Function Extrapolation through Differential Equation Learning

A Thesis

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Christina Seventikidou

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN DATA SCIENCE AND ENGINEERING

University of Ioannina
School of Engineering
Ioannina 2022

Examining Committee:

- **Aristidis Likas**, Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Konstantinos D. Blekas**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Konstantinos Vlachos**, Assist. Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

To my parents, for always loving and supporting me.

Acknowledgements

First, I would like to thank my thesis supervisor, Professor Aristidis Likas, for the opportunity he gave me and for all the support, patience and guidance he provided. His generous sharing of knowledge has been a major inspiration for me while working on this dissertation and also during his graduate courses. I would also like to thank Professor Isaac Lagaris for providing ideas on which this thesis was based.

Furthermore, I would like to express my gratitude to all the members of the Dept. of Computer Science and Engineering with whom I worked, because without the knowledge I received from them, the completion of this thesis would not have been possible.

In addition, I would like to thank the members of the Dept. of Mathematics of the University of Ioannina, for shaping me as a mathematician.

Moreover, I owe special thanks to Giorgos Mitaros, graduate of the Dept. of Computer Science and Engineering, partner and best friend, for his unconditional support and the endless hours of scientific discussions and tutoring in software engineering during our undergraduate studies.

I wish also to thank my fellow students, Christina, Maria, Mary and Rafaela for the support, the discussions and the cooperation during our undergraduate studies in mathematics.

Above all else, I am grateful to my family for their support, love and motivation in my every step.

Table of Contents

List of Figures iii

List of Tables v

List of Algorithms vi

Abstract vii

Εκτεταμένη Περίληψη ix

1 Introduction 1

1.1 Motivation 1

1.2 Approach and contribution 2

1.3 Thesis outline 3

2 Preliminary concepts and methods 4

2.1 Machine Learning 4

2.1.1 Supervised Learning 5

2.1.2 Artificial Neural Networks 6

2.1.3 Feedforward neural networks 8

2.2 Numerical methods for solving ordinary differential equations 13

2.2.1 Ordinary differential equations 13

2.2.2 Initial value problem 14

2.2.3 The Existence and Uniqueness of Solutions 14

2.2.4 Basic concepts of Numerical Methods for Initial-Value Problems 16

2.2.5 Runge Kutta methods 17

2.2.6 Solving ODEs with neural networks 19

2.3 Extrapolation 21

2.3.1 Definition, basic concepts 21

3 The proposed method **23**

3.1 Structure of the method 23

3.1.1 First step - initial neural network 23

3.1.2 Second step – numerical ODE definition 24

3.1.3 Third step – solution of numerical ODE 25

3.2 Summary of the method 25

3.3 Gradient of a neural network 26

3.4 Variations of the method 27

4 Experimental results **29**

4.1 Software requirements 30

4.2 Datasets 30

4.3 Hyperparameters and architecture details 31

4.4 Illustration of the method 32

4.5 Extrapolation results 37

4.6 Data with noise 38

4.7 Results of the alternative ODE models 43

4.8 Experiment with unknown ODE 46

5 Epilogue **50**

5.1 Conclusion 50

5.2 Future work 51

Bibliography **52**

List of Figures

2.1 Architecture of an artificial neural network 8
2.2 Example of a feed forward neural network, here $W(1)$ is $n \times 3$, $W(2)$ is 3×1, $b(1)$ is 3×1, $b(2)$ is 1×1 9
2.3 Example of a neural network that we apply back propagation 11
2.4 Given the data points in white section, the lines represent polynomial models in order to make predictions for intermediate points in white region(interpolation) as well as make predictions for data outside the domain in grey region(extrapolation) 21

3.1 Architecture of $N_i(t)$ 24
3.2 Intervals of the proposed method graphically 26

4.1 Example 1 - ODE 33
4.2 Example 1 - P(t) approximation 33
4.3 Example 2 - ODE 34
4.4 Example 2 - P(t) approximation 34
4.5 Example 3 - ODE 35
4.6 Example 3 - P(t) approximation 35
4.7 Example 4 - ODE 36
4.8 Example 4 - P(t) approximation 36
4.9 Problem 1 - Extrapolation error for various ranges 39
4.10 Problem 2 - Extrapolation error for various ranges 39
4.11 Problem 3 - Extrapolation error for various ranges 40
4.12 Problem 4 - Extrapolation error for various ranges 40
4.13 The real time series where we do not which ODE it satisfies 46
4.14 Case 1 48
4.15 Case 2 48

4.16 Case 3 49

List of Tables

2.1 RK tableau 18

4.1 Hyperparameters of Neural Networks that were trained in the implementation, using models (3.1) or (3.5) 31

4.2 Intervals I, I_1, I_2 32

4.3 Statistics of extrapolation error 41

4.4 Example 1 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$ 42

4.5 Example 2 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$ 42

4.6 Example 3 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$ 43

4.7 Example 4 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$ 43

4.8 Statistics of extrapolation error 45

4.9 Statistics of extrapolation error 45

4.10 Statistics of extrapolation error 46

4.11 Intervals I, I_1, I_2 for the testing time series example 47

4.12 Statistics of the extrapolation error of the testing time series, for $t_i \in T = [x_{max}, x_{max} + 1.5]$, R=1.5, with model (3.1) 49

List of Algorithms

2.1 RK of order 4 algorithm 19
3.1 Algorithm of the proposed method 28

Abstract

Christina Seventikidou, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2022.
Function Extrapolation through Differential Equation Learning.
Advisor: Aristidis Likas, Professor.

With the existence of different phenomena that are evolved in time, often there is the need to find a function that can describe them in order to draw inferences and make predictions. A characteristic example is time series prediction, where we are given the values of a function within an input range and we aim to predict future values outside the range (extrapolation). In continuous problems, the time-evolution can in many cases be described by differential equations. In this thesis we will focus only on problems that can be described by ordinary differential equations and in particular by first order differential equations.

The objective of this thesis is the design and implementation of an approach inspired by neural networks and differential equations, in order to study the extrapolation ability of a function based on a given data set. The typical approach would be to train a machine learning model (here a neural network) based on the available function values and then use this model for the extrapolation task. In the proposed method, we assume that this model is the solution to an unknown differential equation. If we manage to find out the differential equation that describes the model in time evolution, then we can use it to obtain good extrapolation results.

Given a set of examples (t_i, y_i), where t_i belongs to a training domain I, we first train a neural network $N_i(t)$. Since we have $N_i(t)$ we can compute the derivative $\frac{dN_i}{dt}$ and define a differential equation model in the form $\frac{dN_i}{dt} = g(N(t;w))$ where $g(.)$ involves one or more neural networks. The parameters of the differential equation model (w) are specified through training, so that the differential equation is satisfied

at various points of the training interval. Once we obtain the differential equation model, we can solve it with a numerical method and use this solution to find the function values outside of the interval I. The proposed method has been tested on several problems and we will present the experimental results as well as the empirical conclusions regarding the efficiency of extrapolation.
Keywords: machine learning, deep learning, neural networks, differential equations, extrapolation

Εκτεταμενη Περιληψη

Χριστίνα Σεβεντικίδου, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2022.
Πρόβλεψη μελλοντικών τιμών συνάρτησης με μάθηση της διαφορικής εξίσωσης που μοντελοποιεί την εξέλιξη της.
Επιβλέπων: Αριστείδης Λύκας, Καθηγητής.

Με την ύπαρξη διαφορετικών φαινομένων που εξελίσσονται στο χρόνο, συχνά υπάρχει η ανάγκη να βρεθεί μια συνάρτηση που να μπορεί να τα περιγράψει προκειμένου να εξαχθούν συμπεράσματα και να γίνουν προβλέψεις. Ένα χαρακτηριστικό παράδειγμα είναι η πρόβλεψη χρονοσειρών, όπου μας δίνονται οι τιμές μιας συνάρτησης μέσα σε ένα διάστημα τιμών και στοχεύουμε να προβλέψουμε μελλοντικές τιμές εκτός του διαστήματος (παρέκταση, extrapolation). Για συνεχή προβλήματα, η χρονική εξέλιξη μπορεί σε πολλές περιπτώσεις να περιγραφεί με διαφορικές εξισώσεις. Σε αυτή τη διατριβή θα εστιάσουμε μόνο σε προβλήματα που μπορούν να περιγραφούν με συνήθεις διαφορικές εξισώσεις και ιδιαίτερα με διαφορικές εξισώσεις πρώτης τάξης.

Στόχος της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η εφαρμογή μιας προσέγγισης εμπνευσμένης από νευρωνικά δίκτυα και διαφορικές εξισώσεις, προκειμένου να μελετηθεί η δυνατότητα παρέκτασης μιας συνάρτησης, με βάση ένα δοθέν σύνολο δεδομένων. Η τυπική προσέγγιση θα ήταν να εκπαιδεύσουμε ένα μοντέλο μηχανικής μάθησης (εδώ νευρωνικό δίκτυο) χρησιμοποιώντας το διαθέσιμο σύνολο δεδομένων και στη συνέχεια να χρησιμοποιήσουμε αυτό το μοντέλο για την παρέκταση. Στην προτεινόμενη μέθοδο υποθέτουμε ότι αυτό το μοντέλο είναι η λύση μιας άγνωστης διαφορικής εξίσωσης. Αν καταφέρουμε να βρούμε τη διαφορική εξίσωση που περιγράφει το μοντέλο κατά την εξέλιξη του χρόνου, τότε μπορούμε

να τη λύσουμε με μια αριθμητική μέθοδο. Μπορούμε να εκμεταλλευτούμε αυτή τη λύση για να επιτύχουμε καλά αποτελέσματα παρέκτασης.

Λαμβάνοντας υπόψη ένα σύνολο παραδειγμάτων (t_i, y_i), όπου το t_i ανήκει σε ένα διάστημα I, αρχικά εκπαιδεύουμε ένα νευρωνικό δίκτυο $N_i(t)$. Εφόσον έχουμε το $N_i(t)$ μπορούμε να υπολογίσουμε την παράγωγο $\frac{dN_i}{dt}$ και να ορίσουμε ένα μοντέλο διαφορικής εξίσωσης με τη μορφή $\frac{dN_i}{dt} = g(N(t; w))$ όπου το $g(.)$ περιλαμβάνει ένα ή περισσότερα νευρωνικά δίκτυα. Οι παράμετροι του μοντέλου διαφορικής εξίσωσης (w) καθορίζονται μέσω της εκπαίδευσης, έτσι ώστε η διαφορική εξίσωση να ικανοποιείται σε διάφορα σημεία του διαστήματος εκπαίδευσης. Κατά συνέπεια, μόλις ληφθεί το μοντέλο διαφορικής εξίσωσης, μπορούμε να το λύσουμε με αριθμητικές μεθόδους και να χρησιμοποιήσουμε αυτή τη λύση για να βρούμε τις τιμές της συνάρτησης εκτός του διαστήματος I. Η προτεινόμενη μέθοδος έχει δοκιμαστεί σε πολλά προβλήματα και θα παρουσιάσουμε τα πειραματικά αποτελέσματα που προέκυψαν καθώς και τα εμπειρικά συμπεράσματα λαμβάνοντας υπόψην την αποτελεσματικότητα της παρέκτασης.

Λέξεις-κλειδιά: μηχανική μάθηση, βαθιά μάθηση, νευρωνικά δίκτυα, διαφορικές εξισώσεις, παρέκταση(extrapolation)

Chapter 1

Introduction

1.1 Motivation
1.2 Approach and contribution
1.3 Thesis outline

1.1 Motivation

People use their knowledge and instinct to solve mathematical problems every day in real life and obtain inferences. Mathematics are often hidden behind many tasks, hence a solver should educe the necessary information and formulate it in mathematical language to get the answer. Machine solving problems with sophisticated intelligence, as well as mathematical problems has been a topic of interest to scientists for decades [1]. As hardware and software tools have improved and computing power has increased, implementation became faster, making artificial intelligence, and more specifically deep learning, a direction for solving complex mathematical problems. In particular, neural networks have gained great popularity in applied mathematics problems, such as differential equations deploying machine learning and artificial intelligence concepts. An ordinary differential equation is used to describe the dynamics of a physical system that changes over time. These systems span many disciplines, including health sciences, epidemiology, biology, finance and even climate. Modeling the changes in these systems and making predictions can be a difficult task.

Data driven approaches to the study of dynamical systems have gained attention. Recent research in data-driven prediction has been heavily influenced by machine

learning and in particular by neural networks [2] [3]. Neural networks have been used to solve differential equations [4] [5], and recently in a variety of methods such as, ordinary differential equation networks (ODENet) [6], deep residual learning [7] and deep operator networks (DeepONet) [8]. These approaches aim to model the dynamics of an intricate system. Subsequently, an other group of methods has evolved, called physics-informed neural networks [9] [10]. The goal is for neural networks to approximate the evolution of the system after training, by applying physical laws. These methods represent a breakthrough, along with other methods that have followed, addressing the challenges of modeling dynamical systems in a variety of applications. Some examples include health sciences [11] and ecology [12]. Although prediction methods have evolved, the wide range of complex systems in nature still requires the development of new and improved techniques [13].

The motivation for this thesis was triggered by combining neural networks and differential equations, to conduct a study and test a method based on neural networks for extrapolation and future states forecasting. Based on a given set of examples, we aim to exploit differential equations in order to learn a function in the evolution of time.

1.2 Approach and contribution

The main contribution of this thesis is the comprehensive study, design and implementation of an approach, influenced by a range of concepts, from neural networks and numerical methods to differential equations and dynamical systems.

The proposed method attempts to predict the future values of a function and more specifically of a time series. For continuous problems the time-evolution can be described by differential equations in many cases. In this thesis we focus only on problems that can be described by ordinary differential equations (ODEs) and in particular by first order ODEs. Given a data set of examples (t_i, y_i) where t_i belongs in an interval I, we initially train a neural network in order to approximate the function that describes these data in I. Afterwards we discover the differential equation that is satisfied by this function, which is called differential equation model or numerical ODE. The main research question is whether the solution of this differential equation model can achieve good extrapolation, which means whether it can be used to describe

y_i outside I. We aim to compare the proposed method with the initial trained neural network regarding extrapolation performance, and draw conclusions.

1.3 Thesis outline

This thesis consists of 5 chapters:
Chapter 1: It is a general introduction about the scope of this thesis, some related work, the approach and motivation of the thesis.

Chapter 2: It is devoted to an overview of the background and theory relevant to this thesis. We expound on ordinary differential equations and the initial value problem, focusing on numerical methods to solve them. We also analyze and implement a neural network solver for first order differential equations[4]. In addition, we introduce the background theory for machine learning, neural networks and extrapolation.

Chapter 3: A detailed description of the proposed method is presented. The chapter contains the explanatory analysis of each step, as well as the summary of the method.

Chapter 4: In this chapter, the implementation details for the proposed method are presented. Also, the chapter contains the analysis of the experimental results for problems where we know the differential equation as well as for the case where the differential equation is unknown.

Chapter 6: Finally, this chapter contains a discussion of the results, a summary of the thesis and suggestions for future work.

Chapter 2

Preliminary concepts and methods

2.1 Machine Learning

2.2 Numerical methods for solving ordinary differential equations

2.3 Extrapolation

2.1 Machine Learning

The study of machines which perform tasks and solve problems that are considered to require intelligence is called artificial intelligence, AI for short. Machine learning is a discipline of AI and computer science, which uses algorithms and data to mimic the human learning process. In particular ML focuses on the use of parametric models, which are trained on a set of representative examples and have the ability to generalize in order to answer questions for new examples and make predictions. The history of ML dates to many years back, when in 1950 Alan Turing had the idea of “Turing Test” to study if a computer has true intelligence. Afterwards, in 1952, the first computer program that could play checkers was developed by Arthur Samuel, a pioneer in the field of artificial intelligence. The program had the capability to learn new strategies and enhance its performance by playing. Subsequently, Frank Rosenblatt designed perceptron (the first computer neural network) to imitate the human brain. By 1990s, machine learning obtained a more data-driven figure and programs extracted conclusions from big amounts of data training. Since then there have been many breakthroughs in the field that tackle previously unattainable challenges [14].

Machine learning can be viewed as a technological field of study that consists of self learning algorithms that automatically improve themselves and learn by using given data so that they are able to make conclusions, as well as generalize for new data.

The there basic features of a machine learning algorithm (model) are:

1. Define the hypothesis space, which means that we have to make an assumption about the suitable model for our problem, with its corresponding parameters.
2. Furthermore, we have to define an evaluation function, to answer whether a model linked with its parameters is sufficient or not. A good model is the less mistaken with the proper parameters.
3. Define optimization method. We have to determine a way to optimize the parameters during the training.

Machine learning algorithms can be categorized in three basic problems:

- Supervised Learning: It is the machine learning task of learning a function from a given data set with the target variable available. The model is trained on the training data and then it is tested for its generalization ability on new data.
- Unsupervised Learning: Compared to supervised learning, there is no teacher to correct the model, since the given data are unlabeled and the target value is not available. The main problems addressed are finding hidden patterns in the data, such as clustering, anomaly detection and density estimation.
- Reinforcement Learning: Describes a class of problems where an agent operates in an environment and must learn using feedback from this environment.

Due to the algorithms that are used in this thesis, only supervised learning will be analyzed further [15] [16].

2.1.1 Supervised Learning

Supervised earning is so called because the learning process is aided by the given label of observation variables. In particular, a data set consists of examples, each of which has a set of features or attributes and a target variable, called class/label. The goal of supervised learning is to learn the mapping between the features and the target. In order to learn, the data set is divided into a training set and a testing set.

The training set is used during the training process, when the mapping is learned, by using the ground truth labels. The testing set is used only at the end to test whether the inferred learning algorithm achieves generalization, which means whether it can also map the features to the corresponding labels of previously unseen data [17].

The procedure of supervised learning can be described as follows:
Given a set of n data points $\{(x_i, yi)\}_{i=1,2,3...,N}$, where $x_i \in R^n$ is the input vector/features of the i_{th} example and $y_i \in R^d$ is the ground truth label/class, a learning algorithm tries to learn the function $f : R^n \rightarrow R^d$ such that $y_i \approx f(x_i)$. The function f is a parametric model and after the training phase the best parameters are found, based on the training set. The goal is to infer a function f that can also be representative for new data points in the interval of training[18].
Supervised learning can be categorized into two types of problems:
Classification: The algorithms recognize specific input values/features and classify them to categories.
Regression: An algorithm is used to map the input values/features to a continuous value and not a class.

There are many algorithms, methods and applications of supervised learning. Due to the objective of this thesis, further analysis will focus on artificial neural networks, and in particular feed forward neural networks. Artificial neural networks and more specifically deep neural networks are the basis of a subset of machine learning that has evolved, Deep Learning. We will move sequentially from the most basic of neural networks to further extensions.

2.1.2 Artificial Neural Networks

The first computer neural network that attempted to mimic a human neuron is called Perceptron. It can be concerned as a mathematical linear model for binary classification. As a single neuron it can learn only linearly separable patterns, that is, the algorithm takes binary classified input data and outputs a line or hyperplane that attempts to separate data of one class from data of the other. The parameters of the algorithm is w_0, called as bias and a set of weights w_i, that correspond to the feature vector.

Mathematically the model can be described as follows:
Given a training set of N examples each of which has a feature vector $x = [x_1, x_2, ..., x_n]$

and a target variable of two classes t with $t \in \{\omega_1, \omega_2\}$, the problem is to compute the unknown parameters w_0 and w_i, $i = 1, 2, 3, ..., n$ in order to define the decision hyperplane.

Artificial neural networks are computational networks biologically inspired by the human nervous system. Perceptrons can process data with binary output, while there are many more complex problems to be solved. The initial idea behind artificial neural networks is to stack perceptron units, in order to create layers. The layers are in fact linear functions that apply linear transformations to the input vector. After the linear transformation, it is important the processing element to pass through a non linear function, called activation function. The use of nonlinear activation functions after a linear transformation allows the inference of any type of function that maps the inputs to the corresponding outputs, even if it is a complex mapping. Thus they can be viewed as functional approximation machines. Conventionally, the activation function is the same for all the layers and all the neurons are activated except from the input units.

One typical activation function is the Sigmoid, which is non linear as should be and continuous, so the neural network is differentiable and this helps the optimization process. Sigmoid function is given by

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

The architecture of the an artificial neural network is as follows:

- Input layer: Accepts the feature vector $x = [x_1, x_2, .., x_n]$
- Output layer: It is the last layer of the network that has one node for each value of the output.
- Hidden layers: The layers between the input and the output layer.

Some extra important notes for the structure of neural networks include that neurons in the same layer do not connect to each other and the neurons of one layer are linked with the neurons of the next and previous levels only. There has to be at least one hidden layer in the neural network. The number of hidden layers is called the depth of the network – thus the term of deep neural networks and deep learning. Also, the number of nodes in a layer is called the width of the layer.

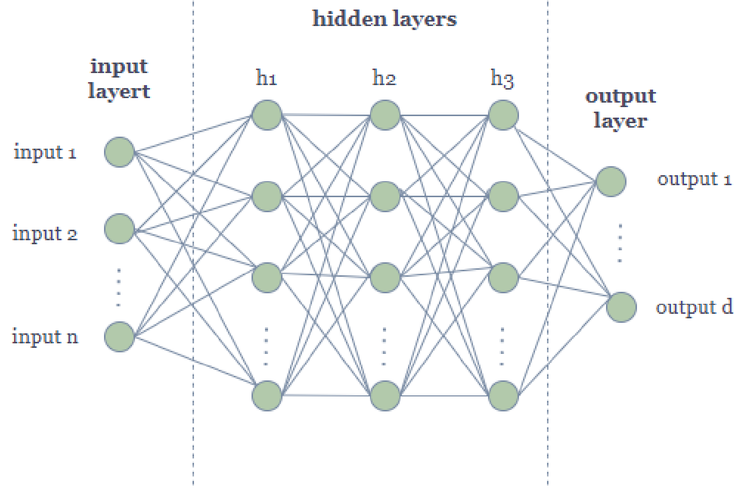

Figure 2.1: Architecture of an artificial neural network

2.1.3 Feedforward neural networks

The Feedforward neural network is in fact a multi layer perceptron (MLP) and is known as the simplest neural network and one of the most popular neural networks[19]. In the feedforward neural network, the output of a layer is input of the immediately following layer, there is no connections between neurons in the same layer and the computations are done sequentially. Firstly we have to define the architecture of the network $\left(H, dh, f\right)$, where H is the number of hidden layers, dh is the number of neurons in each hidden layer and f is the activation functions to be used

The parameters of the model are the weights and biases, given by matrices $W(i)$, $b(i)$ correspondingly, between layers i and i+1, with $1 < i < L - 1$. We assume that we have L layers where the input layer is the first and the output layer the L_{th}.

Particularly $W(i)$ is a $M \times N$ matrix, where M is the number of neurons in layer i and N is the number of neurons in layer i+1. As for biases, they are a vector $M \times 1$, where M is the number of neurons of layer i+1 (figure (2.2)).

Given a training set of N examples, each of them has a feature vector X= $[x_1, x_2, .., x_n]^T$ and a target variable t, the input layer is consisted of the feature vector and the output is computed as follows:

- $W(1)^T X + b(1) = h_1$, h_1 is the vector of neurons of the first hidden layer
- the result is passed through a non linear activation function $f^{(1)}$, so we have $f^{(1)}(h_1)$

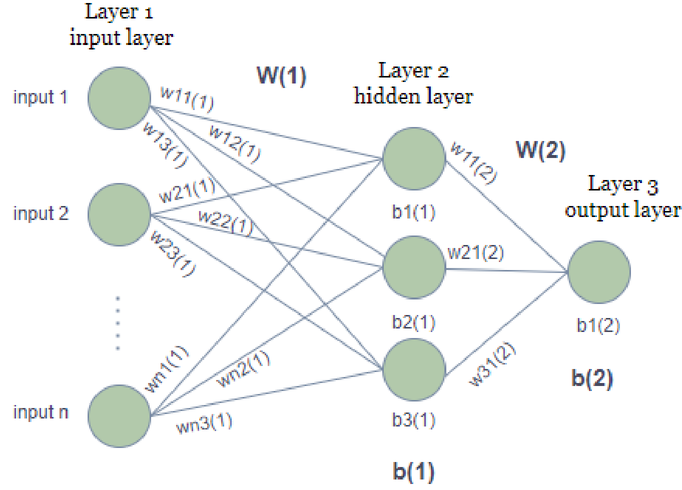

Figure 2.2: Example of a feed forward neural network, here $W(1)$ is $n \times 3$, $W(2)$ is 3×1, $b(1)$ is 3×1, $b(2)$ is 1×1

- $W(2)^T f^{(1)}(h_1) + b(2) = h_2$, where h_2 is the vector of neurons of second hidden layer
- we pass it through an activation function so we have $f^{(2)}(h_2)$
- This process is repeated for all layers until we get the output.

Mathematically, an artificial neural network can be viewed as a function $F : X \to Y$, where $X \in R^n$ is the input space, and $Y \in R^d$ is the output space. The output of the neural network $\hat{y} \in R^d$ for a given input $x \in R^n$ is given by:

$$\hat{y} = F(x) =$$

$$f^{(L)}\Bigg(w^{(L)T} f^{(L-1)}(...(w^{(3)T} f^{(2)}(w^{(2)T} f^{(1)}(w^{(1)T}x + b^{(1)}) + b^{(2)}) + b^{(3)})...\Bigg) + b^{(L)} \quad (2.2)$$

This is as called,the forward pass of the feed forward neural network.

Training of neural networks - backpropagation algorithm

The goal is to learn the optimal parameters of weights and biases in order to infer the model F that maps the input vectors to the corresponding outputs, or $F(x) \approx t$ $\forall(\vec{x}, \vec{t}) \in$ training set. Initially, the parameters are random and a process of training will correct them. The problem is approached as a typical optimization task. Thus an appropriate cost function should be adopted and an algorithmic scheme to optimize it.

Since we refer to a regression problem, a basic choice for the loss function, is the sum of squares. Supposing that $o(x,t)$ denotes the output vector of the network after the forward pass of input vector x, with weights w we have

$$E(w) = \frac{1}{2} \left\| t - o(x,t) \right\|^2 \tag{2.3}$$

The algorithm for the minimization of (2.3) can be gradient descent or a more efficient version called stochastic gradient decent or any other gradient based optimization method. With gradient decent we run through all the samples in the training set for an update of a parameter in a particular iteration, while in stohastic gradient decent, we use a subset of training samples, called minibatch.
The weights are updated iterative following the rule:

$$w_{t+1} = w_t - \rho \frac{\partial E}{\partial w} \tag{2.4}$$

To apply gradient descent, we have to calculate the derivative. In 1986 Hinton proposed backpropagation, influenced by the chain rule [20].

Theorem 2.1. ***(Chain Rule)*** *Let $g = (g_1, .., g_n) : I \to R^n$ a differentiable curve. $U \subset R^n$ open subset with $g(I) \subset U$ and $f : U \to R$ differentiable. Then $f \circ g : I \to R$ differentiable with*

$$(f \circ g)'(t) = grad(f(g(t))(g'(t)) = \frac{\partial f}{\partial x_1}(g(t))g_1'(t) + ... + \frac{\partial f}{\partial x_n}(g(t))g_n'(t)$$

$\forall t \in I$

The theorem follows [21].

Backpropagation intuition

We initialize the network with random parameters and we do a forward pass for an example (x,t). The output of the network is $o(x,t)$ and thus we have the error of the network, given by (2.3), and we can back propagate it. To explain the back propagation algorithm we will present a specific example, and then we will generalize. One important note is that we use sigmoid function as activation function for all the layers, except the first layer, and it is true that $\sigma'(t) = \sigma(t)(1 - \sigma(t))$.

We have:

- $X = [x_1, x_2]^T = a^{(1)}$

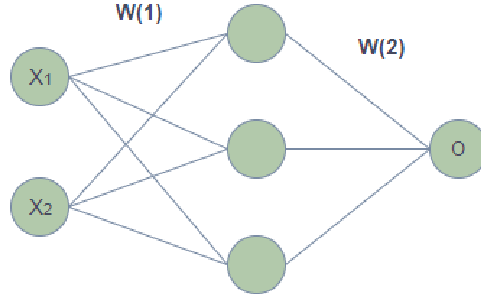

Figure 2.3: Example of a neural network that we apply back propagation

- $h^{(2)} = W(1)^T X \rightarrow a^{(2)} = \sigma(h^{(2)})$
- $h^{(3)} = W(2)^T a^{(2)} \rightarrow o = \sigma(h^{(3)}) = a^{(3)}$
- $E(w) = \frac{1}{2}(a^{(3)} - t)^2$ and we need the derivatives $\frac{\partial E}{\partial W(2)}$ and $\frac{\partial E}{\partial W(1)}$

We start from the output layer, and by using the chain rule we have:

$$\frac{\partial E}{\partial W(2)} = \frac{\partial E}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial W(2)}$$

where

- $\frac{\partial E}{\partial a^{(3)}} = a^{(3)} - t$
- $\frac{\partial a^{(3)}}{\partial h^{(3)}} = \sigma'(h^{(3)}) = \sigma(h^{(3)})(1 - \sigma(h^{(3)})$
- $\frac{\partial h^{(3)}}{\partial W(2)} = a^{(2)}$

$$\Rightarrow \frac{\partial E}{\partial W(2)} = (a^{(3)} - t)\sigma(h^{(3)})(1 - \sigma(h^{(3)})a^{(2)}$$

Afterwards, we continue the same process, starting from the last hidden layer until the input layer, here we have only one hidden layer.

$$\frac{\partial E}{\partial W(1)} = \frac{\partial E}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W(1)}$$

where

- $\frac{\partial E}{\partial a^{(2)}} = \frac{\partial E}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial a^{(2)}}$, where
  - $\frac{\partial E}{\partial a^{(3)}} = a^{(3)} - t$
  - $\frac{\partial a^{(3)}}{\partial h^{(3)}} = \sigma'(h^{(3)}) = \sigma(h^{(3)})(1 - \sigma(h^{(3)})$

- $\frac{\partial h^{(3)}}{\partial a^{(2)}} = W(2)$

- $\frac{\partial a^{(2)}}{\partial h^{(2)}} = \sigma'(h^{(2)}) = \sigma(h^{(2)})(1-\sigma(h^{(2)}))$
- $\frac{\partial h^{(2)}}{\partial W(1)} = X = a^{(1)}$

$$\Rightarrow \frac{\partial E}{\partial W(1)} = (a^{(3)} - t)\sigma(h^{(3)})(1-\sigma(h^{(3)})W(2)\sigma(h^{(2)})(1-\sigma(h^{(2)}))X$$

Backpropagation general form

In order to obtain a general form of the derivatives that backpropagation gives we compute a quantity δ. We assume that we have a network with L layers, that the first layer is the input layer and the output layer is the L_{th} layer.

- Output layer

$$\delta_L = (a^{(L)} - t)\sigma(a^{(L)})(1-\sigma(a^{(L)}))$$

- Hidden layers

$$\delta_i = \delta_{i+1}(\sigma(a^{(i)})(1-\sigma(a^{(i)}))W(i)$$

$i = 2, ..., L-1$

Since it hold that

$$\frac{\partial E}{\partial W(i)} = \delta_{i+1}a^{(i)}$$

we calculate the derivative and apply it in (2.4) for an appropriate value of learning rate ρ.

For the paramater of bias we have that

$$\frac{\partial E}{\partial b(i)} = \delta_i$$

with $i = 2, .., L$

There are many more details about neural networks in order to expand the topic but with respect to the objective of this thesis, no further analysis will be done. The main task in this thesis is regression with neural networks. There are many non linear activation functions, like rectified linear activation function or ReLU, Hyperbolic Tangent or Tanh and SoftMax. In classification problems Categorical cross entropy is the commonly used loss function, that is well combined with SoftMax activation function.

2.2 Numerical methods for solving ordinary differential equations

2.2.1 Ordinary differential equations

Differential equations are a field of both pure and applied mathematics and is a powerful tool in many areas of science, being used extensively in mechanics, physics, biology, and geometry. Physical lows that govern phenomena can be written as ordinary differential equations, so that the equations themselves represent the relationship between physical quantities and their rate of change through these lows.

A differential equation is an equation that represents the relationship between unknown functions and their derivatives. There are some types of differential equations, depending on their characteristics. Some of the most common characteristics that play a role in categorising an equation are the order, whether it is an ordinary or partial equation, linear or non-linear and homogeneous or not. With respect to the scope of this thesis, we mention the ordinary differential equations (ODEs) and the initial-value problem.

ODEs describe the change of a variable $y(x)$ with respect to an independent variable x and the general form of an nth order ODE is given by (2.5)

$$a_0(x)y + a_1(x)y' + a_2(x)y'' + ... + a_n y^{(n)} + p(x) = 0 \tag{2.5}$$

where $y' = \frac{dy}{dx}$ is the first order derivative, $y'' = \frac{d^2y}{dx^2}$ the second derivative and respectively $y^{(n)} = \frac{d^n y}{dx^n}$ the nth derivative.

The equation (2.5) can be written also as an implicit ODE of n order, as is called, given by the form:

$$F(x, y, y', y'', ...y^{(n-1)}, y^{(n)}) = 0 \tag{2.6}$$

or the explicit ODE of n order given by:

$$F(x, y, y', y'', ...y^{(n-1)}) = y^{(n)} \tag{2.7}$$

The solution of an ordinary differential equation requires to define the type of the ODE and follow the corresponding methodologies. Also, we have to mention that the solution is a differentiable function as many times as the order of the ODE, which verifies the ODE. However, in the scope of this thesis, we do not dive deep into how to solve the general ordinary differential equation.

2.2.2 Initial value problem

An ODE that its solution satisfies a given initial condition is known as an initial value problem (IVP). A nth order initial value problem constist of two parts:

- the ODE, given by 2.5
- initial conditions which gives the values of $y(t), y'(t), y''(t), ..., y^{(n)}(t)$ at a particular point, that can be written in the form:

$$\begin{cases} y(t_0) = y_0 \\ y'(t_0) = y_1 \\ y''(t_0) = y_2 \\ \vdots \\ y^{(n)}(t_0) = y_{(n)} \end{cases} \tag{2.8}$$

In the scope of this thesis, we introduce the first order initial value problem that is given from the form

$$\begin{cases} y'(t) = f(t, y(t)), \quad t \in [a, b] \\ y(a) = y_0 \end{cases} \tag{2.9}$$

We suppose that $f \in C([a, b] \times R)$ and the IVP given by (2.9) is satisfied by the function $y \in C^1([a, b])$. Most of real-life situations, require the solution of an initial-value problem to be modelled and they are complicated to be solved exactly. An approach for their solution is to use numerical methods for approximating the solution of the original problem. We present a brief description for some numerical methods further down.

Before presenting methods for approximating the solution of a first order initial-value problem, we need to consider some definitions from the theory of ordinary differential equations. The main things to consider are about the existence and uniqueness of a solution, as well as whether it is well posed.

2.2.3 The Existence and Uniqueness of Solutions

Definition 2.1. A function $f(t, y) : [a, b] \times R \to R$ is said to satisfy a *Lipschitz condition* in the variable y, uniformly in t, if

$$(\exists L > 0)(\forall t \in [a, b])(\forall y_1, y_2) : |f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2| \tag{2.10}$$

The constant L is called a *Lipschitz constant* for f.

Theorem 2.2. *Assume that $f(t,y) : [a,b] \times R \to R$ is continuous, differentiable and satisfies the Lipschitz condition with respect to y, uniformly with respect to t, then for any inital value $y_0 \in R$ the initial value problem*

$$\begin{cases} y'(t) = f(t, y(t)), & t \in [a,b] \\ y(a) = y_0 \end{cases}$$

possesses a unique solution.

Well posed methods

Now that we answered whether an IVP has a unique solution, it is also important to answer to an other question. Initial-value problems describe physical phenomena, thus we need to know especially when we use numerical methods to solve them, whether small changes in the statement of the problem result to correspondingly small changes in the solution.

Definition 2.2. The initial value problem

$$\begin{cases} y'(t) = f(t, y(t)), & t \in [a,b] \\ y(a) = y_0 \end{cases}$$

is said to be well posed if:

- it has a unique solution $y(t) \in C^1[a,b]$
- the initial value problem

  $$\begin{cases} z'(t) = f(t, z(t)), & t \in [a,b] \\ z(a) = z_0, & z_0 \neq y_0 \end{cases}$$

  has a unique solution $z(t) \in C^1[a,b]$ such that

  $$|z(t) - y(t)| < e^{L(t-a)}|z_0 - y_0|$$

Numerical methods for the solutions of an initial value problem require it to be well posed. The following theorem specifies conditions to ensure that an initial-value problem is well-posed.

Theorem 2.3. *Suppose a continues function $f(t, y) : [a, b] \times R \to R$ that satisfies a Lipchitz condition in the variable y, uniformly in t, then the initial value problem given by (2.9) is well posed.*

Some methods of IVPs are described in [22] [23].

2.2.4 Basic concepts of Numerical Methods for Initial-Value Problems

This section is intended to give an idea on how to solve ODEs numerically. Approximating the solution of an initial-value problem given by (2.9) with a numerical method is not to find a continuous, close to the solution approximation, but instead, approximations of $y(t)$ will be generated at various values in the interval $[a, b]$, called mesh points. Once we obtain the approximate solution at these points, we can interpolate to other points of $[a, b]$. The simplest numerical method for initial value problems is Euler's method, that is derived by using Taylor's theorem, as well as higher order Taylor methods. They are seldom used in practice, as these concepts are generalized to more advanced methods, like Runge–Kutta and multistep schemes [22]. In the scope of this thesis, we will introduce Euler's method in order to get an intuition and then we will expand to Runge Kutta.

Euler's method

The goal of Euler's method is to obtain an approximation of the solution $y(t)$ to the well-posed initial-value problem

$$\begin{cases} y'(t) = f(t, y(t)), \quad t \in [a, b] \\ y(a) = y_0 \end{cases}$$

Let $a = t^0 < t^1 < .. < t^N = b$ be an uniform partition of the interval $[a, b]$. Euler's formula is given by:

$$\begin{cases} y^{n+1} = y^n + hf(t^n, y^n) \\ y^0 = y_0 = y(a) \end{cases} \tag{2.11}$$

Once we use a uniform partition, with $N \in \mathbb{N}$ we let the distance between the points to be $h = \frac{b-a}{N}$, called step size, and $t^i = a + hi$, $i = 0, 1, 2, ..N$. Euler's numerical

method gives a set of approximations of $y(t)$ at points t^i. We start with $y^0 = y_0$ and then we produce $y^i \approx y(t_i)$.

Higher order Taylor methods

Euler's method is derived by Taylor's theorem for n=1.

Theorem 2.4. *(Taylor) Suppose that f is defined on some open interval I around a and $f^{(n+1)}(x)$ exists on this interval. Then for each $x \neq a$ in I, there is a value c between x and a, so that*

$$f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n + \frac{f^{(N+1)}(c)}{(N+1)!}(x-a)^{(N+1)} \tag{2.12}$$

$$= f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)}{2!} + ... + f^{(n+1)}(c)\frac{(x-a)^{N+1}}{(N+1)!}$$

Note

By using (2.12), for $y(t^{n+1})$ around t^n with $h = t^{n+1} - t^n$, we get

$$y(t^{n+1}) = y(t^n) + hy'(t^n) + \frac{h^2}{2!}y''(t^n) + ... + \frac{h^{n+1}}{(n+1)!}y^{(n+1)}(c)$$

with $c \in (t^i, t^{i+1})$

The main disadvantage of the Taylor methods is that they require the computation and evaluation of the derivatives of $f(t, y(t)) = y'(t)$. This is a complicated and time-consuming procedure for most problems.

2.2.5 Runge Kutta methods

Runge-Kutta methods have the advantages of the Taylor methods but eliminate the need to compute and evaluate the derivatives of $f(t, y(t))$. The RK (Runge–Kutta) methods are single-step methods, which means that for the computation of the approximation y^{n+1} they use only the approximation at the previous point y^n. We first consider the initial value problem (2.9) and we seek its solution $y : [a, b] \to R$. Each set of constants (A, τ_i, b_i) describes a RK method, that is usually written in the form of a Runge–Kutta tableau (notation of J. Butcher). We have that $i, j = 1, 2, 3, ..., q$ and q are the computational rules. Let $N \in \mathbb{N}$, $h = \frac{b-a}{N}$, $t^i = a + ih$ points in the uniform partition of $[a, b]$ and y^n the approximation of $y(t^n)$. We furthermore introduce the

$$
\begin{array}{cccc|c}
a_{11} & a_{12} & \dots & a_{1q} & \tau_1 \\
a_{21} & a_{22} & \dots & a_{2q} & \tau_2 \\
\vdots & \vdots & & \vdots & \vdots \\
a_{q1} & a_{q2} & \dots & a_{qq} & \tau_q \\
\hline
b_1 & b_2 & \dots & b_q &
\end{array}
= \begin{array}{c|c} A & \tau \\ \hline b^T & \end{array}
$$

Table 2.1: RK tableau

intermediate points $t^{n,i} = t^n + \tau_i h$ with $y^{n,i}$ the approximation of $y(t^{n,i})$. The general RK method with q intermediate stages is described by:

$$
\begin{cases}
y^0 = y_0 \\
y^{n,i} = y^n + h \sum_{j=1}^{q} a_{ij} f(t^{n,j}, y^{n,j}) \\
y^{n+1} = y^n + h \sum_{i=1}^{q} b_i f(t^{n,i}, y^{n,i})
\end{cases}
\tag{2.13}
$$

The stages $y^{n,i}$ are approximations to $y(t^{n,i})$ but are only used for the computation of the approximations y^{n+1}.

One common used Runge Kutta method is of order four, that is also used in this thesis, given by (2.14).

$$
\begin{aligned}
y^0 &= y_0 = y(a) \\
k_1 &= hf(t^n, y^n) \\
k_2 &= hf\left(t^n + \frac{h}{2}, y^n + \frac{1}{2}k_1\right) \\
k_3 &= hf\left(t^n + \frac{h}{2}, y^n + \frac{1}{2}k_2\right) \\
k_4 &= hf(t^{n+1}, y^n + k_3) \\
y^{n+1} &= y^n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
\tag{2.14}
$$

We introduce the notation k_1, k_2, k_3, k_4 into the method to eliminate the need for successive nesting in $f(t, y)$ [22] [23]. The RK of order four to approximate the solution of an initial value problem

$$
\begin{cases}
y'(t) = f(t, y(t)), \quad t \in [a, b] \\
y(a) = y_0
\end{cases}
$$

for $(N+1)$ equally spaced numbers in the interval $[a,b]$ is given by algorithm 3.1.

Algorithm 2.1 RK of order 4 algorithm

Require: integer N, initial condition $y_0 = y(a)$, an interval $[a,b]$

1: $h = \frac{b-a}{N}$
2: $y^0 = y_0$
3: **for** n=0 to N-1 **do**
4: $\quad t^n = a + nh$
5: $\quad k_1 = hf(t^n, y^n)$
6: $\quad k_2 = hf\left(t^n + \frac{h}{2}, y^n + \frac{1}{2}k_1\right)$
7: $\quad k_3 = hf\left(t^n + \frac{h}{2}, y^n + \frac{1}{2}k_2\right)$
8: $\quad k_4 = hf(t^{n+1}, y^n + k_3)$
9: $\quad y^{n+1} = y^n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
10: **end for**

2.2.6 Solving ODEs with neural networks

In respect with the topic of this thesis, we will do a briefly description of the method that is presented in [4]. We followed the methodology for solving an ODE of first order.

Description of the method

Assume a general differential equation definition

$$G\left(x, \Psi(x), \nabla\Psi(x), \nabla^2\Psi(x)\right) = 0, \quad \forall x \in D \subset R^n \tag{2.15}$$

subject to certain boundary conditions (BC's) (for instance Dirichlet and/or Neumann), D is the definition domain and $\Psi(x)$ is the solution of this ODE.

In order to solve this ODE we have to use a set of points of the discretization of the domain D and its boundary S, $\hat{D}$ and $\hat{S}$ respectively. The problem is then transformed into the following system of equations:

$$G\left(x_i, \Psi(x_i), \nabla\Psi(x_i), \nabla^2\Psi(x_i)\right) = 0, \quad \forall x_i \in \hat{D} \tag{2.16}$$

subject to the constraints imposed by the B.Cs.

Suppose that $\Psi_t(x,p)$ is the trial solution with adjustable parameters p. Then the problem is transformed to:

$$min_p \sum_{x_i \in \hat{D}} \Big(G\big(x_i, \Psi_t(x_i,p), \nabla\Psi_t(x_i,p), \nabla^2\Psi_t(x_i,p)\big)\Big)^2 \tag{2.17}$$

subject to the constraints imposed by the B.Cs.

We choose a form for the trial function such that by construction satisfies the BC's. This is achieved by writing it as a sum of two terms:

$$\Psi_t(x) = A(x) + F(x, N(x,p)) \tag{2.18}$$

where A(x) satisfies the boundary conditions and do not contain adjustable parameters and $F(x, N(x,p))$ contains the single-output feedforward neural network $N(x,p)$ with adjustable parameters p, which are the weights and biases that are adjusted in order to deal the minimization problem 2.17.

With respect to this thesis, we deal with problems of the form:

$$\frac{d\Psi(x)}{dx} = f(x, \Psi) \tag{2.19}$$

with $x \in [a,b]$, $\Psi(a) = A$ and solution $\Psi(x)$. For the solution of the problem 2.19 we have the trial solution

$$\Psi_t(x) = A + (x-a)N(x,p)$$

and we have to find the optimal parameters p (weights, biases) in order to minimize the loss function given by 2.20.

$$E_p = \sum_i \left\{ \frac{d\Psi_t(x_i)}{dx} - f(x_i, \Psi_t(x_i)) \right\}^2 \tag{2.20}$$

The derivative $\Psi_t'(x) = \frac{d\Psi_t(x)}{dx}$ is given by:

$$\Psi_t'(x) = N(x,p) + (x-a)\frac{dN(x,p)}{dx} \tag{2.21}$$

After the process of minimizing 2.20 we have the optimal parameters p and thus we have $\Psi_t(x) \approx \Psi(x)$. The gradient of the network with respect to inputs is described in section 3.3.

2.3 Extrapolation

2.3.1 Definition, basic concepts

In mathematics extrapolation is a type of estimation of some function at points that are outside the range of known values. Through extrapolation we make inferences about a hypothetical situation based on known facts, that is, we can do forecasting. Interpolation is a related term, which involves determining a function's value at intermediate points based on the value of other points.

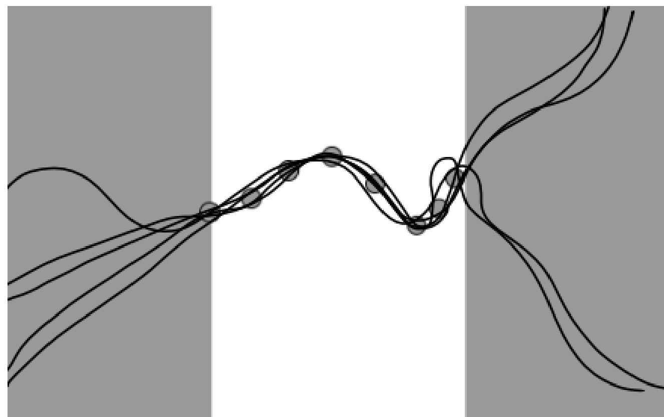

Figure 2.4: Given the data points in white section, the lines represent polynomial models in order to make predictions for intermediate points in white region(interpolation) as well as make predictions for data outside the domain in grey region(extrapolation)

We can see that there is a risk in extrapolation as the model predictions outside of the training domain are sensitive and can result in unpredictable behaviour.

In order to choose the optimal method for extrapolation we have to initially answer to some questions that concern the problem and the data that are available. The estimations have high risk of uncertainty as we expand known experience into an area not known. Some of the methods that are used are:

Linear extrapolation that is used for data that can be represented by a line or a hyperplane and it is expanded to regression techniques.

Polynomial extrapolation that is typically done by Lagrange interpolation or using Newton's method of finite differences to create a Newton series that fits the data. The resulting polynomial may produces estimations for extrapolation.

There are also other methods of extrapolation but with respect to the topic of this thesis we will not expand further.

Extrapolation of neural networks

It is well known that neural networks are universal function approximators. However, neural networks are not expected to extrapolate well for data points in a non trained region, in most nonlinear tasks. In the scope of this thesis we care about the extrapolation ability of neural networks, through a proposed method that is presented further down. We exploit the ability of differential equations to follow the function evolution and we try to discover if the neural networks can be used in a simplified way for extrapolation purpose.

Chapter 3

The proposed method

3.1 Structure of the method
3.2 Summary of the method
3.3 Gradient of a neural network
3.4 Variations of the method

3.1 Structure of the method

The proposed method consists of three basic steps, that are applied sequentially. Assuming we have a problem that can be described by two variables in a certain domain, for example, a time series, a neural network can model the relationship between the variables in this domain. However, one limitation is that it has no extrapolation capability. Therefore, it may not be appropriate to use neural networks to make reliable predictions outside the domain for which they were trained. The method proposed here attempts to obtain extrapolation results for the problem described and is presented analytically in this section.

3.1.1 First step - initial neural network

The problem consists of n data points $(t_i, P(t_i))$ corresponding to an unknown function P(t). We assume that $t_i \in I = [x_{min}, x_{max}]$. We approximate the unknown function P(t) by training the neural network $N_i(t)$ in I. As we said, a neural network can be

trained to approximate a function regardless of the complexity in the defined domain, so let us assume that

$$P(t) \approx N_i(t)$$

in I. However, for $t_i > x_{max}$, $N_i(t)$ yields results that cannot be related to $P(t)$, and the extrapolation may fail.

Neural network description

$N_i(t)$ is a single-linear-output feedforward neural network with one input unit t and one hidden layer consisted of H sigmoid units. For a given input t the output of the network is $N = \sum_{i=1}^{H} v_i \sigma(z_i)$,where $z_i = w_i t + u_i$, w_i denotes the weight from the input unit to the hidden unit i, v_i denotes the weights of the hidden unit i to the output, u_i denotes the bias of the hidden unit i and $\sigma(z)$ is the sigmoid transfer function.

The architecture of $N_i(t)$ is given graphically in figure 3.1.

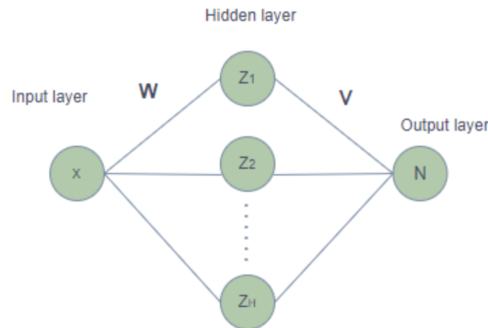

Figure 3.1: Architecture of $N_i(t)$

3.1.2 Second step – numerical ODE definition

Suppose that $P(t)$, with $P(t) \approx N_i(t)$, is the solution of an unknown ordinary differential equation (ODE) of first order. The idea of the proposed method is to train a model to learn this ODE. Differential equations can express laws of physics and predict the evolution of a system, hence it would be useful to explore the ODE that gives $P(t)$ as solution, in order to obtain predictions for future data. We define this ODE by equation (3.1)

$$\frac{dN_i(t)}{dt} = N_o(t) N_i(t) \tag{3.1}$$

$N_o(t)$ is a feed-forward neural network with the same architecture as $N_i(t)$ and parameters θ which are the corresponding weights and biases.The network $N_o(t)$ is trained in the interval $I_1 = [s_{min}, s_{max}]$, where
$I_1 \subset I$ and $s_{min} > x_{min}$.

The error quantity to be minimized for training $N_0(t)$ is given by

$$E(\theta) = \sum_i \{\frac{dN_i(t)}{dt} - N_o(t)N_i(t)\}^2 \tag{3.2}$$

After the training of $N_o(t)$ we have the numerical ODE (3.1) and we can finally solve this ODE to check if its solution approximates P(t).

3.1.3 Third step – solution of numerical ODE

ODE (3.1) should give accurate solution at least in the domain I and most desirable after x_{max}. The two methods that were chosen to solve the numerical ODE are:

- Runge Kutta of order 4, as described in 2.2.5.
- Neural network solution, as described in 2.2.6.

We solve the numerical ODE (3.1) in the domain $I_3 = [s_{min}, f_{max}]$ with $f_{max} > x_{max}$, as we care about the extrapolation results.

3.2 Summary of the method

In total the following three steps constitute the proposed method,

- Given n data points $(t_i, P(t_i))$
- $I = [x_{min}, x_{max}]$: domain of the initial neural network $N_i(t)$ training
  - $P(t) \approx N_i(t)$
- Numerical ODE (3.1) : $\frac{dN_i}{dt} = N_o(t)N_i(t)$
- $I_1 = [s_{min}, s_{max}]$: domain of (1) definition and $N_o(t)$ training
  - $I_1 \subset I$ and $s_{min} > x_{min}$
- $I_2 = [s_{min}, f_{max}]$: domain of ODE (1) solution with $f_{max} > x_{max}$.

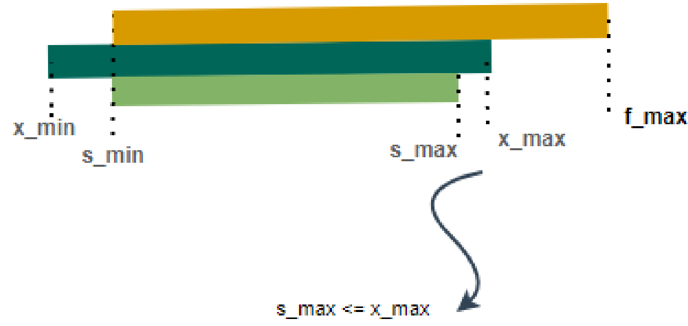

Figure 3.2: Intervals of the proposed method graphically

The illustration of the intervals is given in figure 3.2.

The reason for choosing these intervals is that we are mainly concerned with the property of extrapolation. It has beedn observed that ignoring the starting points from x_{min} helps to get better results after x_{max}. Better results in extrapolation mean that the numerical ODE (3.1) manages to approximate the exact ODE with solution $P(t)$ after x_{max}. The value of s_{max} is chosen to be less or equal to x_{max} because the initial neural network $N_i(t)$ near the edge may not be able to approximate P(t). The accuracy of $N_i(t)$ is very important for the performance of $N_o(t)$.

3.3 Gradient of a neural network

The efficient minimization of the error quantity given by equation (3.2) requires the gradient of the neural network $N_i(t)$ with respect to its inputs. Since an artificial neural network is a differential approximation function, we can compute the derivatives. The proposed method was tested for first order ODEs, since the main purpose is to model data (ti,P(ti)) coming from a function P(t) that is the solution of an ODE we do not know, so we assume it is a first order differential equation. It would be simple to extend the method for second or higher order ODEs, since we can have the derivatives of N(t).

- first derivative

$$\frac{dN_i}{dt} = \sum_{i=1}^{H} v_i \sigma'(z_i) = \sum_{i=1}^{H} v_i \sigma(z_i)(1 - \sigma(z_i)) w_i$$

- second derivative

$$\frac{d^2N}{dt^2} = \sum_{i=1}^{H} v_i \sigma''(z_i) = \sum_{i=1}^{H} v_i \sigma(z_i)(1-\sigma(z_i))(1-2\sigma(z_i))w_i^2$$

Similarly we can find the higher order derivatives.

3.4 Variations of the method

This section presents some different models that have been tested instead of the model described by equation (3.1). There are many variants that could be tried, three of them are presented here.

Add a second neural network - correction network

The ODE model (3.1), attempts to approximate the exact ODE with solution $P(t) \approx N_i(t)$ can be modified into a more sophisticated model, given by equation (3.3),

$$\frac{dN_i}{dt} = N_o(t)N_i(t) + N_c(t) \tag{3.3}$$

where N_c(t) is a feed-forward neural network with the same architecture as $N_i(t)$ and $N_o(t)$ and the corresponding parameters of weights and biases. This approach was inspired because the model from equation (3.1) does not give accurate results at the points where $N_i(t) = 0$. This is because the form of (3.1) is $\frac{dN_i}{dt} = net_o(t)N_i(t)$. Thus, when $N_i(t)$ becomes zero, the derivative is also zero and the ODE model (3.1) cannot approximate the exact ODE with solution P(t). In the experimental part, it is found that the ODE (3.1) generates spikes at these points t_i with $N_i(t_i) = 0$. The idea is that the second network can correct these spikes, contributing to a better extrapolation. There are two approaches:

- train the two networks simultaneously
- train $N_o(t)$ first and then $N_c(t)$

The simplified model

One simple variation of ((3.1)) is to keep $N_o(t)$and $N_c(t)$ and train them to learn the derivative of the network $N_i(t)$ without including the model $N_i(t)$:

$$\frac{dN}{dt} = N_o(t) + N_c(t). \tag{3.4}$$

Also the two networks can be trained together or separately. Furthermore, we have considered the model

$$\frac{dN_i}{dt} = N_o(t) \tag{3.5}$$

which is the most simple model that can be used, where we just approximate the derivative with $N_o(t)$.

The method that is proposed in this chapter follows algorithm 3.1.

Algorithm 3.1 Algorithm of the proposed method

Require: a data set of examples $(t_i, P(t_i))$, the interval I=$[x_{min}, x_{max}]$ of t_i

1: train the initial network $N_i(t)$ in I
2: select the numerical ODE model
3: define I_1
4: train the networks of the defined numerical ODE in I_1
5: define I_2
6: solve the numerical ODE in I_2 with Runge Kutta of order 4 and Neural network
7: plot together: $P(t)$, the initial $N_i(t)$, the Runge Kutta solution of the numerical ODE and the neural network solution of the numerical ODE and observe the extrapolation results.

Chapter 4

Experimental results

4.1 Software requirements

4.2 Datasets

4.3 Hyperparameters and architecture details

4.4 Illustration of the method

4.5 Extrapolation results

4.6 Data with noise

4.7 Results of the alternative ODE models

4.8 Experiment with unknown ODE

In this section the experimental process is presented and the results that came up. An important note is that the performance of the proposed method strongly depends on the initialization of the intervals I, I_1, I_2. Therefore, each of the experiments had to be performed several times, with different values of the intervals and different parameter settings. Each step of the implementation depends on the previous step and strongly affects the extrapolation. For example, poor training for $N_i(t)$ near x_{max}, affects the next steps and thus the extrapolation. Once the experimental results were collected, an analysis was performed to draw conclusions. It is important to say that after the experimental study the conclusion is that the model (3.1) gives the best extrapolation results, compared to the other models.

4.1 Software requirements

The code of this thesis was written in Python programming language using the following libraries:

- Numpy library, a fundamental package for scientific computing with Python and mathematical computations with matrices and vectors.
- Pytorch, an open source machine learning framework which uses Tensors, useful for calculus problems with derivatives, building and training artificial neural networks.
- Matplotlib, a library for creating different kind of visualizations.

4.2 Datasets

In all experiments and examples presented, the initial dataset consists of one feature t and one target value $P(t)$, with $t \in I = [x_{min}, x_{max}]$.

- For the training of $N_i(t)$ we consider a uniform partition of $[x_{min}, x_{max}]$ with data points $(t_i, P(t_i))$ where $P(t)$ is supposed to be the solution of a first order ordinary differential equation. In total, the method was tested on 4 examples with different characteristics. Each data set is divided into two subsets: training set and test set, consisting of 100 and 50 points respectively.
- In the second phase of the implementation we need to learn the differential equation whose solution is $N_i(t) \approx P(t)$. For this purpose, we trained in I_1, $N_o(t)$ for the model (3.1) and also $N_c(t)$ for the other models. Since we know $N_i(t)$ and thus $\frac{dN_i}{dt}$, we can define the number of data points that will be used in this step. The goal is to minimize the error quantity given by (3.2). After research, in order to avoid overffiting and obtain fast results, for the specific examples that were tried, we decided to use 800 data points for the training set and 1000 or more new points for the testing.
- In the third stage we have the numerical differential equation (the model) in I_1 and we solve it in I_2. For the solution, we can also define the number of data points that can be used for the Runge Kutta method, as well as for the training

of the neural network $N_s(t)$. The number of data points for training is set to 800 and for testing to 2000 new points.

We also repeated the experiments with the same data sets with the addition of noise in $P(t)$ to test how the noise affects the results.

4.3 Hyperparameters and architecture details

Determining the hyperparameters and the proper architecture of a neural network in advance is a difficult task. Based on investigations, all the neural networks trained during the implementation and used in the proposed models, have the same architecture. They are fully connected with 3 layers: an input layer, one hidden layer and an output layer. The hidden layer consists of $H = 10$ neurons and Sigmoid was chosen as the activation function to represent the nonlinear relationships between the feature and the target.

For training, we used the Adam optimizer and Adam's hyperparameter η has the value of 0.01 for all the trained neural networks. In table 4.1, we present the values of the hyperparameters for the models given by the equations (3.1) and (3.5) as a whole.

Table 4.1: Hyperparameters of Neural Networks that were trained in the implementation, using models (3.1) or (3.5)

Neural Network	**hidden layers**	**Neurons**	**training points**	**test points**	**epochs**
$N_i(t)$	1	10	100	50	$[2000 - 4000]$
$N_o(t)$	1	10	800	1500	$[8000 - 10000]$
$N_c(t)$	1	10	800	2000	$[1000 - 2000]$

For the models given by (3.3) and (3.4), we have to train also $N_c(t)$:

- if we train the two networks N_o and N_c together, then we train for $[8000 - 10000]$ epochs.
- if we train the two networks separately, then we train first N_o for $[9000 - 10000]$ epochs and after N_c with $[1000 - 1500]$ epochs.

The number of epochs is defined regarding the examples used in this thesis. The best choice for the number of epochs is based on the problem and on the purpose about how much we can reduce the error without causing overfitting.

4.4 Illustration of the method

This section provides an illustration of the method and how it performs in 4 different cases that we know the ODE and the solution P(t) for $t \in I$ and we want to investigate whether the proposed method can extrapolate. The model that is used for these results is 3.1, because as mentioned it gives the best extrapolation results. For each problem is shown:

- The ODE that gives the solution $P(t)$, which is tested with the numerical ODE(model 3.1). They should be identical or very similar. We can see how the numerical ODE behaves in the domain and also outside of the domain for untrained data.
- A figure containing the comparison between the solution of numerical ODE with neural network, the solution of the numerical ODE with Runge Kutta, the exact function $P(t)$ and the approximation of the initial network $N_i(t)$, $N_i(t) \approx P(t)$. Again, we can see the result in the domain and also for extrapolation points.

It is of major importance to refer that we assume, without loss of generality, that all the examples used follow the theory described in section 2.2.

The choice of intervals I, I_1 and I_2 was crucial. Each example was tested for many combinations of the intervals, larger or smaller. The illustration of the method follows the intervals shown in table 4.2.

Table 4.2: Intervals I, I_1, I_2

Problem	$I = [x_{min}, x_{max}]$	$I_1 = [s_{min}, s_{max}]$	$I_2 = [s_{min}, f_{max}]$
1	$[-4, 4]$	$[0, 4]$	$[0, 6]$
2	$[-4, 4]$	$[1, 4]$	$[1, 6]$
3	$[0, 5]$	$[1, 5]$	$[1, 7]$
4	$[0, 7]$	$[1, 7]$	$[1, 9]$

Problem 1

$$\frac{dP}{dt} = cost$$

with $P(-4) = sin(-4)$ and $t \in [-4, 4]$. The analytical solution is $P(t) = sint$.

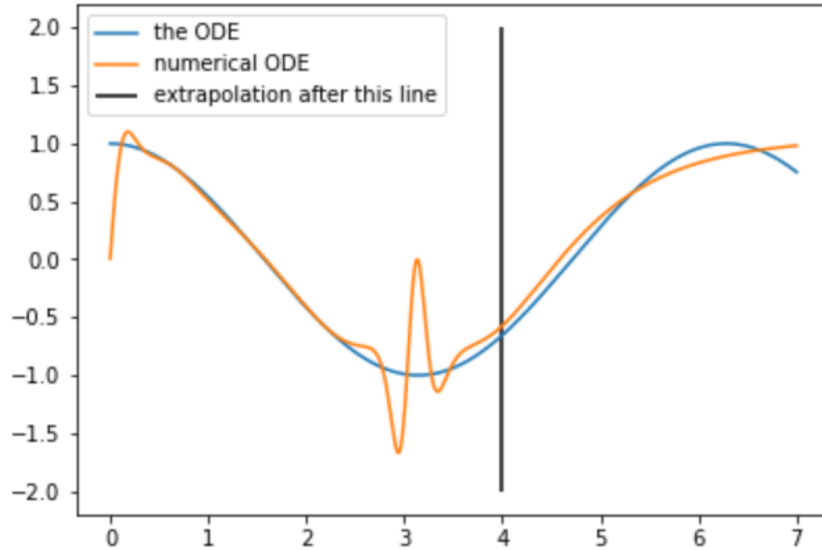

Figure 4.1: Example 1 - ODE

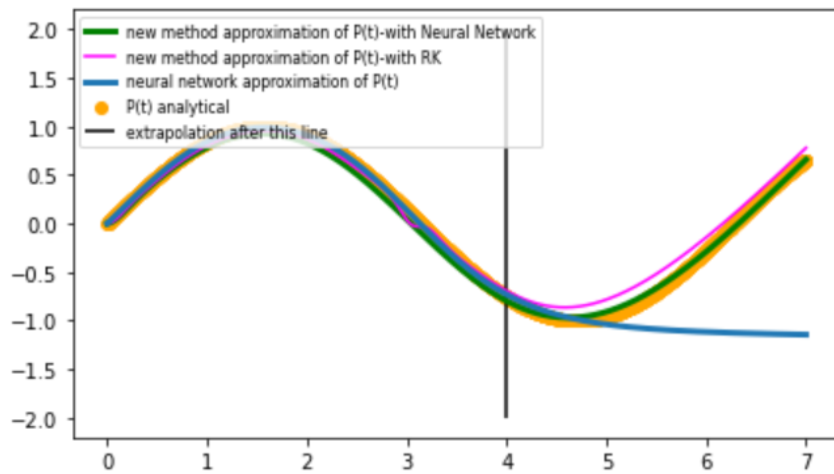

Figure 4.2: Example 1 - P(t) approximation

Problem 2

$$\frac{dP}{dt} = -\frac{1}{5}P + e^{-\frac{t}{5}}cost$$

with $P(-4) = e^{-\frac{4}{5}}sin(-4)$ and $t \in [-4, 4]$. The analytical solution is

$$P(t) = e^{-\frac{t}{5}}sint.$$

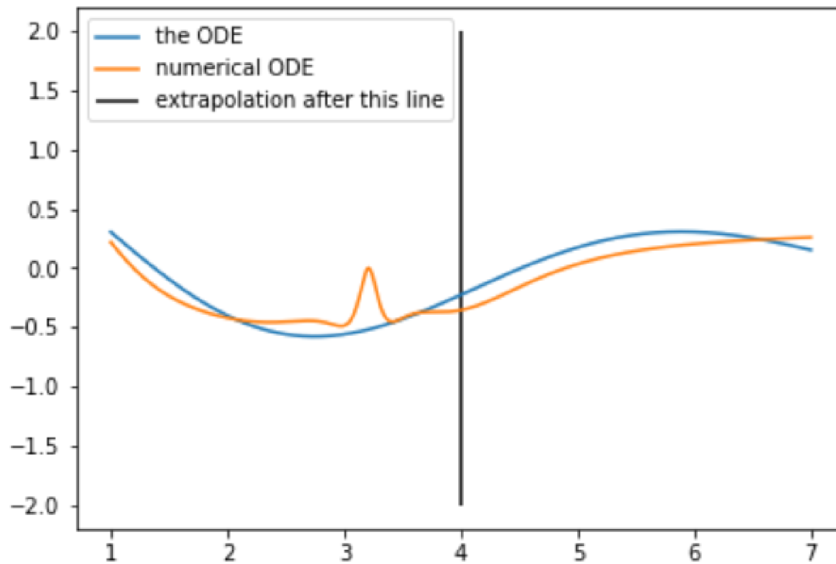

Figure 4.3: Example 2 - ODE

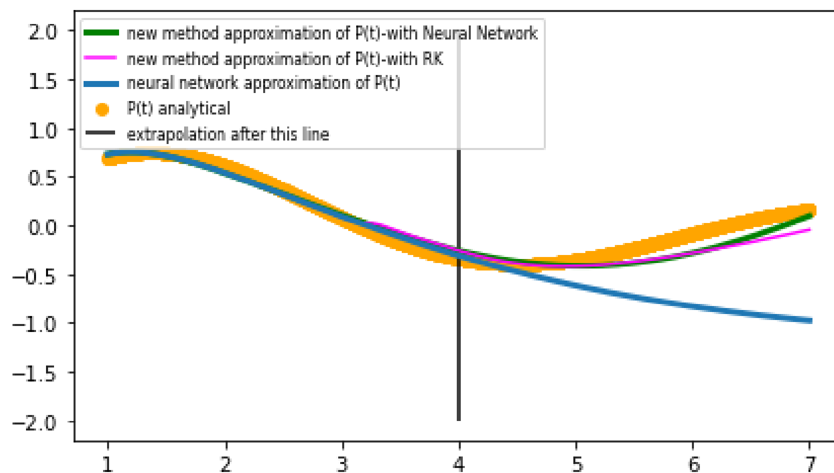

Figure 4.4: Example 2 - P(t) approximation

Problem 3

$$\frac{dP}{dt} = -2P + 12sin(2t)$$

with P(0)=5 and $t \in [0, 5]$. The analytical solution is

$$P(t) = -3cos(2t) + 3sin(2t) + 8e^{-2t}.$$

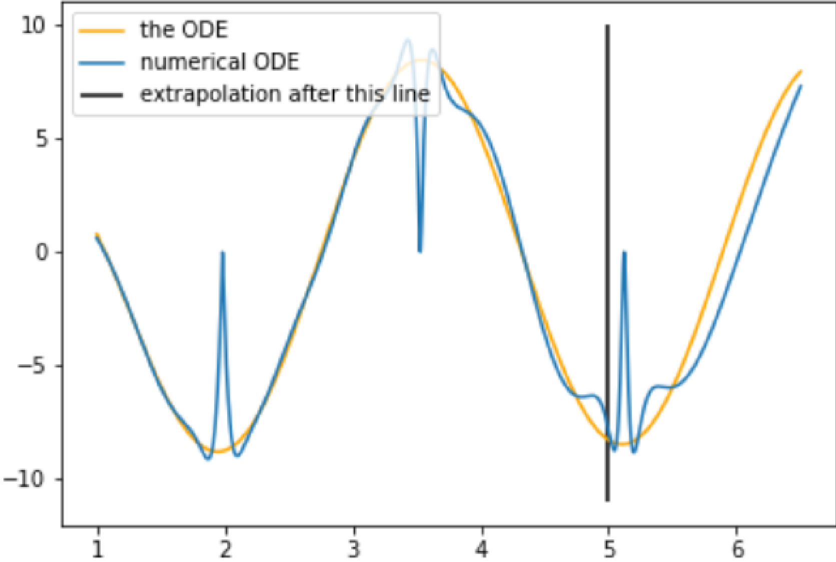

Figure 4.5: Example 3 - ODE

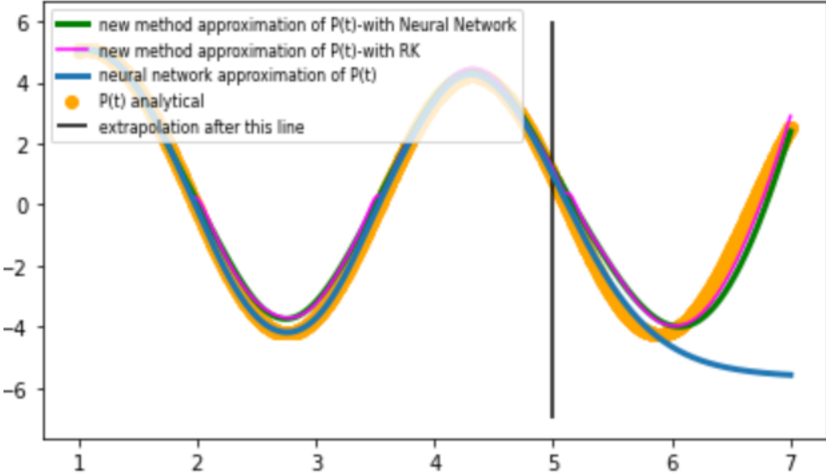

Figure 4.6: Example 3 - P(t) approximation

Problem 4

$$\frac{dP}{dt} + cos(tP) = 2cost$$

with $P(0) = 3$ and $t \in [0, 7]$. The analytical solution is

$$P(t) = 2 + e^{-sint}.$$

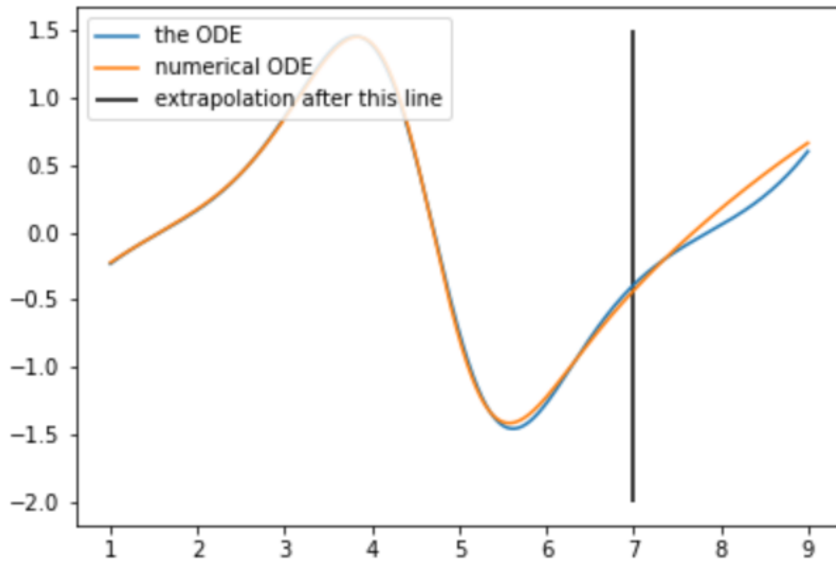

Figure 4.7: Example 4 - ODE

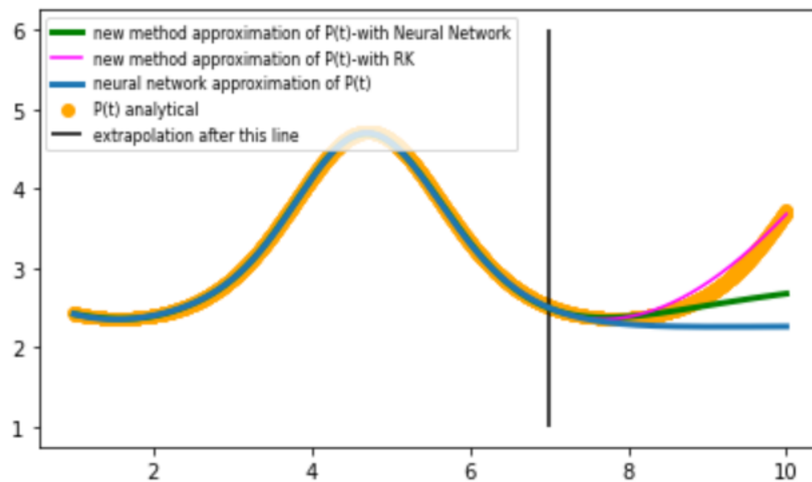

Figure 4.8: Example 4 - P(t) approximation

4.5 Extrapolation results

The first question that should be answered is whether the proposed method gives better results from $N_i(t)$ in terms of extrapolation or not. Of course, we are also interested about interpolation. A second question is how we can measure the results in order to obtain conclusions.
After the experimental phase where all the results were collected, we present here an analysis of the mean extrapolation error to draw inferences. The model used in this section is the model (3.1).

- An interesting thing that would be useful is to define the *range of extrapolation*. In this section we analyse for each example, a range of how far out of the domain is safe to extrapolate.
- We aslo make a quantitative comparison between the two methods used to solve the numerical ODE, which are Runge Kutta and neural network. In this way we test also the ground truth (comparison with P(t)).

In table 4.3 we can observe that the initial approximation of the function P(t) with $N_i(t)$ gives a larger mean extrapolation error for all the examples. This was not unpredictable, as it was expected that $N_i(t)$ would have a low mean error only in the training domain.

Extrapolation error

We define the extrapolation error as:

$$Error_{extr}(t_i) = |prediction(t_i) - P(t_i)| \tag{4.1}$$

with $t_i \in T = [x_{max}, f_{max}]$

- $prediction$ is $N_i(t)$ or the solution of the numerical ODE either with Runge Kutta or neural network for points $t_i \in T$.
- $P(t_i)$ is the value of $t_i \in T$.

Range of extrapolation

We define the range of extrapolation as

$$R = f_{max} - x_{max} \tag{4.2}$$

with $f_{max} > x_{max}$. Figures (4.9) to (4.12) show the mean value of the extrapolation error for different cases of R comparing the Runge Kutta solution and the neural network. The conclusion is that, for these examples, we can extrapolate up to two units of x_{max} for both methods. We see that there is no conclusion which method works better. For example, in problem 1 the results are very close and there is no significant difference because the extrapolation error is very low for both methods. In contrast, the neural network solution for problem 2 yields a larger mean error. In the illustration of the method, we can see that the main feature in each example is that the numerical ODE approaches the exact ODE after x_{max}. If we have the numerical ODE, then we can solve it and get an accurate solution. In all examples, we used $s_{max} = x_{max}$ because we achieved to train $N_i(t)$ with a very low error even in the neighborhood of the boundary x_{max}. However, the result is the same if we choose $x_{max} = s_{max} + \epsilon$ and it might help to kind of examples with approximation problems at the boundary. The training of $N_i(t)$ is very important to be precise especially in the points around x_{max} because it also affects the numerical ODE due to the derivative $\frac{dN}{dt}$. In table 4.3 we can see the statistics of the extrapolation error for $t_i \in T = [x_{max}, f_{max}] = [x_{max}, x_{max} + 2]$.

4.6 Data with noise

In all the examples, the data are synthetic which we obtained from a function P(t), which is solution of known ODE. They do not contain any kind of noise. However, adding noise to the data presents a more realistic case and increases the difficulty. In this section, we present the results of adding random noise from 0.2 to 0.4 for all the used problems. We observe the statistics of the extrapolation error as the noise becomes bigger and in tables 4.4 - 4.7 is shown that the extrapolation error is getting more as the noise is increased. Actually, for noise value of 0.4 the extrapolation error is almost equal to the mean error of $N_i(t)$ when there is no noise, which means that the proposed method fails to extrapolate better than $N_i(t)$. In contrast with noise values of 0.2 and 0.3 where the extrapolation error is lower. This will also be shown

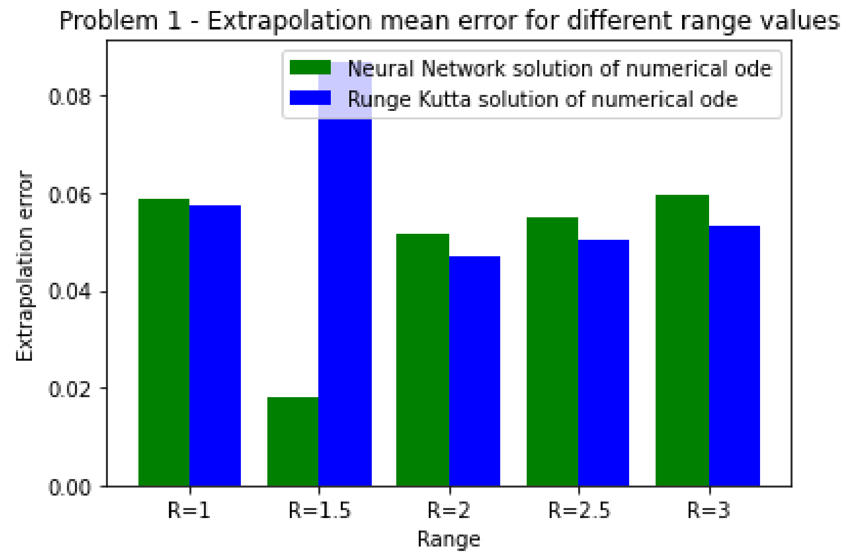

Figure 4.9: Problem 1 - Extrapolation error for various ranges

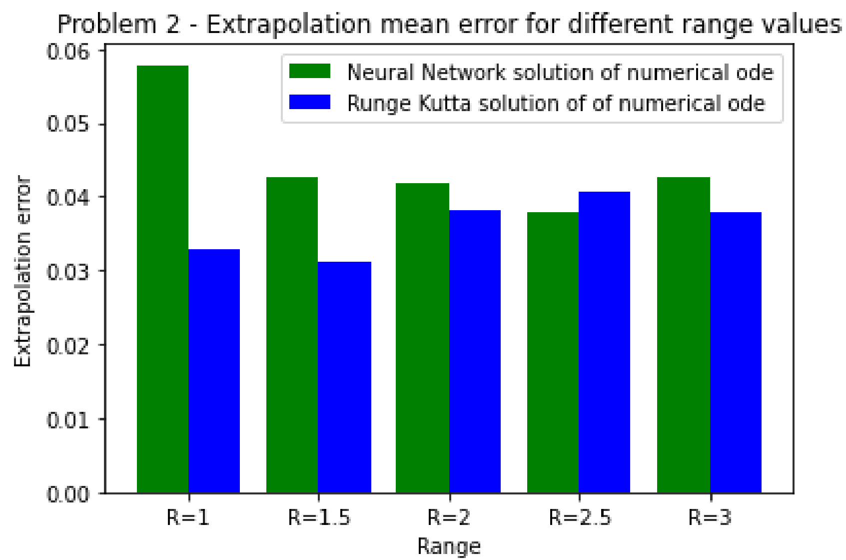

Figure 4.10: Problem 2 - Extrapolation error for various ranges

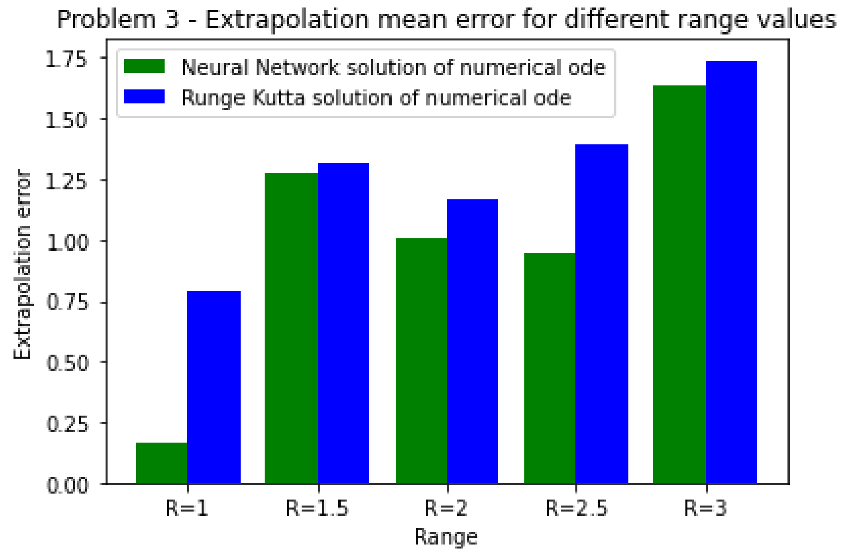

Figure 4.11: Problem 3 - Extrapolation error for various ranges

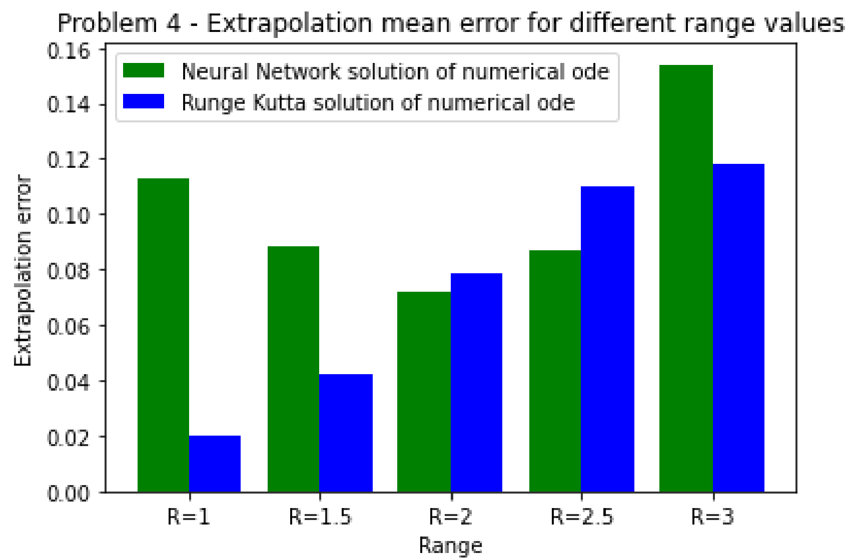

Figure 4.12: Problem 4 - Extrapolation error for various ranges

Table 4.3: Statistics of extrapolation error

for $t_i \in T = [x_{max}, x_{max} + 2]$

Method	**Problem**	**Mean**	**St.Dev.**	**Max**
Runge Kutta	1	0.0469	0.0198	0.0784
	2	0.0380	0.0208	0.0717
	3	1.1671	0.5485	1.9121
	4	0.0788	0.0720	0.2270
Neural Network	1	0.0514	0.0232	0.0894
	2	0.0418	0.0238	0.0830
	3	1.0081	0.6579	1.9044
	4	0.0721	0.0450	0.1347
initial $N_i(t)$	1	1.2123	0.512	2.0433
	2	1.3227	0.3641	1.6302
	3	2.1147	0.7514	3.1215
	4	1.5957	0.1425	1.9403

in section (4.8), where we consider a real dataset, which of course contains noise.

Table 4.4: Example 1 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$

Method	**Noise**	**Mean**	**St.Dev.**	**Max**
Runge Kutta	0	0.0469	0.0198	0.0784
	0.2	0.0779	0.3071	0.5321
	0.3	0.6711	0.2910	1.8156
	0.4	1.2132	0.6201	2.0112
Neural Network	0	0.0514	0.0232	0.0894
	0.2	0.0823	0.2031	0.1209
	0.3	0.8399	0.5188	1.1045
	0.4	1.5103	0.6233	1.9934

Table 4.5: Example 2 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$

Method	**Noise**	**Mean**	**St.Dev.**	**Max**
Runge Kutta	0	0.0380	0.0208	0.0717
	0.2	0.0522	0.3291	1.1201
	0.3	0.8191	0.4920	1.8526
	0.4	1.3212	0.7581	1.8822
Neural Network	0	0.0418	0.0238	0.0830
	0.2	0.0623	0.4322	0.7939
	0.3	0.7989	0.5988	1.4565
	0.4	1.5231	0.6523	1.9883

Table 4.6: Example 3 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$

Method	Noise	Mean	St.Dev.	Max
Runge Kutta	0	1.1671	0.5485	1.9121
	0.2	1.456	0.4181	1.7201
	0.3	1.5881	0.7812	2.556
	0.4	1.862	0.8281	2.722
Neural Network	0	1.0081	0.6579	1.9044
	0.2	1.1221	0.6622	1.3439
	0.3	1.3399	0.7988	1.6655
	0.4	1.5523	0.8933	2.1083

Table 4.7: Example 4 - Statistics of extrapolation error for data with noise for $t_i \in T = [x_{max}, x_{max} + 2]$

Method	Noise	Mean	St.Dev.	Max
Runge Kutta	0	0.0788	0.0720	0.2270
	0.2	0.1223	0.2291	0.4021
	0.3	0.6981	0.5610	0.8156
	0.4	1.0662	0.681	1.722
Neural Network	0	0.0721	0.0450	0.1347
	0.2	0.2321	0.0722	0.4239
	0.3	0.6599	0.1588	1.0543
	0.4	1.0323	1.2323	1.3083

4.7 Results of the alternative ODE models

Throughout chapter 2 we mention the ODE model given by equation (3.1) as having the best extrapolation ability, in contrast with the other alternative models presented. There are many combinations of these models that can yield new ODE models. In this thesis we tested three alternative models, which are :

- $\frac{dN_i}{dt} = N_o(t)N_i(t) + N_c(t)$, model given by (3.3)

- $\frac{dN}{dt} = N_o(t) + N_c(t)$, model given by (3.4)
- $\frac{dN}{dt} = N_o(t)$, model given by (3.5)

Main idea

Recalling again the model given by (3.1): $\frac{dN_i}{dt} = N_o(t)N_i(t)$, we can see that $N_i(t)$ is a part of it. This is because we initially thought that an ODE could contain its own solution. However, as we mentioned earlier, when $N_i(t)$ becomes zero, the derivative $\frac{dN_i}{dt}$ also becomes zero. In these cases, the model given by (3.1) cannot approximate the exact ODE and spikes occur. We can also observe this phenomenon in the illustration of the method, in section 4.4. The main idea behind the alternative models is to avoid these spikes.

Results

The model given by (3.3) can be trained in two ways, either by training N_o and N_c together, or separately. In the first way, the spikes vanish since the derivative $\frac{dN_i}{dt}$ does not become zero when N_i is zero. However the extrapolation results are worse than those of the model 3.1 and the ODE model does not fit exactly the ODE outside the training domain, so the solution of the numerical ODE does not fit with P(t) either. In the second way where they are trained separately, the spikes are still present. This is because we first train $N_o(t)$ to learn the derivative, so we have spikes. Then, we train $N_c(t)$ to improve or correct the numerical ODE. In this way the model does overfitting. As with the model (3.4), we can also train the two networks together or separately. By using this model there are no spikes in the numerical ODE, but this fact does not improve the extrapolation. Moreover, when the two networks are trained separately, the model does overfitting. Finally, model (3.5) also shows no spikes in the numerical ODE, but the extrapolation is worse, compared to model (3.1). Tables 4.8 to 4.10 show the statistical results of the extrapolation error for R=2, for solving the numerical ODE with Runge Kutta and neural network, as well as the ground truth with the initial network $N_i(t)$. We can observe, that compared to the results of the model (3.1), keeping the same range of extrapolation (R=2), the mean error is worse.

Numerical ODE behavior

The performance of each model is based in the property of numerical ODE to be accurate out of the domain, which is something that deeply depends in the training on $N_i(t)$. Since we define how far out of the domain the numerical ODE can be accurate, we can solve it. In conclusion, these alternative models can not produce a numerical ODE that is precise after x_{max}, although the training of $N_i(t)$ is the same.

Table 4.8: Statistics of extrapolation error

for $t_i \in T = [x_{max}, x_{max} + 2]$ with model (3.3):$\frac{dN_i}{dt} = N_o(t)N_i(t) + N_c(t)$

Method	Problem	Mean	St.Dev.	Max
Runge Kutta	1	0.8181	0.3918	1.2784
	2	0.6380	0.4258	1.6756
	3	1.8782	0.8485	2.1234
	4	0.8088	0.8422	1.4270
Neural Network	1	0.8644	0.4532	1.3884
	2	0.5928	0.4238	0.9809
	3	1.7881	0.8599	2.1044
	4	0.8721	0.8450	1.2637

Table 4.9: Statistics of extrapolation error

for $t_i \in T = [x_{max}, x_{max} + 2]$ with model (3.4):$\frac{dN_i}{dt} = N_o(t) + N_c(t)$

Method	Problem	Mean	St.Dev.	Max
Runge Kutta	1	0.5469	0.5198	0.9784
	2	0.6380	0.4208	1.0726
	3	1.5672	0.8485	2.0524
	4	0.7673	0.3740	1.0237
Neural Network	1	0.0514	0.0232	0.0894
	2	0.5446	0.5269	0.8930
	3	1.6784	0.8649	2.9342
	4	0.7521	0.7942	1.1481

Table 4.10: Statistics of extrapolation error

for $t_i \in T = [x_{max}, x_{max} + 2]$ with model (3.5):$\frac{dN_i}{dt} = N_o(t)$

Method	Problem	Mean	St.Dev.	Max
Runge Kutta	1	0.6469	0.7198	1.0593
	2	0.6838	0.5901	1.0875
	3	1.8570	0.9460	2.6801
	4	1.2488	0.5781	1.72270
Neural Network	1	0.6734	0.5232	0.9098
	2	0.6741	0.6578	1.1850
	3	1.9857	0.8579	2.8334
	4	1.2681	0.5908	1.9793

4.8 Experiment with unknown ODE

For testing the method we use a time series with real data $(t_i, P(t_i))$, therefore we do not know the ordinary differential equation that is hidden behind P(t)(if any exists). We suppose that the time series is a solution of an ODE, which is approximated by model (3.1).

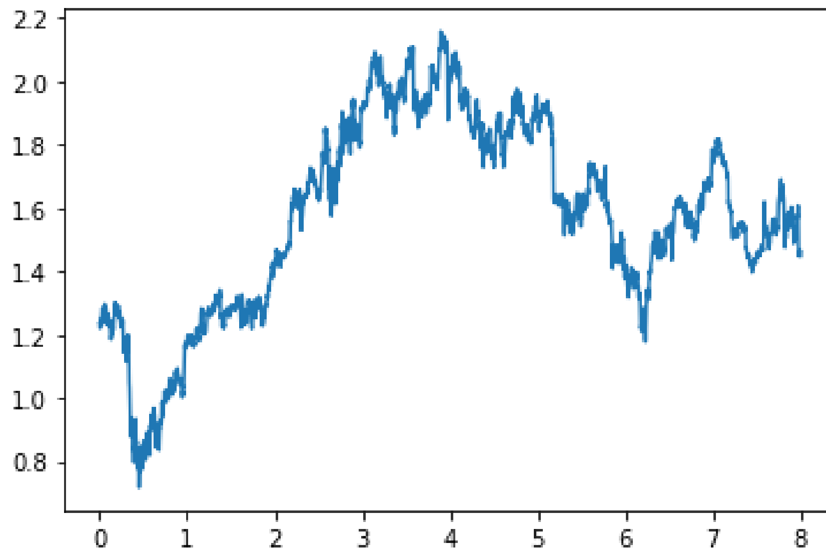

Figure 4.13: The real time series where we do not which ODE it satisfies

Results of the method with unknown ODE

The process for identifying the extrapolation result includes the following steps:

- train the initial neural network $N_i(t)$ to learn P(t) in $I = [x_{min}, x_{max}]$.
- train $N_o(t)$ in order to obtain the numerical ODE, for different intervals I_1.
- solve the numerical ODE in I_2, for R=1.5.

Figures (4.14) to (4.16) illustrate the performance of the method, for different cases of x_{max}. After the black horizontal line, there are extrapolation data, that were not used for training. We can notice that the proposed method can approximate P(t) after x_{max} up to a maximum of 1.5 units. This happens due to the fact that we have real data that contain noise. Unfortunately, we do not have the exact ODE, to test the behavior of the numerical ODE after x_{max}. However, it turns out that the numerical ODE approximation is effective for 0.5 to 1 unit, so the solution of the numerical ODE can approximate P(t). The training of $N_i(t)$ again played an important role, as it must be as precise as possible, without overfitting. The interval values for each case are listed in the table 4.11. For the training of $N_i(t)$ we had to increase the number of epochs to 13.000, while keeping the same hyperparameters that we mentioned earlier. Table 4.12 shows the statisticals for each case, with the extrapolation range R = 1.5. It is important to say that the results are not always the same when we run the experiments.This happens because of the deep dependence on $N_i(t)$ training but also on $N_o(t)$ training.

Table 4.11: Intervals I, I_1, I_2 for the testing time series example

Case	$I = [x_{min}, x_{max}]$	$I_1 = [s_{min}, s_{max}]$	$I_2 = [s_{min}, f_{max}]$
1	$[0, 3]$	$[1, 3]$	$[1, 4]$
2	$[0, 2]$	$[1, 2]$	$[1, 3]$
3	$[1, 3.5]$	$[2, 3.5]$	$[2, 4.5]$

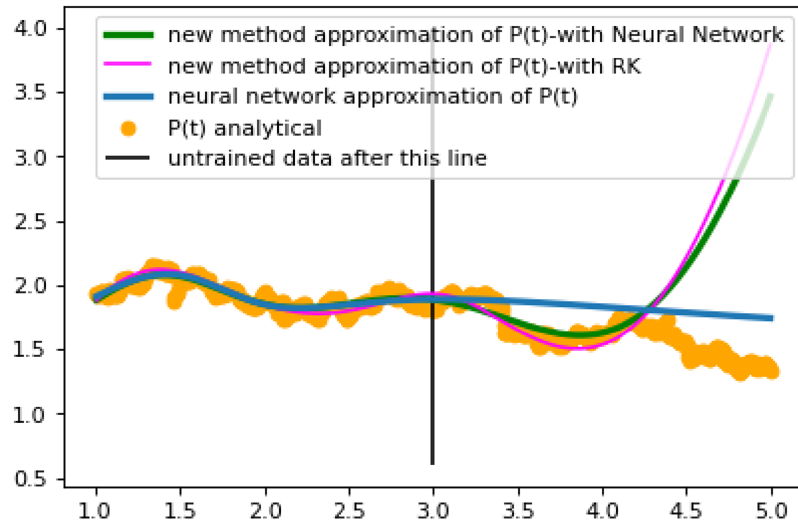

Figure 4.14: Case 1

Illustration of case 1

Illustration of case 2

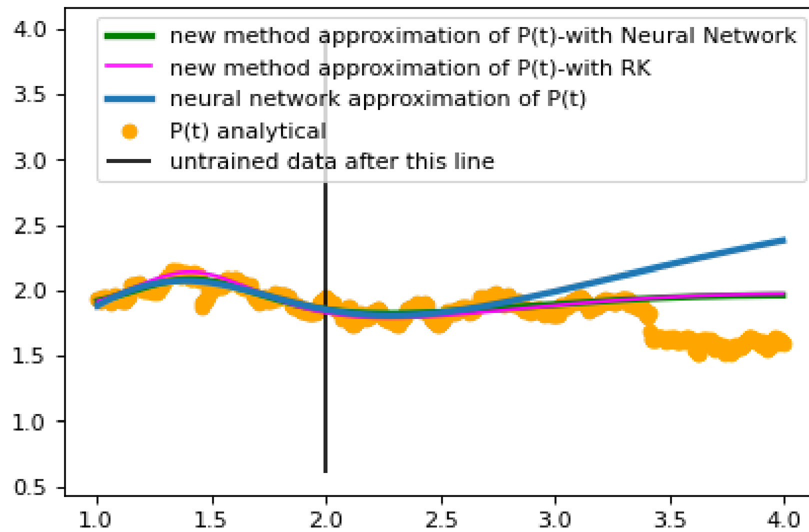

Figure 4.15: Case 2

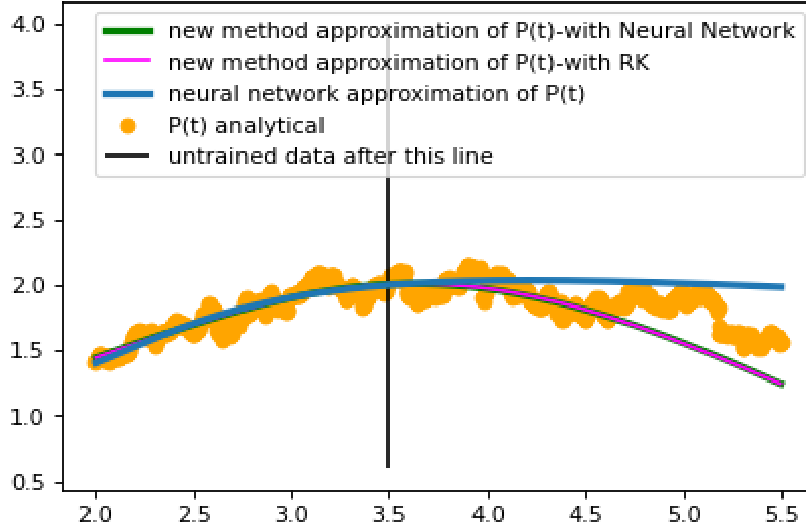

Figure 4.16: Case 3

Table 4.12: Statistics of the extrapolation error of the testing time series, for $t_i \in T = [x_{max}, x_{max} + 1.5]$, R=1.5, with model (3.1)

Method	Case	Mean	St.Dev.	Max
Runge Kutta	1	0.0899	0.0828	0.8989
	2	0.0780	0.0628	0.5217
	3	0.0771	0.05365	0.2021
Neural Network	1	0.0918	0.0832	0.8994
	2	0.0782	0.0631	0.5221
	3	0.0769	0.5367	0.2021
initial $N_i(t)$	1	0.3133	0.0512	0.5532
	2	0.1227	0.0641	0.3321
	3	0.2137	0.0514	0.2151

Illustration of case 3

We can observe in figures 4.14 to 4.16 as well as in table 4.12 that the proposed method has a significant better extrapolation ability than the initial neural network in the three cases. The Runge Kutta method and the Neural Network solution have similar performance and it is shown that there is no important difference between them, regarding the extrapolation error.

Chapter 5

Epilogue

5.1 Conclusion
5.2 Future work

5.1 Conclusion

In this thesis we studied a method inspired by neural networks and numerical methods for solving differential equations, for function extrapolation. The study of the background theory and the design of the proposed method led to the implementation of several variants and experiments. The proposed method attempts to solve the problem of extrapolation in a time series problem where we have a set of data $(t_i, P(t_i))$, with $t_i \in$ I. We initially start by training a neural network in I to approximate P(t), and then we discover the first-order ODE, which is satisfied by P(t), using a numerical ODE model. We then solve the numerical ODE using a numerical method. Specifically, we use Runge Kutta and a neural network to solve the numerical ODE. The solution is expected to yield efficient extrapolation performance. Experimental results for the examples used show that the proposed method achieves better extrapolation than an initial feedforward neural network for a given data set. Runge Kutta and neural network method do not have significant differences in solving the numerical ODE. However, we must say that the extrapolation results are influenced by the type of the problem and the training phase.

5.2 Future work

The proposed method can theoretically be adapted to any time series problem with two variables and can achieve reasonable extrapolation. However, extrapolation in general carries a large risk due to uncertainty and randomness. There is a high dependence on the training process and parameter setting. Thus, there is still much research to be done in order to obtain even clearer and more reliable results.

- It would be interesting to try more models as ODE models, combine neural networks and use other alternatives.
- Another thing to try is to explore more hyperparameter setups in order to fine-tune the models.
- We can also try to re-train the model using the extrapolation results from the previous intervals and investigate how far we can extrapolate using this methodology.
- As mentioned earlier, it is important for the initial network to be accurate at the boundary x_{max}. It would be intriguing to make the proposed method more systematic and automatically check in the edge whether the interval needs to be shortened a bit or not, as well as how this affects the extrapolation.

Bibliography

[1] L. Siklóssy, "On the evolution of artificial intelligence," *Information Sciences*, vol. 2, no. 4, pp. 369–377, 1970. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020025570900344

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[4] I. Lagaris, A. Likas, and D. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998.

[5] C. Michoski, M. Milosavljević, T. Oliver, and D. R. Hatch, "Solving differential equations using deep neural networks," *Neurocomputing*, vol. 399, pp. 193–212, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220301909

[6] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[7] B. Lusch, J. N. Kutz, and S. L. Brunton, "Deep learning for universal linear embeddings of nonlinear dynamics," *Nature communications*, vol. 9, no. 1, pp. 1–10, 2018.

[8] L. Lu, P. Jin, and G. E. Karniadakis, "Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *arXiv preprint arXiv:1910.03193*, 2019.

[9] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999118307125

[10] G. Karniadakis, Y. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," pp. 1–19, 05 2021.

[11] G. Kissas, Y. Yang, E. Hwuang, W. R. Witschey, J. A. Detre, and P. Perdikaris, "Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4d flow mri data using physics-informed neural networks," *Computer Methods in Applied Mechanics and Engineering*, vol. 358, p. 112623, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782519305055

[12] S. Christin, E. Hervet, and N. Lecomte, "Applications for deep learning in ecology," *bioRxiv*, 2018. [Online]. Available: https://www.biorxiv.org/content/early/2018/05/30/334854

[13] E. B. Ghadami A, "Data-driven prediction in dynamical systems: recent developments," 2022. [Online]. Available: https://royalsocietypublishing.org/doi/epdf/10.1098/rsta.2021.0213

[14] P. Sodhi, N. Awasthi, and V. Sharma, "Introduction to machine learning and its basic application in python," *SSRN Electronic Journal*, 01 2019.

[15] J. Brownlee, *Master Machine Learning Algorithms: Discover How They Work and Implement Them From Scratch*, 2016. [Online]. Available: https://books.google.gr/books?id=PdZBnQAACAAJ

[16] C. M. Bishop, *Pattern Recognition and Machine Learning*, 2006.

[17] A. Hurson and S. Wu, *AI and Cloud Computing*, ser. ISSN, 2021. [Online]. Available: https://books.google.gr/books?id=4WIFEAAAQBAJ

[18] "Supervised learning (2022, august 31). wikipedia. retrieved september 29, 2022." [Online]. Available: https://en.wikipedia.org/wiki/Supervised_learning

[19] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608014002135

[20] "Backpropagation." [Online]. Available: https://en.wikipedia.org/wiki/Backpropagation

[21] �. Γιαννούλης, "Γιαννούλης, Ι. (2015). Διανυσματική Ανάλυση [Προπτυχιακό εγχειρίδιο]. Κάλλιπος, Ανοικτές Ακαδημαϊκές Εκδόσεις. http://hdl.handle.net/11419/1201." [Online]. Available: http://hdl.handle.net/11419/1201

[22] G. D. Akrivis, "Numerical methods for initial value problems, georgios d. akrivis." [Online]. Available: http://www.bcamath.org/documentos_public/courses/AKRIVIS20121119-23.pdf

[23] R. Burden, J. Faires, and A. Burden, *Numerical Analysis*. Cengage Learning, 2015. [Online]. Available: https://books.google.gr/books?id=9DV-BAAAQBAJ

Short Biography

Christina Seventikidou was born in Ptolemaida, Greece, in 1995. In 2013 she enrolled in the undergraduate program of Mathematics of the University of Ioannina and earned her Degree in 2018. In 2020 she enrolled in the Graduate Program of the Department of Computer Science and Engineering of University of Ioannina, and is persuiting a MSc Degree entitled ”Data and Computer Systems Engineering”.