (e.g. A in Figure 1.1a) is composed of several nodes. In the input layer, the nodes represent an input variable for the system and take that variable value. For the hidden layers (e.g. B, C and D in Figure 1.1a), every node receives information from the nodes in the previous layer (i.e. B from A and C from B in Figure 1.1a); that information is weighted and defines the value of the nodes. Finally, each node in the output layer (e.g. E in Figure 1.1a) represents an output variable of the system and takes values using the weighted values from the last layer.

In Figure 1.1b, an elementary neuron in the Lth layer with j inputs is shown. The a variables in the input and output represent the values of the neurons, with the superscript being the layer number and the subscript the position in the layer. The input vector contains j values a^{L-1}; these are the values of the j neurons of the $L-1$ layer. The single output a_k^L is the value of the kth neuron of the Lth layer. On the other hand, each neuron has a set of parameters w (i.e. weights) and b (i.e. biases) that are constant and are used to calculate the neuron values a each time the input is changed.

The process in Figure 1.1b starts with a set of inputs a that are weighted with a specific constant, w, the weighted inputs are summed, and a bias term b is added. The intermediate result output of the operations involving the parameters w and b is defined as z. This z is then input into a transfer function σ to generate the neuron value a (Equation 1.1). The use of transfer functions, more specifically nonlinear transfer functions, is required because it allows the network to learn nonlinear relationships between input and output vectors.

$$a_k^L = \sigma\left(\sum_j (a_j^{L-1} w_{jk}^L) + b_k^L\right) \tag{1.1}$$

Equation 1.1 applies for layers starting with the second layer up to and including the last layer. The neuron values a of the first layer (i.e. A in Figure 1.1a) are just the external inputs of the neural network, thus no calculations are done in this first layer. The a values in layer one then serve as the initial state for the process in Equation 1.1.

The back-propagation is the learning process to find the best weights and biases for an ANN given an input-output and was introduced by Rumelhart et al. (1986). The development of this algorithm made it possible to effectively train an ANN and was a breakthrough in the machine learning field. This algorithm first defines a cost function proportional to the sum of the square differences between the outputs of the ANN and the expected outputs for all the data points. Then, using the chain rule, it calculates the partial derivative of this cost function with respect to each parameter of the neural network in one backward pass. The parameters can then be updated in the opposite direction of the gradient, in the direction of the steepest descent.

If an ANN with L layers is considered, the total cost is defined as:

$$C = \frac{1}{2} \sum_i \sum_k ((y_k)_i - (a_k^L)_i)^2 \tag{1.2}$$

Where i is the index of input-output data points and k is the index of the neurons of the output layer, a is the output of the ANN and y are the desired outcomes. The back-propagation starts finding the partial derivative of the cost with respect to the single output a_k^L:

$$\frac{\partial C}{\partial a_k^L} = y_k - a_k^L \tag{1.3}$$

Then, using the chain rule, the partial derivative of the cost with respect to the intermediate value z_k^L is:

$$\frac{\partial C}{\partial z_k^L} = \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L} = (y_k - a_k^L)\sigma'(z_k^L) \tag{1.4}$$

As z_k^L is a linear function of the parameters w_{jk}^L and b_k^L, the partial derivative with respect to these parameters can be easily calculated:

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^L} &= \frac{\partial C}{\partial z_k^L} \frac{\partial z_k^L}{\partial w_{jk}^L} = (y_k - a_k^L)\sigma'(z_k^L) a_j^{L-1} \\ \frac{\partial C}{\partial b_k^L} &= \frac{\partial C}{\partial z_k^L} \frac{\partial z_k^L}{\partial w_{jk}^L} = (y_k - a_k^L)\sigma'(z_k^L) \end{aligned} \tag{1.5}$$

Moving one layer backward to the $L-1$ layer, the derivative of the cost with respect to a neuron value a_j^{L-1} will be a sum of the contributions emanating from it to the layer L. Each of these contributions will be the product of the weight w_{jk}^L (connecting the neuron j of the $L-1$ layer to the neuron k of the L layer) by the partial derivative of the cost with respect to the intermediate value z_k^L.

$$\frac{\partial C}{\partial a_j^{L-1}} = \sum_k \frac{\partial C}{\partial z_k^L} w_{jk}^L \tag{1.6}$$

Then the same process follows in Equation (1.4), and (1.5) can be used to find the derivative of the cost with respect to the parameters w_{jk}^{L-1} and b_{jk}^{L-1} (i.e. θ_{jw}^{L-1}). The trainable parameters (i.e. weights and biases) in an ANN are called parameters and

are group in the variable θ, while the non-trainable parameters (e.g. number of layers, activation function) in an ANN are called hyperparameters.

The process described in Equations (1.3),(1.4),(1.5) and (1.6) can be repeated to find the gradient of the cost with respect to all the trainable parameters θ. This back-propagation algorithm was a breakthrough because it can calculate the derivative of the cost with respect to all the parameters for an input-output pair in one backwards pass.

To be able to perform the back-propagation, the activation function must be differentiable for all possible z values, that is $z \in R$. The transfer function used by Rumelhart et al. (1986) was the logistic sigmoid (Equation 1.7) that maps the entire number line into the range $(0, 1)$ (Figure 1.2). But Glorot & Bengio (2010) showed that the sigmoid activation caused the last layer of deep networks to saturate towards zero, causing the training to slow down and even to never converge to a minimum. In the same document Glorot & Bengio (2010) showed that the hyperbolic tangent sigmoid (Equation 1.8) that is similar to the standard sigmoid, but maps the line into the range $(-1, 1)$, does not suffer from the same type of saturation as the standard sigmoid, and can give better results when the parameters are properly initialized.

More recently the rectified linear activation function (ReLU) was introduced; it outputs 0 for negative inputs and the same input for positive inputs (Equation 1.9). It was first published by Hahnloser et al. (2000), justifying it as a better model of a biological neuron. Glorot et al. (2011) showed empirically that the ReLU offered a better test error for some benchmark problems compared to the hyperbolic tangent sigmoid. Since then it had become one of the most popular activations in deep neural networks.

$$\sigma(z_k^L) = \frac{1}{1 + e^{-z_k^L}} \tag{1.7}$$

$$\sigma(z_k^L) = \frac{e^{-2z_k^L} - 1}{e^{-2z_k^L} + 1} \tag{1.8}$$

$$\sigma(z_k^L) = \begin{cases} z_k^L & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.9}$$

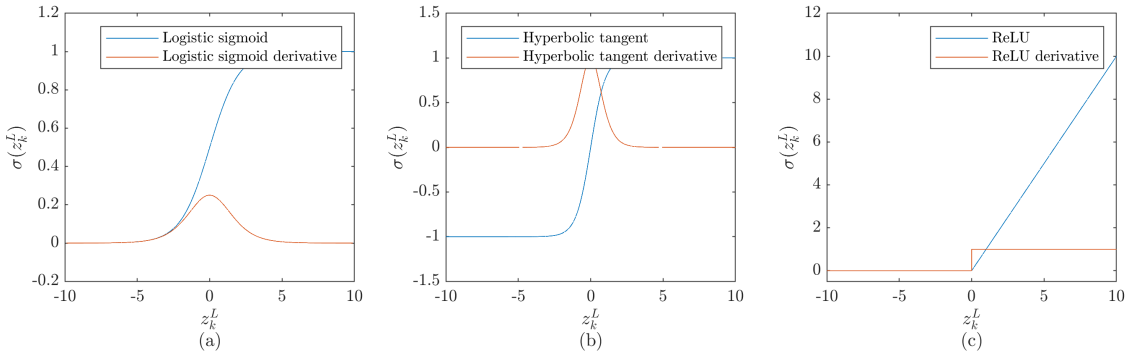


Figure 1.2: Activation functions and its derivatives, (a) Logistic sigmoid, (b) Hyperbolic tangent sigmoid, (c) ReLU.

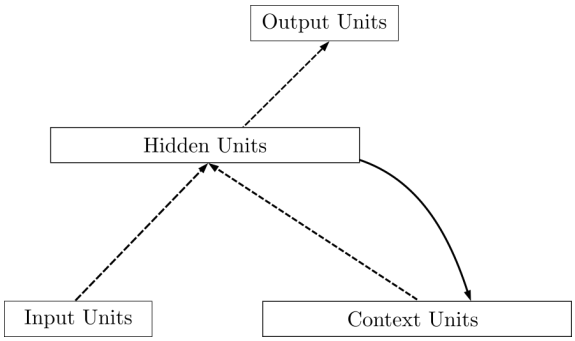


Figure 1.3: Recurrent neural network (Modified from source: Elman (1990)).

Feed-forward ANNs have many applications, but they cannot be used to model sequential data because they assume independence between the measurements. In a feed-forward ANN, a single piece of data is fed and then a response is obtained, but when another piece of data is fed, the ANN goes to an initial state and forgets all the information from the previous event. Although these structures cannot model continuous systems ruled by ODEs, the feed-forward ANNs are the building blocks for all the other types of ANNs presented in this document.

1.1.2 Recurrent Neural Networks

This limitation was addressed by Elman (1990). He proposed the use of feed-forward ANNs, however using what he called context. The ANN is fed with the first sequential data and an initial context, and then the hidden unit will generate an output and an updated context. This context will then be fed back to the network and then be used in the next time step (Figure 1.3). This context works as a memory for the network, and then information can flow over time. This architecture is called recurrent neural network (RNN).

However there are some issues with RNN described by Hochreiter & Schmidhuber (1997) when the sequences are long. The influence in the loss function of an input early in the sequence will explode or vanish depending on the sizes of the weights. In the case of exploding gradients, this will lead to oscillating weights, and in the case of vanishing gradients the long-term information will be lost.

1.1.3 Long-Short Term Memory

In the same work, Hochreiter & Schmidhuber (1997) proposed a new architecture called long-short term memory (LSTM) whose main feature was to let information flow through the network in time without applying a continuous scaling and a nonlinear activation on each step. This extra piece of information that is passed to the next time step is called cell state. Since the introduction of the LSTM by Hochreiter & Schmidhuber (1997), this

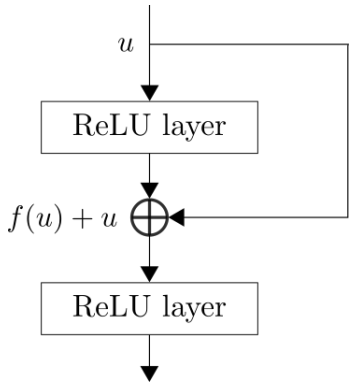


Figure 1.4: Residual learning: a building block (Modified from source: He et al. (2016)).

architecture has been used successfully in many areas, showing exceptional results.

Even Though LSTMs are successful in many areas such as speech recognition, image classification, and music composition; they have fundamental limitations to model continuous dynamic systems. Their architecture is built to have equally-spaced time series inputs-outputs that are not compatible with the measurement of dynamic systems in which measurements can be missing or not taken in constant time intervals. Most importantly, due to their discrete nature, LSTMs tend to be affected by noise, and they also struggle to capture the underlying dynamics in systems ruled by ODEs (Chen et al. 2018).

1.2 Current Ideas in the Topic

1.2.1 Residual Neural Networks

Recently, He et al. (2016) introduced an ANN architecture called residual neural networks (ResNETs) in order to overcome the difficulty of training very deep neural networks used in image recognition. He et al. (2016) found that after a certain number of layers, increasing the depth caused an increment in training and test error, which is counter-intuitive because one will expect a better fit with increased flexibility of the model. They addressed the problem by creating shortcut connections that feed-forward the inputs of layers to later layers, skipping one or more layers (Figure 1.4). In this way, the ANN will need to learn only the residuals of the change of the input vector. It turns out that this was a much better architecture to train deep networks, and the ResNET won the ILSVRC (ImageNet Large Scale Visual Recognition Competition) in 2015 in the image classification task (Zhai et al. (2020)).

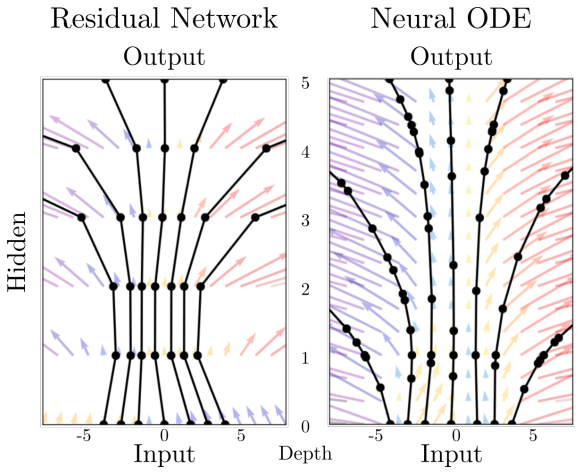


Figure 1.5: ResNets compared with NODEs (Modified from source: Chen et al. (2018)).

1.2.2 Neural Ordinary Differential Equations

Inspired by the ResNETs, Chen et al. (2018) presented a new architecture called neural ordinary differential equations (NODE). They were based on the idea that the residual architecture of a ResNET can be seen as an Euler discretization of an ODE:

$$U_{t+1} = U_t + \Delta t f(U_t, \theta_t) \tag{1.10}$$

Where $t \in 0, 1, .., T$ is the layer sequence, $\Delta t = 1$, U_t is the vector containing the state of the neurons at layer t, and f represents a feed-forward ANN with parameters θ_t. Then, if the steps are made smaller and smaller, the state U will become continuous and the derivative will become the neural network itself:

$$\frac{dU(t)}{dt} = f(U(t), \theta) \tag{1.11}$$

Starting from the initial condition $U(0)$ an ODE solver can be used to calculate the output of the network $U(T)$ using a feed-forward ANN as the source of the gradient of $U(t)$ (Equation 1.12). The NODE transforms the state vector $U(T)$ continuously, while the ResNet has a discrete sequence of finite transformations (Figure 1.5). The depth of the NODE is determined by the ODE solver, and it is equivalent to the number of times the gradient is evaluated. A key difference between ResNets and NODEs is that the parameters θ_t in ResNets can variate between layers because it has a finite number of layers T (i.e. $t \in 0, 1, .., T$); on the contrary, in NODEs these parameters θ must be constant as the t interval is continuous (i.e. $t \in (0, T]$).

$$U(T) = U(0) + \int_0^T f(U(t), \theta) \tag{1.12}$$

The NODE can evaluate the state vector at any time forward in time; this enables the NODE to be trained by back-propagation, using a time series with irregular sample steps. This offers a great advantage over the LSTM and RNN architectures that have a fixed

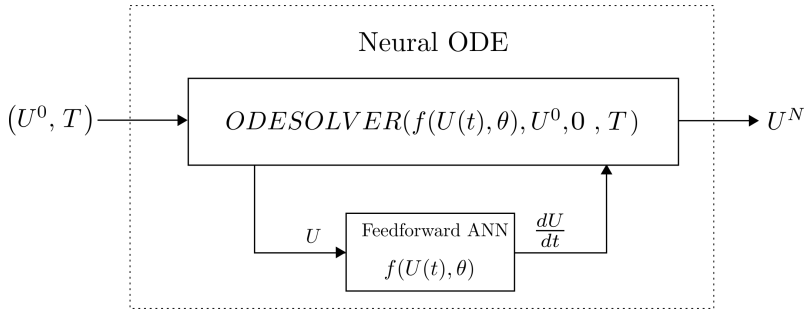


Figure 1.6: Simplified block diagram of a NODE.

time step and cannot be trained or evaluated at irregular time steps. Then for example, a real dataset that has missing or invalid points can be used to train NODEs, and the resulting NODE can be evaluated at any time t as well as between training points.

Figure 1.6 shows a simplified block diagram of the NODE; it shows that the ODE solver can be treated as a black box, thus allowing different ODE solvers to be used, depending on the type of problem and the accuracy required. This can be done without changing the structure of the NODE.

Chen et al. (2018) showed that NODEs trained with data series contaminated with gaussian noise can recover the original trajectory and successfully extrapolate the behaviour of the underlying phenomenon.

In order to train the NODE, the gradient of the cost with respect to the parameters θ of the underlying feed-forward ANN have to be calculated. This can be done by back-propagating through the operations of the ODE solver and then through the feed-forward ANN. But for doing this, the exact operations of the ODE solver have to be known, and then the ODE solver cannot be treated any longer as a black box.

As an alternative to the back-propagation method, Chen et al. (2018) proposed a method that treats the ODE solver as a black box and computes the gradient using the adjoint sensitivity method. This method is based on the idea that the state vector is continuous and then the gradients of the cost with respect to the parameters only depend on the gradients of the underlying feed-forward ANN.

The method defines an adjoint state that is the gradient of the cost with respect to the state vector $U(t)$ at each instant $a(t) = \frac{\partial C}{\partial U(t)}$. The dynamics of this adjoint state is defined for another ODE that can be thought of as the instantaneous chain rule:

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f(U(t),\theta)}{\partial U} \tag{1.13}$$

With this, the adjoint state $a(t)$ can be found:

$$a(t) = a(T) + \int_T^t a(s)\frac{\partial f(U(s),\theta)}{\partial U} ds \tag{1.14}$$

The gradient of the cost with respect to the parameters θ can be calculated again using the instantaneous chain rule:

$$\frac{dC}{d\theta} = \int_T^0 a(t)\frac{\partial f(U(t),\theta)}{\partial \theta} dt \tag{1.15}$$

Finally, the Equation 1.12 needs to be reversed:

$$U(t) = U(T) - \int_T^t f(U(s), \theta) ds \tag{1.16}$$

Equations (1.14), (1.15) and (1.16) can be concatenated and be solved in a single call to an ODE solver from T to 0, thus obtaining the total gradient $\frac{dC}{d\theta}$. The initial conditions $U(T)$ and $a(T)$ need to be found beforehand with a forward pass of Equation 1.12. Note that Equations (1.12), (1.14), (1.15) and (1.16) involved in the calculations only depend on the gradient of the underlying feed-forward ANN, so the ODE solver can be treated as a black box.

But Hasani et al. (2020) claimed that the adjoint sensitivity method produces gradients with lower accuracy, compared with the back-propagation method. This is caused by the numerical errors generated in the recovery of the state vector $U(t)$ in the backward-pass using Equation 1.16. The state vector $U(t)$ needs to be recovered in the adjoint sensitivity method, because the ODE solver used in the backward-pass needs the value of the state vector $U(t)$ at some specific times t in order to find the total gradient $\frac{dC}{d\theta}$.

Recently, Kidger et al. (2021) proposed a modification of the adjoint sensitivity method that improves the speed of the calculation. Kidger et al. (2021) noticed that the Equation 1.15 is not an ODE in the sense that errors do not propagate in time; it is just an integral once $a(t)$ and $U(t)$ are known. Thus, if the accuracy requirements for the Equation 1.15 are relaxed when solving it simultaneously with Equations (1.14) and (1.16), the calculation needs less gradient evaluations. This supposedly improves the speed, but maintains a similar accuracy corresponding to the original adjoint sensitivity method.

Inspired by this, a modification of the adjoint method to find the gradient in NODEs is proposed in this document. It uses the state values $U(t)$ of the forward-pass to evaluate numerically the integrals in Equations (1.14) and (1.15) to find the total gradient $\frac{dC}{d\theta}$. Because it uses the value of $U(t)$ of the forward-pass, the numerical errors mentioned by Hasani et al. (2020) are avoided, and the problem is simplified to that of solving numerically two integrals. As the adjoint method, this method only requires the gradients of the underlying feed-forward ANN; then there is no need to back-propagate through the ODE solver.

1.3 Knowledge Gap

Karlsson & Svanström (2019) and Chen et al. (2018) showed some examples of the outstanding capabilities of the NODEs for modelling dynamic systems. However, there is no comprehensive study that evaluates the dependency of the performance of these structures when the underlying ANN used to model the gradient is changed. An example would be using different depths or changing the activation function of the underlying ANN.

Chen et al. (2018) proposed the adjoint sensitivity method for finding gradient in NODEs, because it explicitly controls numerical error an has a constant memory cost, but Hasani et al. (2020) argued that the gradient generated with the adjoint sensitivity method had lower accuracy than the gradient generated by the backpropagation method. However

there is no direct comparison of the accuracy-speed of the gradients obtained with these methods. Moreover, there is no comparison of the models obtained with different gradient calculation methods.

As a result, there is not a basic starting point for selecting the underlying ANN structures for NODEs, or criteria for selecting a gradient algorithm when facing a modeling problem.

In this study, a set of trajectories generated by an ODE system were used to train a set of NODEs with different hyperparameters (e.g. number of layers, activation function, training algorithm, and gradient computation algorithm); then these NODEs were used to replicate a set of new trajectories generated by the same ODE systems. The distances between the true trajectories and the approximations generated by the NODEs were used to quantify the effects of those hyperparameters in the NODEs obtained, and to evaluate in general the performance of NODEs.

1.4 Aim of the Study

The aim of this study is to evaluate the ability of NODEs to model different dynamic systems governed by ODEs and to quantify the influence of the hyperparameters (i.e. number of layers, activation function) of the underlying feed-forward ANN, the optimization algorithm and the gradient algorithm in the accuracy of the model obtained. Between the gradient algorithms to evaluate, a new method proposed in this document referred as the adjoint-modified method is included.

1.5 Objectives

The main objectives of this thesis are:

- Select several systems of ordinary differential equations that can be used as ground truth which will be used to generate synthetic data.
- Use object-oriented programming to develop classes, objects and methods that allow the testing and training of NODEs. This implementation should be flexible enough to accept different underlying ANNs structures and also to train the networks using the back-propagation method, the adjoint sensitivity method, and the adjoint-modified method.
- Use a set of trajectories generated by an ODE system to train a NODE (i.e. learn the parameters θ of the underlying ANN); then using the trained NODE, try to replicate another set of trajectories generated by the same ODE system. With this, assess the ability of NODEs to learn the training data and to extrapolate the behaviour of the dynamic systems selected.

- Quantify the influence of changing different hyperparameters in the underlying ANN in the ability of NODEs to learn the training trajectories and to approximate the testing trajectories.
- Quantify the cost-benefit of using the back-propagation, the adjoint sensitivity or the adjoint-modified methods to compute gradients for the NODEs implemented.

.

Chapter 2

Methods

This chapter discusses the details of the methodology followed in the study to achieve the aim. First, a general description of the implementation of Neural ODE built in this study is given. Then a detailed description of the main parts of the implementation is presented, including the derivation of the mathematical expressions that were used. Subsequently, the selected ODE systems used to generate synthetic data are shown. Finally, the experiments proposed to quantify the effect of the NODE's hyperparameters in its performance are described.

2.1 Neural ODE Implementation

An approach based on object-oriented programming was selected over procedural programming because it allows encapsulation of data and behaviours in the same entity. This effectively adapts to the modeling of ANNs because these structures are a blend of parameters and hyperparameters (data) and actions (behaviour). Then specific instances of NODEs with different hyperparameters can be created, used, and stored with ease.

Matlab was chosen as a programming language, because it was designed specifically to work with matrices. As most of the functionality related to ANN involves matrix operations, this suite is a good fit for the problem in hand. It also has a well-documented graphics library that allows the creation of 2D and 3D plots for visualising and presenting the results. Besides that, the author and the advisor had previous experience with the programming language, which made it convenient to use.

The implementation proposed uses objects from two classes: an ANN class that represents the underlying feed-forward ANN; and a NODE class that uses an instance of the ANN class, and that encapsulates the whole structure. There are two high level methods that the NODE must execute. First, the forward pass in which an initial condition and a time interval is given and the NODE returns a trajectory. Second, the backward pass or training, in which the NODE is given training data that it uses to update its internal parameters.

The Figure 2.1 shows the block diagram of the program implemented. The ANN object represents the feed-forward ANN; it has methods to feed-forward an input, and to find

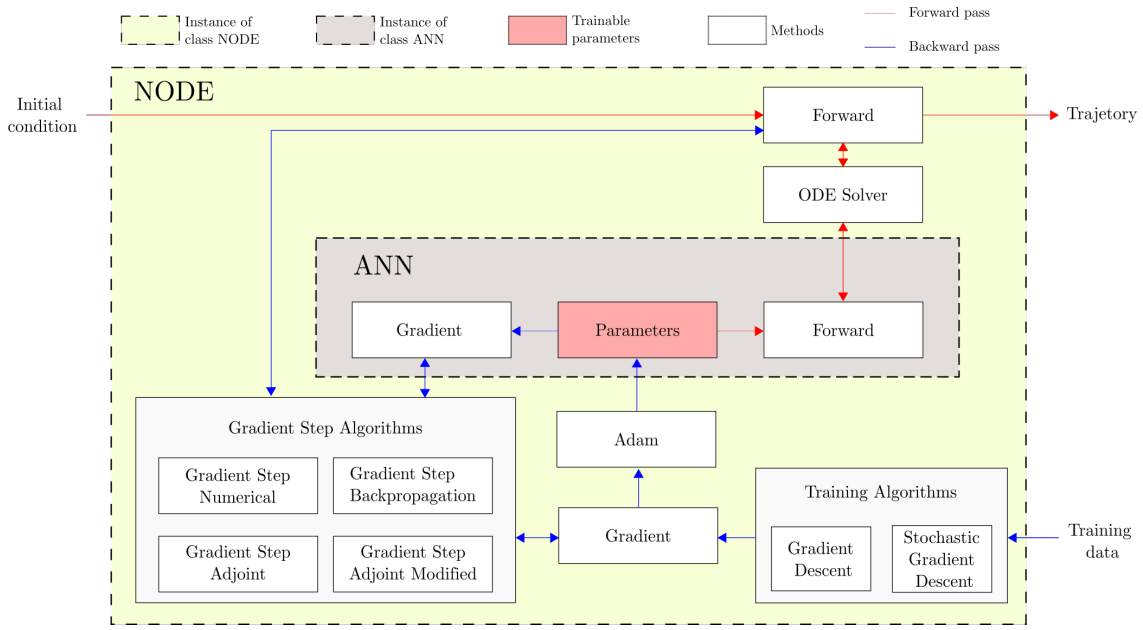


Figure 2.1: NODE implementation block diagram.

the gradient of the output with respect to its parameters and inputs. As the trainable parameters of the NODE are the parameters of the feed-forward ANN, this object stores and uses these parameters for its methods.

The object NODE has a forward method that uses an ODE solver and an ANN instance as a source for gradient. The training method uses the gradient method to find total gradients, that in turn uses the Adam optimizer to update the parameters of the ANN instance. To find the total gradient, the gradient method can use one of the four gradient step algorithms that find the gradient for a pair of points in the training data.

The unified modeling language (UML) diagram in Figure 2.2 gives a complete overview over the structure of the NODE developed. The main methods and properties (highlighted in blue in Figure 2.2) for the two classes are going to be described in the following two sections.

2.2 The Feed-forward ANN Class

The ANN class is an implementation of a feed-forward ANN. An instance of this class will serve as a source of gradient for the NODE class. The attributes necessary to create an instance of this class are the *size* and *activ* vectors. In the *size* vector, the number of elements represent the number of layers, and the element values represent the number of neurons in each layer. *Activ* is a vector of the same length as the *size* vector and it encodes the activation function of each layer. In this way each layer can have a different activation function.

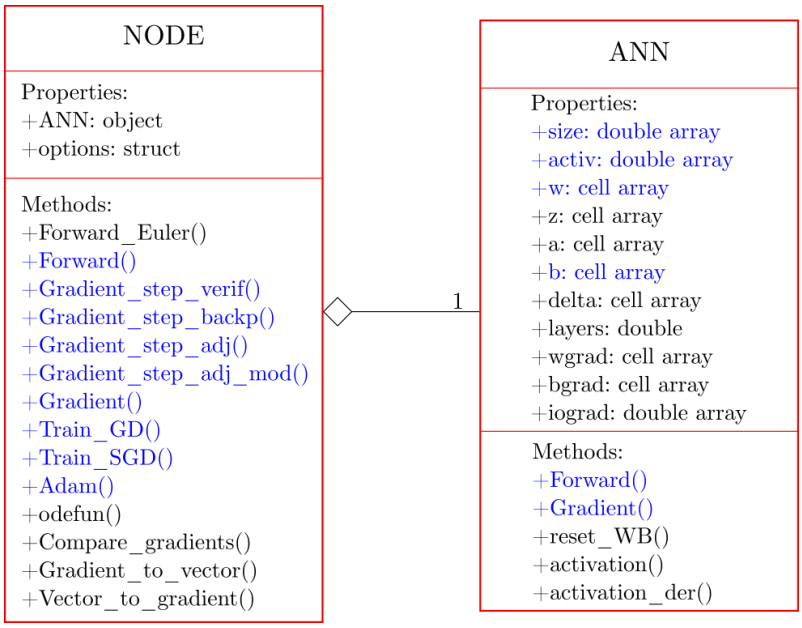


Figure 2.2: Neural ODE implementation UML diagram.

2.2.1 Forward method

The forward method calculates the output of the feed-forward ANN for a given input. Figure 2.3 shows the basic ANN structure and the nomenclature used in this document, for each variable the super-index indicates the layer and the sub-index the neuron of the layer. W^i and B^i are the matrices containing the weights and biases for the interface between layer $i-1$ and i , and a^i_j is the value of the neuron j in the layer i.

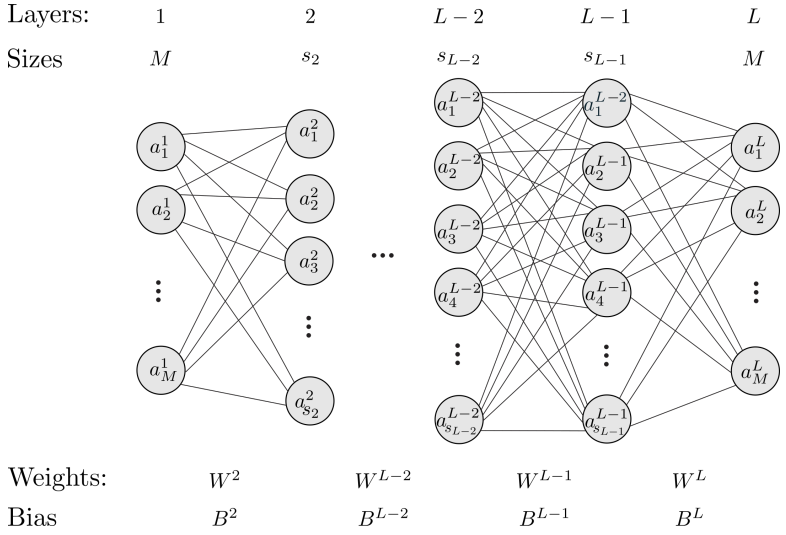


Figure 2.3: Detailed feed-forward ANN structure.

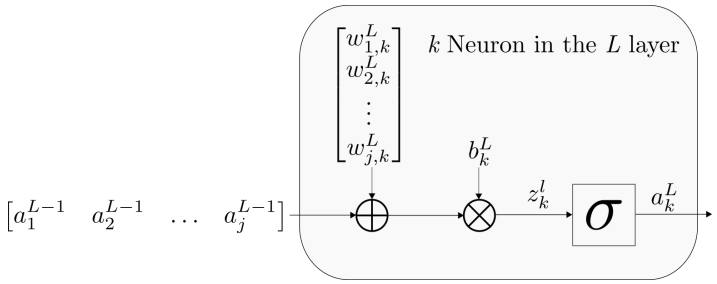


Figure 2.4: Individual Neuron.

In Figure 2.4, the *kth* neuron of the *Lth* layer detailed internal operation is shown. The intermediate z^L_k term is obtained by the product of the vector containing the output of the neurons in the previous layer with the *kth* column of the W^L matrix, and then adding the *kth* element of the bias matrix B^L. The intermediate term z^L_k is then passed through the activation function σ, resulting in the neuron value a^L_k.

The operation shown in Figure 2.4 can be done matricially to get all the intermediate terms z^l in the layer l in a vector Z^l, as shown in Equation 2.1. The vector containing the neuron values a of the layer $l-1$ times the matrix containing the weights w of layer l, this is added to the vector containing the bias terms b of layer l.

Then, the vector with Z^l can be passed through the activation function σ to obtain the vector with the neuron values for layer l, A^l. The process in Equation 2.1 can be looped from layer 2 to the output layer to get the outputs of the feed-forward ANN.

$$Z^l = \begin{bmatrix} a_1 & a_2 & \cdots & a_{s_{l-1}} \end{bmatrix}^{l-1} \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,s_l} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,s_l} \\ \vdots & \vdots & \vdots & \vdots \\ w_{s_{l-1},1} & w_{s_{l-1},2} & \cdots & w_{s_{l-1},s_l} \end{bmatrix}^l + \begin{bmatrix} b_1 & b_2 & \cdots & b_{s_l} \end{bmatrix}^l$$

$$Z^l = A^{l-1}W^l + B^l$$
$$A^l = \sigma(Z^l) \tag{2.1}$$

2.2.2 Gradient method

In a normal feed-forward ANN the gradient of interest is the gradient of the cost with respect to the parameters θ (i.e. weights and biases). For the case of Neural ODEs, the gradient calculation for the underlying feed-forward ANN is different. First, it is necessary to calculate the gradient of each output with respect to the parameters, if the vector state has M variables, then the gradient will be M groups of matrices. Also it is necessary to calculate the gradient of the outputs with respect to the inputs; this will be a matrix with MxM elements in the case of a vector state with M elements. In order to find these gradients a back-propagation process from the output to the input needs to be done.

From Equation 2.1 the derivative of one of the outputs a_j^L can be calculated with respect to the previous intermediate state Z^L as:

$$\frac{\partial a_j^L}{\partial Z^L} = \begin{bmatrix} \frac{\partial a_j^L}{\partial z_j^L} & \frac{\partial a_j^L}{\partial z_2^L} & \dots & \frac{\partial a_j^L}{\partial z_M^L} \end{bmatrix} = \begin{bmatrix} 0 & \dots & \sigma'(z_j^L) & \dots & 0 \end{bmatrix} = \delta_j^L \tag{2.2}$$

This derivative is a vector with all the elements equal to zero except the jth element. The jth element is equal to the derivative of the activation function σ' evaluated at z_j^L. For convenience the derivative of an output of the network a_j^L with respect to a intermediate state Z^i is called δ_j^i.

One step back, the derivative of the output a_j^L with respect to the cell values of the previous layer A^{L-1} by chain of rule is δ_j^L multiplied by the derivative $\frac{\partial Z^L}{\partial A^{L-1}}$ that is the jth column of the weights matrix W^L (Equation 2.3).

$$\frac{\partial a_j^L}{\partial A^{L-1}} = \frac{\partial a_j^L}{\partial Z^L}\frac{\partial Z^L}{\partial A^{L-1}} = \delta_j^L (W^L)^T = \begin{bmatrix} 0 & \dots & \sigma'(z_j^L) & \dots & 0 \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} & \dots & w_{s_{l-1}1} \\ w_{12} & w_{22} & \dots & w_{s_{l-1}2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{12} & w_{22} & ... & w_{s_{l-1}2} \end{bmatrix}^L \tag{2.3}$$

Where $(W^L)^T$ is the transpose of the W^L matrix. Going one step backwards the derivative of the outputs a_j^L with respect to the intermediate state Z^{L-1} will be:

$$\frac{\partial a_j^L}{\partial Z^{L-1}} = \frac{\partial a_j^L}{\partial A^{L-1}}\frac{\partial A^{L-1}}{\partial Z^{L-1}} = \delta_j^L (W^L)^T \odot \sigma'(Z^{L-1}) = \delta_j^{L-1} \tag{2.4}$$

Where $\odot$ represent the element-wise product operation. This process can continue until the derivative of an output with respect to the inputs is found as follows:

$$\frac{\partial a_j^L}{\partial A^1} = \frac{\partial a_j^L}{\partial Z^2}\frac{\partial Z^2}{\partial A^1} = \delta_j^2 (W^2)^T \tag{2.5}$$

In order to find the derivatives of the outputs with respect to the inputs, the δ_j^i from $i = L \dots 2$ for each of the state variables j has to be calculated. The following expressions summarise the process:

$$\begin{aligned} \delta_j^L &= \begin{bmatrix} 0 & \dots & \sigma'(z_j^L)) & \dots & 0 \end{bmatrix} \\ \delta_j^i &= \delta_j^{i+1}(W^{i+1})^T \odot \sigma'(Z^i) \\ \frac{\partial a_j^L}{\partial A^1} &= \frac{\partial a_j^L}{\partial Z^2}\frac{\partial Z^2}{\partial A^1} = \delta_j^2 (W^2)^T \end{aligned} \tag{2.6}$$

Now, it is necessary to find an expression for the derivative of outputs of the feed-forward ANN with respect to the parameters (i.e. weights and biases). Using the derivatives of the outputs with respect to the intermediate states Z^i found previously, the derivatives are straightforwardly found as follows:

$$\frac{\partial a_j^L}{\partial W^i} = \frac{\partial a_1^L}{\partial Z^i}\frac{\partial Z^i}{\partial W^i} = ((\delta_j^i)^T A^{i-1})^T$$
$$\frac{\partial a_j^L}{\partial B^i} = \frac{\partial a_1^L}{\partial Z^i}\frac{\partial Z^i}{\partial B^i} = \delta_j^i \tag{2.7}$$

2.2.3 Parameters initialization

The trainable parameters of a NODE are the parameters θ of its underlying ANN, so the initialization of these parameters is done inside the ANN object. The initialization of the weights matrix W proposed by Glorot et al. (2011) was selected because they demonstrated that it can overcome the vanishing gradient problem in deep networks that cause slow convergence. The initialization depends on the layer sizes s_{l-1} and s_l of the layers that the weight matrix W^l connects (Figure 2.3). Each element of the matrix W^l is sampled from a uniform distributions as follows:

$$w_{ij}^l = U\left[-\frac{\sqrt{6}}{\sqrt{s_{l-1}+s_l}}, \frac{\sqrt{6}}{\sqrt{s_{l-1}+s_l}}\right] \tag{2.8}$$

The biases vectors B were initialized to zero.

2.3 The Neural ODE Class

The NODE class is an implementation of Neural ODEs that uses an instance of the ANN class as a building block. The attributes necessary to create an instance of the NODE class are the *size* and *activ* attributes that dictate the structure of the underlying feed-forward ANN.

2.3.1 Forward method

The forward method calculates the output at time t^N of a NODE with initial condition vector U at time t^0. The main forward method uses the *Ode45* solver from the Matlab suite that is based on the explicit Runga-Kutta method Shampine & Reichelt (1997).

2.3.2 Gradient method

The training data for a NODE consist of a series of Y points at different time steps not necessarily equally separated. The objective of the NODE is to join each pair of consecutive points and then be able to reconstruct the true trajectory. For a pair of

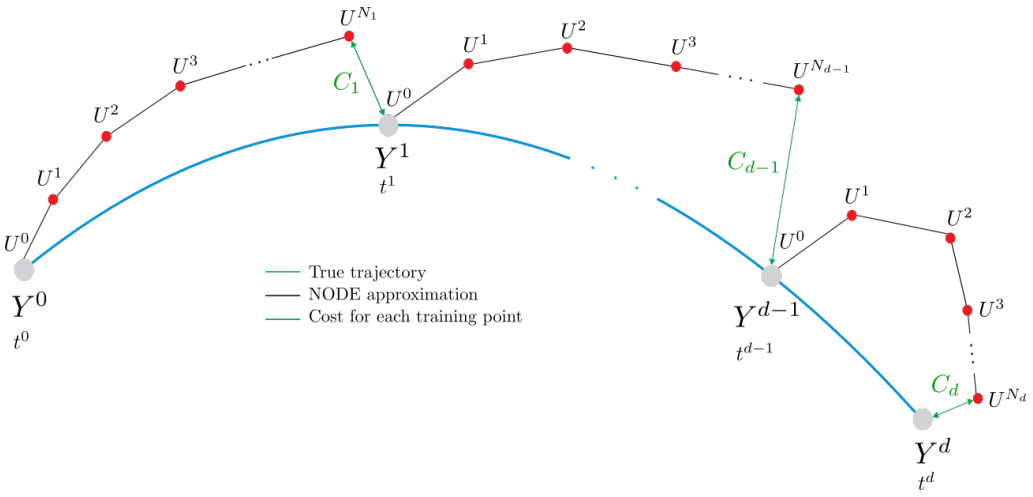


Figure 2.5: NODE cost calculation.

consecutive training points Y^0 and Y^1 given at times t^0 and t^1, the cost is a function of the distance between Y^1 and the NODE approximation at that point, that is the state vector U at time t^1, i.e. U^{N_1} (Figure 2.5). The super-index N_1 represents the number of time steps that the NODE has done between the points Y^0 and Y^1.

The cost for two consecutive points for simplicity is defined as:

$$C_n = \frac{1}{2}\sum_{i=1}^{M}(y_i^n - u_i^{N_n})^2 \tag{2.9}$$

Where C_n is the cost associated to the nth point Y^n, M is the number of variables in the state vector, and $u_i^{N_n}$ is the ith element of the vector U^{N_n} containing the approximation of the NODE for the nth point.

The total cost for all the d training points is:

$$C = \frac{1}{d-1}\sum_{n=1}^{d} C_n \tag{2.10}$$

Then the gradient of the total cost with respect to the parameters θ of the underlying feed-forward ANN is:

$$\frac{\partial C}{\partial \theta} = \frac{1}{d-1}\sum_{n=1}^{d} \frac{\partial C_n}{\partial \theta} \tag{2.11}$$

In order to find the total gradient of the cost with respect to the parameters θ, it is necessary to find the gradient of the partial cost C_n with respect to the parameters θ for every pair of points of the training dataset. This process is going to be done separately by another method called gradient step.

2.3.3 Gradient step methods

These methods calculate the gradient of the partial cost C_d for two consecutive points in the training dataset Y^d and Y^{d-1} as shown in Figure 2.5. Different approaches are going to be considered for this gradient calculation.

Gradient step numerical method

This method is based on the symmetric definition of derivative in the Equation 2.12. A parameter is disturbed by $-\epsilon$ and $+\epsilon$ and the cost is calculated for each case using the forward method, with this the gradient with respect to that parameter can be estimated. The error in the estimation of the gradient will be then proportional to ϵ^2. This method is going to be the basis for checking the accuracy of the other methods.

$$f'(x_0) = \lim_{\epsilon \to 0} \frac{f(x_0 + \epsilon) - f(x_0 - \epsilon)}{2\epsilon} \tag{2.12}$$

Gradient step back-propagation method

In Figure 2.6 the back-propagation procedure to find the cost for each pair of points Y^r and Y^{r+1} is shown. Starting from $U^0 = Y^r$ a forward pass is performed using the parameters θ to obtain U^{Nr+1}. Then the cost can be calculated using the following equation:

$$C_{r+1} = \frac{1}{2} \sum_{i=1}^{M} (y_i^{r+1} - u_i^{N_{r+1}})^2 \tag{2.13}$$

For simplicity, the $r + 1$ index is dropped. The partial derivative of the cost with respect to U^N is as follows:

$$\left. \frac{\partial C}{\partial U^N} \right|_{\theta} = \begin{bmatrix} u_1^N - y_1 & u_2^N - y_2 & \dots & u_M^N - y_M \end{bmatrix} \tag{2.14}$$

One step backward from Figure 2.6, it can be seen that U^N is a function of U^{N-1} and the parameters θ. The partial derivative of the cost with respect to the parameters θ having U^{N-1} constant is as follows:

$$\left. \frac{\partial C}{\partial \theta} \right|_{U^{N-1}} = \left(\frac{\partial C}{\partial U^N} \frac{\partial U^N}{\partial \theta} \right) \Bigg|_{U^{N-1}} \tag{2.15}$$

The partial derivative of the cost with respect to U^{N-1} having the parameters θ constant is then as follows:

$$\left. \frac{\partial C}{\partial U^{N-1}} \right|_{\theta} = \frac{\partial C}{\partial U^N} \frac{\partial U^N}{\partial U^{N-1}} \tag{2.16}$$

One more step backwards from Figure 2.6, it can be seen that U^{N-1} is a function of U^{N-2} and the parameters θ. The partial derivative of the cost with respect to the parameters θ having U^{N-2} constant is now the sum of the two ways that the parameters

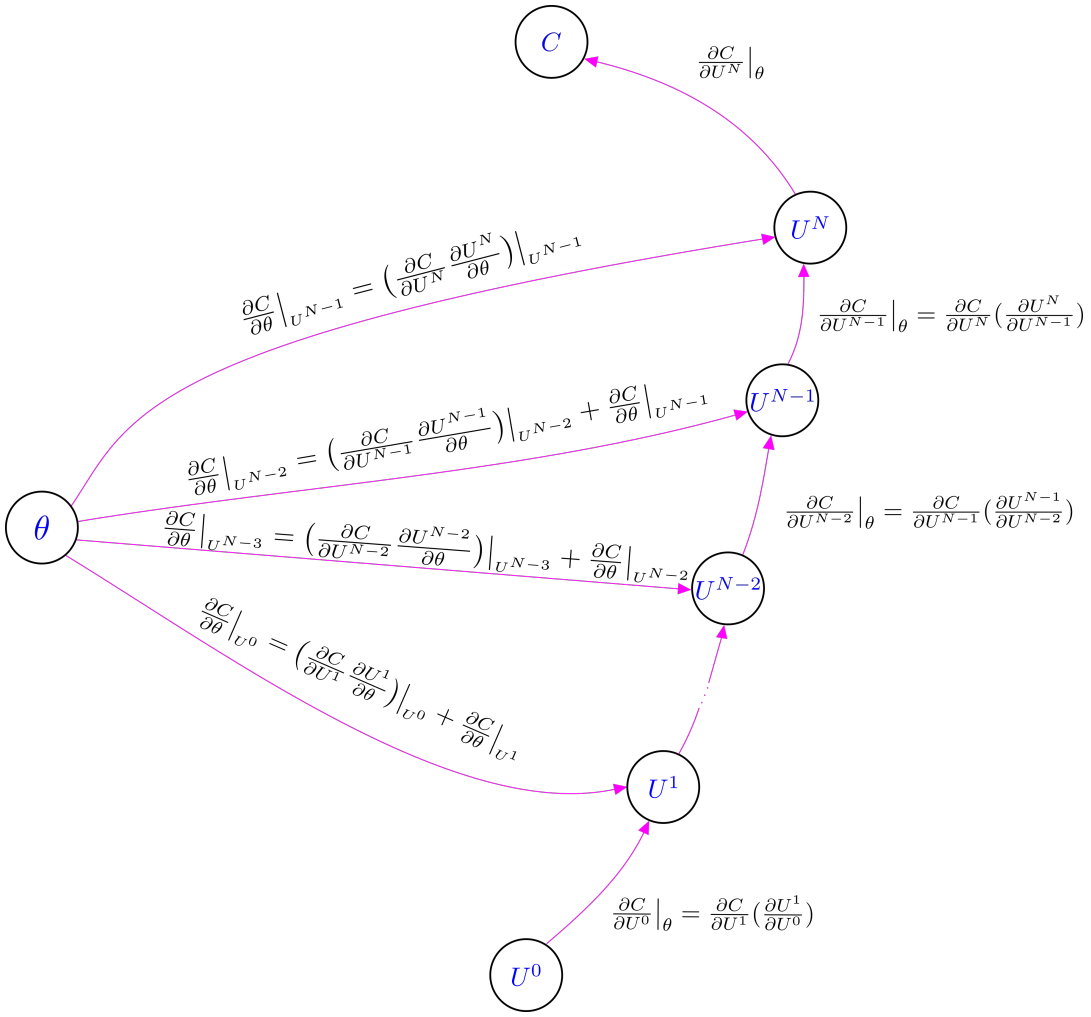


Figure 2.6: NODE backpropagation.

θ affect the cost. One way going through U^{N-1} and the other way going through U^N as follows:

$$\left.\frac{\partial C}{\partial \theta}\right|_{U^{N-2}} = \left(\frac{\partial C}{\partial U^{N-1}}\frac{\partial U^{N-1}}{\partial \theta}\right)\Bigg|_{U^{N-2}} + \left(\frac{\partial C}{\partial U^{N}}\frac{\partial U^{N}}{\partial \theta}\right)\Bigg|_{U^{N-1}} \tag{2.17}$$

That can be written as:

$$\left.\frac{\partial C}{\partial \theta}\right|_{U^{N-2}} = \left(\frac{\partial C}{\partial U^{N-1}}\frac{\partial U^{N-1}}{\partial \theta}\right)\Bigg|_{U^{N-2}} + \left.\frac{\partial C}{\partial \theta}\right|_{U^{N-1}} \tag{2.18}$$

The partial derivative of the cost with respect to U^{N-2} having the parameters θ constant is then as follows:

$$\left.\frac{\partial C}{\partial U^{N-2}}\right|_\theta = \frac{\partial C}{\partial U^{N-1}}\frac{\partial U^{N-1}}{\partial U^{N-2}} \tag{2.19}$$

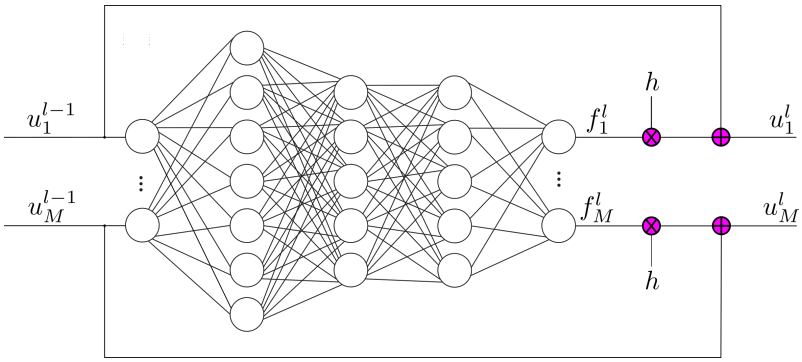


Figure 2.7: Single step in the NODE using the Euler ODE solver.

This process continues until the derivative of the cost with respect to the parameters θ with only the input U^0 constant.

$$\left.\frac{\partial C}{\partial \theta}\right|_{U^0} = \left(\frac{\partial C}{\partial U^1}\frac{\partial U^1}{\partial \theta}\right)\Bigg|_{U^0} + \left.\frac{\partial C}{\partial \theta}\right|_{U^1} \tag{2.20}$$

But to be able to evaluate this expression it is necessary to know the partial derivatives $\frac{\partial U^l}{\partial U^{l-1}}$ and $\frac{\partial U^l}{\partial \theta}$ for $l \in N, N-1, \ldots, 1$. These partial derivatives depend on the operations of the ODE solver used by the NODE and need to be recalculated in case the ODE solver is changed. This is a major drawback of the backpropagation method.

For this implementation the Euler method was selected as the ODE solver for simplicity. In Figure 2.7 the diagram for the calculation of U^l from U^{l-1} is shown. The output is calculated from the following equation:

$$u_1^l = u_{l-1}^l + h f_1^l(u_1^{l-1}, ..., u_M^{l-1}, \theta) \tag{2.21}$$

The matrices with the derivatives of U^l with respect to U^{l-1}, and U^l with respect to the parameters θ are:

$$\frac{\partial U^l}{\partial U^{l-1}} = \begin{bmatrix} \frac{\partial u_1^l}{\partial u_1^{l-1}} & \cdots & \frac{\partial u_1^l}{\partial u_M^{l-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_M^l}{\partial u_1^{l-1}} & \cdots & \frac{\partial u_M^l}{\partial u_M^{l-1}} \end{bmatrix} = \begin{bmatrix} 1 + h\frac{\partial f_1^l}{\partial u_1^{l-1}} & \cdots & h\frac{\partial f_1^l}{\partial u_M^{l-1}} \\ \vdots & \ddots & \vdots \\ h\frac{\partial f_M^l}{\partial u_1^{l-1}} & \cdots & 1 + h\frac{\partial f_M^l}{\partial u_M^{l-1}} \end{bmatrix}$$

$$\frac{\partial U^l}{\partial \theta} = \begin{bmatrix} h\frac{\partial f_1^l}{\partial \theta} \\ h\frac{\partial f_2^l}{\partial \theta} \\ \vdots \\ h\frac{\partial f_M^l}{\partial \theta} \end{bmatrix} \tag{2.22}$$

These derivatives are in terms of the derivatives $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial \theta}$ that are calculated by the gradient method of the ANN object. With this, the back-propagation implementation is completed.

Gradient step adjoint sensitivity method

The adjoint sensitivity method is an alternative method to find the gradients that does not require back-propagation through the operations of the ODE solver. This method was proposed in Chen et al. (2018) claiming that it lowers the numerical error and has less memory cost when compared with the backpropagation method.

Two new quantities are introduced $a(t)$ and $m(t)$. Let's consider a trajectory of the state vector U(t) from t^0 to t^N, then $a(t)$ is the gradient of the cost with respect to the hidden state $U(t)$, considering only the trajectory from t to t^N. On the other hand, $m(t)$ is the gradient of the cost with respect to the parameters, considering only the trajectory from t to t^N.

$$\begin{aligned} a(t) &= \frac{\partial C}{\partial U(t)}, \text{ Considering the trajectory from } t \text{ to } t^N \\ m(t) &= \frac{\partial C}{\partial \theta}, \text{ Considering the trajectory from } t \text{ to } t^N \end{aligned} \tag{2.23}$$

If the $U(t)$ trajectory is considered continuous by the instant chain rule:

$$\begin{aligned} \frac{dU(t)}{dt} &= f(U(t), \theta) \\ \frac{da(t)}{dt} &= -a(t)\frac{f(U(t), \theta)}{\partial U} \\ \frac{dm(t)}{dt} &= -a(t)\frac{f(U(t), \theta)}{\partial \theta} \end{aligned} \tag{2.24}$$

Solving the ODE system in Equation 2.24 from t^N to t^0 will give us $m(t^0)$, which is the gradient of the cost with respect to the parameters for the whole trajectory. The initial conditions for the ODE system in Equation 2.24 at t^N are:

$$\begin{aligned} & U^N \\ a(t^N) = \frac{\partial C}{\partial U^N} &= \begin{bmatrix} u_1^N - y_1 & u_2^N - y_2 & ... & u_M^N - y_M \end{bmatrix} \\ & m(t^N) = 0 \end{aligned} \tag{2.25}$$

In order to find the initial conditions in Equation 2.25, a forward pass of the NODE needs to be performed a priori. The ODE system in Equation 2.24 is only in terms of $\frac{\partial f}{\partial U}$ and $\frac{\partial f}{\partial \theta}$ that are calculated by the gradient method of the ANN object.

Gradient step adjoint-modified method

The Equations in 2.24 can be written as:

$$\begin{aligned} U(t) &= U(t^N) - \int_{t^N}^{t} f(U(s), \theta) \\ a(t) &= a(t^N) + \int_{t^N}^{t} a(s)\frac{f(U(s), \theta)}{\partial U} ds \\ m(t) &= \int_{t^N}^{t} a(s)\frac{f(U(s), \theta)}{\partial \theta} ds \end{aligned} \tag{2.26}$$

Kidger et al. (2021) noticed that the expression for $m(t)$ (Equation 2.26) is not an ODE, but rather an integral, in the sense that small errors do not propagate to create large errors. Then it is much more important for the accuracy of the solution to have an accurate $U(t)$ and $a(t)$ that are truly ODEs. When the Equations 2.26 that include $m(t)$ are solved with a ODE solver, it may take many unnecessary steps due to $m(t)$ that will still not improve the overall error. In Kidger et al. (2021) a method for only taking into account $U(t)$ and $a(t)$ in the steps rejection algorithm of the ODE solver is proposed, but this method is not analysed in this study.

Instead, based on the ideas in Kidger et al. (2021), an original alternative method called the adjoint-modified method is proposed in this study. The method consists in using the $U(t)$ obtained in a forward-pass to evaluate the integrals for $a(t)$ and $m(t)$ in Equation 2.26 numerically. Therefore, no ODE solver call is needed for finding the total gradient in the backward-pass, which could make this method very fast. Keeping $U(t)$ from the forward-pass instead of reconstructing it in a backward-pass helps with the speed and accuracy of the method. But because the only times t that $U(t)$ values are available are the ones from the forward-pass, the accuracy in the evaluation of the integrals of $a(t)$ and $m(t)$ is expected to be lower than the standard adjoint method.

The method for approximating the integrals is the trapezoidal rule in Equation 2.27.

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2}(f(a)+f(b)) \tag{2.27}$$

Using the approximation in Equation 2.27 to solve for $a(t^{N-1})$ and $m(t^{N-1})$ in the Equation 2.26 gives:

$$\begin{aligned} a(t^{N-1}) &= \left(a(t^N) + \frac{(t^N - t^{N-1})}{2} a(t^N) \frac{\partial f(U(t^N),\theta)}{\partial U}\right)\left(I - \frac{(t^N - t^{N-1})}{2}\frac{\partial f(U(t^{N-1}),\theta)}{\partial U}\right)^{-1} \\ m(t^{N-1}) &= \frac{(t^N - t^{N-1})}{2}\left(a(t^N)\frac{\partial f(U(t^N),\theta)}{\partial \theta} + a(t^{N-1})\frac{\partial f(U(t^{N-1}),\theta)}{\partial \theta}\right) \end{aligned} \tag{2.28}$$

Doing a loop with Equation 2.28 from t^N to t^0 using the steps obtained in the forward pass for $U(t)$ will produce an approximation for $m(t^0)$ that is the gradient $\frac{\partial C}{\partial \theta}$.

2.3.4 Training Methods

Gradient Descent

Gradient descent (GD), or batch optimization, is the most basic optimization algorithm to find the minimum of a cost function as it uses the whole dataset for gradient calculations.

Before the parameters θ_{t-1} from time step $t-1$ are updated, the gradient is calculated for the whole training dataset with d data points (Equation 2.29). Then the parameters θ_{t-1} are updated in the direction of the steepest descent that corresponds to the negative of the gradient; the gradient is weighted by a constant learning rate α (Equation 2.30) to obtain θ_t. This process is done in a loop while the cost is bigger than the maximum cost, and the number of epochs is less than the maximum number of epochs.

$$\frac{\partial C}{\partial \theta} = \frac{1}{d-1} \sum_{n=1}^{d-1} \frac{\partial C_n}{\partial \theta} \tag{2.29}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\partial C}{\partial \theta} \tag{2.30}$$

As the parameters are updated only once for each pass through the entire dataset, the algorithm tends to converge slowly to the optimal parameters.

Stochastic Gradient Descent

In the case of continuous datasets, GD tends to evaluate similar gradients of adjacent points, this causes redundancy that slows the training process unnecessarily. For this reason, the stochastic gradient descent SGD method was implemented as an alternative to the GD method in the NODE class. This method is based on the idea proposed by Robbins & Monro (1951) for finding roots of a function with an stochastic approximation method. The SGD calculates the gradient for updating the parameters using a random sample over the training data. It samples without replacement a *mini-batch* number of data points in a set n_s, and with this sample calculates the gradient as:

$$\frac{\partial C}{\partial \theta} = \frac{1}{|n_s|} \sum_{n_s} \frac{\partial C_n}{\partial \theta} \tag{2.31}$$

Where $|n_s|$ is the number of elements in n_s. With this gradient, in the same way as GD, SGD updates the parameters using the gradient weighted by a constant learning rate α (Equation 2.30). This process is continued until all the data points in the training dataset are sampled, completing one *epoch*. Then a new sample over the whole dataset can start again.

2.3.5 Adam learning rate optimization method

But the challenge with GD and SGD is the tuning of the hyperparameter α, which is crucial for convergence of the training algorithm. The optimal hyperparameter α is different for each set of initial parameters, each training dataset and each time step. If a constant α is used, it is required that it is tuned for each set of initial parameters θ and each training dataset to ensure convergence. For this reason the Adam adaptive learning rate method was implemented in the NODE class. The Adam method was proposed in Kingma & Ba (2017) and it estimates adaptive learning rates based on approximations of the first and second momentum of the gradient; accelerating the learning in relevant directions and slowing it down in irrelevant directions.

The Adam optimization was selected because it only requires the first-order gradient,it has a simple implementation, and requires minimum hyperparameter tuning. To obtain the updated parameters θ_t, the bias-corrected first $\hat{m}_t$ and second $\hat{v}_t$ momentum are used as follows:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.32}$$

Where α is the step size and was set to 0.001 as suggested by Kingma & Ba (2017). The bias-corrected momentums are calculated based on the biased first m_t and second v_t momentum as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned} \tag{2.33}$$

Where $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are the exponential decay rates for the moment estimates, and were selected as suggested by Kingma & Ba (2017). Finally, the biased first and second momentums are calculated using the gradient of the cost function with respect to the parameters and the biased momentums of the previous step $t - 1$ as:

$$\begin{aligned} m_t &= \beta_1 . m_{t-1} + (1 - \beta_1) \frac{\partial C}{\partial \theta} \\ v_t &= \beta_2 . v_{t-1} + (1 - \beta_2) \left(\frac{\partial C}{\partial \theta} \right)^2 \end{aligned} \tag{2.34}$$

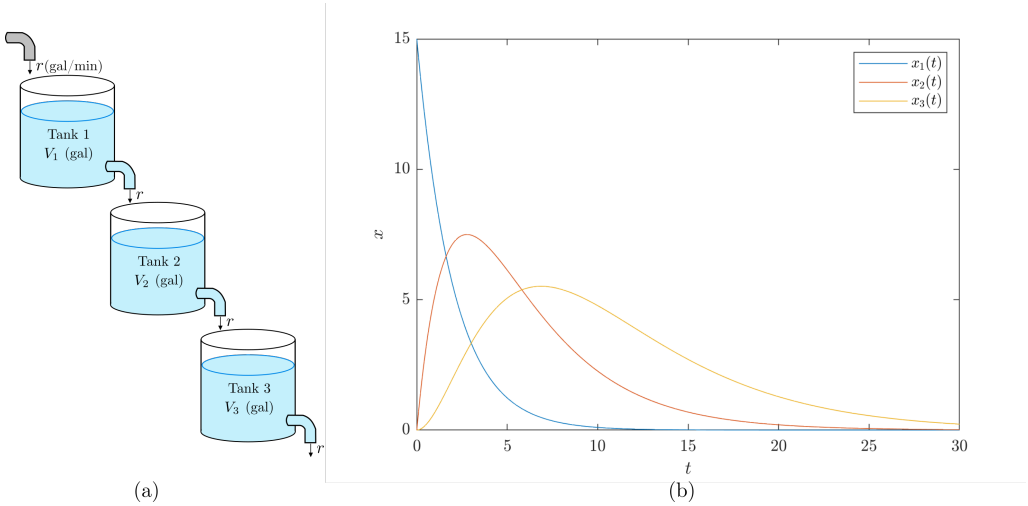
The initial values of the biased first m_0 and second v_0 momentum are zero. With this, the updated parameters θ_t can be obtained.

2.4 Systems of ODE selected and synthetic datasets

Three systems of ODE that model physical phenomena were selected to generate the synthetic data necessary to train and test the NODE implementation. The intention was to select systems that represent general groups of ODE systems and that increase in complexity. The first and simpler system selected was a linear system of ODE that models a three stage tank salt content. An almost linear system of ODE that models the movement of a damped pendulum was selected as second system. Lastly, the most complex model is a non-linear ODE system modeling a predator-prey system. The following sections describe these models briefly.

2.4.1 Linear ODE system: Three stage tank salt content

Figure 2.8a shows a three stage tank system with volumes V_1, V_2 and V_3 (gallons) containing brine. Fresh water enter the system in thank 1 with rate r (gals/min), while mixed brine flows down to tank 2 and 3 with the same rate r. The salt content (pounds) in each tank is denoted with x_1, x_2 and x_3 for tanks 1, 2 and 3 respectively. The ODE system in Equation 2.35 models the salt content in the tanks over time, where $k_i = r/V_i$.



(a) (b)

Figure 2.8: (a) Three stage tank system, (b) Salt content function (Equation 2.36) for $X_0 = \begin{bmatrix} 15 & 0 & 0 \end{bmatrix}$ (Modified from source: Edwards et al. (2007)).

$$\begin{aligned} \frac{dx_1}{dt} &= -k_1 x_1 \\ \frac{dx_2}{dt} &= k_1 x_1 - k_2 x_2 \\ \frac{dx_3}{dt} &= k_2 x_2 - k_3 x_3 \end{aligned} \tag{2.35}$$

The parameters selected for the system used in this study were $k_1 = 0.5$, $k_2 = 0.25$ and $k_3 = 0.2$, with this the linear ODE system becomes:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -0.5 & 0 & 0 \\ 0.5 & -0.25 & 0 \\ 0 & 0.25 & 0.2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.36}$$

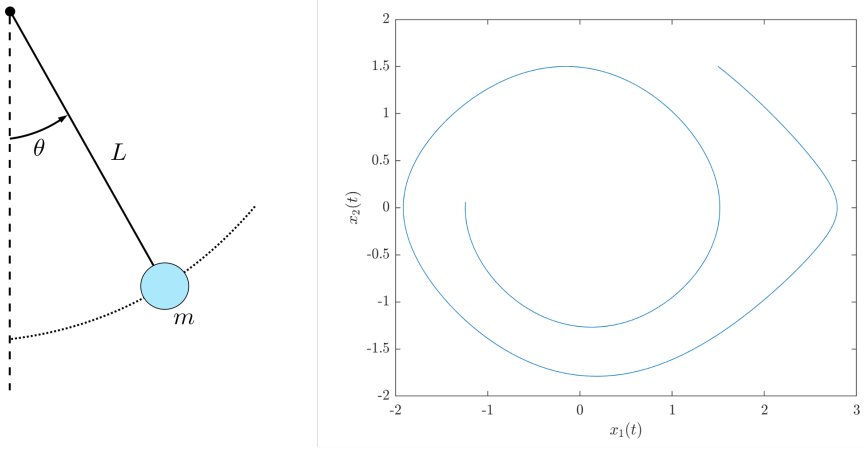
Figure 2.8b shows an example of the solution of Equation 2.36 with initial condition $X_0 = \begin{bmatrix} 15 & 0 & 0 \end{bmatrix}$.

Using the ODE solver ODE45 from the Matlab suite the *training-dataset-0* containing three trajectories with initial conditions $\begin{bmatrix} 10.3 & 2.0 & 3.5 \end{bmatrix}$, $\begin{bmatrix} 13.2 & 2.8 & 4.2 \end{bmatrix}$ and $\begin{bmatrix} 10.5 & 1.7 & 0.2 \end{bmatrix}$ in the time interval $\begin{bmatrix} 0 & 30 \end{bmatrix}$ was generated as training data. The initial conditions were generated sampling from a uniform distribution in the case of x_1 from the interval $\begin{pmatrix} 10 & 15 \end{pmatrix}$ and for x_2 and x_3 from the interval $\begin{pmatrix} 0 & 5 \end{pmatrix}$. The *testing-dataset-0* was generated using 1000 trajectories, the initial conditions for these trajectories were generated, sampling from the same distribution as in the training dataset.

2.4.2 Almost linear ODE system: Damped pendulum

Figure 2.9a shows a simple pendulum, a mass m swinging back and forth, attached to a mass-less rod of length L; the position of the pendulum on time is described by the angle $\theta(t)$ to the vertical. The second order ODE in Equation 2.37 describes the angle $\theta(t)$, where μ is a constant accounting for the air resistance.

$$\frac{d^2\theta}{dt^2} + \mu\frac{d\theta}{dt} + \frac{g}{L}\sin(\theta) = 0 \tag{2.37}$$



(a) (b)

Figure 2.9: (a) Damped pendulum, (b) Solution of Equation 2.39 with initial condition $X_0 = \begin{bmatrix}1.5 & 1.5\end{bmatrix}$.

Doing the variable substitution $x_1 = \theta$ and $x_2 = \frac{d\theta}{dt}$, the almost linear first-order ODE system equivalent to Equation 2.37 is:

$$\begin{aligned}\frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -\frac{g}{L}\sin x_1 - \mu x_2\end{aligned} \tag{2.38}$$

The parameters selected were $g/L = 1$ and $\mu = 0.1$, then the system becomes Equation 2.39. Figure 2.9b shows and example of a trajectory with initial condition $X_0 = \begin{bmatrix}1.5 & 1.5\end{bmatrix}$ for the time interval $\begin{bmatrix}0 & 15\end{bmatrix}$.

$$\begin{aligned}\frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -\sin x_1 - 0.1x_2\end{aligned} \tag{2.39}$$

Using the ODE solver ODE45 from the Matlab suite the *training-dataset-1* containing one trajectory with initial condition $\begin{bmatrix}2 & 1\end{bmatrix}$ in the time interval $\begin{bmatrix}0 & 40\end{bmatrix}$ was generated as training data. Due to the spiral behaviour that winds towards zero, one long trajectory

was used as training data. The *testing-dataset-1* was generated using 1000 trajectories, and the initial conditions were generated, sampling from a uniform distribution over the interval $\begin{pmatrix}-1.5 & 1.5\end{pmatrix}$ for both x_1 and x_2.

2.4.3 Nonlinear ODE system: Predator-prey system

The nonlinear ODE system in Equation 2.40 is known as the predator-prey equation that is used to describe the dynamics of the interaction between predator and prey in a natural environment. In this model, the number of prey is denoted by $x_1(t)$ and the number of predators is denoted by $x_2(t)$; a is the natural grow rate of prey in the absence of predators; b is the natural decline rate of predators in the absence of prey; p and q are the constants modelling the interactions between prey and predators.

$$\begin{aligned}\frac{dx_1}{dt} &= ax_1 - px_1x_2 \\ \frac{dx_2}{dt} &= -bx_2 + qx_1x_2\end{aligned} \tag{2.40}$$

The parameters selected were $a = 200$, $b = 150$, $p = 4$ and $q = 2$. An example of the system obtained is shown in Figure 2.10

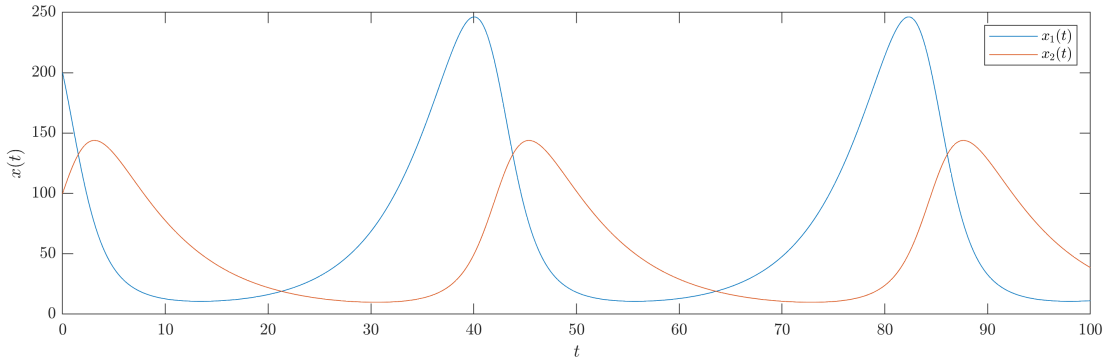


Figure 2.10: Solution of Equation 2.40
with parameters $a = 200$, $b = 150$, $p = 4$, and $q = 2$, for initial condition $X_0 = \begin{bmatrix}200 & 100\end{bmatrix}$.

Using the ODE solver ODE45 from the Matlab suite the *training-dataset-2*, containing three trajectories with initial conditions $\begin{bmatrix}328.2 & 32.2\end{bmatrix}$, $\begin{bmatrix}122.9 & 21.5\end{bmatrix}$, and $\begin{bmatrix}108.0 & 119.2\end{bmatrix}$ in the time interval $\begin{bmatrix}0 & 50\end{bmatrix}$, was generated as training data. The initial conditions were generated, sampling from a uniform distribution over the interval $\begin{pmatrix}0 & 350\end{pmatrix}$ for x_1 and $\begin{pmatrix}0 & 200\end{pmatrix}$ for x_2. The *testing-dataset-2* was generated using 1000 trajectories, the initial conditions for these trajectories were generated, sampling from the same distribution as for the training dataset.

2.5 Experiments and Experiments Metrics

In this section, the experiments and experimental metrics proposed to reach the aims are described. The results of these experiments are presented in the next chapter.

2.5.1 Experiment 1 - Testing gradient step methods on single pairs of points

The NODE class has four gradient step methods for evaluating the gradient between two consecutive points Y^i and Y^{i+1} in a training dataset. As these gradient step algorithms are the basis to find the total gradients, it is of paramount importance that these gradients between two points are accurate. The objective in this experiment is to evaluate the accuracy of these methods; the true gradient was considered to be the numerical method. It is also of interest to evaluate the time per gradient evaluation for each method.

For testing these methods, 1000 pair of points with two elements $Y^i = \begin{bmatrix} y_1^i & y_2^i \end{bmatrix}$ and $Y^{i+1} = \begin{bmatrix} y_1^{i+1} & y_2^{i+1} \end{bmatrix}$ were generated randomly, each element was sample from an uniform distribution over the interval $\begin{pmatrix} -0.1 & 0.1 \end{pmatrix}$. Then, the gradient was evaluated with the four gradient step methods and the time per gradient evaluation was recorded.

After that, the distance between the true gradient (Numerical method) and the other methods was evaluated. The metric selected to measure the distance between the gradients was the euclidean distance normalized by the sum of the norms (Equation 2.41). This measurement gives a distance normalized between 0 and 1. Finally, an average of the distances for the 1000 points was used to evaluate the accuracy.

$$Distance = \frac{\left\| \frac{dC}{d\theta} - (\frac{dC}{d\theta})_{approx} \right\|_2}{\left\| \frac{dC}{d\theta} \right\|_2 + \left\| (\frac{dC}{d\theta})_{approx} \right\|_2} \tag{2.41}$$

2.5.2 Experiment 2 - Evaluating gradient step methods on complete training datasets

After finding the accuracy of the different gradient steps algorithms and their speed, it is of interest to evaluate how these differences in gradients of single pairs of points affect the model obtained after a training session, using a complete training dataset.

For this experiment a NODE with one hidden ReLU layer with 100 neurons was used. This NODE was trained using the four different gradient step algorithms, and the cost trajectory for each was recorded. The initial parameters of the NODE were the same at the beginning of each training session, and the GD was used as training algorithm with 100 epochs. With this setup, the cost trajectory and the final model obtained would be the same if the gradient step algorithms give the same gradient.

The cost trajectory, the final cost, and the final model trajectory for the training datasets were used to evaluate the impact of the use of different gradient step algorithms.

2.5.3 Experiment 3 - Investigating the NODE's underlying ANN hyperparameter's influence in final model performance

Now the effect of the NODE's underlying ANN hyperparmeter's (i.e. number of layers and activation functions) in the resulting model precision was evaluated. For this, NODEs with different underlying ANN number of layers and different activation functions were trained. The training error and the testing error were calculated and presented for each combination of hyperparameters. To lower the influence on the results of the initial parameters of the underlying ANN, five different tests with different initial parameters for each combination of hyperparameters were performed, and the average result presented.

2.5.4 Experiment 4 - Exploring different SGD mini-batch size effect on the training cost trajectory

The effect on the mini-batch size of the SGD training algorithm on the final cost and cost trajectory was evaluated. For this, a NODE with an underlying feed-forward ANN with the best hyperparameters from experiment 4 was trained using different batch-sizes; in order to see the advantages of the SGD over the GD algorithm, a test with GD was also done. To lower the influence on the results of the initial parameters of the underlying ANN, five different tests with different initial parameters for each batch-size were performed, and the average result presented.

2.5.5 Experiment 5 - Final model evaluation

NODEs with the gradient step algorithm selected from experiment 2, best underlying ANN hyperparameters selected from experiment 3, and best mini-batch size from experiment 4, were trained using the training datasets. The test error over the testing datasets including 1000 trajectories was evaluated.

In order to get a normalized test and training error, the normalized mean-squared error NMSE shown in Equation 2.42 was used to evaluate the distance between the true trajectory and the NODE approximation.

$$\begin{aligned} MSE(Y_i, U_i) &= \frac{1}{n}\sum_{i=1}^{n}(Y_i - U_i)^2 \\ NMSE(Y_i, U_i) &= \frac{MSE(Y_i, U_i)}{MSE(Y_i, 0)} \end{aligned} \tag{2.42}$$

Where Y^i is the true trajectory, U^i is the approximated trajectory and n is the number of points. A plot showing the frequencies of the NMSE for the test trajectories compared with the NMSE of the training trajectory was used to evaluate the ability of the model to generalize.

Chapter 3

Results

In this chapter the results of the experiments planned in the previous chapter are shown. These experiments were designed to give information that contributes to reaching the aims. The chapter is divided in three sections containing the experiments applied to the data generated, using the three systems of ODE selected. These systems of ODE were selected with the intent that they increase in complexity, starting from a linear system, then going to an almost linear system and ending with a nonlinear system; in this way a wide range of ODE systems is covered that allows generalisation.

The implementation of NODE, built using the Matlab suite, was used to carry out the experiments. Appendix A presents the source code for the NODE class, while appendix B shows the source code for the ANN class.

3.1 Linear ODE System. The Three Stage Tank Salt Content

The following linear system of ODE with three variables was selected as a test system:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -0.5 & 0 & 0 \\ 0.5 & -0.25 & 0 \\ 0 & 0.25 & 0.2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3.1}$$

The *training-dataset-0* shown in Figure 3.1 was generated solving Equation 3.1, it contains three trajectories with initial conditions $\begin{bmatrix} 10.3 & 2.0 & 3.5 \end{bmatrix}$, $\begin{bmatrix} 13.2 & 2.8 & 4.2 \end{bmatrix}$ and $\begin{bmatrix} 10.5 & 1.7 & 0.2 \end{bmatrix}$ in the time interval $\begin{bmatrix} 0 & 30 \end{bmatrix}$. The vector field in three planes for Equation 3.1 is also shown in Figure 3.1.

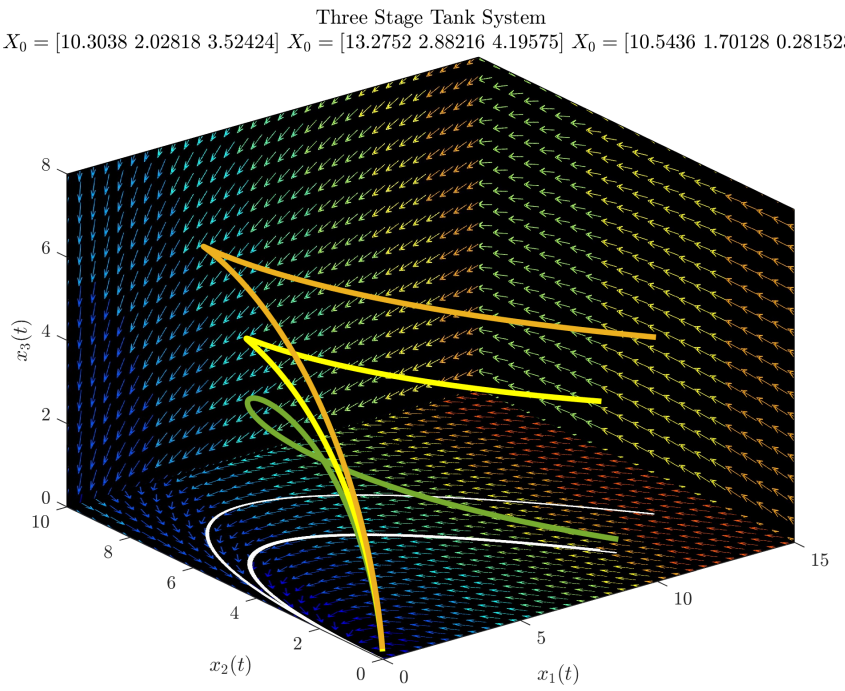


Figure 3.1: Linear case: Equation 3.1 vector field and trajectories for $X_0 = \begin{bmatrix}10.3 & 2.0 & 3.5\end{bmatrix}$, $\begin{bmatrix}13.2 & 2.8 & 4.2\end{bmatrix}$ and $\begin{bmatrix}10.5 & 1.7 & 0.2\end{bmatrix}$.

3.1.1 Experiment 1 - Testing gradient step methods on single pairs of points

The NODE class has four methods for evaluating the gradient between two consecutive points Y^i and Y^{i+1}. For testing these methods, one thousand pairs of points with two elements $Y^i = \begin{bmatrix}y_1^i & y_2^i\end{bmatrix}$ and $Y^{i+1} = \begin{bmatrix}y_1^{i+1} & y_2^{i+1}\end{bmatrix}$, in which each element y was generated randomly from an uniform distribution in the interval $\begin{pmatrix}-0.1 & 0.1\end{pmatrix}$ were used.

A NODE with underlying ANN with one hidden ReLU layer with 100 neurons was used for this test. The gradient was calculated for each of the one thousand pairs of points, and then the distance between the true gradient (i.e. Numerical method) and the other methods was measured. The metric used to measure the distance between the gradients was the euclidean distance normalised by the sum of the norms NMSE (Equation 2.41).

As the accuracy of the back-propagation method based on the Euler method depends on the step size selected, five different step sizes were evaluated, from $1e-6$ to $1e-2$; the adjoint, and adjoint-modified methods use the $ODE45$ solver that selects the step size automatically. Figure 3.2b shows the error in the gradient calculation from the three methods considered, back-propagation, adjoint and adjoint-modified. When the step size is set very small (e.g. $1e-6$ to $1e-4$) for the back-propagation method, it becomes the most accurate method, and then as the step size is set bigger (e.g. $1e-2$), the error increases over the two other methods. The adjoint-modified has an average error of $3.15e-4$ that is approximately double the error of the adjoint method, i.e. $1.46e-4$.

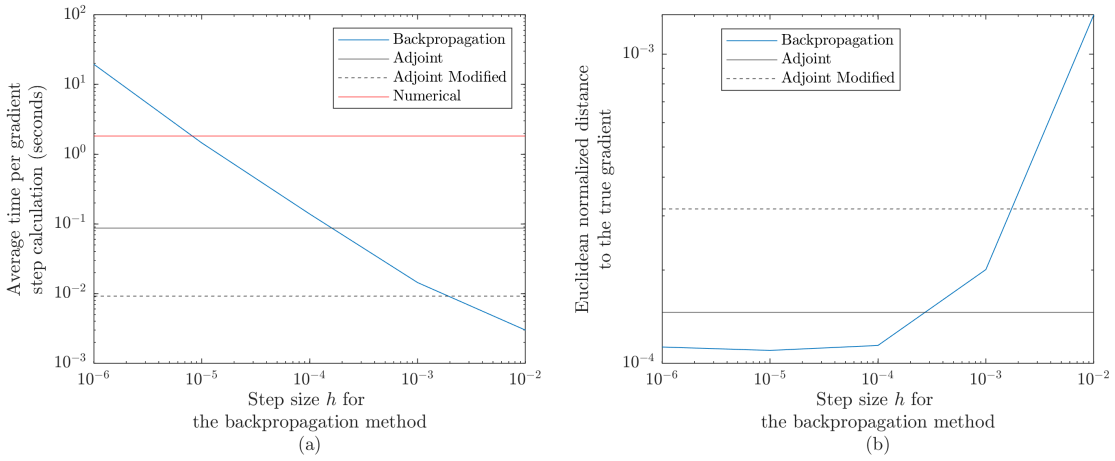


Figure 3.2: One-thousand gradient calculation for two consecutive points, (a) Average time per gradient calculation, (b) Average Euclidean normalized distance between the numerical calculated gradient and the gradient calculated by back-propagation, adjoint and adjoint-modified methods

Figure 3.2a shows the adjoint-modified method had a time per gradient step approximately 10 times smaller than the adjoint method, and 100 times smaller than the numeric method. The back-propagation time per gradient step depends greatly on the time step selected, having a time per gradient similar to the adjoint and adjoint-modified methods when the accuracy is matched.

3.1.2 Experiment 2 - Evaluating gradient step methods on complete training datasets

To test the effect of the differences in the gradient calculated for two consecutive points observed in Figure 3.2, four NODEs were trained with the *training-dataset-0* using the four distinct gradient algorithms. The NODEs used had an underlying ANN with one hidden ReLU layer with 100 neurons. For each training session, the same initial parameters (weights and biases) were loaded to the NODE and the gradient descent method was used as a training algorithm. This implies that if the gradients obtained with different methods were equal, the cost trajectory and trained NODE approximation of the training trajectory should be exactly the same.

The cost trajectory for each method (Figure 3.3a) remained perfectly overlapped for the whole training session; this means that the gradient calculations for the different methods remained very close. Figure 3.3(b)(c)(d)(e) shows that the paths reconstructed by the NODEs, generated with gradients calculated by all the methods, are indistinguishable. Thus, any of the methods had produced the same final parameters. Due to the speed, all the tests below in this section were done using the adjoint-modified method for gradient calculations.

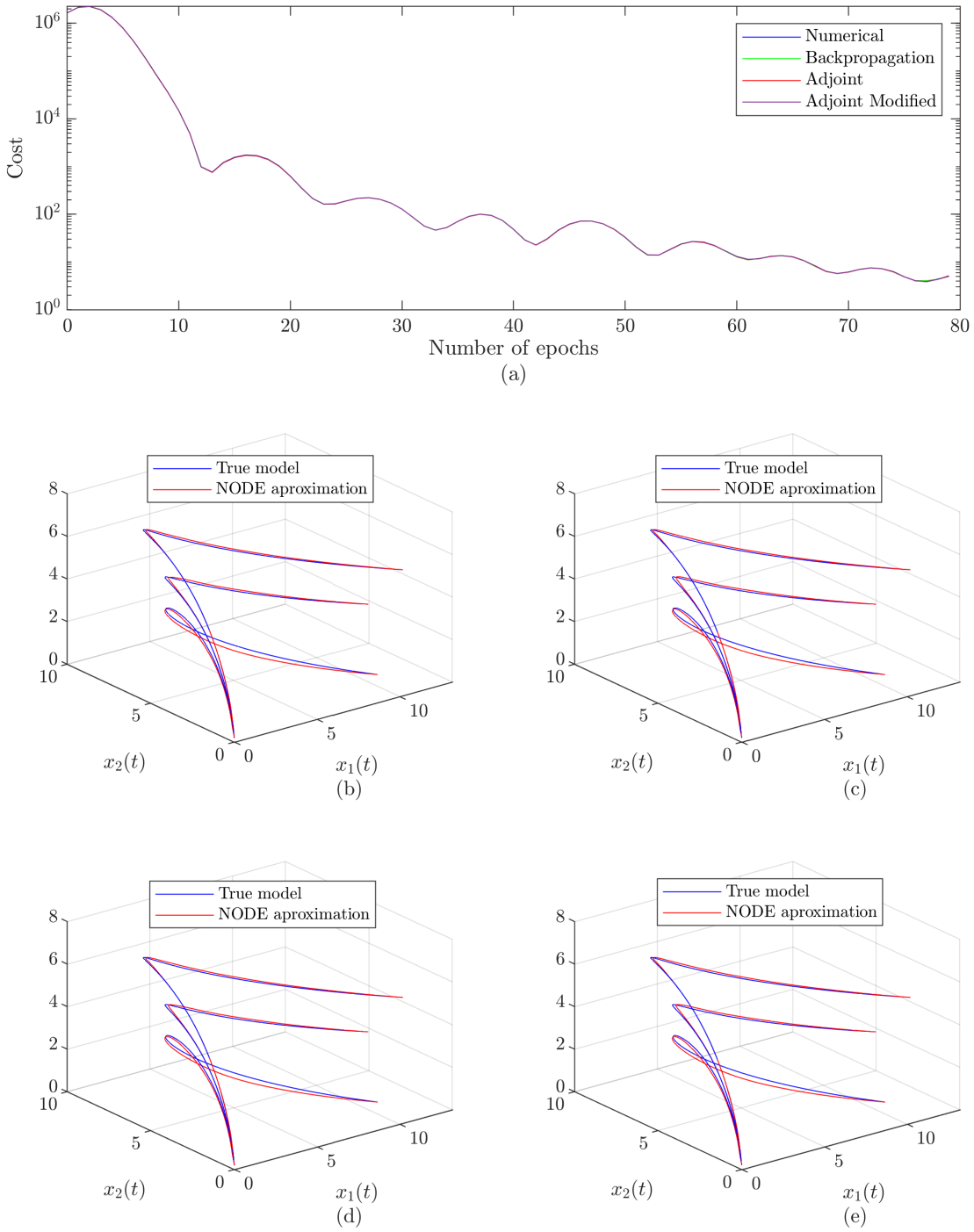


Figure 3.3: Linear case: Training of a NODE with different gradient step algorithms, (a) Learning curves, cost versus epochs, (b) Numerical gradient, (c) Backpropagation gradient, (d) Adjoint gradient, (e) Adjoint-modified gradient.

3.1.3 Experiment 3 - Investigating the NODE's underlying ANN hyperparameter's influence in final model performance

Now the effect of the hyperparameters (i.e. number of layers and activation function) of the underlying ANN is going to be evaluated. The *training-dataset-0* was used as a training dataset. One trajectory of the *testing-dataset-0* was used as a testing dataset. Eighteen NODEs with unique hyperparameters were tested. The activation functions considered were: sigmoid, hyperbolic tangent sigmoid, and ReLU. Six sizes of NODE were tested, all had a total of 100 neurons in the hidden layers evenly distributed, as shown in Figure 3.4. Each test was repeated five times and the average results presented, this to reduce the influence of the random parameters initialization in the results.

$$
\begin{array}{ll}
\text{NODE-1:} & \begin{bmatrix} 3 & 100 & 3 \end{bmatrix} \\
\text{NODE-2:} & \begin{bmatrix} 3 & 50 & 50 & 3 \end{bmatrix} \\
\text{NODE-3:} & \begin{bmatrix} 3 & 33 & 34 & 33 & 3 \end{bmatrix} \\
\text{NODE-4:} & \begin{bmatrix} 3 & 25 & 25 & 25 & 25 & 3 \end{bmatrix} \\
\text{NODE-5:} & \begin{bmatrix} 3 & 20 & 20 & 20 & 20 & 20 & 3 \end{bmatrix} \\
\text{NODE-6:} & \begin{bmatrix} 3 & 15 & 17 & 17 & 17 & 17 & 17 & 3 \end{bmatrix}
\end{array}
$$

Figure 3.4: Linear case: NODE sizes for hyperparameters tests.

The ReLU activation performed better for any size of NODE (Figure 3.5a). On the other hand, Figure 3.5(b)(c) shows that the best NODE performer in terms of testing and training error was the shallowest network with only one hidden layer.

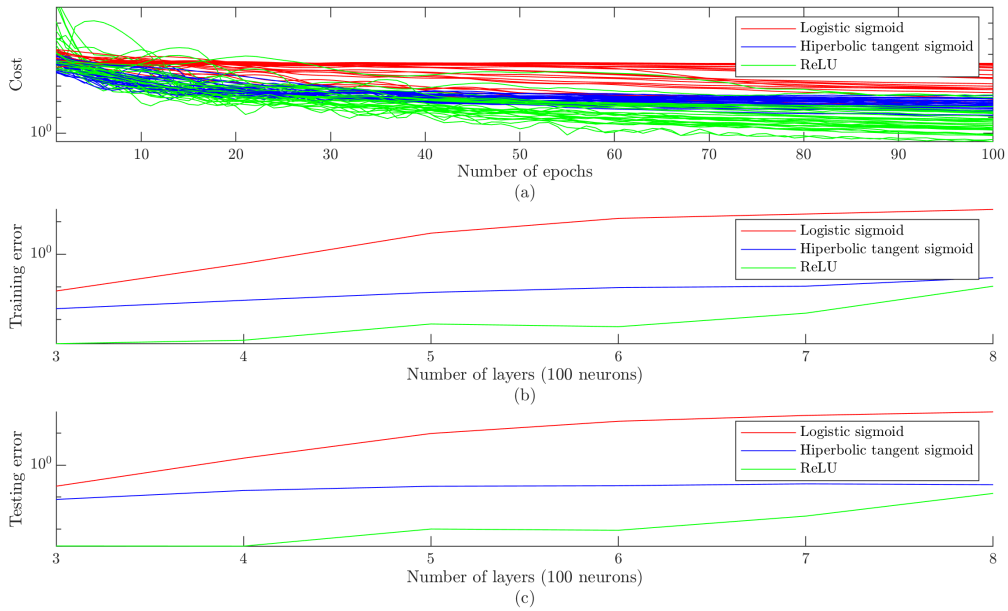


Figure 3.5: Linear case: Underlying ANN hyperparameter tests, (a) Cost trajectories for each test, (b) Average training error, (c) Average testing error.

3.1.4 Experiment 4 - Exploring different training algorithms and their effect on the training cost trajectory

In this section, two different training algorithms were evaluated, the gradient descent that had been used in all the tests in this section and the stochastic gradient descent with different *mini-batch* sizes. The underlying ANN hyperparameters that were used for all the NODEs were the optimal ones found in the previous section, one hidden ReLU layer with 100 neurons. Five tests were done for each training algorithm, and average values were presented.

In Figure 3.6 it can be seen that the SGD algorithm with any of the considered *mini-batch* sizes performed better than the GD algorithm. The GD cost curve is more stable but it decreases very slowly, compared with the SGD algorithm. The SGD with *mini-batch* sizes of 5, 10 and 20 reached the absolute minimum faster, but their training trajectories were very unstable. For this reason, the SGD with a *mini-batch* size of 50 was selected as the best performer.

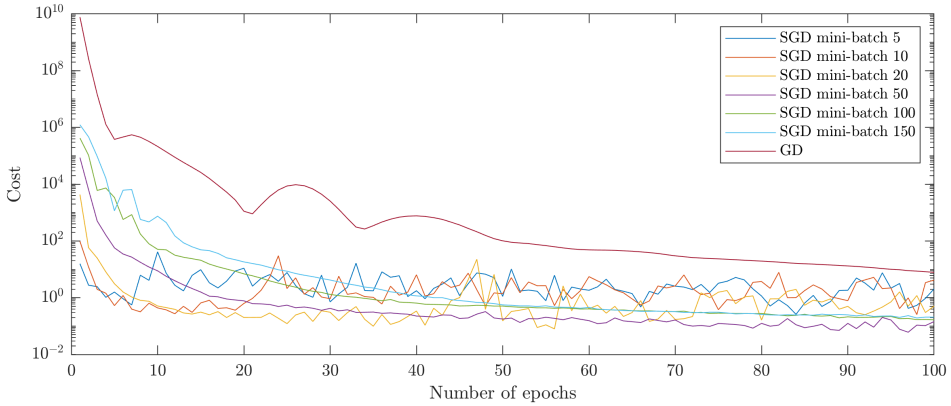


Figure 3.6: Linear case: Average cost trajectory for the gradient descent algorithm and the stochastic gradient descent (different *mini-batch* sizes) for five tests.

3.1.5 Experiment 5 - Final model evaluation

A final NODE with the optimal hyperparameters found in the previous sections was trained with *training-dataset-0* (3 trajectories) and its performance evaluated with the *testing-dataset-0* (1000 trajectories). A NODE with an underlying ANN with one ReLU hidden layer with 100 neurons was used. It was trained with SGD with *mini-batch* size of 50. All gradients were calculated using the adjoint-modified method.

Figure 3.7a shows the training process using SGD of the optimal NODE, the training was restricted to 100 epochs. The resulting NODE can reproduce the training trajectory with a high degree of accuracy (Figure 3.7b). The first four testing trajectories and the corresponding NODE approximation are shown in Figure 3.7(c)(d)(e)(f). These trajectories were effectively reconstructed even though they had never been seen by the NODE.

The trajectory in Figure 3.7c is even closer to the true model than the training data. To measure the distance between trajectories, the NMSE metric was used (Equation 2.42).

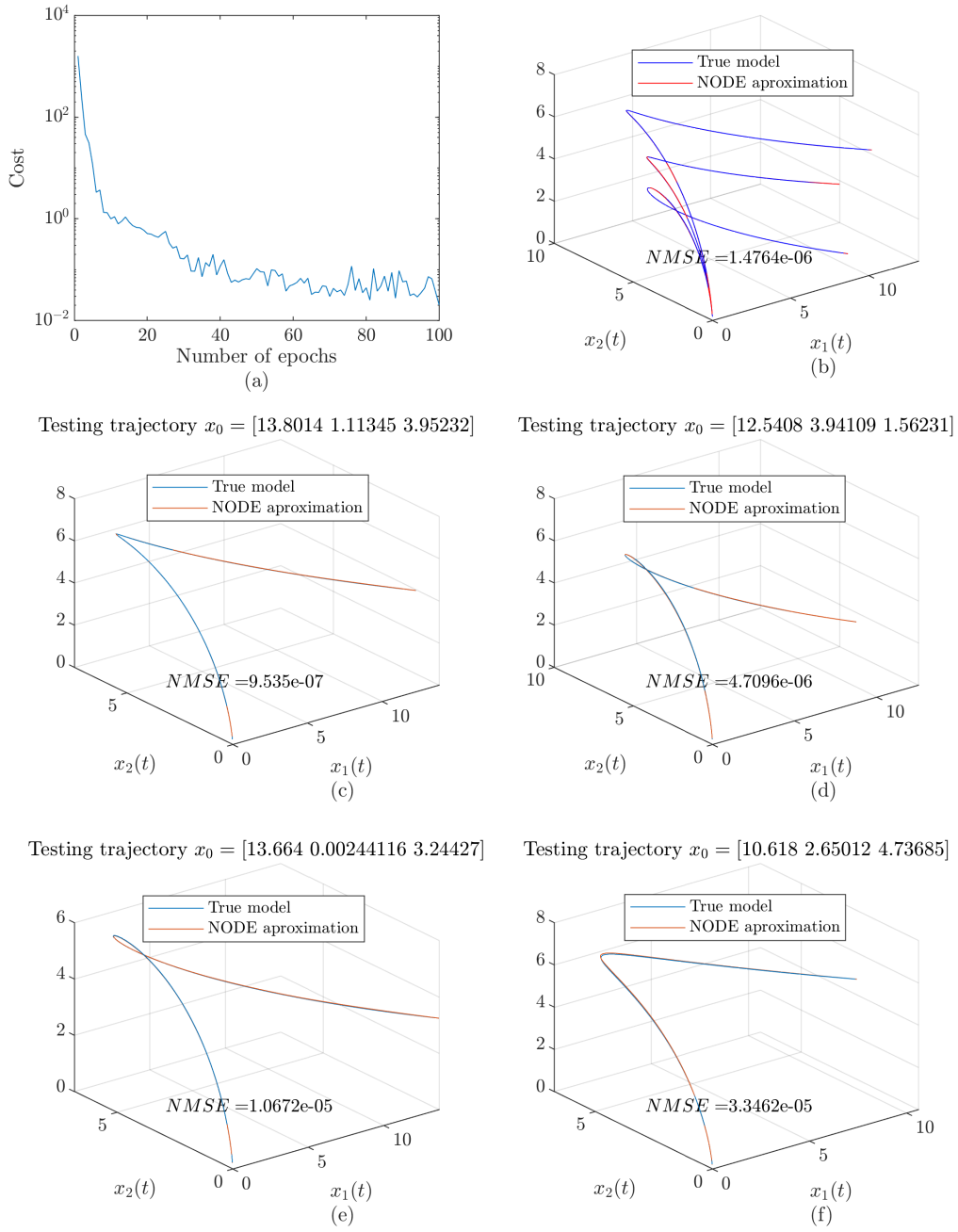


Figure 3.7: Linear case: (a) Cost trajectory for NODE with optimal hyperparameters trained with *training-dataset-0*, (b) NODE approximation for the training trajectory, (c) NODE approximation for first trajectory in *testing-dataset-0*, (d) NODE approximation for second trajectory in *testing-dataset-0*, (e) NODE approximation for third trajectory in *testing-dataset-0*, (f) NODE approximation for fourth trajectory in *testing-dataset-0*.

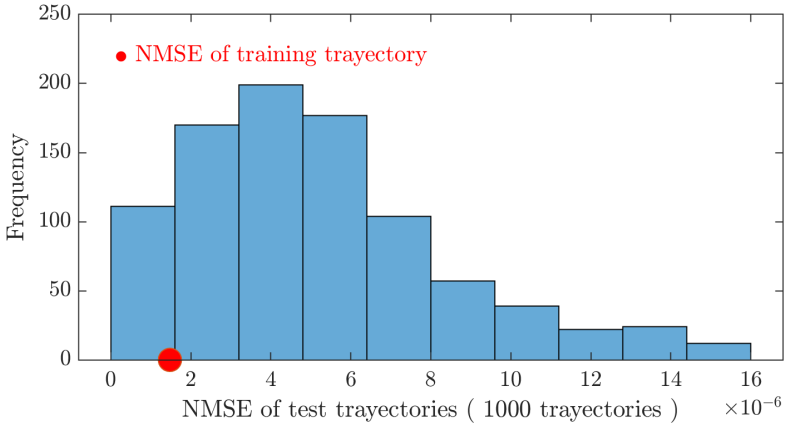


Figure 3.8: Linear case: NMSE for NODE approximation of the *testing-dataset-0* (1000 trajectories).

A histogram with the summary of the distance between the NODE approximation and the true trajectory for the *testing-dataset-0* with one thousand trajectories is shown in the Figure 3.8. With NMSE in the order of 10^{-5} and 10^{-6}, the NODE was able to decode the underlying dynamics of the ODE system in Equation 3.1. More than 10% of the testing trajectories approximated by the NODE are closer to the truth trajectory than the NODE approximation of the training data.

3.2 Almost Linear ODE system. The Damped Pendulum

The following almost linear system of ODE with two variables was selected as a test system:

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -\sin x_1 - 0.1x_2 \end{aligned} \tag{3.2}$$

The *training-dataset-1* shown in Figure 3.9 was generated solving Equation 3.2 using the ODE solver ODE45 from the Matlab suite; it contains one trajectory with initial condition $X_0 = \begin{bmatrix} 2 & 1 \end{bmatrix}$, in the time interval $\begin{bmatrix} 0 & 40 \end{bmatrix}$. The vector field for Equation 3.2 is also shown in Figure 3.9.

3.2.1 Experiment 2 - Evaluating gradient step methods on complete training datasets

To evaluate the effect of the differences in the gradient calculated for two consecutive points observed in Figure 3.2, four NODEs were trained with *training-dataset-1* using the

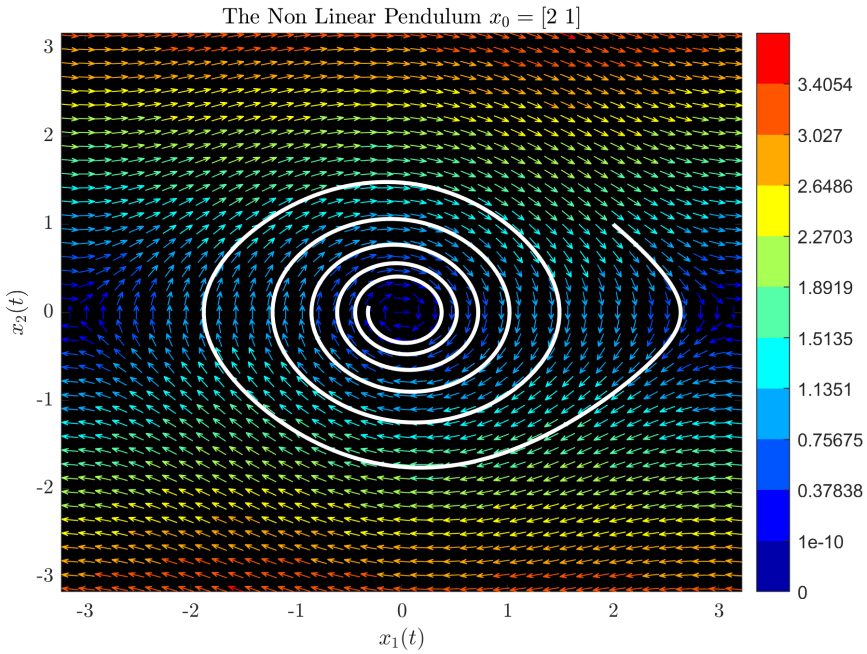


Figure 3.9: Almost linear case: Equation 3.2 vector field and trajectory for $X_0 = \begin{bmatrix} 2 & 1 \end{bmatrix}$.

four distinct gradient algorithms. The NODEs used have an underlying ANN with one hidden ReLU layer with 100 neurons. For each training session the same initial parameters were loaded to generate the same cost trajectory, in the case that the gradient calculated by the different methods were the same.

The cost trajectory for each method (Figure 3.10a) overlapped at the beginning of the training process until around epoch 50; after that, the trajectories separated, but followed almost the same general trend. This indicates that the gradients found by the four methods differed slightly, but these small differences did not affect the final cost in any significant way.

Figure 3.10(b)(c)(d)(e) shows that the paths reconstructed by the NODEs generated with gradients calculated by the numerical method, the adjoint and the adjoint-modified, are almost indistinguishable. These results demonstrate that the small differences in the gradient calculated by different algorithms did not affect the training process, as the final cost and final model differed in a minor way. Due to the speed, all the tests within this section will use the adjoint-modified method for gradient calculations.

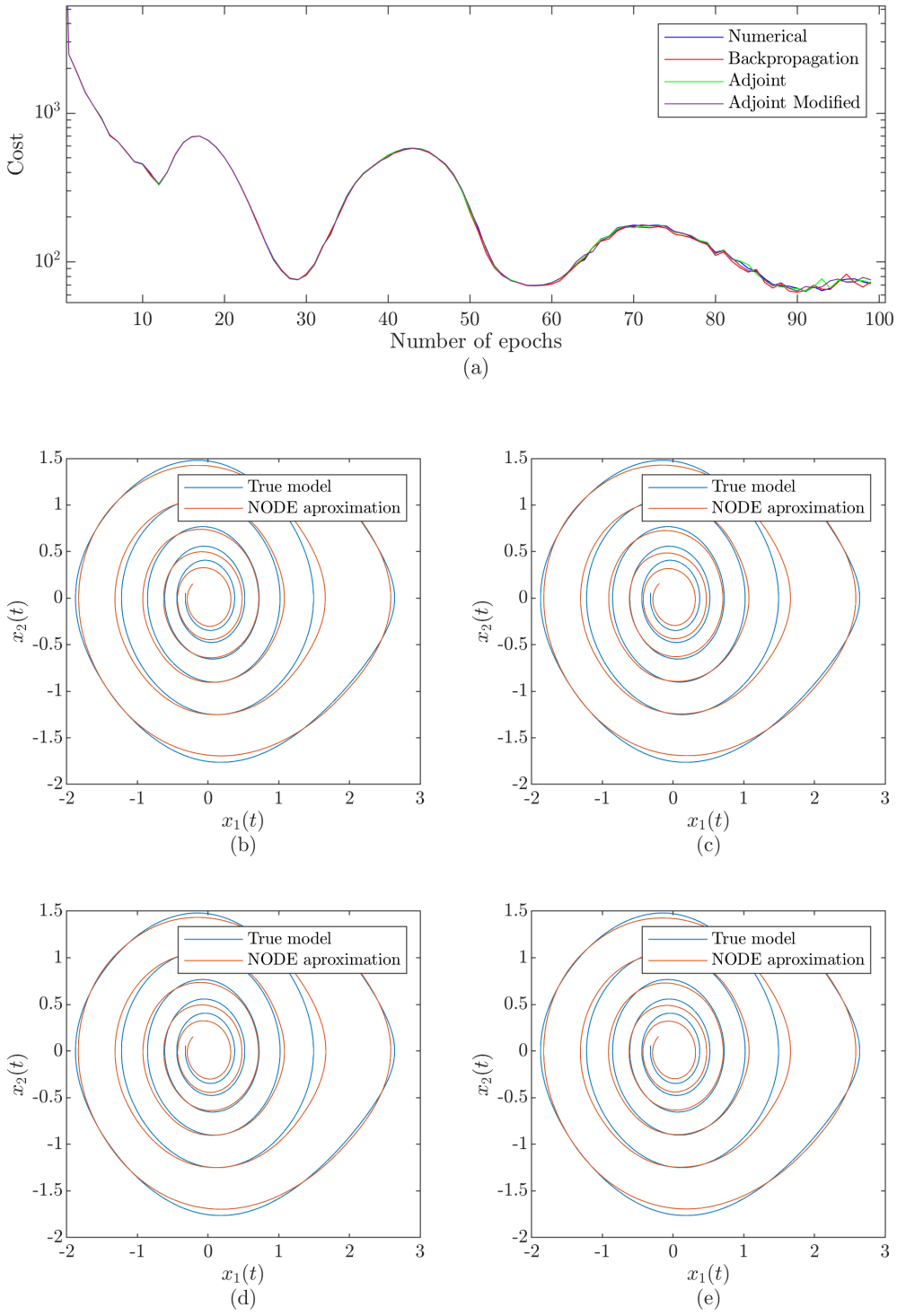


Figure 3.10: Almost linear case: Training of a NODE with different gradient step algorithms, (a) Learning curves, cost versus epochs, (b) Numerical gradient, (c) Backpropagation gradient, (d) Adjoint gradient, (e) Adjoint-modified gradient.

3.2.2 Experiment 3 - Investigating the NODE's underlying ANN hyperparameters influence in final model performance

Now the effect of the hyperparameters of the underlying ANN is going to be evaluated. The *training-dataset-1* was used as a training dataset and one trajectory of the *testing-dataset-1* was used as a testing trajectory. Thirty NODEs with unique hyperparameters were tested. The activation functions considered were: Sigmoid, hyperbolic tangent sigmoid, and ReLU. Ten sizes of NODE were tested. All had a total of 100 neurons in the hidden layers, evenly distributed, as shown in Figure 3.11. Each test was repeated five times and the average results presented, this to reduce the influence of the random parameters initialization in the results.

```
NODE-1: [2  100  2]
NODE-2: [2  50  50  2]
NODE-3: [2  33  34  33  2]
NODE-4: [2  25  25  25  25  2]
NODE-5: [2  20  20  20  20  20  2]
NODE-6: [2  15  17  17  17  17  17  2]
NODE-7: [2  14  14  14  16  14  14  14  2]
NODE-8: [2  12  12  12  16  12  12  12  12  2]
NODE-9: [2  11  11  11  11  12  11  11  11  11  2]
NODE-10: [2  10  10  10  10  10  10  10  10  10  10  2]
```

Figure 3.11: Almost linear case: NODE sizes for hyperparameters tests.

All the NODEs that used the sigmoid activation function performed poorly. The hyperbolic tangent sigmoid and the ReLU activation performed similarly, but the best performance was achieved with the ReLU activation and four layers (two hidden layers); the hidden layers had 50 neurons per layer. From Figure 3.12, it is clear that the hyperparameter selection of the underlying ANN affects the performance of the NODE to a great extent.

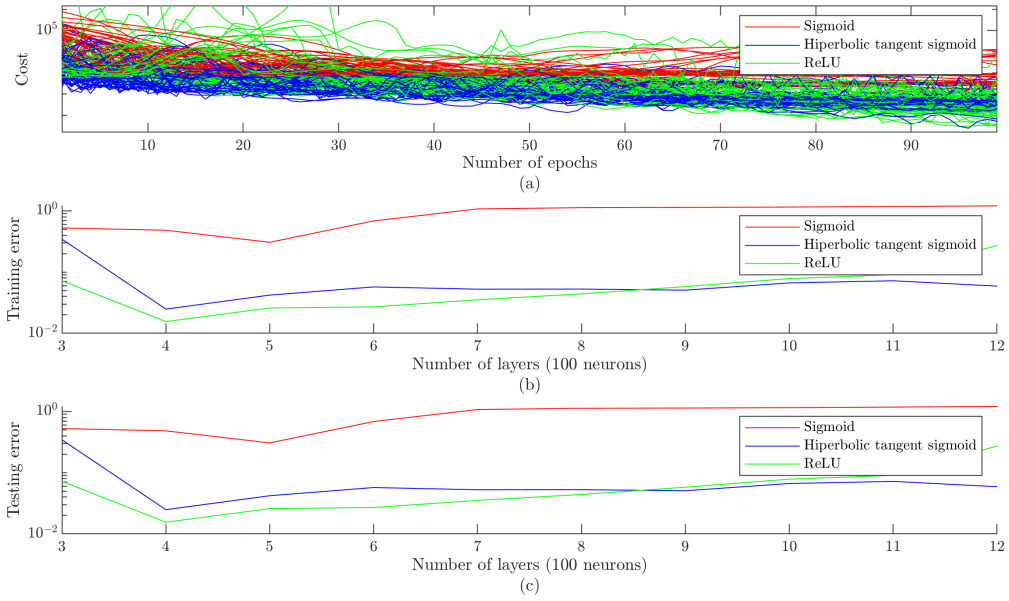


Figure 3.12: Almost linear case: Underlying ANN Hyperparameters tests, (a) Cost trajectories for each test, (b) Average training error, (c) Average testing error.

3.2.3 Experiment 4 - Exploring different training algorithms and their effect on the training cost trajectory

Even though the hyperparameters of the NODE for the test done in Figure 3.10 were very close to the optimal parameters, the results were very poor after 100 epochs. In this section, two different training algorithms are going to be evaluated, the gradient descent that had been used in all the tests in this section and the stochastic gradient descent with several *mini-batch* sizes. The underlying ANN hyperparameters that were used for all the NODEs were the optimal ones found in the previous section. Five tests were done for each training algorithm.

In Figure 3.13, it can be seen that the SGD algorithm with any of the considered *mini-batch* sizes performed better than the GD algorithm. The SGD with batch size of 25 reaches the absolute minimum faster, but it is very unstable, going even above the GD cost at the end of the training process. For this reason, the SGD with a *mini-batch* size of 50 was selected as the best performer.

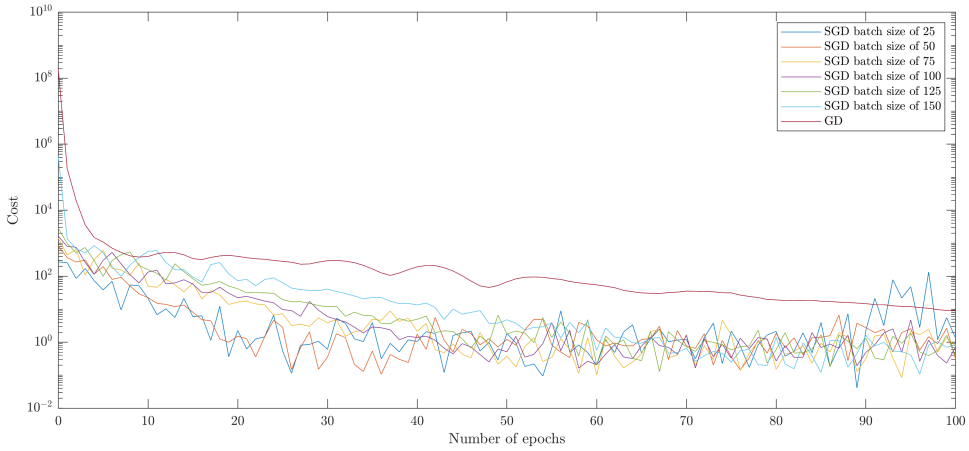


Figure 3.13: Almost linear case: Average cost trajectory for gradient descent algorithm and stochastic gradient descent (different *mini-batch* sizes) for five tests.

3.2.4 Experiment 5 - Final model evaluation

The Figure 3.14a shows the training process using the optimal training algorithm SGD (50 elements *mini-batch* size) of the optimal NODE (Two hidden ReLU layers with 50 neurons). The training process found the target cost only after 80 epochs. The resulting NODE can reproduce the training data with high degree of accuracy, as shown in Figure 3.14b.

The *testing-dataset-1* with 1000 trajectories was used to test the model obtained. In Figures 3.14(c)(d)(e)(f) the first four testing trajectories are shown, accompanied by the approximation generated by the trained NODE. These trajectories could be reconstructed with a high degree of accuracy, even though they had never been seen by the NODE. The

trajectories approximated by the NODE in Figures 3.14(e)(f) are even closer to the true model than the approximation of the NODE for the training data.

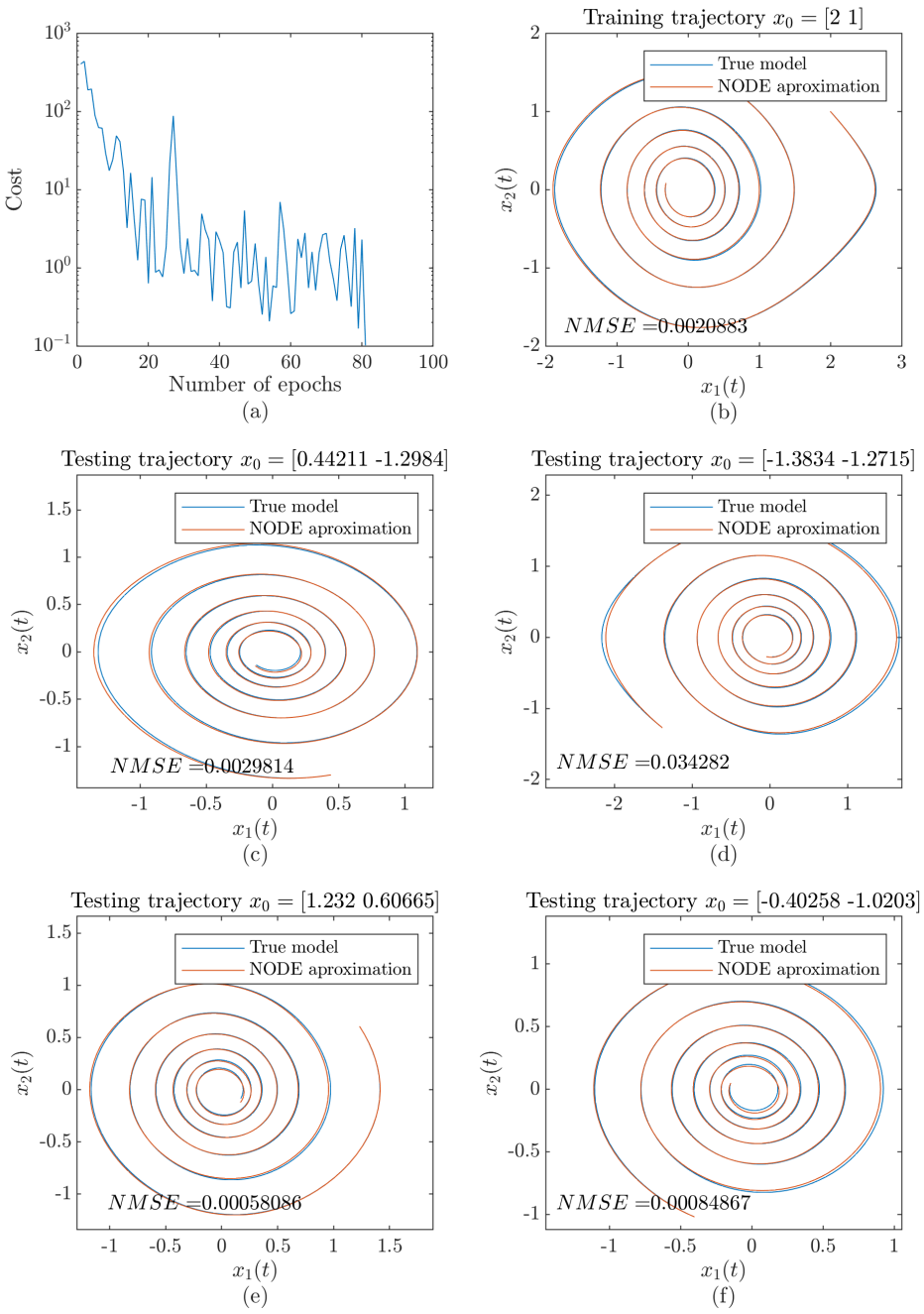


Figure 3.14: Almost linear case: (a) Cost trajectory for NODE with optimal hyperparameters trained with *training-dataset-1*, (b) NODE approximation for the training trajectory, (c) NODE approximation for first trajectory in *testing-dataset-1*, (d) NODE approximation for second trajectory in *testing-dataset-1*, (e) NODE approximation for third trajectory in *testing-dataset-1*, (f) NODE approximation for fourth trajectory in *testing-dataset-1*.

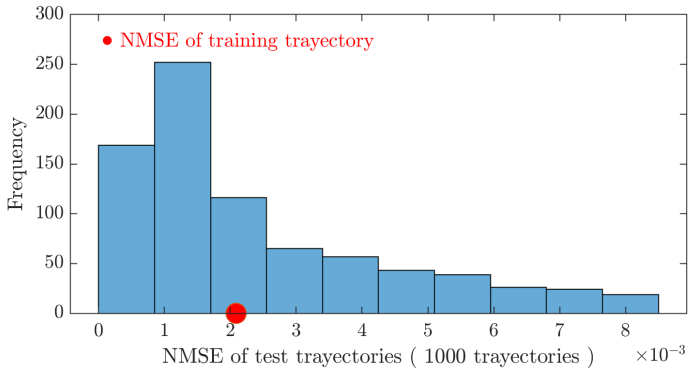


Figure 3.15: Almost linear case: NMSE for NODE approximation of the *testing-dataset-1* (1000 trajectories).

A histogram with the summary of the result of the distance (i.e. error) of the approximation of the trained NODE with the one thousand trajectories in the *testing-dataset-1* is shown in Figure 3.15. Nearly half of the approximated trajectories have the same NMSE or better NMSE than the approximation of the training trajectory. The NODE trained with only one trajectory was able to decode effectively the behaviour of the ODE system in Equation 3.2.

3.3 Non-linear ODE system: Predator-prey system

The following non-linear system of ODE with two variables was selected as a test system:

$$\begin{aligned} \frac{dx_1}{dt} &= 200x_1 - 4x_1x_2 \\ \frac{dx_2}{dt} &= -150x_2 + 2x_1x_2 \end{aligned} \tag{3.3}$$

The *training-dataset-2* shown in Figure 3.16 was generated solving Equation 3.3. It contains three trajectories with initial conditions $X_0 = \begin{bmatrix}328.2 & 32.2\end{bmatrix}$, $\begin{bmatrix}122.9 & 21.5\end{bmatrix}$ and $\begin{bmatrix}108.0 & 119.3\end{bmatrix}$ in the time interval $\begin{bmatrix}0 & 50\end{bmatrix}$. The vector field in three planes for Equation 3.3 is also shown in Figure 3.16.

3.3.1 Experiment 2 - Evaluating gradient step methods on complete training datasets

Due to the differences in the gradient calculated between two points using different methods observed in experiment 1 (Figure 3.2), four NODEs were trained with the *training-dataset-2*, using three gradient algorithms, and the resulting models and training trajectory were evaluated. Initially a NODE with an underlying ANN with one hidden ReLU layer

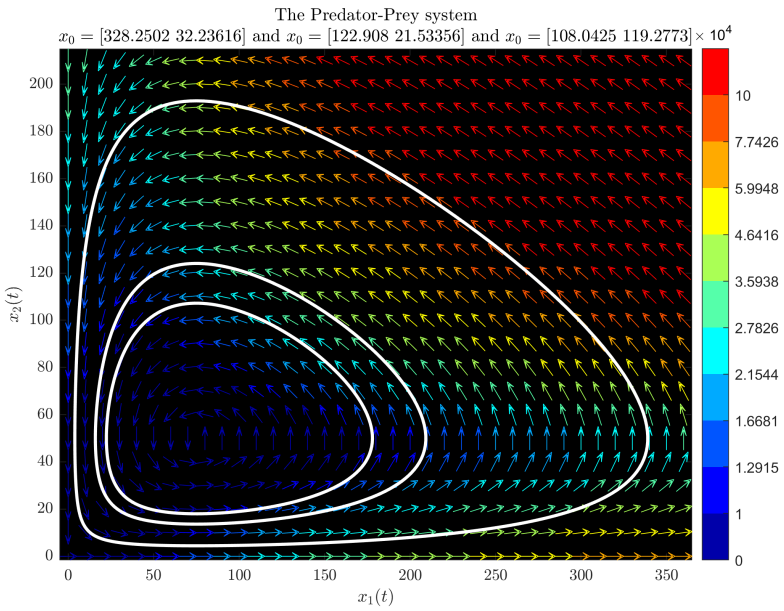


Figure 3.16: Non-linear linear case: Equation 3.3 vector field and trajectories for $X_0 = \begin{bmatrix}328.2 & 32.2\end{bmatrix}$, $\begin{bmatrix}122.9 & 21.5\end{bmatrix}$ and $\begin{bmatrix}108.0 & 119.3\end{bmatrix}$.

with 100 neurons was attempted, but the model was not able to converge with either GD or SGD with different *mini-batch* sizes. The number of layers had to be increased to two, and the SGD algorithm had to be used to achieve convergence. The NODE used for this test had an underlying ANN with two ReLU layers with 100 neurons each; it was trained using SGD with *mini-batch* size of 50. Due to the size of the NODE, it was not feasible to train the network with the gradient generated with the numerical method, because the training time was several days; the NODE was trained using the back-propagation, adjoint and adjoint-modified gradient step methods.

Although it was expected that the cost trajectories would have remained similar, in this case the trajectories diverged substantially as shown in Figure 3.17a. This indicates that the gradients calculated with the different methods were different. But even though the trajectories were different, they all found a similar minimum cost at different places in the training session.

Figure 3.17(b)(c)(d) shows the paths reconstructed by the NODEs generated with gradients calculated by three methods. Despite the fact that the cost trajectories followed different paths, the approximations of the training trajectories found by the different NODEs do not seem substantially different. This is because the minimum cost for each method was similar, and the parameters saved at the end of the training (by design of the NODE class) are the ones that generated the minimum cost in the whole training session. Due to the benefit in the speed and the final similar results, the adjoint-modified method was used for the following experiments 3 and 4.

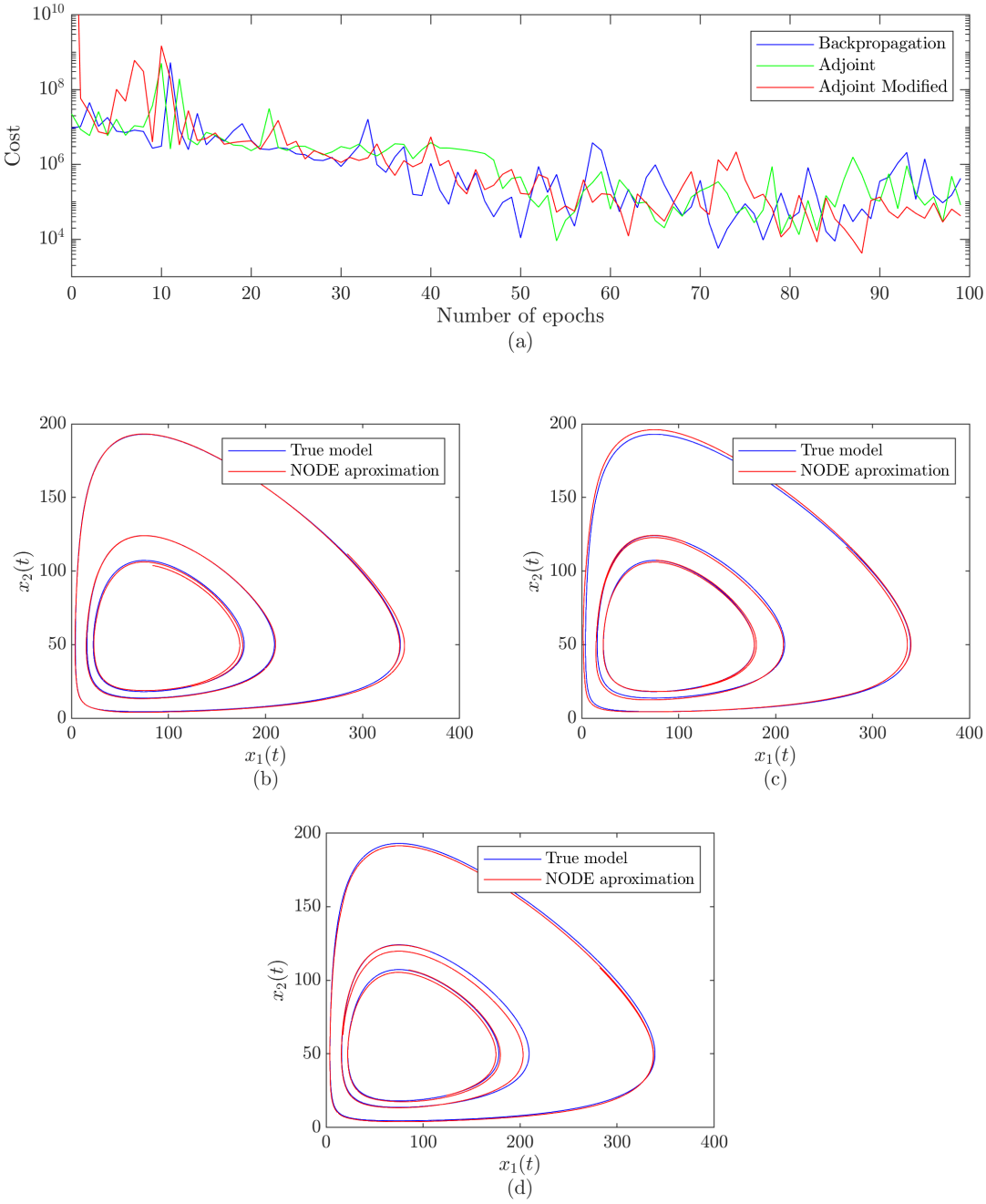


Figure 3.17: Non-linear linear case: Training of a NODE with different gradient step algorithms, (a) Learning curves, cost versus epochs, (b) Backpropagation gradient, (c) Adjoint gradient, (d) Adjoint-modified gradient.

3.3.2 Experiment 3 - Investigating the NODE's underlying ANN hyperparameters influence in final model performance

The effect of the hyperparameters (i.e. number of layers and activation function) of the underlying ANN was evaluated with this experiment. The *training-dataset-2* was used as a training dataset; one trajectory of the *testing-dataset-2* was used as a testing dataset. Twenty-four NODEs with unique hyperparameters were tested. Six sizes of NODE were tested, each with a total of 200 neurons in the hidden layers, distributed as shown in Figure 3.18. As before, each test was repeated five times.

NODE-1: [2 100 100 2]
NODE-2: [2 67 66 67 2]
NODE-3: [2 50 50 50 50 2]
NODE-4: [2 40 40 40 40 40 2]
NODE-5: [2 33 33 35 33 33 33 2]
NODE-6: [2 28 28 28 32 28 28 28 2]
NODE-7: [2 25 25 25 25 25 25 25 25 2]
NODE-8: [2 22 22 22 22 24 22 22 22 22 2]

Figure 3.18: NODE sizes for hyperparameters tests.

As for the other cases studied, the ReLU activation performed better for any size of NODE (Figure 3.19a). But contrary to what was seen in the other two cases studied, deeper networks performed better; the network with 9 layers was selected as the best performer (Figure 3.19(b)(c)).

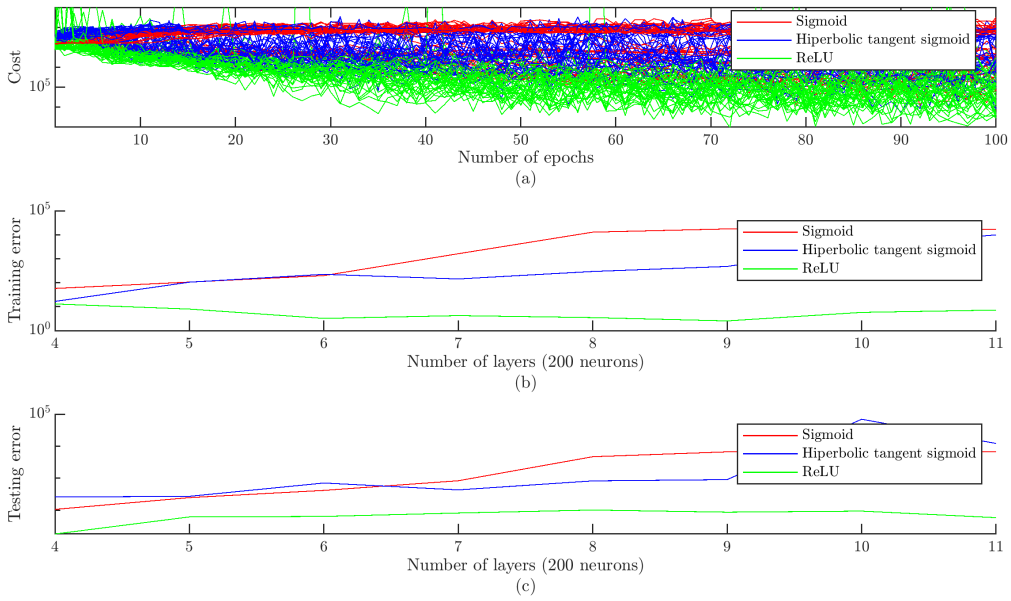


Figure 3.19: Non-linear linear case: Underlying ANN Hyperparameter tests, (a) Cost trajectories for each test, (b) Average training error, (c) Average testing error.

3.3.3 Experiment 4 - Exploring different training algorithms and their effect on the training cost trajectory

With this experiment, the effect on the *mini-batch* size of the SGD algorithm was studied. The underlying ANN hyperparameters that were used for all the NODEs were the optimal ones found in the previous section, which was the NODE-6 in Figure 3.18. Five tests were done for each training algorithm, and average values were presented.

In this case the training was more unstable, compared with the previous cases studied. The SGD with *mini-batch* size 50 reached a better minimum that the other *mini-batch* sizes; for this reason it was selected as the better performer.

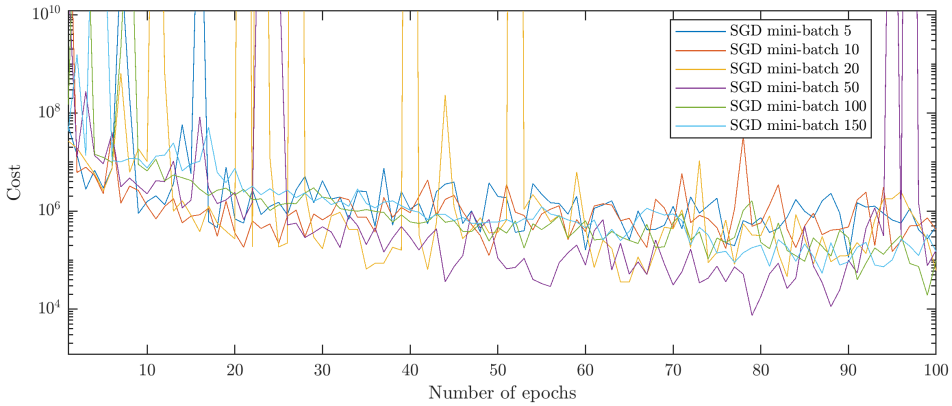


Figure 3.20: Non-linear linear case: Average cost trajectory for gradient descent algorithm and stochastic gradient descent (different *mini-batch* sizes) for five tests.

3.3.4 Experiment 5 - Final model evaluation

The optimal NODE-6 (Figure 3.18) with ReLU hidden layers was trained with 3 trajectories (*training-dataset-2*) and its performance evaluated with one thousand trajectories (*testing-dataset-2).* The NODE was trained with SGD with *mini-batch* size of 50. Due to the results in experiment 2 that showed that the models using different gradient step algorithms differed slightly, the optimal NODE was trained using the adjoint and the adjoint-modified gradient step methods.

Figure 3.21a shows the cost trajectories for both gradient step algorithms. As seen in experiment 2, the trajectories differed. When the adjoint method was used, the cost reached a lower minimum than when the adjoint-modified method was used. This caused the approximated trajectory of the training data of the NODE trained with the adjoint method to be closer (Lower NMSE) to the true model than the NODE trained with the adjoint-modified (Figure 3.21b).

But an interesting result can be observed in Figures 3.21(c)(d)(e)(f) that show that in three of the first four trajectories of the testing data, the NODE trained with the adjoint-modified method is closer to the true trajectory than the NODE trained with the adjoint

method. This being the case, even though the NODE trained with the adjoint method had a lower cost in the training session than the NODE trained with the adjoint-modified method.

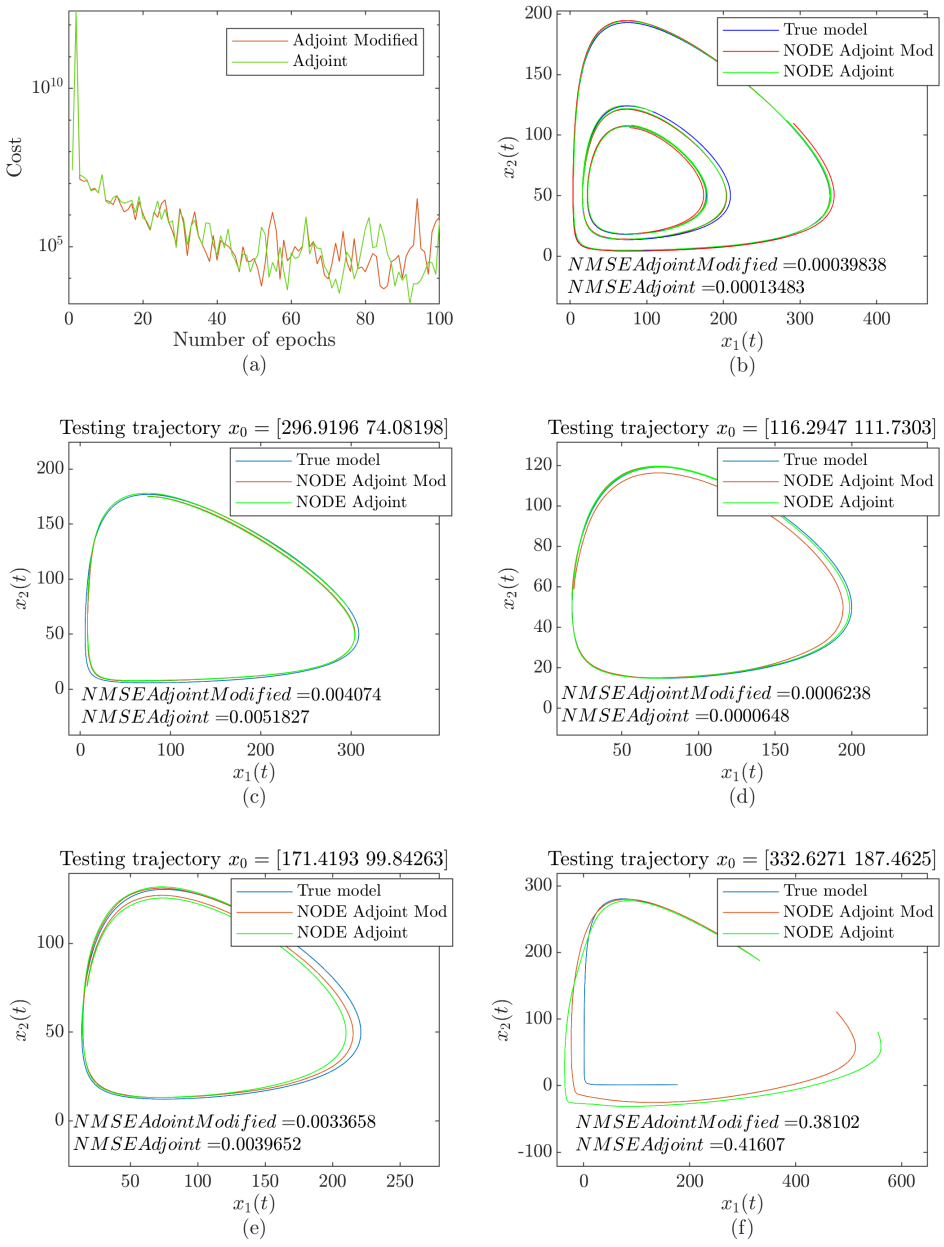


Figure 3.21: Non-linear linear case: (a) Cost trajectory for NODE with optimal hyper-parameters trained with *training-dataset-2*, (b) NODE approximation for the training trajectory, (c) NODE approximation for first trajectory in *testing-dataset-2*, (d) NODE approximation for second trajectory in *testing-dataset-2*, (e) NODE approximation for third trajectory in *testing-dataset-2*, (f) NODE approximation for fourth trajectory in *testing-dataset-2*.

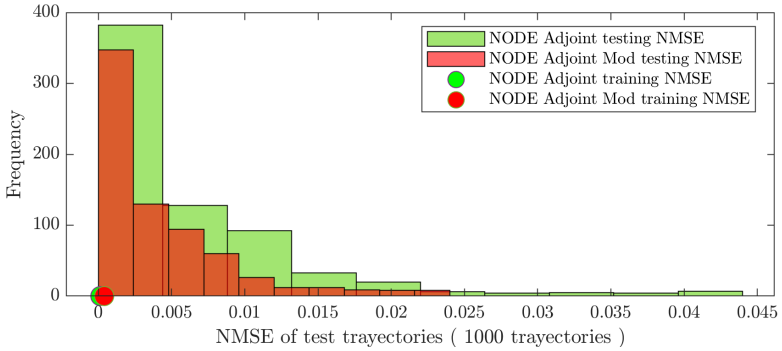


Figure 3.22: Non-linear linear case: NMSE for NODE approximation of the *testing-dataset-2* (1000 trajectories).

A plot with the summary of the test error of the approximations made by the NODEs trained with adjoint and adjoint-modified methods is shown in Figure 3.22; the one thousand trajectories in *testing-dataset-2* were used. The training error in both cases is lower than the testing error for most of the testing trajectories. This means that the training trajectory was approximated better than most of the testing trajectories. But it is important to note that in general the NODE trained with the adjoint-modified produced trajectories closer to the true test trajectory than the NODE trained with the adjoint method. In Figure 3.22 the histogram with the error from the NODE trained with the adjoint-modified has a distribution closer to the zero error than the error distribution from the NODE trained with the adjoint method.

Chapter 4

Discussion and Conclusion

To reach the aim, a series of experiments were conducted and the results were presented in the previous chapter. In the first section of this chapter a discussion of these results in connection with the aims is presented. The first two experiments had the objective of evaluating the effects of the gradient algorithm on the NODE obtained after training. The third experiment assessed the impact of the hyperparameters (i.e. number of layers and activation function) of the NODE underlying ANN. The fourth experiment focused on evaluating the effects of the optimization algorithm on the learning curves of the NODEs. And the last experiment evaluated the ability of the NODEs to model the different cases considered.

After the discussion of the results, a review of the contributions of this study is given. Following, an examination of the limitations of this study and future research opportunities to address the limitations of the study are presented. Finally, the conclusion of this study is presented.

4.1 Discussion

4.1.1 Experiment 1 - Testing gradient step methods on single pairs of points

In this test, the accuracy and speed of the different methods to find a gradient between two points of a training dataset in a NODE was studied. These methods execute the basic step, which is repeated for all the data points in the training dataset. Therefore the overall speed and accuracy of the training depend to a large degree on these methods. For studying the accuracy of the methods, the numerical method was considered as the truth gradient.

As the back-propagation method implemented in the NODE class was based on an Euler ODE solver, the accuracy of it depends on the time step selected: When the step was selected small enough, the back-propagation gave the best approximation of the gradient over the adjoint and adjoint-modified methods. This result is in accordance with the

results reported by Hasani et al. (2020) that showed empirically that the back-propagation provided more accurate gradient calculations than the adjoint sensitivity method. But this high accuracy comes with a cost, because the time per gradient step calculation increases inversely proportional to the time step size, even reaching a time per step above the one of the numerical method.

As expected, the accuracy of the gradient step method proposed in this document (i.e. adjoint-modified) was lower than the adjoint method; in this specific dataset of one thousand data points, the error produced by the adjoint-modified method was approximately double that which was obtained with the original adjoint sensitivity method. But due to the simplicity of the calculations done by the adjoint-modified method, it was ten times faster than the original adjoint method and one hundred times faster than the numerical approximation method.

The findings of this experiment suggest that the trade-off between the accuracy and speed of the adjoint-modified method proposed in this study could be superior to the other methods. But this data must be interpreted with caution, because it was obtained with a small database of one thousand data points generated randomly, and the measurement per gradient was obtained by measuring the running time.

4.1.2 Experiment 2 - Evaluating gradient step methods on complete training datasets

The results of experiment 1 showed that there is a difference in the gradient calculated between two points. So, with this experiment, the objective was to quantify the influence of these differences in the total gradient calculated for a complete training session. The total gradient was monitored following the cost trajectory.

This experiment indicated that the differences of total gradients calculated by different methods increased with increased problem complexity. For the linear case, the cost trajectories overlap completely, suggesting that the gradients generated using different methods were approximately the same. For the almost linear case, the cost trajectories had only slight differences when different methods were used. In contrast, in the nonlinear case the trajectories had considerably different trajectories. This indicates that the gradients generated were significantly different.

For the first two cases, the linear and almost linear, the final NODEs obtained were nearly the same, regardless of the gradient computation algorithm used. For the nonlinear case, although the trajectory followed using different gradient computation methods, was different, the resulting NODEs did not have major differences; when these NODEs were evaluated in the training trajectories they reconstructed almost the same trajectories.

Taken together these results suggest that the ability of the NODEs to learn the data obtained with the three ODE systems selected was not impacted in a significant way by the gradient step algorithm used. In this case, due to the speed of the calculations, the adjoint-modified could offer an advantage over the other methods. But it is important to bear in mind that only three datasets were used to test the method, and such a small sample does not allow for generalisation.

4.1.3 Experiment 3 - Investigating the NODE's underlying ANN hyperparameter's influence in final model performance

In this experiment, the objective was to evaluate the influence of the hyperparameters of the underlying ANN in the performance of the NODE. Three activation functions were evaluated: logistic sigmoid, tangent hyperbolic sigmoid and ReLU. The data collected for the three study cases suggested that the ReLU activation is the activation that performs better in NODEs used to model dynamical systems; this coincides with the results reported by Glorot et al. (2011) in which ReLUs outperformed the logistic sigmoid and the hyperbolic tangent activations in several task with deep ANNs.

Keeping the same number of hidden neurons, the effect of the depth of the network on the accuracy of the final NODE was also evaluated. The evidence from the three study cases suggests that when the training data is extracted from a simple model (e.g. linear ODE system), a shallow network performs better, but when the data is extracted from a more complex model (e.g. non-linear ODE system), deeper networks perform better.

4.1.4 Experiment 4 - Exploring different training algorithms and their effect on the training cost trajectory

The objective of this experiment was to quantify the influence in the training trajectory of the training algorithm used. Two training algorithms were tested, the gradient descent and the stochastic gradient descent, both using the Adam optimization algorithm. The data indicated that batch optimization that uses the whole training data converges at a slower pace, compared with the stochastic optimization algorithm. Besides this, the SGD was able to find a lower minimum than the GD. It was observed that with small *mini-batch* sizes (e.g. 5, 10 ,20) the learning trajectory was unstable, but the best *mini-batch* size was 50 for the cases analysed in this document.

4.1.5 Experiment 5 - Final model evaluation

The purpose of this final experiment was to assess the ability of NODEs to learn the training trajectory, and to extrapolate the behaviour of the dynamic systems selected. For this, the hyperparameters that performed better from experiment 3 and the *mini-batch* size that produced better results found in experiment 4 were used. With this, the NODEs were trained with the training datasets and tested with the testing datasets, including one thousand trajectories each.

The data reported in this experiment for the linear and almost linear case suggest that NODEs are able to learn the underlying behaviour of these ODE systems effectively, and that they are able to extrapolate that behaviour to other initial conditions. In the almost linear case, the NODE examined was able to approximate half of the testing trajectories better than the training trajectories. But these results need to be interpreted with caution, because the initial conditions selected for the testing trajectories were sampled from the

same distribution as the training dataset.

For the nonlinear case the observations suggest that the extrapolation ability of NODEs is inferior compared with the linear and almost linear cases. This is because extrapolations in the non-linear case had in general a higher error than the training trajectory.

Due to the results in experiment 2 for the non-linear case that indicated that the gradient computation method affected the learning trajectory, the NODE for the non-linear case was trained with the adjoint and adjoint-modified methods to evaluate the differences in the models obtained. The models obtained were slightly different, and the NODE trained with the adjoint method was able to better approximate the training trajectory, but the NODE trained with the adjoint-modified method was able to extrapolate the testing trajectories better. As the differences were small, there is no evidence that any of these methods for computing gradients produces a better model. However, with only one case studied, caution must be applied when interpreting this result.

4.2 Contributions

The algorithm for gradient computation referred to as the adjoint-modified method was the most significant contribution of this study. The data suggest that it could offer advantages in terms of speed over other methods analysed in this study, when training NODEs with certain types of datasets.

Another contribution of this study is the step by step explanation of the back propagation process in NODEs. Even though the process is just the application of the chain rule backwards successively, the graphical explanation developed in this study could be useful for understanding the process.

The classes developed in the Matlab suite offer a modular piece of software that could be expanded easily to test other ideas related to Neural ODEs. It was implemented using the *Parallel Computing Toolbox* ™ that can use the full processing power in multicore computers, so that it could be used to test more complex systems in multiprocessor computers.

Finally, the study of the effect of the underlying ANN hyperparameters and the training algorithm on the performance of NODEs can give a notion of a good initial approach when facing a more complex modelling problem using NODEs.

4.3 Limitations

Only data generated from a group of three different ODE systems was used for this study. Moreover, only synthetic data was used, so the results remained specific to that group of ODE systems and limited the generalisability of the results.

Even though the *Parallel Computing Toolbox* ™ was used to take advantage of the full computational power available, only a personal computer was available to run the tests. This limited the quantity of hyperparameters tested (e.g. activation functions, parameter

initialization techniques), and this could have impacted the performance of the NODEs evaluated, as a better hyperparameter was not able to have been evaluated.

4.4 Future work

Connected to the limitations of this study, a future investigation could include more ODE systems to generate synthetic data or even data from real dynamic systems. This to confirm the results in this study, and especially to assess the performance of the adjoint-modified gradient computation method in a more real setting. Also, more hyperparameter instances could be considered, this to ensure that the parameters that maximise the performance of NODEs are selected.

New types of NODEs such as the Liquid Time-Constant Networks LTCs proposed by Hasani et al. (2020) that states a better performance in the modelling of dynamic systems could be included in future investigations. It would be interesting to see the final models obtained with LTCs and standard NODEs and the effect of the gradient algorithms used.

The gradient calculation method proposed by Kidger et al. (2021) (i.e. adjoint via semi-norms), that lies in between of the adjoint-modified method and the adjoint method, would be interesting to assess as it claims to offer improved speed with the same accuracy as the adjoint method.

4.5 Conclusion

Overall, the data suggest that NODEs are suitable structures to model dynamic systems. The results obtained for the three cases analyzed in this study showed the ability of the NODEs to learn the training trajectory effectively. In contrast, the ability of NODEs to extrapolate varied with the complexity of the ODE system used to generate the data; the extrapolations obtained in the linear and almost linear ODE systems were in general much better than the one for the non-linear case.

But this performance was also influenced heavily by the hyperparameters selected for the NODE. For example, the findings of this study suggest that the use of the ReLU activation produces NODEs that performed better than the sigmoid activations in terms of training and testing error. In addition, the data reported in this study appears to support the idea that the number of layers that give better approximations increases with complexity of the derivatives of the problem to model. Moreover, the stochastic gradient descent proved to be key in the training of the NODEs studied, and it was not only faster to get to absolute minimums in the cost function, but it happened that this method also found better minimums than the batch training algorithm.

On the other hand, the gradient algorithm that was used influenced the speed of a training session, and it also affected the final model obtained, depending on the accuracy of the gradient. In this study, three different gradient computation algorithms were

analyzed, the back-propagation method and two methods based on the adjoint. The back-propagation method offered the best gradient approximation, but only if the step size was small enough, which increased the time per calculation over the other methods, making it the slowest. Additionally, it cannot treat the ODE solver as a black box.

Alternatively, the adjoint methods can treat the ODE solver as a black box, this allows the user to test different ODE solvers, without changing the structure of the NODE. The first adjoint method analyzed was the standard adjoint sensitivity method that showed a slightly worse approximation of the gradient than the back-propagation method, this possibly due to the fact that it forgets the forward pass of the state vector and has to reconstruct it in the backward pass introducing numerical error in the process. In terms of speed, the adjoint method did not offer significant advantages over the back-propagation method.

The other adjoint method analyzed was the adjoint-modified method, which is an original contribution of this study. This method uses the forward-pass state vector values to solve the adjoint equations numerically as integrals. This makes this method extremely fast: Some tests performed indicated a time per gradient 10 times smaller than the adjoint method. But as expected, the accuracy is degraded: Some tests done showed that the adjoint-modified method gives an error in the gradient calculation of about double that of the adjoint method. However, when the method was used to model the case studied, no significant differences were found in the final models obtained.

All things considered, the NODEs are structures that are able to model and predict the behaviour of dynamic systems, but the hyperparameters of the underlying feed-forward ANN and the NODE in general have a great influence on the accuracy of the resulting model. As these parameters are not trainable, many training sessions need to be done in order to find a set of hyperparameters that optimise the final NODE, however the running time limits the quantity of test that can be performed. Here, the adjoint-modified proposed in this study could be use to test a broader set of hyperparameters in order to select the optimal ones, this due to its exceptional speed. And, after that the final NODE with optimal hyperparameters can be trained with a more accurate gradient computation method, such as the standard adjoint method.

Bibliography

Chen, R., Rubanova, Y., Bettencourt, J. & Duvenaud, D. (2018), Neural ordinary differential equations, Vol. 2018-December, pp. 6571–6583. ISSN: 1049-5258.

Edwards, C. H., Penney, D. E. & Calvis, D. (2007), *Elementary Differential Equations*, 6th edition edn, Pearson College Div, Upper Saddle River, N.J.

Elman, J. (1990), 'Finding structure in time', *Cognitive Science* **14**(2), 179–211.

Glorot, X. & Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* 'Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics', JMLR Workshop and Conference Proceedings, pp. 249–256. ISSN: 1938-7228.
URL: *https://proceedings.mlr.press/v9/glorot10a.html*

Glorot, X., Bordes, A. & Bengio, Y. (2011), Deep Sparse Rectifier Neural Networks, *in* 'Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics', JMLR Workshop and Conference Proceedings, pp. 315–323. ISSN: 1938-7228.
URL: *https://proceedings.mlr.press/v15/glorot11a.html*

Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J. & Seung, H. S. (2000), 'Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit.', *Nature* **405**(6789), 947–951. Num Pages: 5.

Hasani, R., Lechner, M., Amini, A., Rus, D. & Grosu, R. (2020), 'Liquid Time-constant Networks', *arXiv:2006.04439 [cs, stat]* . arXiv: 2006.04439.
URL: *http://arxiv.org/abs/2006.04439*

He, K., Zhang, X., Ren, S. & Sun, J. (2016), Deep residual learning for image recognition, Vol. 2016-December, pp. 770–778. ISSN: 1063-6919.

Hochreiter, S. & Schmidhuber, J. (1997), 'Long Short-Term Memory', *Neural Computation* **9**(8), 1735–1780.

Karlsson, D. & Svanström, O. (2019), 'Modelling Dynamical Systems Using Neural Ordinary Differential Equations'. Accepted: 2019-07-05T11:53:06Z.
URL: *https://odr.chalmers.se/handle/20.500.12380/256887*

Kidger, P., Chen, R. T. Q. & Lyons, T. (2021), "Hey, that's not an ODE": Faster ODE Adjoints via Seminorms, Technical Report arXiv:2009.09457, arXiv. arXiv:2009.09457

[cs, math] type: article.
URL: *http://arxiv.org/abs/2009.09457*

Kingma, D. P. & Ba, J. (2017), Adam: A Method for Stochastic Optimization, Technical Report arXiv:1412.6980, arXiv. arXiv:1412.6980 [cs] type: article.
URL: *http://arxiv.org/abs/1412.6980*

Robbins, H. & Monro, S. (1951), 'A Stochastic Approximation Method', *The Annals of Mathematical Statistics* **22**(3), 400–407. Publisher: Institute of Mathematical Statistics.
URL: *http://www.jstor.org/stable/2236626*

Rumelhart, D., Hinton, G. & Williams, R. (1986), 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.

Shampine, L. F. & Reichelt, M. W. (1997), 'The MATLAB ODE Suite', *SIAM Journal on Scientific Computing* **18**(1), 1–22.
URL: *http://epubs.siam.org/doi/10.1137/S1064827594276424*

Zhai, J., Shen, W., Singh, I., Wanyama, T. & Gao, Z. (2020), A review of the evolution of deep learning architectures and comparison of their performances for histopathologic cancer detection, Vol. 46, pp. 683–689. ISSN: 2351-9789.
Thesis.bib

Appendix A

NODE Class

```
classdef NODE < matlab.mixin.Copyable
    %Implements a NODE

    properties
        ANN; options;
    end

    methods
        function obj = NODE(size,activ,varargin)
            %NODE Construct an instance of this class
            %   Creates a NODE with an underlying ANN
            obj.ANN = ANN(size,activ);
            if isempty(varargin)
                obj.options = odeset('RelTol',1e-5,'AbsTol',1e-7);
            else
                obj.options = varargin{1};
            end
        end

        function [steps,u] = Forward_Euler(obj,u0,h,tspan)
            %Forward takes and input for the ANN and generate and output
            %using the current ANN
            t0=tspan(1); tN=tspan(2);
            u(1,:) = u0;
            steps = floor((tN-t0)/h); %Number of time steps to perform
            if h*steps < tN-t0
                steps = steps + 1;
            end
            %Go throug the all the time steps and store interm. results
            for i = 1:steps-1
                u(i+1,:) = u(i,:)+h.*obj.ANN.Forward(u(i,:));
            end
            u(steps+1,:)= u(steps,:)+(tN-h*(steps-1)-t0)...
                .*obj.ANN.Forward(u(steps,:));
        end

        function [t,u] = Forward(obj,u0,tspan)
            %Takes and input for the ANN and generate and output
            %using the current ANN as a source of gradient
            [t,u]=ode45(@(t,u) obj.ANN.Forward(u')',tspan,u0,obj.options);
```

```
        end

        function [C,dCdw,dCdb] = Gradient_step_verif(obj,u0,y,h,t0,tN)
            %Calculates the gradient and cost for a pair of points u0 and y
            %perturbing the parameters
            eps=1e-5;
            w_backup = obj.ANN.w;
            b_backup = obj.ANN.b;
            for l = 2:1:obj.ANN.layers
                for j = 1:obj.ANN.size(l)
                    for i = 1:obj.ANN.size(l-1)
                        obj.ANN.w{l}(i,j) = obj.ANN.w{l}(i,j)-eps;
                        [~,u] = obj.Forward(u0,[t0 tN]);
                        Cm = sum((y-u(end,:)).^2)/2;
                        obj.ANN.w{l}(i,j) = obj.ANN.w{l}(i,j)+2*eps;
                        [~,u] = obj.Forward(u0,[t0 tN]);
                        Cp = sum((y-u(end,:)).^2)/2;
                        dCdw{l}(i,j) = ((Cp-Cm)./(2*eps));
                        obj.ANN.w = w_backup;
                    end
                    obj.ANN.b{l}(1,j) = obj.ANN.b{l}(1,j)-eps;
                    [~,u] = obj.Forward(u0,[t0 tN]);
                    Cm = sum((y-u(end,:)).^2)/2;
                    obj.ANN.b{l}(1,j) = obj.ANN.b{l}(1,j)+2*eps;
                    [~,u] = obj.Forward(u0,[t0 tN]);
                    Cp = sum((y-u(end,:)).^2)/2;
                    dCdb{l}(j) = ((Cp-Cm)./(2*eps));
                    obj.ANN.b = b_backup;
                end
            end
            C=sum((y-u(end,:)).^2)/2; %Returns the total cost
        end

        function [C,dCdw,dCdb] = Gradient_step_backp(obj,u0,y,h,t0,tN)
            [steps,u] = obj.Forward_Euler(u0,h,[t0 tN]);
            for i = steps:-1:1 %Backpropagates the error from the last step
                obj.ANN.Gradient(u(i,:));
                if i==steps %Calculates dC/du for the last step
                    dCdu{steps} = u(steps+1,:)-y;
                else %Calculates dC/du for all the steps but the last
                    dCdu{i} = dCdu{i+1}*(obj.ANN.iograd*h+...
                        eye(size(obj.ANN.iograd)));
                end
                for l = obj.ANN.layers:-1:2 %Calculates dC/dw and dC/db
                    dCdw{l,i} = 0;
                    dCdb{l,i} = 0;
                    for j = 1:obj.ANN.size(end)
                        dCdw{l,i} = h*dCdu{i}(j).*obj.ANN.wgrad{l,j}...
                            +dCdw{l,i};
                        dCdb{l,i} = h*dCdu{i}(j).*obj.ANN.bgrad{l,j}...
                            +dCdb{l,i};
                    end
                    if i~=steps
                        dCdw{l,i} = dCdw{l,i}+dCdw{l,i+1};
                        dCdb{l,i} = dCdb{l,i}+dCdb{l,i+1};
                    end
```

```
                end
            end
            %Return just the gradient dC/dw dC/db with backpropagation all
            % the way back
            dCdw=dCdw(:,1);
            dCdb=dCdb(:,1);
            C=sum((y-u(steps+1,:)).^2)/2; %Returns the total cost
        end

        function [C,dCdw,dCdb] = Gradient_step_adj(obj,u0,y,h,t0,tN)
            %Calculates the gradient and cost for a pair of points u0 and y
            %Using the adjoint sensitivity method
            [~,u] = obj.Forward(u0,[t0 tN]); %Calculates U(tN)
            i = obj.ANN.size(1); %Number of components of the U vecor
            ivector(1:i) = u(end,:);%Insert U(tN) in the init. vec.
            ivector(i+1:i*2) = u(end,:)-y; %Insert dC/dU(tN) in init. vec.
            parameters=0;
            for l = 2:1:obj.ANN.layers
                parameters=obj.ANN.size(l-1)*obj.ANN.size(l)+...
                    obj.ANN.size(l)+parameters;
            end
            ivector(i*2+1:i*2+parameters)=0;
            ivector = ivector'; %ODE solver requires columns vectors
            [~,vector] = ode45(@(t,vector) obj.odefun(t,vector),...
                [tN t0],ivector,obj.options); %Finds the dC/dtheta
            %Only interested in the values in the end time t0
            vector = vector(end,:);
            %Converts the vector resulting in the dCdw and dCdb matrices
            i2 = i*2+1;
            [dCdw,dCdb] = obj.Vector_to_gradient(vector(i2:end));
            C=sum((y-u(end,:)).^2)/2; %Returns the total cost
        end

        function [C,dCdw,dCdb] = Gradient_step_adj_mod(obj,u0,y,h,t0,tN)
            %Calculates the gradient and cost for a pair of points u0 and y
            %Using the adjoint sensitivity method modified
            [times,u] = obj.Forward(u0,[t0 tN]); %Calculates U(tN)
            steps = length(times);
            a(steps,:) = u(end,:)-y;
            obj.ANN.Gradient(u(steps,:));
            ANN_= obj.ANN;
            for i=2:obj.ANN.layers
                dCdb_{i,steps} = zeros(1,obj.ANN.size(i));
                dCdw_{i,steps} = zeros(obj.ANN.size(i-1),obj.ANN.size(i));
                temp_b{i} = zeros(1,obj.ANN.size(i));
                temp_w{i} = zeros(obj.ANN.size(i-1),obj.ANN.size(i));
            end
            for i = steps-1:-1:1
                obj.ANN.Gradient(u(i,:));
                a(i,:) = (a(i+1,:)+(times(i+1)-times(i))/2.*(a(i+1,:)...
                    *ANN_.iograd))*inv(eye(obj.ANN.size(1))-...
                    (times(i+1)-times(i))/2.*obj.ANN.iograd);
                for l=obj.ANN.layers:-1:2 % For each layer
                    for j=1:obj.ANN.size(end)
                        temp_w{l} = a(i+1,j).*ANN_.wgrad{l,j}+a(i,j)...
                            .*obj.ANN.wgrad{l,j}+temp_w{l};
```

```
                        temp_b{l} = a(i+1,j).*ANN_.bgrad{l,j}+a(i,j)...
                            .*obj.ANN.bgrad{l,j}+temp_b{l};
                    end
                    temp_w{l} = temp_w{l}.*(times(i+1)-times(i))/2;
                    temp_b{l} = temp_b{l}.*(times(i+1)-times(i))/2;
                    dCdw_{l,i} = dCdw_{l,i+1}+temp_w{l};
                    dCdb_{l,i} = dCdb_{l,i+1}+temp_b{l};
                    temp_b{l} = zeros(1,obj.ANN.size(l));
                    temp_w{l} = zeros(obj.ANN.size(l-1),obj.ANN.size(l));
                end
                ANN_= obj.ANN;
            end
            for l=obj.ANN.layers:-1:2 % For each layer
                dCdw{l} = dCdw_{l,1};
                dCdb{l} = dCdb_{l,1};
            end
            C=sum((y-u(end,:)).^2)/2;
        end

        function [C_total,dCdw_total,dCdb_total] = ...
                Gradient(obj,targets,times,h,grad_alg)
            data_points = length (targets{1}(:,1));
            for i=1:obj.ANN.layers-1 %Initializes total gradients to zero
                dCdw_total{i+1}=zeros(obj.ANN.size(i),obj.ANN.size(i+1));
                dCdb_total{i+1}=zeros(1,obj.ANN.size(i+1));
            end
            C_total = 0; %Initializes total cost to zero
            C = zeros(1,data_points);
            %Go through pairs of consecutive data points evaluating
            % gradients and cost
            parfor i=1:data_points
                if grad_alg == 1
                    [C(i),dCdw_{i},dCdb_{i}] = obj.Gradient_step_verif...
                        (targets{1}(i,:),targets{2}(i,:),h,times{1}(i),...
                        times{2}(i));
                elseif grad_alg == 2
                    [C(i),dCdw_{i},dCdb_{i}] = obj.Gradient_step_backp...
                        (targets{1}(i,:),targets{2}(i,:),h,times{1}(i),...
                        times{2}(i));
                elseif grad_alg == 3
                    [C(i),dCdw_{i},dCdb_{i}] = obj.Gradient_step_adj...
                        (targets{1}(i,:),targets{2}(i,:),h,times{1}(i),...
                        times{2}(i));
                elseif grad_alg == 4
                    [C(i),dCdw_{i},dCdb_{i}] = obj.Gradient_step_adj_mod...
                        (targets{1}(i,:),targets{2}(i,:),h,times{1}(i),...
                        times{2}(i));
                end
            end
            for i=1:data_points
                dCdw = dCdw_{i};
                dCdb = dCdb_{i};
                for j=2:obj.ANN.layers
                    dCdw_total{j} = dCdw_total{j}+dCdw{j}./(data_points-1);
                    dCdb_total{j} = dCdb_total{j}+dCdb{j}./(data_points-1);
                end
```

```
            end
            C_total = sum(C);
        end

        function [cost] = train_GD(obj,targets,times,h,max_epochs,...
                max_cost,grad_alg,varargin)
            %Gradient decent algorith that uses input-output data
            %to update the weights and bias
            data_points = length (targets(:,1));
            %If data containes more than one trajectory segmented=true
            if nargin > 7
                segmented = true;
                segments = length(varargin{1})+1;
                borders = varargin{1};
            else
                segmented = false;
            end
            %Initializes intermediate variables for Adam
            for i=2:obj.ANN.layers
                mt{i,1}=obj.ANN.w{i}*0;
                mt{i,2}=obj.ANN.b{i}*0;
                vt{i,1}=obj.ANN.w{i}*0;
                vt{i,2}=obj.ANN.b{i}*0;
            end
            t=0;beta1=0.9;beta2=0.999; alpha=0.005;eps=1e-8; %Adam para
            epochs = 1;
            cost=zeros(1,max_epochs);
            [~,dCdw,dCdb] = obj.Gradient(targets,times,h,grad_alg);
            [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,vt,dCdw,dCdb);
            ANN_best = copy(obj.ANN);
            figure(20);
            set(gca,'TickLabelInterpreter','latex');
            costs = animatedline('Color','b');
            set(gca, 'YScale', 'log')
            ylabel('Cost','Interpreter','latex')
            xlabel('Number of epochs','Interpreter','latex')
            figure(21);
            hold on
            if segmented
                for i = 1:length(borders)-1
                    [~,u] = obj.Forward(targets{2}(borders(i),:),...
                        times{2}(borders(i):borders(i+1)-1));
                    plot(targets{2}(borders(i):borders(i+1)-1,1),...
                        targets{2}(borders(i):borders(i+1)-1,2),'b')
                    uplot(i) = plot(u(:,1),u(:,2),'r');
                    cost(epochs) = sum(sum((targets{2}(borders(i):...
                        borders(i+1)-1,:)-u).^2,2)/2) + cost(epochs);
                end
            else
                [~,u] = obj.Forward(targets{1}(1,:),times{2});
                plot(targets{2}(:,1),targets{2}(:,2))
                uplot = plot(u(:,1),u(:,2));
                cost(epochs) = sum(sum((targets{2}-u).^2,2)/2);
            end
            set(gca,'TickLabelInterpreter','latex');
            ylabel('$x_2(t)$','Interpreter','latex')
```

```
            xlabel('$x_1(t)$','Interpreter','latex')
            legend('True model','NODE aproximation','Interpreter','latex')
            fprintf('Gradient Decent. Number of epochs %d Cost %f\n',...
                epochs,cost(epochs));
            addpoints(costs,epochs,cost(epochs));
            refreshdata;
            drawnow
            while cost(epochs)>max_cost && epochs<max_epochs
                epochs = epochs +1;
                [~,dCdw,dCdb] = obj.Gradient(targets,times,h,grad_alg);
                [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,vt,...
                    dCdw,dCdb);
                if segmented
                    for i = 1:length(borders)-1
                        [~,u] = obj.Forward(targets{2}(borders(i),:),...
                            times{2}(borders(i):borders(i+1)-1));
                        uplot(i).XData = u(:,1); uplot(i).YData = u(:,2);
                        cost(epochs) = sum(sum((targets{2}(borders(i):...
                            borders(i+1)-1,:)-u).^2,2)/2) + cost(epochs);
                    end
                else
                    [~,u] = obj.Forward(targets{1}(1,:),times{2});
                    uplot.XData = u(:,1); uplot.YData = u(:,2);
                    cost(epochs) = sum(sum((targets{2}-u).^2,2)/2);
                end
                if min(cost(1:epochs-1))>cost(epochs)
                    ANN_best = copy(obj.ANN);
                end
                fprintf('Gradient Decent. Number of epochs %d Cost %f\n'...
                    ,epochs,cost(epochs));
                addpoints(costs,epochs,cost(epochs));
                refreshdata;
                drawnow
            end
            obj.ANN = copy(ANN_best);
            hold off; figure(20); hold off;
        end

        function [cost] = train_SGD(obj,targets,times,h,max_epochs,...
                max_cost,grad_alg,batch_size,varargin)
            %Stochastic Gradient decent algorith that uses input-output
            % data to update the weights and bias
            data_points = length (targets{1}(:,1)); %Number of data points
            if nargin > 8 %If data containes more than one trajectory
                segmented = true;
                segments = length(varargin{1})+1;
                borders = varargin{1};
            else
                segmented = false;
            end
            if nargin==10 %If random index vector was input
                index = varargin{2};
            else %If index was not input it generates one
                [~,index] =  sort(rand(max_epochs,length(targets{1})),2);
            end
             %Initializes intermediate variables for Adam
```

```
            for i=2:obj.ANN.layers
                mt{i,1}=obj.ANN.w{i}*0;
                mt{i,2}=obj.ANN.b{i}*0;
                vt{i,1}=obj.ANN.w{i}*0;
                vt{i,2}=obj.ANN.b{i}*0;
            end
            t=0;beta1=0.9;beta2=0.999; alpha=0.005;eps=1e-8; %Adam para
            epochs = 1;
            cost=zeros(1,max_epochs);
            iter = floor(data_points/batch_size);
            for k = 1:iter-1
                    selected_targets{1} = targets{1}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size),:);
                    selected_targets{2} = targets{2}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size),:);
                    selected_times{1} = times{1}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size));
                    selected_times{2} = times{2}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size));
                    [~,dCdw,dCdb] = obj.Gradient(selected_targets,...
                        selected_times,h,grad_alg);
                    [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,...
                        vt,dCdw,dCdb);
            end
            selected_targets{1} = targets{1}(index(epochs,(iter-1)...
                *batch_size+1:end),:);
            selected_targets{2} = targets{2}(index(epochs,(iter-1)...
                *batch_size+1:end),:);
            selected_times{1} = times{1}(index(epochs,(iter-1)...
                *batch_size+1:end));
            selected_times{2} = times{2}(index(epochs,(iter-1)...
                *batch_size+1:end));
            [~,dCdw,dCdb] = obj.Gradient(selected_targets,...
                selected_times,h,grad_alg);
            [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,vt,dCdw,dCdb);
            ANN_best = copy(obj.ANN);
            figure(20);
            set(gca,'TickLabelInterpreter','latex');
            costs = animatedline('Color','b');
            set(gca, 'YScale', 'log')
            ylabel('Cost','Interpreter','latex')
            xlabel('Number of epochs','Interpreter','latex')
            figure(21);
            hold on
            if segmented
                for i = 1:length(borders)-1
                    [~,u] = obj.Forward(targets{2}(borders(i),:)...
                        ,times{2}(borders(i):borders(i+1)-1));
                    plot(targets{2}(borders(i):borders(i+1)-1,1)...
                        ,targets{2}(borders(i):borders(i+1)-1,2),'b')
                    uplot(i) = plot(u(:,1),u(:,2),'r');
                    cost(epochs) = sum(sum((targets{2}(borders(i)...
                        :borders(i+1)-1,:)-u).^2,2)/2) + cost(epochs);
                end
            else
                [~,u] = obj.Forward(targets{1}(1,:),times{2});
```

```
                plot(targets{2}(:,1),targets{2}(:,2))
                uplot = plot(u(:,1),u(:,2));
                cost(epochs) = sum(sum((targets{2}-u).^2,2)/2);
            end
            set(gca,'TickLabelInterpreter','latex');
            ylabel('$x_2(t)$','Interpreter','latex')
            xlabel('$x_1(t)$','Interpreter','latex')
            legend('True model','NODE aproximation','Interpreter','latex')
            fprintf(['Stochastic Gradient Decent. Number of epochs' ...
                ' %d Cost %f\n'],epochs,cost(epochs));
            addpoints(costs,epochs,cost(epochs));
            refreshdata;
            drawnow
            while cost(epochs)>max_cost && epochs<max_epochs
                epochs = epochs +1;
                iter = floor(data_points/batch_size);
                for k = 1:iter-1
                    selected_targets{1} = targets{1}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size),:);
                    selected_targets{2} = targets{2}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size),:);
                    selected_times{1} = times{1}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size));
                    selected_times{2} = times{2}(index(epochs,(k-1)...
                        *batch_size+1:k*batch_size));
                    [~,dCdw,dCdb] = obj.Gradient(selected_targets,...
                        selected_times,h,grad_alg);
                    [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,...
                        vt,dCdw,dCdb);
                end
                selected_targets{1} = targets{1}(index(epochs,(iter-1)...
                    *batch_size+1:end),:);
                selected_targets{2} = targets{2}(index(epochs,(iter-1)...
                    *batch_size+1:end),:);
                selected_times{1} = times{1}(index(epochs,(iter-1)...
                    *batch_size+1:end));
                selected_times{2} = times{2}(index(epochs,(iter-1)...
                    *batch_size+1:end));
                [~,dCdw,dCdb] = obj.Gradient(selected_targets,...
                    selected_times,h,grad_alg);
                [mt,vt,t] = obj.Adam(beta1,beta2,alpha,eps,t,mt,vt,...
                    dCdw,dCdb);

                if segmented
                for i = 1:length(borders)-1
                    [~,u] = obj.Forward(targets{2}(borders(i),:),...
                        times{2}(borders(i):borders(i+1)-1));
                    uplot(i).XData = u(:,1); uplot(i).YData = u(:,2);
                    cost(epochs) = sum(sum((targets{2}(borders(i)...
                        :borders(i+1)-1,:)-u).^2,2)/2) + cost(epochs);
                end
                else
                    [~,u] = obj.Forward(targets{1}(1,:),times{2});
                    uplot.XData = u(:,1); uplot.YData = u(:,2);
                    cost(epochs) = sum(sum((targets{2}-u).^2,2)/2);
                end
```

```
433                 if min(cost(1:epochs-1))>cost(epochs)
434                     ANN_best = copy(obj.ANN);
435                 end
436                 fprintf(['Stochastic Gradient Decent. Number of epochs' ...
437                     ' %d Cost %f\n'],epochs,cost(epochs));
438                 addpoints(costs,epochs,cost(epochs));
439                 refreshdata;
440                 drawnow
441             end
442             obj.ANN = copy(ANN_best);
443             figure(21); hold off; figure(20); hold off;
444         end
445
446
447         function [mt,vt,t] = Adam(obj,beta1,beta2,alpha,eps,t,mt,vt, ...
448                 dCdw,dCdb)
449             %Adam optimization algorithm
450             t=t+1;
451             alphat=alpha*((1-beta2^t)^0.5)/(1-beta1^t);
452             for i=2:obj.ANN.layers
453                 mt{i,1}=beta1.*mt{i,1}+(1-beta1).*dCdw{i};
454                 mt{i,2}=beta1.*mt{i,2}+(1-beta1).*dCdb{i};
455                 vt{i,1}=beta2.*vt{i,1}+(1-beta2).*dCdw{i}.^2;
456                 vt{i,2}=beta2.*vt{i,2}+(1-beta2).*dCdb{i}.^2;
457                 obj.ANN.w{i} = obj.ANN.w{i}-alphat*(mt{i,1}./(vt{i,1} ...
458                     .^0.5+eps));
459                 obj.ANN.b{i} = obj.ANN.b{i}-alphat*(mt{i,2}./(vt{i,2} ...
460                     .^0.5+eps));
461             end
462         end
463
464         function dydt = odefun(obj,t,y)
465             %Function that build a vector containing the derivatives
466             %dU/dt, da/dt, dm/dt.  This vector is use to solve for the
467             %gradient dC/dtheta using an ODE solver
468             i = obj.ANN.size(1);
469             obj.ANN.Gradient(y(1:i)');
470             dydt(1:i) = obj.ANN.a{end};
471             dydt(i+1:i*2) = -y(i+1:i*2)'*obj.ANN.iograd;
472             for l = 2:1:obj.ANN.layers
473                 dmdt_w{l} = zeros(obj.ANN.size(l-1),obj.ANN.size(l));
474                 dmdt_b{l} = zeros(1,obj.ANN.size(l));
475                 for j = 1:i
476                     dmdt_w{l} = -y(i+j).*obj.ANN.wgrad{l,j}+dmdt_w{l};
477                     dmdt_b{l} = -y(i+j).*obj.ANN.bgrad{l,j}+dmdt_b{l};
478                 end
479             end
480             vector = obj.Gradient_to_vector(dmdt_w,dmdt_b);
481             dydt= [dydt,vector];
482             dydt = dydt';
483         end
484
485         function dist = Compare_gradients(obj,dCdw,dCdb,dCdw_,dCdb_)
486             %Returns the euclidean distance normalized by the sum of the
487             % norm of the vectors
488             gvector = obj.Gradient_to_vector(dCdw,dCdb);
```

```
            gvector_ = obj.Gradient_to_vector(dCdw_,dCdb_);
            dist = norm(gvector-gvector_)/(norm(gvector)+norm(gvector_));
        end

        function gvector = Gradient_to_vector(obj,dCdw,dCdb)
            %Convert dCdw and dCdb from structures to concatenated
            % 1D vector
            gvector = [];
            for l = 2:1:obj.ANN.layers
                for j = 1:obj.ANN.size(l)
                    for i = 1:obj.ANN.size(l-1)
                        gvector = [gvector,dCdw{l}(i,j)];
                    end
                    gvector = [gvector,dCdb{l}(j)];
                end
            end
        end

        function [dCdw,dCdb] = Vector_to_gradient(obj,gvector)
           %Convert a 1D vector contaning gradients to structures
           index = 1;
            for l = 2:1:obj.ANN.layers
                for j = 1:obj.ANN.size(l)
                    for i = 1:obj.ANN.size(l-1)
                        dCdw{l}(i,j) = gvector(index);
                        index = index+1;
                    end
                    dCdb{l}(j) = gvector(index);
                    index = index+1;
                end
            end
        end

    end
end
```

Appendix B

ANN Class

```
classdef ANN < matlab.mixin.Copyable
    %ANN Feedforward Artifitial Neural Network Class

    properties
        size; activ; w; z; a; b; delta; layers; wgrad; bgrad; iograd;
    end

    methods
        function obj = ANN(size,activ)
            %Creates a feedforward ANN
            %Creates weights, bias and gradients matrices and initializes
            %them
            obj.size = size;
            obj.activ = activ;
            obj.layers = length(size);
            obj.a{1} = zeros(1,obj.size(1));
            for i=2:obj.layers
                obj.w{i} = (2*rand([obj.size(i-1) obj.size(i)],'double')...
                    -1)*sqrt(6 / (obj.size(i-1)+obj.size(i)));% "Glorot"
                obj.b{i} = zeros(1,obj.size(i));
                obj.z{i} = zeros(1,obj.size(i));
                obj.a{i} = zeros(1,obj.size(i));
                for j = 1:obj.size(end)
                    obj.bgrad{i,j} = zeros(1,obj.size(i));
                    obj.wgrad{i,j} = zeros(obj.size(i-1),obj.size(i));
                end
            end
        end

        function f = Forward(obj,input)
            %Takes and input for the ANN and generate and output using the
            %current weights and bias
            obj.a{1} = input;
            for i=2:obj.layers
                obj.z{i} = obj.a{i-1}*obj.w{i}+obj.b{i};
                obj.a{i} = obj.activation(obj.z{i},obj.activ(i));
            end
            f = obj.a{end};
        end

```

```
41          function Gradient(obj,input)
42              %Finds the gradient of the output with respect to the
43              %parameters (weights and bias) and the input
44              obj.Forward(input);
45              for j=1:obj.size(end)
46                  obj.delta{obj.layers,j} = zeros(1,obj.size(end));
47                  obj.delta{obj.layers,j}(1,j) = obj.activation_der(obj.z...
48                      {obj.layers}(1,j),obj.activ(obj.layers));
49                  obj.wgrad{obj.layers,j} = (obj.delta{obj.layers,j}'*obj.a
50                      {obj.layers-1})';
51                  obj.bgrad{obj.layers,j} = obj.delta{obj.layers,j};
52                  for l=obj.layers-1:-1:2
53                      obj.delta{l,j} = (obj.delta{l+1,j}*obj.w{l+1}')...
54                          .*obj.activation_der(obj.z{l},obj.activ(l));
55                      obj.wgrad{l,j} = (obj.delta{l,j}'*obj.a{l-1})';
56                      obj.bgrad{l,j} = obj.delta{l,j};
57                  end
58                  obj.iograd(j,:)=obj.delta{2,j}*obj.w{2}';
59              end
60          end
61
62          function reset_WB(obj)
63              %Clear weights and bias
64              for i=2:obj.layers
65                  obj.w{i} = (2*rand([obj.size(i-1) obj.size(i)],'double')...
66                      -1)*sqrt(6 / (obj.size(i-1)+obj.size(i)));% "Glorot".
67                  obj.b{i} = zeros(1,obj.size(i));
68                  obj.z{i} = zeros(1,obj.size(i));
69                  obj.a{i} = zeros(1,obj.size(i));
70                  for j = 1:obj.size(end) %Creates gradient matrices for w
71                      % and b for each output
72                      obj.bgrad{i,j} = zeros(1,obj.size(i));
73                      obj.wgrad{i,j} = zeros(obj.size(i-1),obj.size(i));
74                  end
75              end
76          end
77
78          function output = activation(obj,input,activ)
79              %Returns the activation function evaluated at input
80              switch activ
81                  case 1 %Identity
82                      output = input;
83                  case 2 %Sigmoid
84                      output = logsig(input);
85                  case 3 %Hiperbolic tangent sigmoid
86                      output = tanh(input);
87                  case 4 %ReLU
88                      output = zeros(1,length(input));
89                      for i=1:length(input)
90                          if input(i)>0
91                              output(i)=input(i);
92                          end
93                      end
94              end
95          end
96
```

```
97          function output = activation_der(obj,input,activ)
98              %Returns the activation derivative evaluated at input
99              switch activ
100                 case 1 %Identity
101                     output = ones(1,length(input));
102                 case 2 %Sigmoid
103                     output = dlogsig(input,logsig(input));
104                 case 3 %Hiperbolic tangent sigmoid
105                     output = dtansig(input,tansig(input));
106                 case 4 %ReLU
107                     output = zeros(1,length(input));
108                     for i=1:length(input)
109                         if input(i)>0
110                             output(i)=1;
111                         end
112                     end
113             end
114         end
115
116     end
117 end
```