University of Potsdam
Institute of Mathematics
Master of Science Mathematics

Master Thesis

Parameter estimation for differential equations:

A comparison between the use of deep learning and data assimilation

Author: Janik Raue
Matriculation number: 780899
Submission date: September 14, 2022
First Supervisor: Prof. Dr. Melina Freitag
Second Supervisor: Dr. Thomas Mach

Abstract

To describe the laws of nature, differential equations are usually used. Often differential equations contain parameters for modelling different processes of the same kind. These are usually not known exactly and can be estimated with the help of observations. There are different possibilities for this, two of which are presented and compared in this master thesis. On the one hand, a deep learning based approach using physics-informed neural networks and on the other hand, a conventional data assimilation method using the Kalman filter.
The comparison focusses on the required computational effort and the accuracy of the estimates with respect to the number of data and the amount of noise. For this purpose, synthetic data were generated for the Lorenz system using the Runge-Kutta method. To simulate possible deviations, a normally distributed noise term was added to each time point. The Kalman filter was able to estimate the parameters well with only a few observations. Moreover, it takes less time to compute than physics-informed neural networks need. These, on the other hand, were able to provide more precise estimates if there was enough data for the differential equation. They were also not as sensitive to noisy data as the Kalman filter.
Finally, the comparison was extended to real data from the Robert Koch Institute which provides data on the infection incidence of the coronavirus disease 2019 epidemic in Germany. To model the process, a system of differential equations can be used, for example the SIR model. Only the Kalman filter was able to estimate plausible parameters.

Zusammenfassung

Um Gesetzmäßigkeiten der Natur zu beschreiben, werden normalerweise Differentialgleichungen verwendet. Zum Modellieren von verschiedenen Vorgänge der gleichen Art sind in Differentialgleichungen Parameter enthalten. Diese sind häufig nicht exakt bekannt und müssen mit Hilfe von Beobachtungen geschätzt werden. Dafür gibt es verschiedene Möglichkeiten, von denen zwei in dieser Masterarbeit vorgestellt und verglichen werden. Zum einen ein Deep Learning basierter Ansatz mittels Physik-informierter Neuronaler Netze und zum anderen mit dem Kalman Filter eine herkömmliche Dataassimilationsmethode.
Beim Vergleich wurde auf den benötigten Rechenaufwand und die Genauigkeit der Schätzungen in Abhängigkeit von Datenanzahl, sowie Stärke des Rauschens, geachtet. Dafür wurden für das Lorenz System synthetische Daten via Runge-Kutta-Verfahren erzeugt. Um mögliche Abweichungen zu simulieren wurde zu jeden Zeitpunkt ein normalverteilter Rauschterm addiert. Dabei konnte der Kalman Filter mit wenigen Beobachtungen die Parameter gut schätzen. Außerdem benötigt er zum Berechnen weniger Zeit, als die Physik-informierten Neuronalen Netze. Diese konnten dagegen präzisere Schätzungen liefern, wenn für die Differentialgleichung ausreichend viele Daten vorhanden waren. Dabei reagierten sie auch nicht so empfindlich auf verrauschte Daten, wie der Kalman Filter.
Um den Vergleich auf echte Daten auszuweiten, konnten vom Robert Koch-Institut Daten zum Infektionsgeschehen der COVID-19 Epidemie in Deutschland erhalten werden. Um den Verlauf zu modellieren, kann ein System von Differentialgleichungen verwendet werden, zum Beispiel das SIR-Model. Dabei konnte nur der Kalman Filter plausible Parameter schätzen.

Erklärung

Hiermit versichere, dass ich Janik Raue die Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln (z.B. Nachschlagewerke oder Internet) angefertigt habe. Alle Stellen der Arbeit, die ich aus diesen Quellen und Hilfsmitteln dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht und im Literaturverzeichnis aufgeführt. Die „Richtlinie zur Sicherung guter wissenschaftlicher Praxis für Studierende an der Universität Potsdam (Plagiatsrichtlinie) - Vom 20. Oktober 2010“, im Internet unter `https://www.uni-potsdam.de/am-up/2011/ambek-2011-01-037-039.pdf`, ist mir bekannt.

.....	.....
Ort, Datum	Unterschrift

Contents

Abbreviations

AI artificial intelligence

DL deep learning

ML machine learning

NN neural network

FNN feedforward neural network

ReLU rectified linear unit

PINN physics-informed neural network

AD automatic differentiation

SGD stochastic gradient descent

ADAM adaptive moment estimation

BFGS Broyden-Fletcher-Goldfarb-Shanno

SIR susceptible-infected-removed

COVID-19 coronavirus disease 2019

1 Introduction

Differential equations are frequently used in various fields, for example in physics, biology, chemistry, astronomy. Differential equations or systems of differential equations are mathematical models that can formally describe the laws of nature. With the help of these models it is possible to predict future states (state estimation). In order to do this as accurately as possible, it is necessary to have full knowledge about the differential equation. Usually they contain parameters that vary between problems of the same type, a very precise estimation of the parameters is essential (parameter estimation).
Often it is impossible to find the solution of a system of differential equations analytically, therefore the use of numerical methods is necessary. With these methods and small errors of the parameters, minor deviations at earlier states can be potentiated and lead to significantly larger errors at later states.
Nowadays, it is becoming increasingly easy to collect large amounts of data on a variety of processes, including those that can be modelled using differential equations. There is a strong link between big data and artificial intelligence (AI), or more precisely deep learning (DL). DL has recently made an impact in many different research areas, including state and parameter estimation of differential equations. Most common deep learning algorithms such as neural networks (NNs) only learn with data to approximate the underlying function. A much more advanced approach are physics-informed neural networks (PINNs), which are NNs respecting any given laws of physics described by differential equations [Raissi et al., 2019].
Another approach is to look at both observations and the underlying mathematical model and build a new more accurate model from them. This concept is called data assimilation [Law et al., 2015, Freitag, 2020].
A well-known and often used data assimilation algorithm is the Kalman filter, named after its creator Rudolf E. Kalman [Kalman, 1960], where the state estimation is recursively improved by minimising the mean-square error of the observed current state and the model prediction. This is rather conventional and has been used in various problems, such as the Apollo moon mission and in numerical weather prediction [Grewal and Andrews, 2001].
Section 2 introduces differential equations, explains what an initial value problems is, when there exists a solution and how to find it numerically, for example using the Runge-Kutta method. Furthermore, this part describes what an inverse problem is in general and in the context of differential equations. The formal aspects are illustrated by an example, the Lorenz system.
DL, especially feedforward neural networks (FNNs) are explained in section 3. The general structure of such networks is presented and important components such as activation, cost and loss functions are described. The methods with which NNs can learn from data and make predictions are also introduced. Training algorithms, such as gradient-based optimisation, adaptive moment estimation (ADAM) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, but also techniques for efficient differentiation, such as automatic differentiation, are to be mentioned here. In addition, in this section PINNs are described in general, as well as their ability to estimate states and parameters of the

system of differential equations. To demonstrate the functionality of PINNs, this method was applied to the Lorenz system.
The next section (4), is about data assimilation, with a particular focus on the Kalman filter and how it works for state and parameter estimation in differential equations. Once again, the Lorenz system was used to illustrate the theoretical foundations of the Kalman filter with an example.
Finally, in section 5, a comparison is made between DL algorithms, such as PINNs, and the Kalman filter, a data assimilation method. On the one hand, the theoretical aspect is examined with the help of an error analysis, and on the other hand, the practical aspect is evaluated with regard to computational effort and accuracy using the example of the Lorenz system. Furthermore both methods are tested on real-world data, more precise on data from the current coronavirus disease 2019 (COVID-19) pandemic For this purpose the susceptible-infected-removed (SIR) model, originally developed by [Kermack and McKendrick, 1927], is introduced. This is a system of differential equations that models a pandemic. It will be shown whether the Kalman filter is still best suited for parameter estimation or whether it is outperformed by PINNs.

2 Differential equations

Differential equations can be used to explain and predict new facts for about anything that changes continuously according to [Farlow, 2006]. There are a variety of applications. For example, it is possible to compute the movement of objects, to represent chemical reactions or to model the development of a biological population.
To introduce differential equations and systems of differential equations mathematically we go along with [Strehmel et al., 2012]. For our purpose it is sufficient to use first order differential equation. A system of $n \in \mathbb{N}$ differential equations of order 1 in explicit form is given by

$$\begin{aligned} y_1'(x) &= F_1(x, y_1(x), \ldots, y_n(x)), \\ y_2'(x) &= F_2(x, y_1(x), \ldots, y_n(x)), \\ &\ldots, \\ y_n'(x) &= F_n(x, y_1(x), \ldots, y_n(x)), \end{aligned} \tag{2.1}$$

where $y_i : I \to \mathbb{R}$ with $y_i' = \frac{dy_i}{dx}$ is the first derivative of function y_i with respect to $x \in I \subset \mathbb{R}$ and $F_i : I \times \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, n$. Often differential equations contain parameters that can vary between different cases, but remain constant for each case.
Let $y(x) = \begin{pmatrix} y_1(x) \\ \ldots \\ y_n(x) \end{pmatrix}$, $F(x, y) = \begin{pmatrix} F_1(x, y_1(x), \ldots, y_n(x)) \\ \ldots \\ F_n(x, y_1(x), \ldots, y_n(x)) \end{pmatrix}$ then we can rewrite equation 2.1 in a compact manner

$$y' = F(x, y), \tag{2.2}$$

where $F : I \times \mathbb{R}^n \to \mathbb{R}^n$.
Often we talk about time-dependent differential equations, where t, interpreted as time, is written instead of x and y' is replaced by $\dot{y} = \frac{dy}{dt}$.
If the differential equation has a different form than the given one in equation 2.2 it is called implicit.
The solution of a differential equation, is a vector function y with the following properties

- y is continuously differentiable on I,
- $(x, y(x)) \in I \times \mathbb{R}^n$,
- $y' = F(x, y) \qquad$ for all $x \in I$.

Usually we are not interested in the set of solutions, but in the solution for a given initial value, the so-called initial value problem.

2.1 Initial value problem

In an initial value problem, in addition to a system of differential equations, as in 2.2, informations about the initial conditions are also known.

We aim to find a solution for a given initial condition $y(x_0) = y_0 \in \mathbb{R}^n$ with fixed value $x_0 \in I$. Thus, an initial value problem is then determined by the following equations

$$\begin{aligned} y' &= F(x, y), \\ y(x_0) &= y_0. \end{aligned} \tag{2.3}$$

For such problems, it is often impossible to determine a solution and state it in closed form. So it is very important to know when a solution exists and under which circumstances it is unique.
A theorem for existence and uniqueness of such a solution is theorem 2.1 from Émile Picard and Ernst Lindelöf as given in [Strehmel et al., 2012]. First, to define is Lipschitz continuity, which is an important property for the theorem.

Definition (Lipschitz continuity):
A function $F(x, y)$ is Lipschitz continuous with respect to y on a closed domain

$$G := \{(x, y) : |x - x_0| \leq a, \|y - y_0\| \leq b\}, \quad a, b > 0,$$

if there exists a constant $L > 0$, such that

$$\|F(x, u) - F(x, v)\| \leq L\|u - v\| \quad \textit{for all } (x, u), (x, v) \in G. \tag{2.4}$$

Now the Picard-Lindelöf theorem can be stated.

Theorem 2.1 (Picard-Lindelöf):
Let the function $F(x, y)$ be continuous and Lipschitz continuous on

$$S := \{(x, y) : x_0 \leq x \leq x_e, y \in \mathbb{R}^n\}$$

then for every $y_0 \in \mathbb{R}^n$ the initial value problem 2.3 has a unique continuously differentiable solution $y(x)$ in the interval $[x_0, x_e]$.

Often it is very difficult or impossible to determine the exact solution to the initial value problem. Therefore the initial value problem (2.3) has to be solved numerically. We consider one particular type of solution methods next.

2.2 Runge-Kutta methods

In numerics, methods have been developed to approximate the solution of an initial value problem. Generally, a distinction is made between two classes, single-step and multi-step methods. Both methods are based on a discretisation, a decomposition of the integration interval, while single-step methods compute approximations for each time individually, multi-step methods also use previous approximation to compute the new one [Strehmel et al., 2012].
Only single-step methods are introduced here, more precisely their important class, the Runge-Kutta methods, as these are sufficient for the purpose of this thesis.

In this section we assume, that an Initial Value Problem like in equation 2.3 with is given. Furthermore the conditions of theorem 2.1 are assumed to be satisfied. In other words, $F(x, y)$ is continuous and Lipschitz continuous on S with respect to y. Thus, it is know that a unique solution exists, which can now be approximated using a Runge-Kutta method. According to [Strehmel et al., 2012] the Runge-Kutta method of order $s \in \mathbb{N}$ is given by :

$$\begin{aligned} r_1 &= h \cdot F(x_k, y(x_k)), \\ r_2 &= h \cdot F(x_k + c_2 h, y(x_k) + a_{21} r_1), \\ r_3 &= h \cdot F(x_k + c_3 h, y(x_k) + a_{31} r_1 + a_{32} r_2), \\ &\vdots, \\ r_s &= h \cdot F(x_k + c_s h, y(x_k) + \sum_{j=1}^{s-1} a_{s,j} r_j), \\ y(x_{k+1}) &= y(x_k) + \sum_{j=1}^{s} b_j r_j. \end{aligned} \tag{2.5}$$

Here h denotes the step size and is given by $h := x_{k+1} - x_k$. The coefficients are summarised in a so called Butcher-Schema

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ c_4 & a_{41} & a_{42} & a_{43} & & \\ \vdots & \vdots & \vdots & & \ddots & \\ c_s & a_{s,1} & a_{s,2} & a_{s,3} & \dots & a_{s,s-1} \\ \hline & b_1 & b_2 & b_3 & \dots & b_{s-1} & b_s \end{array} . \tag{2.6}$$

In the further course of the work, two Runge-Kutta methods are used. First, the simplest variant, the explicit Euler method with order $s = 1$. In addition, the classical Runge-Kutta method with order $s = 4$. The corresponding Butcher schemes are described in [Hairer et al., 1993]:

- explicit Euler:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \quad (y' = F(x, y) \text{and initial condition} y(x_0) = y_0), \tag{2.7}$$

- classical Runge-Kutta:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} . \tag{2.8}$$

The Runge Kutta methods introduced here are explicit methods, which are sufficient to consider.

2.3 Inverse problems

In the context of this thesis, the main interest is not in solving an initial value problem, but in finding the parameters of the differential equation that best describe the given realisations. This type of problem is known as an inverse problem.
In [Kirsch, 2021] it is said that two problems are inverse to each other, if full or partial knowledge of one problem is necessary for the formulation of the other. This definition does not contain any information about which problem is the direct and which is the inverse. Often for the direct problem the definition of well-posedness by [Hadamard, 1923] holds, i.e. it fulfils the properties of uniqueness, existence, and stability of the solution. The problem is said to be ill-posed, if one of the properties fails to hold. That is normally the case for inverse problems.
A formal example for a direct and its corresponding inverse problem is given in [Freitag and Potthast, 2013]. Suppose an operator which describes the relation between a source and the effect is given. The source can be very different, for example the model parameters or the initial condition can be meant. Whereas observations or realisations, i.e. data, can be interpreted as effect. Now the direct problem is determining the effect while having knowledge about the source. Accordingly, the inverse problem is to draw conclusions about the source when the effect is known.
Now we consider the initial value problem (2.3) with additional model parameter λ, i.e. $y' = F(x, y; \lambda)$. The direct problem is finding a solution y while the model F and its parameters λ are already determined. The corresponding inverse problem is to identify the optimal parameters λ of the differential equation for given observations y^*.
In practice, the true underlying model F is unknown and often it is not possible to know the exact values y of a system. This can have many reasons, usually the measurements are inaccurate. To simulate imprecise values, a noise term z_k can be added in each iteration k, which leads to the observations

$$y^*(t_k) = y(t_k) + z_k. \tag{2.9}$$

The noise term can be chosen arbitrarily, usually as a centred normal distribution with variance $\sigma_z^2 \in (0, \infty)$, i.e. $z_k \sim \mathcal{N}(0, \sigma_z^2)$.
To make accurate predictions it is useful to combine model and noisy data. The goal is to find the best possible model. This is an inverse problem. There are several ways to

solve it, including PINNs and the Kalman filter. These methods will be described later in this thesis.
Next, the Lorenz system is introduced. For this system of differential equations, data are generated using the Runge Kutta method. These simulations will later be used to test the two different methods.

2.4 Example: Lorenz system

The Lorenz system, introduced by [Lorenz, 1963] to model the conditions in the atmosphere. This is an example of a system of time-dependent differential equations of order 1 with $n = 3$ different functions y_1, y_2, y_3 and given by

$$\begin{aligned} \dot{y}_1 &= \rho(y_2 - y_1), \\ \dot{y}_2 &= y_1(\sigma - y_3) - y_2, \\ \dot{y}_2 &= y_1 y_2 - \beta y_3. \end{aligned} \tag{2.10}$$

When an initial condition $y(0) = (y_1(0), y_2(0), y_2(0))^T$ is given and the parameters $\rho, \sigma, \beta \in \mathbb{R}$ are known it is possible to formulate an initial value problem.
Since the Picard-Lindelöf theorem 2.1 is fulfilled, it is possible to apply the Runge-Kutta method to forecast the system of differential equation and obtain datapoints $y(t_i)$ for time $t_i \in [0, T), i = 1, \ldots, l,$ with $l \in \mathbb{N}$.
The development of the variable $y = (y_1, y_2, y_3)^T$ over the time t of the Lorenz system with parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial condition $y(0) = ((-5, 10, 25)^T$ computed with the Runge-Kutta method is shown in figure 1.
In the Lorenz system, it is possible that small changes in the parameters and / or initial conditions can lead to strong deviations in the further process. Such a problem is visualised in figure 2, where the so-called butterfly effect can be seen in the illustration on the right.
Because of such characteristics, it is of immense importance in inverse problems to determine the parameters as exactly as possible. For such problems the system of differential equation is known but not the parameters.
In figure 3 the Lorenz system with noisy observations y^* obtained with equation 2.9 is given. The goal is to estimate the parameters $\lambda = (\rho, \sigma, \beta)$, therefore several techniques has been discovered, two of them will be presented in this thesis, the so called PINNs a technique based on DL and a data assimilation method, the Kalman filter.

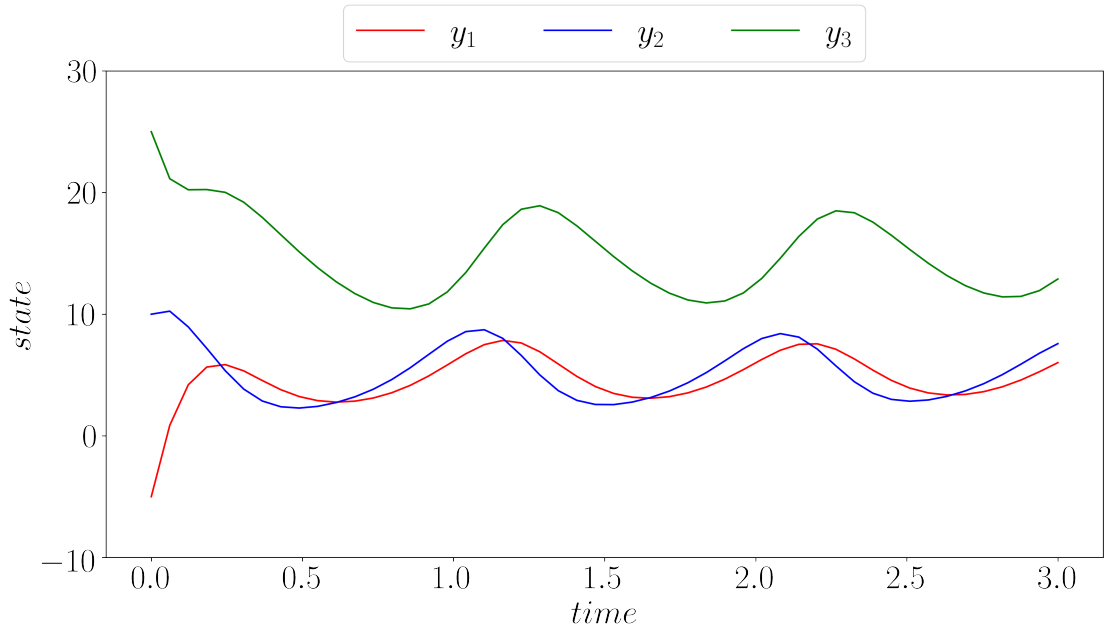

Figure 1: A two-dimensional visualisation of the Lorenz system, where the time course of the individual states y_1, y_2 and y_3 is shown. The simulation of the Lorenz system was generated using the classical Runge-Kutta methods with the parameters $\rho = 8$, $\sigma = 16$ and $\beta = 2$ and the initial condition $y(0) = (-5, 10, 25)^T$.

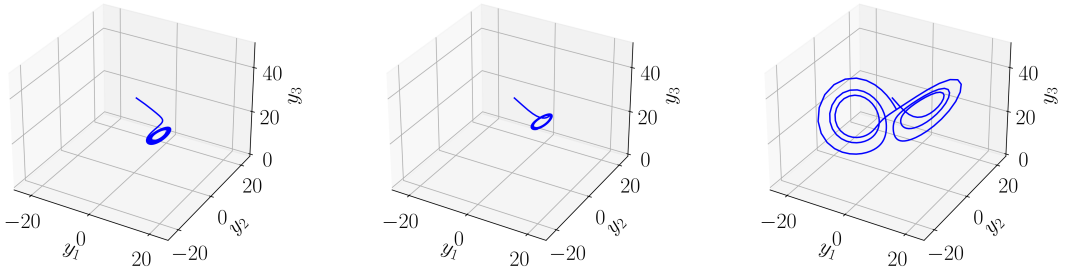

Figure 2: Here the differences between individual parameter constellations are visible in a three-dimensional representation. All simulations shown here were created using the classical Runge Kutta method with the initial condition $y(0) = (-5, 10, 25)^T$. The underlying parameters in the left image are $\rho = 8$, $\sigma = 16$ and $\beta = 2$, in the middle $\rho = 8$, $\sigma = 22$ and $\beta = 2$ and in the right image $\rho = 8$, $\sigma = 28$ and $\beta = 2$.

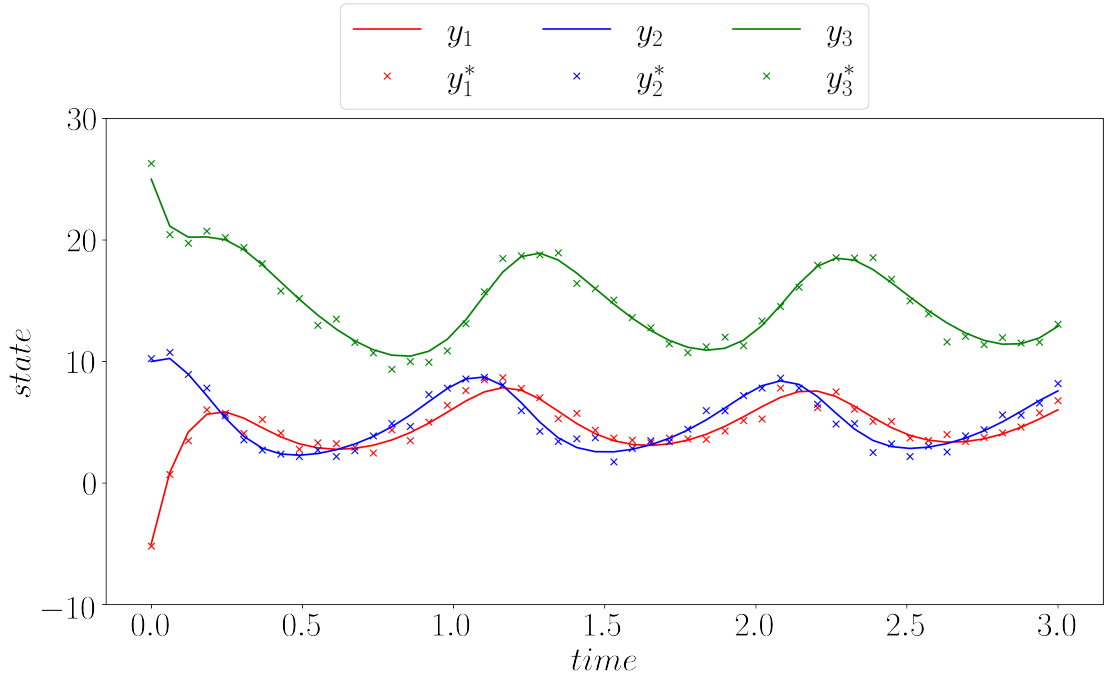

Figure 3: In this figure, the true states y_1, y_2 and y_3 of the Lorenz system are shown as solid lines. The states of the noisy observations y_1^*, y_2^* and y_3^* are marked as crosses. Observations were made at 30 time points in the $[0, 3]$ time interval. The true states were generated using the classical Runge-Kutta methods with the parameters $\rho = 8$, $\sigma = 16$ and $\beta = 2$ and the initial condition $y(0) = (-5, 10, 25)^T$. On these, additive noise z was added, which is normally distributed with mean 0 and variance 0.5, i.e. $z \sim \mathcal{N}(0, 0.5)$.

3 Deep learning

The term deep learning (DL) is closely linked to the terms artificial intelligence (AI), machine learning (ML) or neural networks (NNs). AI is the umbrella term for machines that can almost think like humans and are thus able to perform everyday tasks such as communication and object recognition, to name just a few. ML describes the methodology of gaining knowledge from experience and observations, i.e. from a large amount of data. One approach to ML is DL, where patterns are extracted hierarchically by breaking down the complex input data step by step into simpler and simpler contexts. NNs are particularly well suited for this and are the main component of deep learning. There are different ways of constructing NNs the most commonly used types are FNNs, convolutional NNs and recurrent NNs. For solving differential equations or estimating parameters, FNNs are perfectly adequate ([Lu et al., 2020]), therefore the other types are not considered in this thesis. For more information about them and DL in general please refer to [Goodfellow et al., 2016]. A graphical representation of the relation between the above mentioned terms is figure 4.

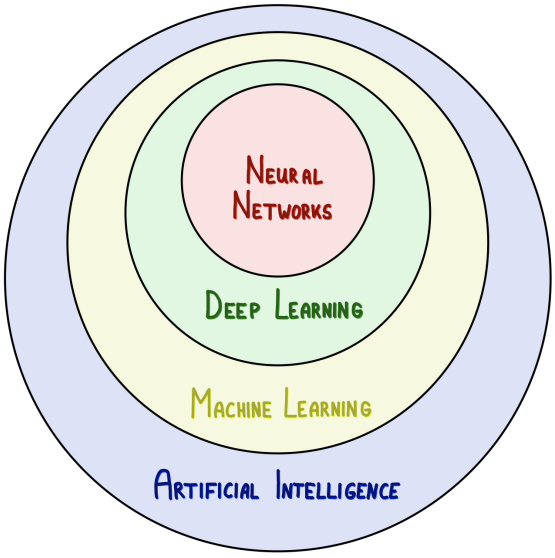

Figure 4: This is an illustration about the relationship of the terms AI, ML, DL and NN inspired by [Goodfellow et al., 2016]. AI is the generic term for intelligent machines. One way to teach computers new things is via ML. There are various approaches, for example DL with NNs.

This section first introduces FNNs and their architecture as well as the different activation functions. Then the optimisation process of NNs is considered, i.e. how insights are drawn from data. For this purpose, the cost and loss functions are described and algorithms are introduced that can minimise such functions, such as (stochastic) gradient descent, ADAM and BFGS. Since these algorithms are based on the first and partly the

second derivative, a technique is needed to determine them efficiently, therefore automatic differentiation is explained. Finally, it is explained how NNs have to be adapted in order to be suitable for our task, the estimation of parameters of differential equations. For this purpose, the so-called PINNs are described in detail and their basic algorithm is presented.

3.1 Feedforward neural network

FNNs, or multilayer perceptrons, are described in [Goodfellow et al., 2016] as a quintessential DL model. Whose goal is to approximate a function f^*, which maps an input $\mathbf{x}$ to an output $\mathbf{y}$, i.e. $f^*(\mathbf{x}) = \mathbf{y}$. The function can be very different, for example a classifier or a time-dependent differential equation with $\mathbf{x} = t$ as input and output $\mathbf{y} = y(t)$. A FNN defines a mapping

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{y} \tag{3.1}$$

with $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}, \mathbf{y} \in \mathbb{R}^{d_{\text{out}}}$. The parameters $\boldsymbol{\theta} \in \mathbb{R}^{d_{\text{param}}}$ are then optimised so that the target function f^* is approximated as best as possible, i.e. $f^*(\mathbf{x}) \approx f(\mathbf{x})$.
An FNN is a hierarchly concept, where linear and nonlinear transformations are applied one after the other to the input $\mathbf{x}$. In this process, the input is broken down step by step into simpler patterns and can be seen as a composition of multiple functions

$$f(\mathbf{x}) = f^L(f^{L-1}(\dots(f^2(f^1(f^0(\mathbf{x})))))). \tag{3.2}$$

3.1.1 Architecture

A basic FNN is visualised in figure 5. As can be seen, a distinction is made between different layers, each of which is described by a function f^l from the decomposition of f, as in equation 3.2.
Each layer consists of neurons, also called units, and can be interpreted as a vector. In [Higham and Higham, 2019] the layers are divided into hidden, input and output layers. In the hidden layers, each neuron receives information from every neuron in the previous layer and computes a real value from all the received information. This is passed on to the next layer, or more precisely to every neuron in this layer. Taken together, each neuron of a layer is connected to all neurons of the neighbouring layers, which is why the term fully connected layer is often used for these type of layers. There can be several hidden layers within a NN; in contrast, there is exactly one input and one output layer. The input layer is the first layer, so there is no previous layer, and the input vector $\mathbf{x}$ is used instead. The output layer lacks the subsequent layer, as it is the last layer, the values of its neurons then determine the output of the entire network.
For each layer l there is a certain weight matrix $\mathbf{W}^l \in \mathbb{R}^{d_l \times d_{l-1}}$. The number of columns of this matrix reflects the number of neurons of the previous layer d_{l-1} and the number of rows is identical to the number of neurons of the current layer d_l. In addition, there is a bias vector $\mathbf{b}^l \in \mathbb{R}^{d_l}$ for each layer l, the size corresponds to the number of neurons d_l in this layer. All weights $\mathbf{W}$ and biases $\mathbf{b}$ are summarised as parameters $\boldsymbol{\theta}$.

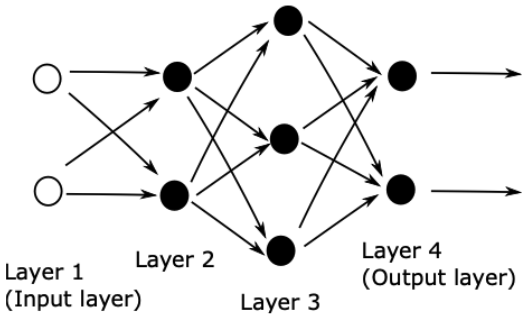

Figure 5: This figure shows a FNN with four layers. On the lefthand side the input layer can be seen. Here the informations are entered into the NN. This is followed by the first hidden layer, which in this example has two neurons and is linked to all input neurons and the three neurons of the second hidden layer. These are linked to the output layer, which is two-dimensional in this specific representation. Usually , much larger networks are used, where the number of hidden layers is increased just like the number of neurons per hidden layer. Input and output layers depend on the task and are included in each FNN. The graphic was taken from [Higham and Higham, 2019].

The layer functions are defined as

$$\begin{aligned} &f^l : \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l}, \qquad l \in 0, \ldots, L, \\ &f^l(x) := \boldsymbol{\sigma}(\mathbf{W}^l x + \mathbf{b}^l) \end{aligned} \tag{3.3}$$

Here, $\boldsymbol{\sigma} : \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$ stands for an activation function applied element wise to the input. They will be introduced in the next section.

3.1.2 Activation functions

Activation functions are an indispensable part of FNN and NNs in general, because without them a NN would only be a special linear regression according to [Sharma et al., 2020]. To differ from linear regression, it is important that the activation function is non-linear in at least one layer of the NN, but preferably in all of them. This makes it possible to map non-linear relations between input and output data, as they often occur in the real world. Besides the non-linearity just discussed, it is important that an activation function is differentiable, as this is essential for gradient-based training algorithms, which we will introduce later.
There are a variety of options in the choice of activation function. In [Sharma et al., 2020], different variants are presented. There are simple activation functions, such as the linear function. However, these cannot recognise complex non-linear relationships.

The rectified linear unit (ReLU), hyperbolic tangent function and sigmoid functions are well suited for this. For classification problems, the softmax function is usually used in the output layer. Then the output can be interpreted as the probability of the individual classes. The NNs implemented in this thesis contain only the hyberbolic tangent function. Non-smooth activation functions, such as the ReLU function, often do not lead to the optimal solution because they do not converge consistently, as stated in [Markidis, 2021].
The hyberbolic tangent function and its gradient is defined as

$$\begin{aligned} \boldsymbol{\sigma}_{\tanh}(x) &= \tanh x \\ &= \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}, \\ \boldsymbol{\sigma}'_{\tanh}(x) &= 1 - \tanh(x)^2. \end{aligned} \tag{3.4}$$

The original values are mapped to the interval $[-1, 1]$ and are centred around 0. Problems can occur here, as the difference between particularly small values can almost disappear due to the mapping. The same applies to particularly large values. This phenomenon can lead to difficulties with gradient-based optimisation methods. How exactly this optimisation works is explained in the next subsection.

3.2 Optimisation of neural networks

According to [Goodfellow et al., 2016], the optimisation of a NN is the process of finding the optimal parameters that lead to an optimal performance, which can be measured with a cost function $J(\boldsymbol{\theta})$.
In mathematical terms, this is an unconstrained optimisation problem, more precisely a minimisation problem given by

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{3.5}$$

where $\boldsymbol{\theta} \in \mathbb{R}^{d_{\text{param}}}$ are the parameters of the NN, summarised in a vector. As described $J : \mathbb{R}^{d_{\text{param}}} \to \mathbb{R}$ is the cost function, which should be smooth.
The goal is now to find the global minimum of J. This definition and the following ones are based on [Nocedal and Wright, 2006].

Definition (global minimum):
A point $\boldsymbol{\theta}^*$ is a global minimizer if

$$J(\boldsymbol{\theta}^*) \leq J(\boldsymbol{\theta}) \quad \textit{for all } \boldsymbol{\theta} \in \mathbb{R}^{d_{param}}. \tag{3.6}$$

Since a cost function is normally very complex, it can be difficult to find the global minimum. Then it is also possible that a local minimum can lead to a sufficiently good performance of the NN. For this, the concept of neighbourhood must first be introduced.

Definition (neighbourhood [Nocedal and Wright, 2006]):
A Neighbourhood $\mathcal{N}$ of $\boldsymbol{\theta}^*$ is an open set that contains $\boldsymbol{\theta}^*$.

Now we can define a local minimum.

Definition (local minimum [Nocedal and Wright, 2006]):
A point $\boldsymbol{\theta}^*$ is a local minimizer if there is a neighbourhood $\mathcal{N}$ of $\boldsymbol{\theta}^*$ such that

$$J(\boldsymbol{\theta}^*) \leq J(\boldsymbol{\theta}) \quad \text{for all } \boldsymbol{\theta} \in \mathcal{N}. \tag{3.7}$$

To find a local minimum, there are necessary and sufficient conditions, which are described in the following theorems.

Theorem 3.1 (First-Order Necessary Conditions [Nocedal and Wright, 2006]):
If $\boldsymbol{\theta}^*$ is a local minimizer and J is continously differentiable in an open neighborhood of $\boldsymbol{\theta}^*$, then $\nabla J(\boldsymbol{\theta}^*) = 0$

Theorem 3.2 (Second-Order Necessary Conditions [Nocedal and Wright, 2006]):
If $\boldsymbol{\theta}^*$ is a local minimizer of J and $\nabla^2 f$ exists and is continously differentiable in an open neighborhood of $\boldsymbol{\theta}^*$, then $\nabla J(\boldsymbol{\theta}^*) = 0$ and $\nabla^2 J(\boldsymbol{\theta}^*)$ is positive semidefinite.

Theorem 3.3 (Second-Order Sufficient Conditions [Nocedal and Wright, 2006]):
If $\nabla^2 J$ is continous in an open neighborhood of $\boldsymbol{\theta}^*$ and that $\nabla J(\boldsymbol{\theta}^*) = 0$ and $\nabla^2 J(\boldsymbol{\theta}^*)$ is positive definite. Then $\boldsymbol{\theta}^*$ is a strict local minimizer of J.

As described above a cost function can be very complex due to its often multidimensional input. Hence, there is usually a lack of global perspective on the function. Usually only certain values and some derivatives are known. The global minimum is only very rarely reached. The cost function can have many different local minima. In addition, other phenomena, such as saddle points or very flat regions, can occur, which make it difficult to find the best possible parameters. However, these are usually not needed, as parameters that lead to a very small, but not globally minimal, cost function can also provide good results.
NNs are trained similarly to other ML models using supervised learning methods.
This requires data, ideally consisting of pairs of input $\mathbf{x}^{(j)} \in \mathbb{R}^{d_{\text{in}}}$ and associated target $\mathbf{y}^{*(j)} \in \mathbb{R}^{d_{\text{out}}}$. These are also called instances and are summarised in data sets, for example a data set D of size $N \in \mathbb{N}$

$$D = \{(\mathbf{x}^{(1)}, \mathbf{y}^{*(1)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{*(N)}).\} \tag{3.8}$$

The dataset is then often split into D_{train} and D_{test}, a training and a test dataset. The training data set is generally larger and is used to find the optimal parameters, which can then be evaluated on the test data set.
In this section, the cost and loss functions are first described in more detail and frequently used functions are given. It also introduces gradient-based optimisation and a technique for efficient gradient computation - automatic differentiation (AD). The backpropagation method, which is essential in almost all NN training algorithms, is explained.

3.2.1 Cost and loss functions

The loss function provides information about how well a NN works on a specific training pair. In contrast, the cost function is a measure of performance on an entire training data set, for which loss functions are usually a basis. For loss functions, it is important to emphasise that the target $\mathbf{y}^*$ represent a vector, so $\mathbf{y}^* = (\mathbf{y}_1^*, \ldots, \mathbf{y}_{d_{out}}^*)^T$. Now L_1- and L_2-loss functions are introduced. For more details about loss functions in general please refer to [Janocha and Czarnecki, 2017]. Firstly, there is the L_1-loss

$$\mathcal{L}_1(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*) = \|\mathbf{y}^* - f(\mathbf{x}, \boldsymbol{\theta})\|_1 = \sum_{i=1}^{d_{out}} |\mathbf{y}_i^* - f(\mathbf{x}, \boldsymbol{\theta})_i|. \tag{3.9}$$

In addition, there is also a more regularised form, the L_2-loss

$$\mathcal{L}_2(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*) = \|\mathbf{y}^* - f(\mathbf{x}, \boldsymbol{\theta})\|_2^2 = \sum_{i=1}^{d_{out}} (\mathbf{y}_i^* - f(\mathbf{x}, \boldsymbol{\theta})_i)^2. \tag{3.10}$$

The loss functions only reflect a very limited picture, as they only depend on one training instance. The aim is to represent the relation between input and output as accurately as possible, i.e. to select the parameters in such a way that the loss for all valid training instances that are not necessarily contained in the data set is as low as possible. In [Bottou et al., 2018], it is assumed that the loss is measured with respect to a probability distribution $\mathcal{P}(\mathbf{x}, \mathbf{y}^*)$, with $\mathcal{P} : \mathbb{R}^{d_{in}} \times \mathbb{R}^{d_{out}} \to [0, 1]$. The expected risk, which applies to the parameter vector $\boldsymbol{\theta}$ with respect to the probability distribution $\mathcal{P}$, can then be taken as the cost function and is defined as

$$J(\boldsymbol{\theta}) = \int_{\mathbb{R}^{d_{in}} \times \mathbb{R}^{d_{out}}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}^*) \mathrm{d}\mathcal{P}(\mathbf{x}, \mathbf{y}^*) = \mathbb{E}[L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}^*)] \tag{3.11}$$

and gives a real-valued measure over all possible instances. The expected risk cannot be computed exactly, as there is a lack of informations regarding $\mathcal{P}$. Hence an estimation with the help of data is necessary and leads to the empirical risk, which is defined as

$$J_D(\boldsymbol{\theta}) = \mathbb{E}_D[L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}^*)] = \frac{1}{N} \sum_{j=1}^{N} L(f(\mathbf{x}^{(j)}; \boldsymbol{\theta}), \mathbf{y}^{*(j)}). \tag{3.12}$$

The empirical risk $J_D : \mathbb{R}^{d_{\text{param}}} \to \mathbb{R}$ is the cost function we want to minimise and can be summarised as the expectation of the loss function $L(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*)$ over a set of data points D. It is also possible to compute the empirical risk on subset $D_M, M \in \mathbb{N}$ of D. To find the minimum of the cost function, an optimisation algorithm is needed. A basic approach is gradient-based optimisation. How this works is explained in the next section. The classical gradient descent algorithm is also presented here.

3.2.2 Gradient-based optimisation

In this section, let a minimisation problem be given as in equation 3.5. Our goal is to minimise the smooth cost function $J : \mathbb{R}^{d_{\text{param}}} \to \mathbb{R}$ with regard to the parameters $\boldsymbol{\theta} \in \mathbb{R}^{d_{\text{param}}}$ First we define the gradient $\nabla J(\boldsymbol{\theta})$ of a function J evaluated at $\boldsymbol{\theta}$ as

$$\nabla J(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} \\ \cdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_{d_{\text{param}}}} \end{pmatrix}. \tag{3.13}$$

This is the vector containing all the partial derivates of J with respect to the parameters $\boldsymbol{\theta}$.
One way to find a minimum of the cost function is by applying a Brute-force search and compute the result for all possible parameters. However, this way is impractical.
According to [Nocedal and Wright, 2006] a more efficient method exists. If J is smooth and twice continuously differentiable, it is sufficient to compute the gradient and the Hessian in order to be able to make a statement about whether $\boldsymbol{\theta}^*$ belongs to a local minimum. The mathematical tool needed is Taylor's theorem. A version adapted to our problem is given in theorem 3.4.

Theorem 3.4 (Taylor's theorem [Nocedal and Wright, 2006]):
Suppose that $J : \mathbb{R}^{d_{param}} \to \mathbb{R}$ is continuously differentiable and that $p \in \mathbb{R}^{d_{param}}$. Then we have that

$$J(\boldsymbol{\theta} + p) = J(\boldsymbol{\theta}) + \nabla J(\boldsymbol{\theta} + tp)^T p \quad \textit{for some } t \in (0, 1). \tag{3.14}$$

Moreover if J is twice continuously differentiable, we have that

$$\begin{aligned} \nabla J(\boldsymbol{\theta} + p) &= \nabla J(\boldsymbol{\theta}) + \int_0^1 \nabla^2 J(\boldsymbol{\theta} + tp) p \; dt, \quad \textit{and that} \\ J(\boldsymbol{\theta} + p) &= J(\boldsymbol{\theta}) + \nabla J(\boldsymbol{\theta})^T p + \frac{1}{2} p^T \nabla^2 J(\boldsymbol{\theta} + tp) p \quad \textit{for some } t \in (0, 1). \end{aligned} \tag{3.15}$$

Now we can derive the gradient descent algorithm, which is basically a line search algorithm with gradient as a search direction. The gradient can be considered as best direction, since Taylor's theorem (theorem 3.4) implies that $J(\boldsymbol{\theta} + \nabla J(\boldsymbol{\theta})) \leq J(\boldsymbol{\theta})$. The basic idea of a line search algorithm is to set a direction $p_k \in \mathbb{R}^{d_{\text{param}}}$ and search along it for an iteration that has a lower function value than the current one $\boldsymbol{\theta}_k$. The best possible step size α can be found by solving the following optimisation problem,

$$\min_{\alpha > 0} J(\boldsymbol{\theta}_k + \alpha p_k). \tag{3.16}$$

Since this problem is also difficult to solve, a few step sizes are tried out and the one that best minimises equation 3.16 is selected. Now we arrive at a new point $\boldsymbol{\theta}_{k+1}$, from there the process is repeated again, a direction p_{k+1} is chosen and a step size α is computed. This process is repeated until a convergence criterion is met, for example the change

between two iterations is only marginal. Since the choice of direction in a general line search algorithm has many degrees of freedom, this can also lead to suboptimal directions. In contrast a gradient descent algorithm uses the gradient $-\nabla J_k$ as search direction, which is also the steepest direction. It is exactly the direction in which the function decreases the most, this follows directly from Taylor's theorem 3.4. The step size can be recalculated in each iteration, as mentioned above, or it can be set uniformly in advance.
The pseudo code of a simple gradient descent algorithm is given in algorithm 1.

Algorithm 1 Gradient Descent

1: **Initialise:** $\varepsilon > 0$ (step size)
2: **Initialise:** $\boldsymbol{\theta}$ (parameter)
3: **while** $\boldsymbol{\theta}$ not converges **do**
4: Compute: $g = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ (compute gradient)
5: Update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \varepsilon g$

In order to compute the derivative needed for the gradient descent algorithm, a method is needed that can do this almost automatically. Such a method is introduced next - AD.

3.2.3 Automatic differentiation and backpropagation

The main component of the Gradient Descent algorithm is the computation of the gradient, i.e. the individual partial derivatives. This should be as simple and fast as possible, as well as automated for different functions. In [Margossian, 2019] four different methods for this purpose are presented, including hand-coded analytical derivative, finite difference or other numerical approximations, symbolic differentiation and AD. An overview over the different methods is given in figure 6.
Since AD is used almost exclusively for our application, we will deal with this method in more detail. Basically, AD is the recursive application of the chain rule given in theorem 3.5.

Theorem 3.5 (chain rule):
Let there be $L \in \mathbb{N}$ different functions $f^1, \dots, f^L$ which yield to $f(x) = f^L(f^{L-1}(\dots(f^1(x))))$. Then the following applies to the partial derivative of f with respect to x

$$\frac{\partial f}{\partial x} = \frac{\partial f^L}{\partial f^{L-1}} \frac{\partial f^{L-1}}{\partial f^{L-2}} \cdots \frac{\partial f^1}{\partial x}. \tag{3.17}$$

"Hence we can break down the action [...] into simple components which we evaluate sequentially." ([Margossian, 2019])
In order to compute all partial derivatives with respect to the parameters, it is necessary to compute some parts several times. This additional effort can be avoided by using the principle of backpropagation by [Rumelhart et al., 1986]. Here, the NN is first evaluated for a given input via a so-called forward pass, i.e. all functions are evaluated and the

Technique	Advantage(s)	Drawback(s)
Hand-coded analytical derivative	Exact and often fastest method.	Time consuming to code, error prone, and not applicable to problems with implicit solutions. Not automated.
Finite differentiation	Easy to code.	Subject to floating point precision errors and slow, especially in high dimensions, as the method requires at least D evaluations, where D is the number of partial derivatives required.
Symbolic differentiation	Exact, generates symbolic expressions.	Memory intensive and slow. Cannot handle statements such as unbounded loops.
Automatic differentation	Exact, speed is comparable to hand-coding derivatives, highly applicable.	Needs to be carefully implemented, alhough this is already done in several packages.

Figure 6: A good overview of the different methods to compute derivatives is given in [Margossian, 2019]. For the four methods hand-coded analytical derivative, finite differentiation, symbolic differentiation and AD advantages and disadvantages are listed. It is clear from this that AD is best suited for computing the derivatives in the optimisation of NNs.

output is computed. A backward pass is then performed. Here, starting from the output, all partial derivatives are computed step by step until one arrives at the input [Higham and Higham, 2019].
The main advantage is that only one forward and one backward pass has to be performed to compute all partial derivatives, no matter how many. It is also possible to compute higher order derivatives, for which the backpropagation with AD can simply be applied again [Lu et al., 2020].

3.3 Algorithms for training of neural networks

In the previous chapter, the basics of NN optimisation were introduced. For this purpose, different loss functions and the cost function were defined. The cost function depends on the parameters of the NN. The smaller the cost function, the better the parameters. The optimisation of a NN, or more precisely its parameters, can thus be expressed as a minimisation problem of the cost function. In addition to the definitions of the global and local minimum, also the necessary and sufficient conditions for the determination of a local minimum were described. Since the explicit computation is unfortunately not possible, the gradient descent method was introduced as a simple numerical method for minimum approximation. This method is a line search algorithm with the gradient as the search direction, which is derived from Taylor's theorem. There are other optimisation algorithms that lead to better approximations of the global minimum or can reach it in

fewer iterations. Some of them are introduced in this section. Basically, these algorithms are based on the gradient descent method. First, the stochastic gradient descent (SGD) is presented. For this it is necessary to explain what batch, mini-batch and stochastic training of NNs means. Basically, the stochastic gradient algorithm uses only a part of the entire training data set, a so-called mini-batch. This method is more efficient, the reasons for which are described in more detail in this section. A special form of the stochastic gradient method is the ADAM algorithm. Here, with the help of estimates of the first and second moment, better step sizes for the optimisation step are computed. These methods are all first-order methods; only the first derivative, i.e. the gradient, is required for their computation. A second-order method is introduced with the BFGS algorithm. Here the second derivative, the Hessian, is also included. However, since this is time-consuming to compute and requires a large amount of memory, an approximation of the Hessian is used in the BFGS algorithm.
We will discuss this in more detail in section 3.3.4. First, we will explain the important concept of batch learning and introduce the closely related SGD algorithm. This is then modified by estimating moments in each iteration and using them in the update step. The resulting algorithm is called ADAM.

3.3.1 Batch, mini-batch and stochastic training of neural networks

If we consider the cost function $J_D(\boldsymbol{\theta})$ of NN, as in equation 3.12. We can see that it is the average of $L(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*)$, the loss function of the individual training pairs, over the data set D. In [Goodfellow et al., 2016], it is therefore described as an expensive computational method, since to determine the gradient of the cost function, the evaluation of each individual training instance and subsequent gradient computation is necessary. This is due to the linearity of the gradient. Applying this property to the gradient of the cost function, we get

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J_D(\boldsymbol{\theta}) =& \nabla_{\boldsymbol{\theta}} \frac{1}{N} \sum_{j=1}^{N} L(f(\mathbf{x}^{(j)}; \boldsymbol{\theta}), \mathbf{y}^{*(j)}) \\ =& \frac{1}{N} \sum_{j=1}^{N} \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^{(j)}; \boldsymbol{\theta}), \mathbf{y}^{*(j)}). \end{aligned} \tag{3.18}$$

Therefore, it may be more practical to compute the gradient only over subsets $D_M \subset D$ of size $M \in \mathbb{N}, M < N$. These subsets are called mini-batches. To create a mini-batch, M different training pairs are randomly selected. The advantage here is that the loss function and the corresponding gradient only have to be computed from M instead of N training pairs in order to perform an update step. Using fewer training samples for the estimates of expected risk leads to less precise estimates. The standard error is a measure for the accuracy. For the standard error SE of the cost function of the complete dataset D the following applies

$$\mathrm{SE}(J_D(\boldsymbol{\theta})) = \frac{\sigma}{\sqrt{N}}, \tag{3.19}$$

where σ is the standard deviation. Similarly, the standard error of the cost function of a mini-batch is given by

$$\mathrm{SE}(J_{D_M}(\boldsymbol{\theta})) = \frac{\sigma}{\sqrt{M}}. \tag{3.20}$$

A ratio between the standard errors can now be derived from equations 3.19 and 3.20. It applies

$$\begin{aligned} \mathrm{SE}(J_{D_M}(\boldsymbol{\theta})) = \frac{\sigma}{\sqrt{M}} = \frac{\sigma}{\sqrt{\frac{MN}{N}}} = \frac{\sigma}{\sqrt{N}}\sqrt{\frac{M}{N}} \\ = \sqrt{\frac{N}{M}} \cdot \mathrm{SE}(J_D(\boldsymbol{\theta})). \end{aligned} \tag{3.21}$$

The computational effort of the gradient determination when using a mini-batch compared to the entire data set is scaled by $\frac{N}{M}$. With the standard error, the difference between the two possibilities scales to this factor in the root. In other words, $\frac{N}{M}$ more calculations are needed to compute the gradient of the cost function over the complete data set compared to the mini-batch. However, the standard error is only lower by factor $\sqrt{\frac{N}{M}}$. This way of training NNs is also called mini-batch training. If the entire data set is used in an algorithm to determine the gradient, it can be assigned to the class of batch training. The extreme case, when the mini-batch contains only one training pair, i.e. $M = 1$, is called stochastic or online training. Using a stochastic training approach, leads to very low computational effort per iteration (optimisation step). However, the gradient depends on only one instance, which can lead to a very inaccurate estimate of the expected risk and thus to a poor search direction. To reduce this risk, small step sizes are often used in stochastic training algorithms [Goodfellow et al., 2016]. In [Bottou et al., 2018], a trade-off is reported between per-iteration costs and per-iteration improvement in terms of minimising the expected risk. Typically chosen mini-batch sizes are numbers of the power of 2 between 32 and 256, according to [Goodfellow et al., 2016]. Hardware components then enable efficient parallelisation in the evaluation of the individual training pairs per iteration.
One algorithm that uses mini-batches is the SGD algorithm. This is based on the gradient descent method but uses a randomly selected partial data set D_M in each iteration instead of the entire training data set D. In the next subsection, the pseudo code of the algorithm is presented and explained. In addition, possibilities for the choice of the step size are given.

3.3.2 Stochastic gradient descent

The SGD method is one of the most frequently used algorithms when it comes to optimising DL models, i.e. NNs [Goodfellow et al., 2016]. This algorithm is based on the use of mini-batches, as just introduced. Essentially, it approximates the true gradient of the cost function with respect to the entire data set by estimating it on a randomly selected portion of the data set. This leads to faster optimisation iterations, with somewhat

noisy gradients. The noise also occurs near a good minimum If the step size remains unchanged throughout the algorithm, this can lead to the optimisation requiring a large number of iterations for a small step size on the one hand. Or on the other hand, if the step size is large, a good local minimum is difficult to achieve. To solve this problem, the step size can be adjusted in the course of training, i.e. starting with a large step size, which then becomes smaller and smaller in order to counteract the noise. A simple possibility is a linear decay of the step size as presented in [Goodfellow et al., 2016]. Here, a maximum ε_0 and a minimum step size ε_τ are defined. Furthermore, a number of iterations τ is defined in which the actual step size develops stepwise from the maximum step size to the minimum step size. Mathematically expressed, the following applies for the step size of iteration $k \in \mathbb{N}$,

$$\varepsilon_k = \begin{cases} (1 - \frac{k}{\tau})\varepsilon_0 + \frac{k}{\tau}\varepsilon_\tau, & 0 \leq k \leq \tau, \\ \varepsilon_\tau, & \text{otherwise.} \end{cases} \tag{3.22}$$

The linear decay of the step size is only one of a multitude of possibilities. It is difficult to specify an optimal variant, so in practice the learning curve, the cost function over time or iterations, is often plotted. This gives good information about the training process. Algorithm 2 shows the pseudo code of the SGD method.

Algorithm 2 Stochastic Gradient Descent

Initialise: $\tau \in \mathbb{N}$ (number of iterations in which the stepwise linear decays)
Initialise: $\varepsilon_0 > 0$ (start step size)
Initialise: $\varepsilon_\tau \in (0, \varepsilon_0)$ (final step size)
Initialise: $k = 0$ (iteration)
Initialise: $\boldsymbol{\theta}$ (parameter)
while $\boldsymbol{\theta}$ not converges **do**
    Create Mini-Batch of size M: $\{(\mathbf{x}^{(1)}, \mathbf{y}^{*(1)}), \ldots, (\mathbf{x}^{(M)}, \mathbf{y}^{*(M)})\}$
    Compute: $g = \frac{1}{M} \sum_{j=1}^{M} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(j)}; \boldsymbol{\theta_t}), \mathbf{y}^{*(j)})$ (gradient approximation)
    $k = k + 1$
    Update: $\varepsilon_k = \begin{cases} (1 - \frac{t}{\tau})\varepsilon_0 + \frac{k}{\tau}\varepsilon_\tau, & 0 \leq k \leq \tau \\ \varepsilon_\tau, & \text{otherwise} \end{cases}$
    Update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \varepsilon_k g$

A more complex algorithm is presented in the next subsection. This is the ADAM algorithm. Similar to the (stochastic) gradient descent method, it is based on the gradient and thus belongs to the first-order methods. Here, instead of the linear decay of the step size in the SGD method, a moment-based adjustment is made.

3.3.3 Adaptive moment estimation

The ADAM algorithm was first presented by [Kingma and Ba, 2017] and was described as a first-order gradient-based optimisation method based on adaptive estimates of lower-order moments. This chapter is based on the explanations given in the paper

Just like the SGD method, ADAM is based on the use of mini-batches, the advantages of which in terms of more efficient optimisation have already been considered in chapter 3.3.1. The main difference between the two algorithms is the step size computation. In contrast to SGD, where the step size often decays linearly and is identical for all parameters, ADAM uses step sizes that are individually adapted to the individual parameters. These are computed by estimating the first and second moment of the respective gradient.
In order to understand the exact process of step size determination, the estimation of the moment is explained first. Suppose we have just computed the gradient $g \in \mathbb{R}^{d_{\text{param}}}$ of a mini-batch. Then we update $s \in \mathbb{R}^{d_{\text{param}}}$ and $r \in \mathbb{R}^{d_{\text{param}}}$, where s is the exponential moving average of the gradient and r of the squared gradient. The exponential moving averages is controlled by the decay rates $\alpha_1, \alpha_2 \in [0, 1)$. Here r and s are computed with the following equations,

$$\begin{aligned} s &= \alpha_1 s + (1 - \alpha_1) g, \\ r &= \alpha_2 r + (1 - \alpha_2) g^2. \end{aligned} \tag{3.23}$$

The variable s can be interpreted as a biased estimate of the first moment, i.e. the mean, and r as a biased estimate of the second moment, i.e. the variance, of the gradient. Since the two variables are usually initialised as vector of zeros with length $d_{\text{param}} \in R$. To avoid this phenomenon, a correction to bias-corrected estimates $\hat{s}$ and $\hat{r}$ follows. In order to do this, the following equations are used

$$\begin{aligned} \hat{s} &= \frac{s}{1 - \alpha_1^k}, \\ \hat{r} &= \frac{r}{1 - \alpha_2^k}, \end{aligned} \tag{3.24}$$

where $k \in \mathbb{N}$ denotes number of the current iteration. Here k is the exponent and not an index. For more details of the bias correction, see [Kingma and Ba, 2017].
After that, from the unbiased estimators $\hat{s}$ and $\hat{r}$, the actual parameter update $\Delta\boldsymbol{\theta}$ is computed with

$$\Delta\boldsymbol{\theta} = \varepsilon \cdot \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}. \tag{3.25}$$

Here $\varepsilon > 0$ denotes the step size and $\delta > 0$ is a very small constant for numerical stability, which ensures that in the course of the optimisation is not accidentally divided by zero. This can happen when $\hat{r}$ disappears. The ratio $\frac{\hat{s}}{\sqrt{\hat{r}}}$ can be interpreted as measure being indirect proportional to the uncertainty, i.e.. a smaller ratio leads to a higher amount of uncertainty. Normally, the ratio becomes smaller near a local minimum. Intuitively, it is more difficult to determine the correct direction here, so the degree of uncertainty is greater. Since the ratio is small there, it follows from equation 3.25 that the parameter update step is also smaller. So if the gradient is very noisy, one does not move too far away from the desired local minimum because of the only slight parameter changes.

In addition to the parameter optimisation advantage just described, the ADAM algorithm is very efficient and requires little memory. Thus, this method is also suitable for large amounts of data or very branched NNs with many layers and neurons per layer, which results in a high number of parameters. The pseudocode of ADAM is given in algorithm 3.

Algorithm 3 ADAM [Kingma and Ba, 2017]

Initialise: $\varepsilon > 0$ (step size, e.g.: $\varepsilon = 0.001$)
Initialise: $\alpha_1 \in [0, 1)$ (exponential decay rate of the first moment estimator, e.g. $\alpha_1 = 0.9$)
Initialise: $\alpha_2 \in [0, 1)$ (exponential decay rate of the second moment estimator, e.g. $\alpha_2 = 0.999$)
Initialise: $\delta > 0$ (constant for numerical stability, e.g. $\delta = 10^{-8}$)
Initialise: $s = 0$ (first moment - mean)
Initialise: $r = 0$ (second moment - variance)
Initialise: $k = 0$ (iteration)
Initialise: $\boldsymbol{\theta}$ (parameter at time $t = 0$)
while $\boldsymbol{\theta}$ not converged **do**
    create a mini-batch of size M: $\{(\mathbf{x}^{(1)}, \mathbf{y}^{*(1)}), \ldots, (\mathbf{x}^{(M)}, \mathbf{y}^{*(M)})\}$
    Compute: $g = \frac{1}{M} \sum_{j=1}^{M} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(j)}; \boldsymbol{\theta_t}), \mathbf{y}^{*(j)})$ (gradient of mini batches)
    $k = k + 1$
    Update: $s = \alpha_1 s + (1 - \alpha_1) g$ (biased estimate of the first moment)
    Update: $r = \alpha_2 r + (1 - \alpha_2) g^2$ (biased estimate of the second moment)
    Correct: $\hat{s} = \frac{s}{1 - \alpha_1^k}$ (unbiased estimate of the first moment)
    Correct: $\hat{r} = \frac{r}{1 - \alpha_2^k}$ (unbiased estimate of the second moment)
    Compute: $\Delta\boldsymbol{\theta} = -\varepsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$
    Update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

Like SGD, ADAM belongs to the first order methods. In the following, a second order method is described, it is BFGS. Here, the second derivative, the Hessian, is used to improve the optimisation with the help of additional information about the parameter space.

3.3.4 Broyden-Fletcher-Goldfarb-Shanno

The conditions for a local minimum are described in chapter 3.2. In addition to first-order conditions, there were also second-order conditions. The algorithms introduced so far are all first-order, so the second-order conditions are not taken into account. This leads to a shorter computation time, as the computation of the second derivative is not necessary. However, the second derivative contains important information about the step size, as can be derived from Taylor's theorem (theorem 3.4). Probably the best known second-order method is Newton's method [Goodfellow et al., 2016]. From equation 3.15

we can derive Newton's update rule

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \mathbf{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{3.26}$$

$\mathbf{H} = \nabla^2 J(\boldsymbol{\theta}) \in \mathbb{R}^{d_{\text{param}} \times d_{\text{param}}}$ denotes the Hessian. If the Hessian is positive definite, i.e. the sufficient condition of second order is fulfilled (theorem 3.3) and in addition the cost function is locally quadratic, only one iteration is necessary to arrive at the local minimum. Since the cost function is very complex and in general not locally quadratic, the direct step does not happen in practice. Furthermore the computation of the inverse of $d_{\text{param}} \times d_{\text{param}}$ Matrix, as $\mathbf{H}^{-1}$, is of order $\mathcal{O}(d_{\text{param}}^3)$. As already mentioned, there can be very many parameters in a neural network and in every Iteration the Hessian $\mathbf{H}$ needs to be computed. Unfortunately, this makes Newton's method very impractical.
Therefore, so-called quasi-Newton methods, like BFGS, named after Broyden, Fletcher, Goldfarb, and Shanno, were invented. Quasi Newton methods in general and the BFGS method in particular are well described in [Nocedal and Wright, 2006]. In the following, these explanations will be used as a guideline. Instead of computing the Hessian $\mathbf{H}$ in each iteration step, only an approximation $\mathbf{M} \in \mathbb{R}^{d_{\text{param}} \times d_{\text{param}}}$ is made. More precisely, this approximation is adjusted in each iteration instead of being completely recomputed. The modified update rule is given by

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \mathbf{M}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{3.27}$$

To derive a formula for fitting the approximation of the Hessian, a modified form of Taylor's theorem 3.4 is used. By adding and subtracting the term $\nabla^2 J(\boldsymbol{\theta})p$ in formula 3.15 we obtain

$$\nabla J(\boldsymbol{\theta} + p) = \nabla J(\boldsymbol{\theta}) + \nabla^2 J(\boldsymbol{\theta})p + \int_0^1 [\nabla^2 J(\boldsymbol{\theta} + tp) - \nabla^2 J(\boldsymbol{\theta})]p \; dt. \tag{3.28}$$

The integral in equation 3.28 is of size $\mathcal{o}(\|p\|)$. If we substitute $\boldsymbol{\theta} = \boldsymbol{\theta}_k$ and $p = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ there, we get

$$\nabla J(\boldsymbol{\theta}_{k+1}) = \nabla J(\boldsymbol{\theta}_k) + \nabla^2 J(\boldsymbol{\theta}_k) \cdot (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) + \mathcal{o}(\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k\|), \tag{3.29}$$

which equivalent to

$$\nabla^2 J(\boldsymbol{\theta}_k) \cdot (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) \approx \nabla J(\boldsymbol{\theta}_{k+1}) - \nabla J(\boldsymbol{\theta}_k). \tag{3.30}$$

The Hessian approximation $\mathbf{M}$ should also satisfy the equation 3.30. For simplicity, we define $s_k := \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ and $r_k := \nabla J(\boldsymbol{\theta}_{k+1}) - \nabla J(\boldsymbol{\theta}_k)$. Therefore,

$$\mathbf{M}_{k+1} s_k = r_k \tag{3.31}$$

must be valid for each approximation, and the symmetry property should also be satisfied, as in the correct Hessian. Furthermore, it is desired that the difference between two successive approximations $\mathbf{M}_k$, $\mathbf{M}_{k+1}$ is a matrix of low rank. The BFGS formula

meets all requirements and is given by

$$\mathbf{M}_{k+1} = \mathbf{M}_k - \frac{\mathbf{M}_k s_k s_k^T \mathbf{M}_k}{s_k^T \mathbf{M}_k s_k} + \frac{r_k r_k^T}{r_k^T s_k}. \tag{3.32}$$

To avoid the inversion of the matrix $\mathbf{M}$, the inverse Hessian $\mathbf{H^{-1}}$ can be approximated directly in each Iteration. In order to do that, the equation 3.32 has to be modified. The resulting updating scheme is then given by

$$\mathbf{M}_{k+1}^{-1} = (\mathbb{1} - \frac{r_k s_k^T}{r_k^T s_k})\mathbf{M}_k^{-1}(\mathbb{1} - \frac{r_k s_k^T}{r_k^T s_k}) + \frac{s_k s_k^T}{r_k^T s_k}. \tag{3.33}$$

With the help of this formula, the BFGS algorithm can now be formulated, as in algorithm 4.

Algorithm 4 Broyden-Fletcher-Goldfarb-Shanno

1: **Initialise:** $\boldsymbol{\theta}$ (parameter)
2: **Initialise:** $M_0 = \mathbb{1}$ (approximation of the inverse Hessian matrix)
3: **Initialise:** $k = 0$ (iteration)
4: **Initialise:** $g_0 = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$ (gradient)
5: **while** $\boldsymbol{\theta}$ not converged **do**
6: Update: $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - M_k g_k$ (update step)
7: Compute: $g_{k+1} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{k+1})$ (gradient)
8: Compute: $s_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$
9: Compute: $r_k = g_{k+1} - g_k$
10: Update: $M_{k+1} = (\mathbb{1} - \frac{s_k r_k^T}{r_k^T s_k})M_k(\mathbb{1} - \frac{r_k s_k^T}{r_k^T s_k}) + \frac{s_k s_k^T}{r_k^T s_k}$ (update of the approximation of the Hessian matrix)
11: $k = k + 1$

The rate of convergence of BFGS is not as high as Newton's method, but it is still superlinear according to [Nocedal and Wright, 2006]. This means that the BFGS algorithm converges faster than the previously presented first-order algorithms. This requires a little more computational effort per iteration, but since the approximation of the inverse Hessian $\mathbf{M}^{-1}$ with equation 3.33 only has to perform operations with a cost of $\mathcal{O}(d_{\text{param}}^2)$, is it acceptable.
In the section just presented, the most important algorithms for optimising general NNs were presented. Starting with the gradient descent method, the stochastic variant and the ADAM algorithm, which are all first order methods, a second order method, the BFGS algorithm, was derived afterwards. AD and backpropagation were introduced in order to compute the necessary derivatives. Before that, the general architecture of FNNs was described. Now the most important tools are available, to introduce NNs for solving differential equations and parameter estimation, PINNs.

3.4 Physics-informed neural network

The basic DL architecture and ML techniques for learning from data have already been introduced. However, known laws of physics have not been taken into account. For this purpose, [Raissi et al., 2019] introduced PINNs - NNs that use laws of physics, which are usually formulated as (partial) differential equations, to achieve better results with less data. PINNs are able to solve differential equations as well as inverse problems, such as estimating parameters.
In this thesis, it is sufficient to describe PINNs for solving time-dependent differential equations. How PINNs can be used for other differential equations is well explained in both [Raissi et al., 2019] and [Lu et al., 2020]. We follow their explanations, but modify them for our application.
Suppose we have a time-dependent system of differential equation as given in equation 2.2,

$$\dot{y} = F(t, y),$$

with $F : I \times \mathbb{R}^n \to \mathbb{R}^n$, $y : I \to \mathbb{R}^n$ and $\dot{y} = (\dot{y}_1, \dots, \dot{y}_n)^T$. Then it is possible to create a NN $f : I \to \mathbb{R}^n$ that is able to approximate the function y with data. A simple FNN as introduced in chapter 3.1 is capable to do that. The number of layers and neurons per layer depends on the complexity of y. The output of the NN is given by $f(t) = \hat{y}(t) = (\hat{y}_1(t), \dots, \hat{y}_n(t)$.
Using AD, the derivative of the NN with respect to its input t, $\frac{d\hat{y}}{dt}$, can be computed efficiently. The biggest differences between PINNs and ordinary NNs are in the loss and cost function. Here, the physical laws are included. Assume a dataset $D = \{t^{(1)}, \dots, t^{(N)}\}$ is given, which is divided into points in the domain D_a and on the boundary D_b. The cost function is given by

$$J_D(\boldsymbol{\theta}) = w_a \frac{1}{|D_a|} \sum_{t \in D_a} L_a(\boldsymbol{\theta}, t) + w_b \frac{1}{|D_b|} \sum_{t \in D_b} L_b(\boldsymbol{\theta}, t), \tag{3.34}$$

where w_a and w_b weights. The first loss function L_a is defined as

$$L_a(\boldsymbol{\theta}, t) = ||F(t, f) - \frac{df}{dt}||_2^2. \tag{3.35}$$

This equation results from the differential equation $F(t, y) = \dot{y}$. The solution y is unknown and is to be approximated by the NN f. So $F(t, f)$ and the derivative of the NN after time $\frac{f}{t}$ should be as similar as possible, i.e. $F(t, f) - \frac{f}{t} \approx 0$. This can be achieved by including the loss function L_a in the cost function. To determine the derivative $\frac{f}{t}$ automatic differentiation is used. This is included in many software packages, such as PyTorch ([Paszke et al., 2019]).
The loss function L_b measures how well the boundary conditions $\mathcal{B}(y, t)$ of the differential equations are fulfilled by the NN f and is given by

$$L_b(\boldsymbol{\theta}, t) = ||\mathcal{B}(f, t)||_2^2. \tag{3.36}$$

If we have an initial value problem, the initial condition is interpreted as a special boundary condition. The loss functions L_a and L_b mentioned above are for the case, when a solution is to be found for the system of differential equations with known parameters. However, if it is an inverse problem, data $D_c = \{(t^{(1)}, \mathbf{y}^{*(1)}), \dots, (t^{(|D_c|)}, \mathbf{y}^{*(|D_c|)})\}$ that can be modelled with the system of differential equations are given, but the parameters $\lambda \in \mathbb{R}^{n_{param}}$ are unknown. Here $n_{param} \in \mathbb{N}$ denotes the number of parameters in the system of differential equations. Then a small change must be made in the cost function. More precisely, the unknown parameters of the differential equations are interpreted as model parameter and an additional term is introduced, which results in

$$J_D(\boldsymbol{\theta}, \lambda) = w_a \frac{1}{|D_a|} \sum_{t \in D_a} L_a(\boldsymbol{\theta}, \lambda, t) + w_b \frac{1}{|D_b|} \sum_{t \in D_b} L_b(\boldsymbol{\theta}, \lambda, t) + w_c \frac{1}{|D_c|} \sum_{t \in D_c} L_c(\boldsymbol{\theta}, \lambda, t), \tag{3.37}$$

with $L_c(\boldsymbol{\theta}, \lambda, t) = ||\mathcal{I}(f, t)||_2^2$ and $\mathcal{I}(y, t) = y(t) - \mathbf{y}^* \approx 0$.
Having a cost function $J(\boldsymbol{\theta})$ makes it possible to optimise the parameters using one of the algorithms in chapter 3.3 presented, SGD, ADAM or BFGS. The basic process to create PINNs for state estimation is described in algorithm 5. Algorithm 6 contains the extension to solve the parameter estimation problem. Figure 7 gives a visual overview of this.

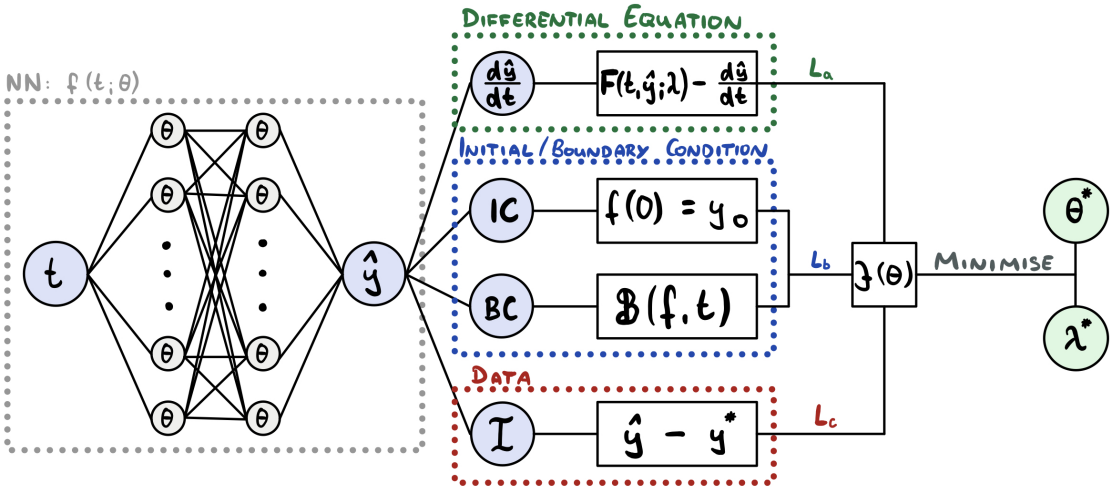

Figure 7: This figure shows the basic structure of PINNs using the Lorenz system as an example. It was inspired by a graphic from [Lu et al., 2020]. The essential part of a PINN forms a NN $f(t; \boldsymbol{\theta})$ with output $\hat{y}$. From this, three separate loss functions are determined, L_a, L_b and L_c. These are finally combined to form the cost function $J(\boldsymbol{\theta})$. L_a represents the differential equation, L_b the boundary and initial condition and L_c the loss of the additional data. The optimal parameters $\boldsymbol{\theta}^*$ of the neural network, as well as the optimal parameters λ^* of the differential equations, are obtained by minimising the cost function $J(\boldsymbol{\theta})$

Algorithm 5 The PINN algorithm for solving differential equations [Lu et al., 2020]

1: **Step 1:** Construct a NN $f : I \to \mathbb{R}^n$ with parameters $\boldsymbol{\theta}$.
2: **Step 2:** Specify the two trainings sets D_a and D_b for the equation and boundary / initial conditions.
3: **Step 3:** Specify a cost function $J(\boldsymbol{\theta})$ by summing the weighted L^2 norm of both the differential equation and boundary condition residuals.
4: **Step 4:** Train the NN to find the best parameters $\boldsymbol{\theta}^*$ by minimizing the loss function $J_D(\boldsymbol{\theta})$.

Algorithm 6 The PINN algorithm for estimating parameters λ of differential equations [Lu et al., 2020]

1: **Step 1:** Construct a NN $f : I \to \mathbb{R}^n$ with parameters $\boldsymbol{\theta}$.
2: **Step 2:** Specify the trainings sets D_a, D_b, D_c for the equation, boundary / initial conditions and additional informations.
3: **Step 3:** Specify a cost function $J(\boldsymbol{\theta})$ by summing the weighted average of the L^2 norms of the differential equation, boundary condition residuals and the additional dataset.
4: **Step 4:** Train the NN to find the best parameters $\boldsymbol{\theta}^*$ and λ by minimising the cost function $J(\boldsymbol{\theta})$.

"Despite the fact that there is no theoretical guarantee that this procedure converges to a global minimum, our empirical evidence indicates that, if the given partial differential equation is well-posed and its solution is unique, our method is capable of achieving good prediction accuracy given a sufficiently expressive neural network architecture and a sufficient number of collocation points" ([Raissi et al., 2019]). Since the physical laws are implemented in a PINN, less data is required compared to conventional NNs.
In the next subsection, a PINN is tested using the Lorenz system from section 2.4 as an example. It describes which adjustments have to be made and examines how different amounts of data and noise affect the performance.

3.5 Example: Lorenz system

The Lorenz system is mathematically represented by equation (2.10). The same parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial conditions $y(0) = (-5, 10, 25)^T$ as in section 2.4 will be used to create the dataset D_c, which consists of 250 pairs $(t_j, y^*(t_j))$. The input t_j represents the time and is equally distributed over the interval $[0, 3]$. The observations y^* are noisy versions of the true system states y with $z \sim \mathcal{N}(0, 0.5)$ as described in equation (2.9).
To implement a PINN for the Lorenz system, the Python package DeepXDE presented in [Lu et al., 2020] was used. It can be found under this link: `https://github.com/lululxvi/deepxde`.

In order to approximate the solution y a simple FNN is chosen. In addition to the usual input and output layers, it consists of three hidden layers. The input layer has only one neuron, the time t_j. A three-dimensional vector, $\hat{y}(t_j) = (\hat{y}_1(t_j), \hat{y}_2(t_j), \hat{y}_3(t_j))$, is the output. The hidden layers each consist of 32 neurons. The *tanh* function defined in equation 3.4 was chosen as the activation function. The NN was optimised using the Adam method. Figure 8 shows the state prediction of the PINN. These predictions approximate the true values very well, as the visualisation of the differences over the time in figure 9 shows. The value of the cost function $J_{D_c}(\boldsymbol{\theta})$ with L_1-loss is

$$J_{D_c}(\boldsymbol{\theta}) = \frac{1}{250}\sum_{j=1}^{250}\mathcal{L}_1(\hat{y}(t_j), y(t_j)) = \frac{1}{250}\sum_{j=1}^{250}\frac{1}{3}\sum_{i=1}^{3}|\hat{y}_i(t_j) - y_i(t_j)| = 0.034.$$

The parameters of the differential equations are also approximated very well. For the parameter ρ the true value 8 was only misestimated by 0.01. The situation is similar for β, where 1.98 was estimated instead of the correct value 2. For σ, the value 16.0 was even estimated completely correctly. The development of the parameter estimation over the epochs is shown in figure 10.

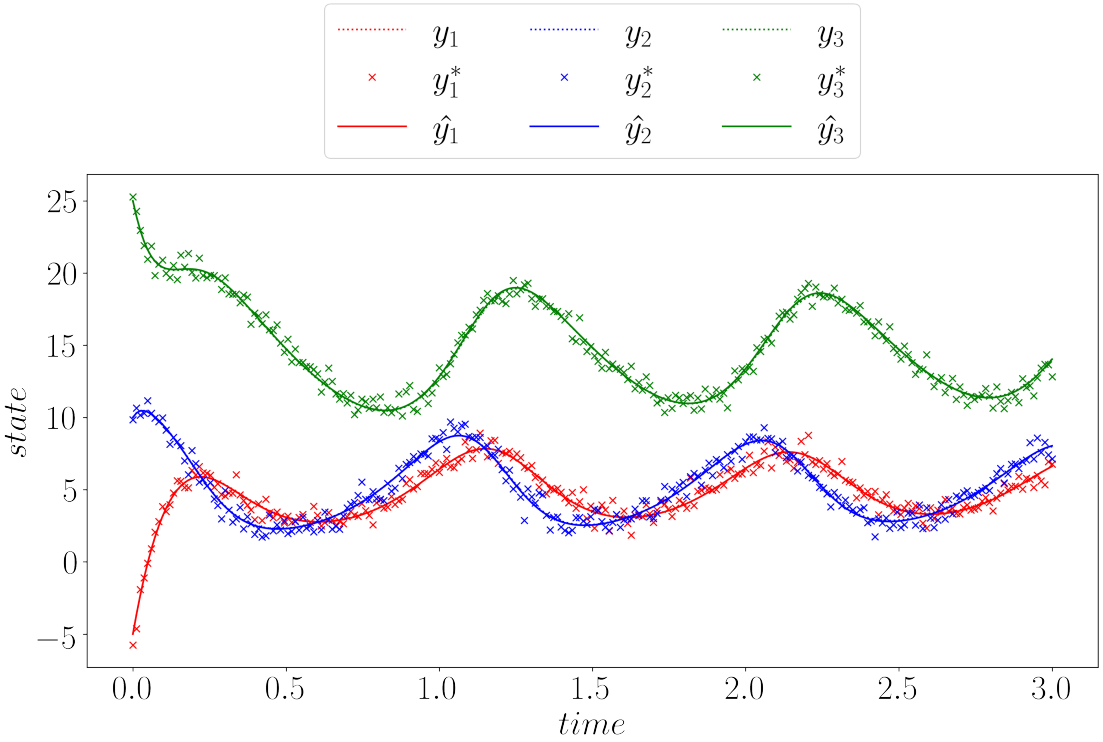

Figure 8: Here the evolution of the states is shown, where the dashed lines represent the true states y. The observations y^* are marked with crosses. The solid line reflects the predictions $\hat{y}$. The dashed lines are almost completely hidden behind the solid lines, implying a good prediction.

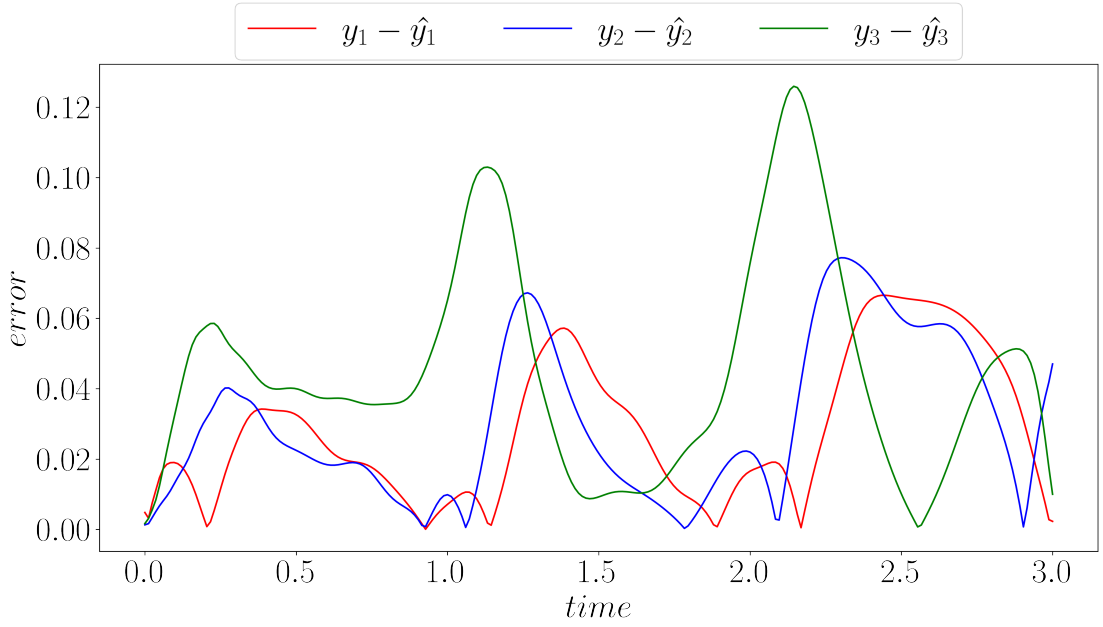

Figure 9: Here the absolute difference between true state y and prediction $\hat{y}$ at the different points in time is shown. The values at time t_j can be computed by $|y_i(t_j) - \hat{y}_i(t_j)|$. The red line represents the values for $i = 1$, the blue for $i = 2$ and the green for $i = 3$. The value of the cost function, i.e the mean over all indices and time points is given by 0.034.

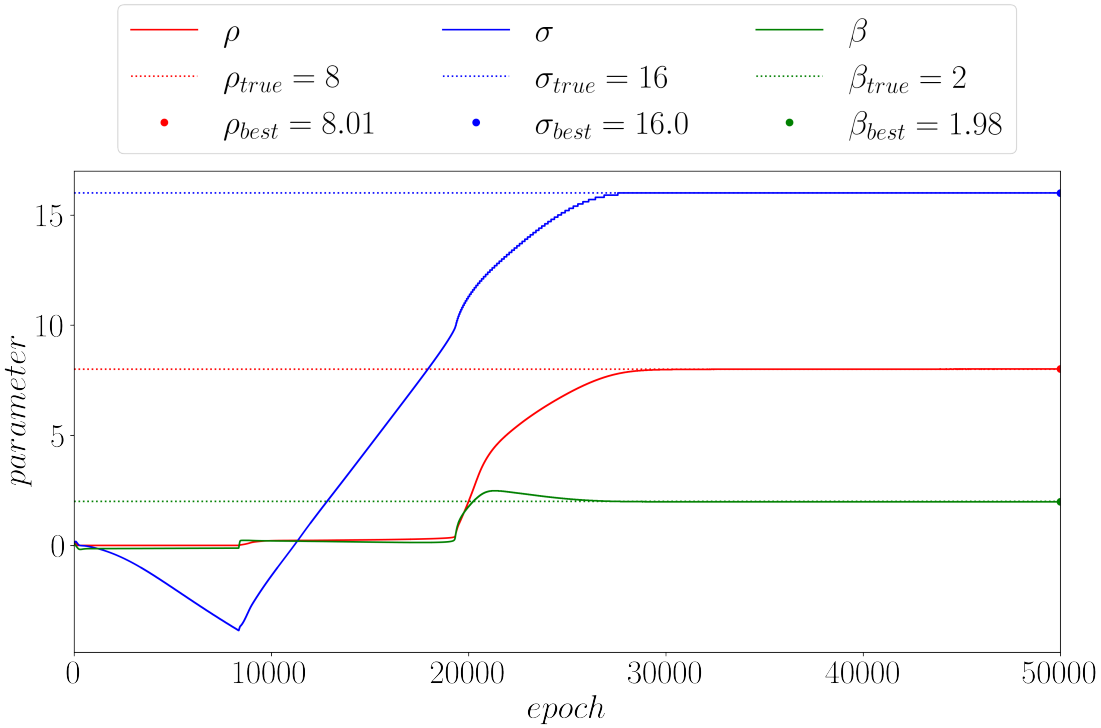

Figure 10: The course of the parameter values over the training epochs are visualised here. The dashed lines represent the true parameter value. The solid lines show the value of the parameter after the respective optimisation epoch. The best parameter is the value after $50,000$ optimisation iterations. The colour red stands for the parameter ρ, blue for σ and green for β. It takes about $30,000$ epochs to find good parameter values for the Lorenz system. But then they are very accurate.

4 Data assimilation

Differential equations unfortunately do not describe natural laws exactly. This is mainly due to complex interactions that often cannot all be included in the system of differential equations. Therefore, the computer model that forecast the state on this basis will also be inaccurate. Often observations of these states are available, but they can also differ from the actual state. There can be many reasons for this, for example other external influences or measurement errors, to name just a few. To minimise errors, the principle of data assimilation can be applied, which according to [Freitag, 2020], is the process of combining model and data to obtain a more informed system. In [Reich and Cotter, 2015], data assimilation is described as a constant comparison between model forecast and actual observations, which leads to adjustments in the computer model and reduces uncertainty in subsequent forecasts. The adjustments can be made in the parameters of the model. Weather forecasting was one of the early fields in the development of data assimilation techniques. Subsequently, these methods have been adapted for a variety of different applications in all kinds of fields, such as economics, biology, chemistry, medicine, physics and many more. So wherever models have been developed to describe processes and observations of states are made at certain points in time.
One of the most important data assimilation techniques is the Kalman filter, which was introduced in [Kalman, 1960]. According to [Humpherys et al., 2012], the Kalman filter minimises the mean squared estimation error between model estimation and observations of the state at fixed times, thus gradually improving the underlying model. When minimising the estimation error, the best unbiased estimate of the true state of the system is obtained. Furthermore, the covariance matrix provides additional informations, which can be interpreted as uncertainty. A major advantage of the Kalman filter is its relatively easy computation, which also allows it to be used in real-time problems.

4.1 Kalman filter

In this section we will introduce the Kalman filter, in order to do that we will go along with [Freitag, 2020]. Basically the Kalman filter consists of two steps, a forecast step and an analysis or correction step.
Assume that noisy observations $y_j^* \in \mathbb{R}^n$ at time $t_j \in \mathbb{R}$ with $j = 0, \dots, N$ and $N \in \mathbb{N}$ are given, which can be modelled by a system of differential equations. The realisations of the Kalman filter at time t_j are divided into forecast realisations $y^F(t_j) \in \mathbb{R}^p$ and analysis realisations $y^A(t_j) \in \mathbb{R}^p$ with $p \in \mathbb{N}$. To set up the algorithm of the Kalman filter, two mappings are needed. One is the mapping from a forecast realisation $y^F(t_j)$ to an observation $y^*(t_j)$ given by

$$\begin{aligned} &\mathcal{H}_j : \mathbb{R}^p \to \mathbb{R}^n, \\ &y^F(t_j) \mapsto y^*(t_j). \end{aligned} \tag{4.1}$$

The other is a mapping to evolve the realisations in time, as going from an analysis realisation $y^A(t_j)$ to the forecast of the following iteration $y^F(t_{j+1})$, which is defined as

$$\begin{aligned} \mathcal{M}_j &: \mathbb{R}^p \to \mathbb{R}^p, \\ & y^A(t_j) \mapsto y^F(t_{j+1}). \end{aligned} \tag{4.2}$$

This defines the forecast step.
For the following computations it is necessary to have linearisations of these mappings. The mapping $\mathcal{H}_j$ from equation 4.1 can be represented by the matrix $H_j = \frac{\partial \mathcal{H}_j}{\partial y^F(t_j)}(y^F(t_j)) \in \mathbb{R}^{n\times p}$. Analogously the linearisation corresponding to the operator $\mathcal{M}_j$ from equation 4.2, is given by $M_j = \frac{\partial \mathcal{M}_j}{\partial y^A(t_j)}(y^F(t_j)) \in \mathbb{R}^{p\times p}$. $\mathcal{M}_j$ is usually determined by the system of differential equations. Now we can set up the discrete-time linear system dynamics

$$\begin{aligned} y^F(t_{j+1}) &= M_i y^A(t_j) + w_j, \\ y_j &= H_j y^F(t_j) + v_j. \end{aligned} \tag{4.3}$$

Here $w_j \in \mathbb{R}^p$ and $v_j \in \mathbb{R}^n$ are Gaussian random vectors, i.e. $w_j \sim \mathcal{N}(0, Q_j)$ with covariance matrix $Q_j = \mathbb{E}[w_j w_j^T] \in \mathbb{R}^{p\times p}$ and $v_j \sim \mathcal{N}(0, R_j)$ with covariance matrix $R_j = \mathbb{E}[v_j v_j^T] \in \mathbb{R}^{n\times n}$. They are assumed to be independent. The random vector w_j can be interpreted as model error and v_j as observation or measurement error.
In the forecast step the analysis $y^A(t_j)$ is given. In order to estimate the new state $y^F(t_{j+1})$, the linearisation M of the model $\mathcal{M}$ has to be applied,

$$y^F(t_{j+1}) = M_j y^A(t_j). \tag{4.4}$$

With the error of the forecast step $e^F_{j+1} = M_j e^A_j + w_j$ we can estimate the corresponding error covariance matrix P^F

$$\begin{aligned} P^F_{j+1} &= \mathbb{E}[e^F_{j+1}(e^F_{j+1})^T] \\ &= \mathbb{E}[(M_j e^A_j + w_j)(M_j e^A_j + w_j)^T] \\ &= M_j P^A_j M^T_j + Q_j. \end{aligned} \tag{4.5}$$

After that, the analysis step can be completed, where the forecast estimate $y^F(t_j)$ is combined with the observation $y^*(t_j)$,

$$y^A(t_j) = y^F(t_j) + K_j(y^*(t_j) - H_j y^F(t_j)). \tag{4.6}$$

In order to do this, the Kalman gain K_j is needed. This is chosen so that the variance of the error covariance of the analysis step $tr(P^A_j)$ is minimal, i.e. the estimate $y^A(t_j)$ is unbiased. For this purpose, $\frac{\partial P^A_j}{\partial K_j} = 0$ is evaluated, from which follows

$$K_j = P^F_j H^T_j (H_j P^F_j H^T_j + R_j)^{-1}. \tag{4.7}$$

The required covariance matrix P_j^A of the analysis step error $e_j^A = K_j(-H_j e_j^F + v_j) + e_j^F$ is given by

$$\begin{aligned} P_j^A &= \mathbb{E}[e_j^A (e_j^A)^T] \\ &= \mathbb{E}[(K_j(-H_j e_j^F + v_j) + e_j^F)(K_j(-H_j e_j^F + v_j) + e_j^F)^T] \\ &= (\mathbb{1} - K_j H_j) P_j^F. \end{aligned} \tag{4.8}$$

These computations are summarised as a pseudocode in algorithm 7.

Algorithm 7 The Kalman filter algorithm [Freitag, 2020]

Determine: Q_j (model error covariance)
Determine: R_j (observation error covariance)
Determine: M_j (linearisation of the model operator $\mathcal{M}_j$)
Determine: H_j (linearisation of the observation operator $\mathcal{H}_j$)
Obtain: $y^*(t_j)$ (noisy observations for $j = 0, \ldots, N$)
Initialise: $y^F(t_0) = y_0$ (initialise first forecast realisation)
Initialise: P_0^F (initialise forecast error covariance)
for $j = 0, \ldots, N$ **do**
    Compute: $K_j = P_j^F H_j^T (H_j P_j^F H_j^T + R_j)^{-1}$ (Kalman gain)
    Compute: $y^A(t_j) = y^F(t_j) + K_j(y^*(t_j) - H_j y^F(t_j))$ (analysis step)
    Compute: $P_j^A = (\mathbb{1} - K_j H_j) P_j^F$ (observation error covariance)
    Compute: $y^F(t_{j+1}) = M_j y^A(t_j)$ (forecast step)
    Compute: $P_j^F = M_j P_j^A M_j^T + Q_j$ (forecast error covariance)

Figure 11 shows the interaction of observation y^*, forecast step y^F and analysis step y^A for the relatively simple one-dimensional case.
A small forecast error P_j^F leads to small Kalman gain K_j, thus $y^A(t_j)$ is heavily geared towards $y^F(t_j)$. Whereas a small observation error R_j leads to Kalman gain $K_j \approx H_j^{-1}$, thus $y^A(t_j)$ is steered towards $y^*(t_j)$. In [Welch and Bishop, 1995] it is described that large values in the covariance matrix R_j lead to small adjustments in the direction of observation. In contrast, small values in this matrix represent small observational errors and the analysis realisations $y^A(t_j)$ are strongly determined by the observations $y^*(t_j)$.
The algorithm just presented is used for state estimation. In order to estimate the parameters as well, there are various possibilities. One simple way is to use Monte Carlo methods. "Monte Carlo methods proceed by drawing random samples, either from the desired distribution or from a simpler one, and using them to compute consistent estimators." ([Luengo et al., 2020]) For this purpose, an interval can be defined for each parameter to be estimated, from which random values are then drawn with the help of the uniform distribution. The Kalman filter is then applied to each value and the forecast realisation is compared with the observations. In order to be able to make a statement about how high the agreement is, a cost function is introduced, based for example on the L_1-loss or L_2-loss. The parameter values with the lowest cost function lead to the model that best describes the observations.

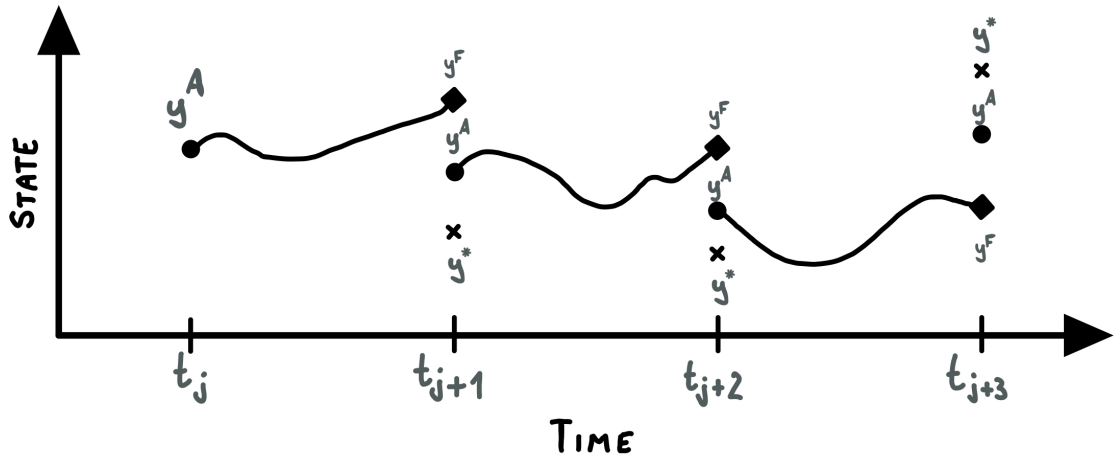

Figure 11: Here is a simplified one-dimensional representation of the Kalman filter. A similar illustration can be found in [Freitag, 2020]. At a time t_j an analysis $y^A(t_j)$ is known. Using the forecast operator $\mathcal{M}$, $y^F(t_{j+1})$ is computed. Subsequently, an analysis step takes place in which the analysis $y^A(t_{j+1})$ is computed from the forecast $y^F(t_{j+1})$ and the observation $y^*(t_{j+1})$. The forecast and the analysis state at the time t_{j+2} and later are computed in the same way.

How the Kalman filter is fitted to a given system of differential equations is presented next. The Lorenz system is used for this purpose.

4.2 Example: Lorenz system

In this subsection, the Kalman filter is applied to the Lorenz system presented in section (2.4). The system of differential equation given in equation (2.10) with parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial conditions $y(0) = (-5, 10, 25)^T$ was used.
In order to test the Kalman filter, observations y^* are required. The true values y are generated using the Runge-Kutta method. 30 observations y^* with noise $z \sim \mathcal{N}(0, 0.5)$ were taken at the time points $t_1, \ldots, t_{30}$. These time points are equispaced over the interval $[0, 3]$.
For the problem of state estimation the parameter values are known, these are initialised first. Then an empty matrix is created for each of the forecast and analysis realisations, which has the dimension 30×3 (number of data $\times$ dimension of the states). The first row of the forecast matrix is set equal to the initial observations. Afterwards the forecast error matrix is initialised as an identity matrix, i.e. $P^F = \mathbb{1}_3$. The covariance matrices R and Q are also chosen as identity matrices for the sake of simplicity, $R = \mathbb{1}_3$ and $Q = \mathbb{1}_3$. Next, the linearisation of the observation operator $\mathcal{H}$ must be determined. Since the observations and realisations agree, this is the identity matrix $H = \mathbb{1}_3$. Now we can iterate over the time index. Here h denotes the difference between the current time and the previous one, i.e. $h = t_j - t_{j-1}$. In each iteration, the Kalman gain is

first computed with the equation (4.7). This makes it possible to complete the analysis step. The current observations are combined with forecast realisation using equation 4.6 to obtain the analysis realisation. The error covariance matrix of the analysis step P^A can then be computed using equation 4.8. In order to complete the final forecast step, it is necessary to determine the linearisation of the forecast operator $\mathcal{M}$. For this, the Runge-Kutta method with order $s = 1$ is used, the Euler method given by

$$\begin{aligned} y(t_{j+1}) &= y(t_j) + h \cdot F(t_j, y(t_j)) \\ &= \begin{pmatrix} y_1(t_j) \\ y_2(t_j) \\ y_3(t_j) \end{pmatrix} + h \cdot \begin{pmatrix} \rho(y(t_j)_2 - y_1(t_j)) \\ y(t_j)_1(\sigma - y_3(t_j)) - y_2(t_j) \\ y_1(t_j)y_2(t_j) - \beta y_3(t_j) \end{pmatrix} \\ &= \begin{pmatrix} y_1(t_j) + h(\rho(y_2(t_j) - y_1(t_j))) \\ y_2(t_j) + h(y_1(t_j)(\sigma - y_3(t_j)) - y_2(t_j)) \\ y_3(t_j) + h(y_1(t_j)y_2(t_j) - \beta y_3(t_j)) \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} 1 - h\rho & h\rho & 0 \\ h\sigma & 1 - h & -hy_1(t_j) \\ hy_2(t_j) & 0 & 1 - h\beta \end{pmatrix}}_{=:M_j} \cdot \underbrace{\begin{pmatrix} y_1(t_j) \\ y_2(t_j) \\ y_3(t_j) \end{pmatrix}}_{y(t_j)}. \end{aligned} \tag{4.9}$$

Thus, $M_j = \begin{pmatrix} 1 - h\rho & h\rho & 0 \\ h\sigma & 1 - h & -hy_1(t_j) \\ hy_2(t_j) & 0 & 1 - h\beta \end{pmatrix}$ is the linearisation of the forecast operator. Using equation 4.4 leads to the forecast realisation for the next time step. The corresponding error matrix P^F can now be computed applying the equation 4.5. In figure 12 a visualisation of the Kalman filter for the state estimation problem is shown.
In order to estimate the parameters, a measure of how good the current parameter values $\lambda = (\rho, \sigma, \beta)$ are, is needed - the cost function. There are several choices here. For the Lorenz system, a cost function based on the L_2-loss was chosen, given by

$$\begin{aligned} J(\lambda) &= \sum_{j=0}^{N} \mathcal{L}_2(y^F(t_j), y^*(t_j)) \\ &= \sum_{j=0}^{N} \sum_{i=1}^{3} (y_i^F(t_j) - y_i^*(t_j))^2. \end{aligned} \tag{4.10}$$

Now random values for the parameters are determined one after the other. For this purpose, a value from the interval $[0, 20]$ is assigned to each parameter using a uniform distribution. The Kalman filter algorithm is then applied to these values and the forecast realisations y^F are determined. Based on the observations y^* and the forecast realisations y^F, the cost function $J(\lambda)$ can be evaluated. The aim is to find the lowest value of the cost function with the associated parameters. If the process consisting of random parameter selection, Kalman filter application and cost function computation is repeated often enough, a good estimator for the minimum is obtained via Monte Carlo method.

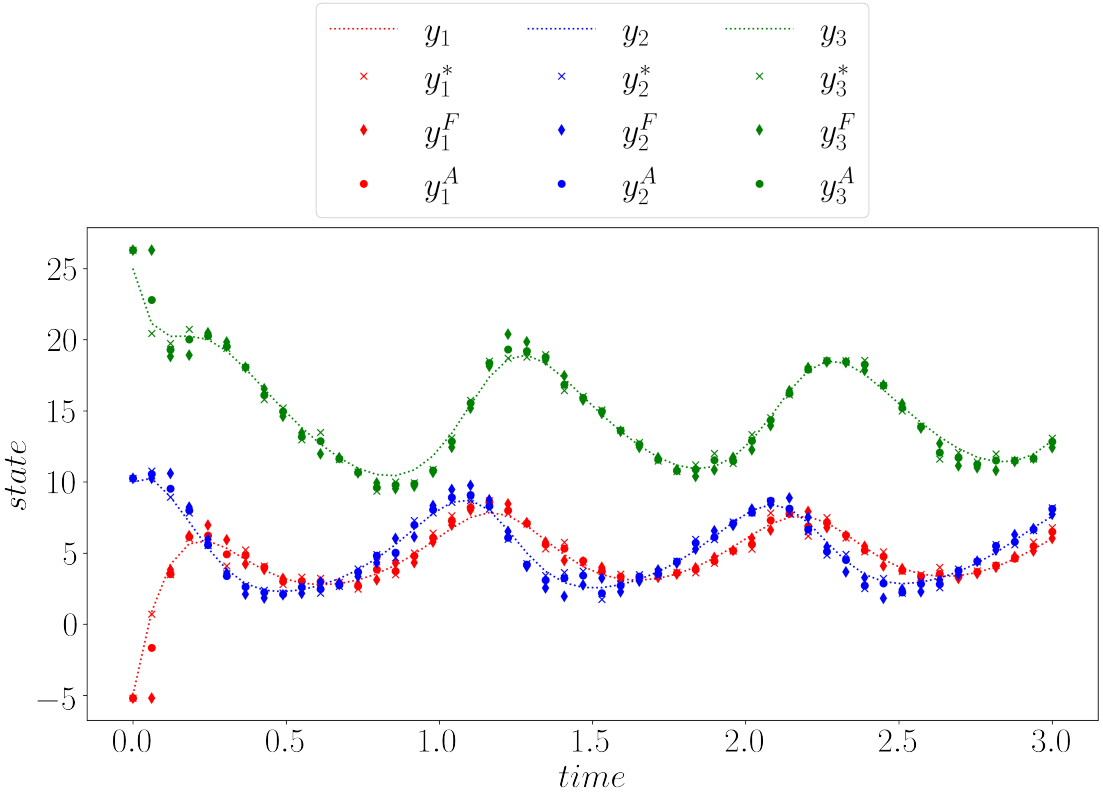

Figure 12: This is a representation of the state estimate of the Kalman filter for the Lorenz system with parameters $\rho = 8$, $\sigma = 16$, $\beta = 2$ and initial condition $y(0) = (-5, 10, 25)^T$. The true states were generated by the explicit Runge-Kutta method of order 4. To simulate real observations, a noise term $z \sim \mathcal{N}(0, 0.5)$ was added at 30 time points. Using these observations y^*, the forecast and analysis states of the Kalman filter could then be determined. In the figure, the true states y are drawn as a dashed line. The observation y^* are marked by crosses. In addition, the forecast states y^F are marked as diamonds and the analysis states y^A as circles.

$50,000$ repetitions were carried out for the Lorenz system. To reduce the time required, the Kalman filter can be evaluated for several random parameter combinations at the same time through parallelisation. Here, the computer architecture is exploited and the Kalman filter for a parameter combination is computed for each available processor core. The progression of the individual best parameter values over the iterations is shown in figure 13.
How the individual parameter values are related to the value of the cost function is visualised in figure 14. The best parameters estimated with this method are $\rho = 7.33$, $\sigma = 15.96$ and $\beta = 2.08$.

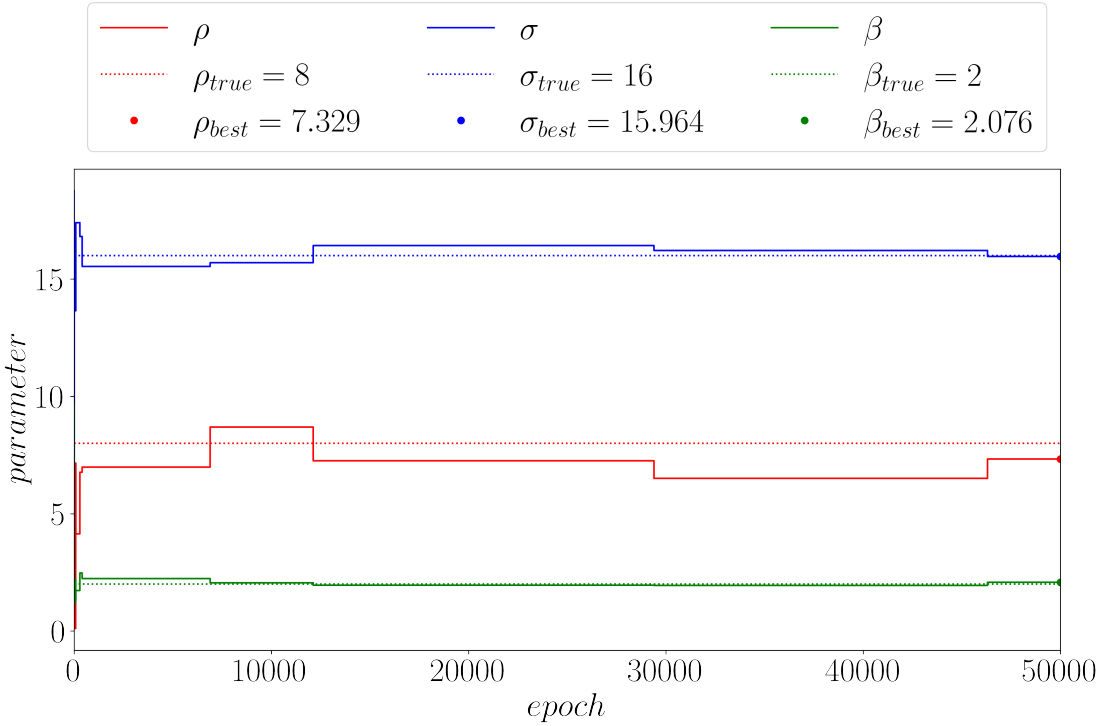

Figure 13: The best parameter values after each Monte Carlo iteration (epoch) of the Kalman filter are shown here. A total of 50,000 epochs were performed. The parameters that led to the lowest value of the cost function after the last epoch are $\rho = 7.33$, $\sigma = 15.96$, $\beta = 2.08$.

However, these parameter estimates cannot be compared with the estimates of the PINN in section 3.5 because less data were used and the specific values of the noise terms were different since they are random. This will be changed in the next chapter, where the two methods will be tested on the same data. Before that, theoretical aspects such as possible errors and the complexity of the implementation are compared.

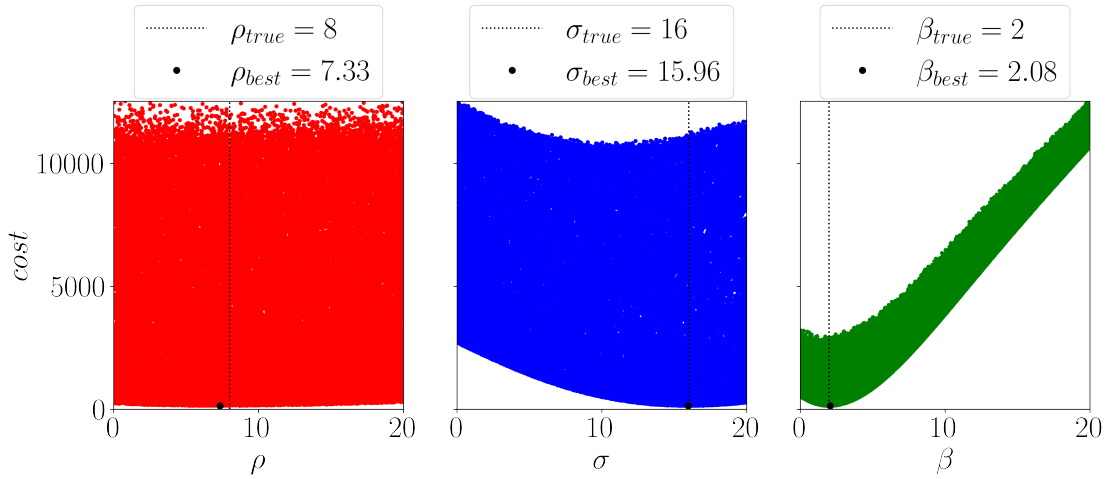

Figure 14: In this figure there are three subplots, one for each parameter ρ, σ and β. In each subplot, all randomly drawn values of the parameter of the Monte Carlo method are plotted with the corresponding value of the cost function. The value that led to the best cost function is marked with a black dot. The dashed line serves as a guide and locates the true parameter value.

5 Comparison between deep learning and data assimilation

After the functionality of the Kalman filter and the PINNs have been introduced and each tested on an example of the Lorenz system, a detailed comparison can now be made. This will first be of a theoretical nature, looking at the resulting errors of both methods. In addition, we will compare how complicated the implementation is. Subsequently, we will see how well the two methods can predict states of the Lorenz system and estimate the parameters, depending on the number of data and the size of the noise term. In doing so, PINN and the Kalman filter are compared with regard to the accuracy of the states and the parameters, as well as the required computing time. Finally, another system of differential equations is introduced, the SIR model. This can be used to model pandemics, like the current COVID-19 pandemic. The two algorithms are tested on both simulated and real data.

5.1 Theoretical comparison

The error analysis for PINNs follows the explanations of [Lu et al., 2020]. Assume we want to approximate the true solution y of a differential equation $\dot{y} = F(y,t)$ by our optimised neural network f with output $\hat{y}$. An optimal FNN y_{NN} with enough layers and neurons can approximate any function y. However there is still a very small approximation error $\varepsilon_{\text{approx}}$, which is defined as

$$\varepsilon_{\text{approx}} = ||y_{\text{NN}} - y||. \tag{5.1}$$

The network architecture y_{arch} we choose for our network is not optimal, which leads to a generation error,

$$\varepsilon_{\text{gen}} = ||y_{\text{arch}} - y_{\text{NN}}||. \tag{5.2}$$

Here we assumed that we reach a global minimum with the chosen architecture y_{arch}. In practice it is very likely to remain at a local minimum at the end of the optimisation process. The obtained solution $\hat{y}$ differ from the optimal solution given the same architectur y_{arch}. This difference can be interpreted as optimisation error

$$\varepsilon_{\text{opt}} = ||\hat{y} - y_{\text{arch}}||. \tag{5.3}$$

The error of PINNs is composed of the types just presented. There is the approximation error from equation (5.1), the generalisation error as in equation (5.2) and in the end the optimisation error given in equation (5.3). In summary, the following upper bound for the overall error of a PINN can be formulated

$$\begin{aligned} \boldsymbol{\varepsilon}_{\text{PINN}} &:= ||\hat{y} - y|| \\ &\leq \varepsilon_{\text{opt}} + \varepsilon_{\text{gen}} + \varepsilon_{\text{approx}}. \end{aligned} \tag{5.4}$$

For the error analysis of the Kalman filter, we already know the two main errors from section 4.1. On the one hand, this is the error of the forecast step

$$e_{j+1}^F = M_j e_j^A + w_j \tag{5.5}$$

and on the other hand that of the analysis step

$$e_j^A = K_j(-H_j e_j^F + v_j) + e_j^F. \tag{5.6}$$

After each time step, the predictions are combined with the observations to obtain the analysis. This combination uses the Kalman gain K, which is chosen to minimise the variance of the error covariance matrix of the analysis step $tr(P_j^A)$. This correction makes the forecast step relatively accurate even for worse models M.
Now we will look into the difficulty of the implementation. It can be seen that PINNs require relatively little or no knowledge about differential equations. If already existing packages, such as DeepXDE ([Lu et al., 2020]), are used, only little knowledge about NNs is needed. Only the basic structure, how many layers, neurons per layer and which activation functions have to be defined. The only thing that needs to be done is to implement the computation of the differential equation and make the data and initial conditions available to the package. However, it may be necessary to make changes to the cost function 3.34 for some differential equations. More precisely, the weights w_a, w_b and w_c have to be adjusted when the values of the differential equations are in different dimensions. For example, if $y_1 > 10^6$ and $y_2 < 10$. The weights should then be chosen that no loss term dominates the cost function, i.e. that they are all approximately identical and ideally in the interval $[0.1, 1.0]$.
With the Kalman filter, the greatest difficulty is to determine the forecast operator $\mathcal{M}$ or its linearisation M. This requires knowledge about the numerical solution of differential equations, for example with Runge-Kutta methods. This can lead to difficulties with some more complex differential equations. Otherwise, there are some subtleties, for example how to best initialise the matrices (P_F, R, Q).
Both methods have advantages and disadvantages. The biggest difference is that the Kalman filter compares the model with observations after each time point. PINNs train a NN on the observations which should approximate the solution of a differential equation. With the help of AD it is then possible to determine the derivatives and to set up a loss term with the help of the given differential equation. Which method is more suitable for which problem in practice will be determined in the next section.

5.2 Computational comparison

Now the two different algorithms are compared with each other using the same data, obtained from the Lorenz system. A total of 12 data sets were created. Between each set the number of data pairs $N \in \{50, 150, 250\}$ and the variance of the additive noise term $\sigma_z^2 \in \{0, 0.1, 0.5, 1.0\}$ were varied. Attention was paid to how long the computations take and how large the difference between the true states or parameters and their estimates are. For the computations a MacBook Pro (14", 2021) with Apple M1 Pro Chip, 10 CPUs

and 32 GB RAM was used. Table 1 shows the results of PINNs and in table 2 those for the Kalman filter are summarised. The first row indicates how many data points N were fed into the particular algorithms. The variance σ_z^2 of the additive noise is given in the first column. The state costs are given by the mean L1-loss between estimated and true state. Similarly, the parameter cost is the absolute distance between the estimated and true parameters averaged over the parameters. These tables are presented graphically in figure 15.
It can be seen that the Kalman filter robustly provides good results and can also handle a small amount of data ($N = 100$). For PINNs, on the other hand, at least $N = 250$ data points are necessary to deliver good results. When 500 data points are available, the PINN provides the most accurate predictions of both the states and the parameters for all compared constellations. The Kalman filter reacts more sensitively to noise than the PINNs. However, if the noise variance $\sigma_z^2 = 1$, then both methods have difficulty in estimating good parameters and states. One advantage of the Kalman filter is the low computational effort, which is reflected in the computation time. The parameter estimation is much faster, although a relatively expensive variant was chosen with the Monte Carlo method. Efficient parallelisation allows simultaneous evaluations of the Kalman filter for several random parameter combinations.
In this comparison, only the Lorenz system was considered. How suitable the two methods are for other systems of differential equations cannot be concluded from this. Using the SIR model, we want to check the generalisability. Furthermore, we want to test whether the two methods are also capable of estimating parameters from real data.

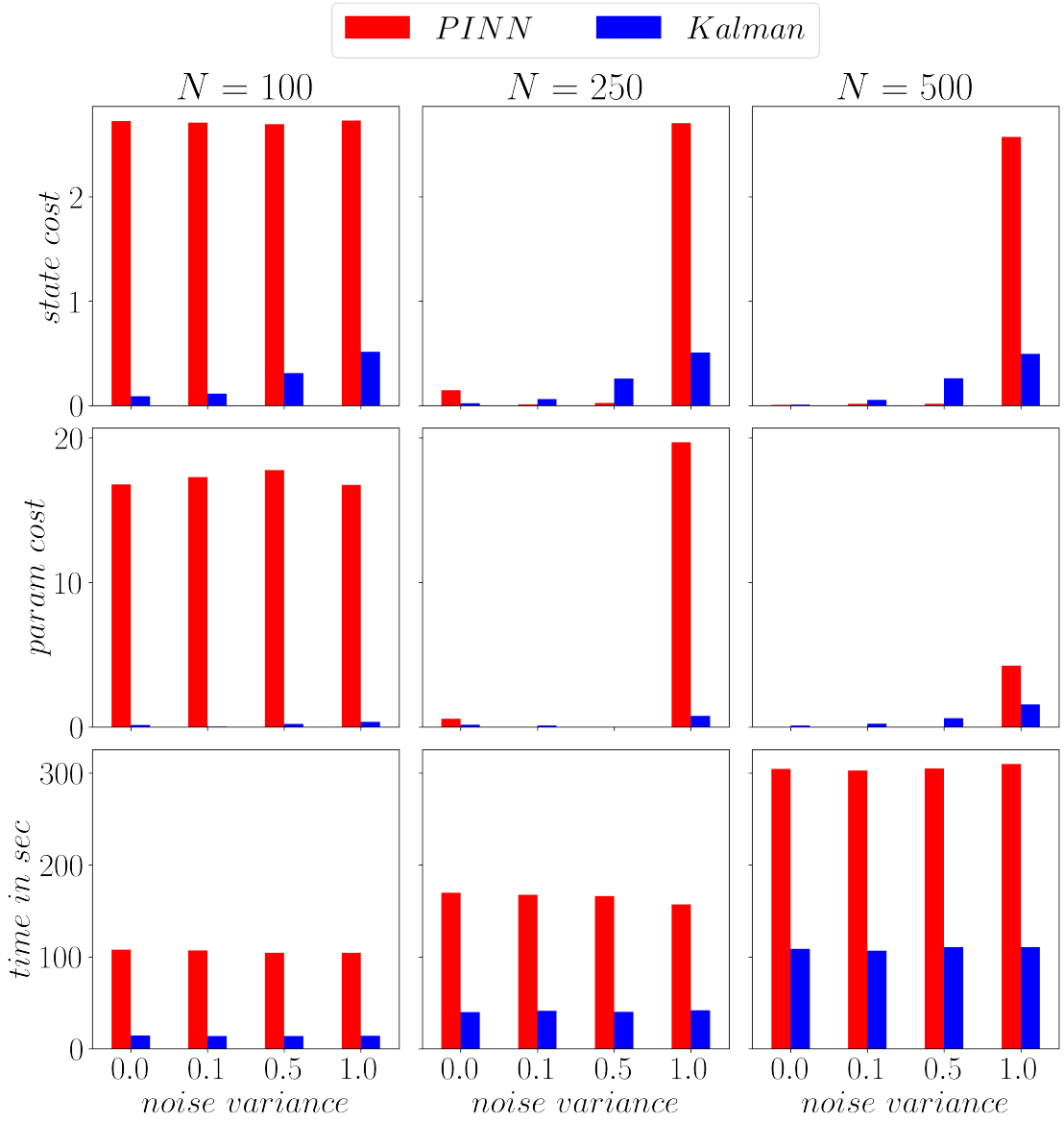

Figure 15: This figure is a graphical representation of the data from table 1 and 2. There, the two methods, PINN and the Kalman filter were applied to the same data, which were obtained from the Lorenz system with parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial condition $y(0) = (-5, 10, 25)^T$ was obtained. Here, the columns represent the number of data used $N \in \{100, 250, 500\}$. The rows indicate which value is being compared, i.e. whether the state cost, the parameter cost or the time in seconds. For each subplot the x-axis is identical and gives the variance σ_z^2 of the additive noise $z \sim \mathcal{N}(0, \sigma_z^2)$. This comes from the set $\{0, 0.1, 0.5, 1.0\}$. The red bars represent the PINNs, while the blue ones represent the Kalman filter.

$N \rightarrow$	100			250			500		
$\sigma_z^2 \downarrow$	state cost	param cost	time (in s)	state cost	param cost	time (in s)	state cost	param cost	time (in s)
0.0	2.723	16.781	107.805	0.146	0.577	169.927	0.006	0.017	304.614
0.1	2.709	17.282	107.150	0.011	0.033	167.492	0.016	0.017	302.727
0.5	2.694	17.782	104.346	0.025	0.030	166.168	0.017	0.027	305.250
1.0	2.729	16.746	104.461	2.704	19.704	157.230	2.572	4.247	310.074

Table 1: The results for the Lorenz system with parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial condition $y(0) = (-5, 10, 25)^T$ using PINNs. The first row indicates the number of data pairs N contained in the data set. The first column refers to the variance σ_z^2 of the normal distribution centred around 0, from which the noise term z was drawn and added to the true data y to obtain the noisy observations y^*. The "state cost" results from the average L_1-loss between the true values y and predicted values $\hat{y}$ over the entire data set. The L_1-loss between true and estimated parameters is called "param cost" in the table.

$N \rightarrow$	100			250			500		
$\sigma_z^2 \downarrow$	state cost	param cost	time (in s)	state cost	param cost	time (in s)	state cost	param cost	time (in s)
0.0	0.090	0.156	14.600	0.022	0.176	40.068	0.010	0.119	108.624
0.1	0.115	0.045	13.854	0.061	0.122	41.483	0.054	0.248	106.689
0.5	0.310	0.229	13.851	0.257	0.017	40.134	0.261	0.622	110.608
1.0	0.514	0.371	14.176	0.508	0.780	41.842	0.494	1.579	110.731

Table 2: The results for the Lorenz system with parameters $\rho = 8, \sigma = 16, \beta = 2$ and initial condition $y(0) = (-5, 10, 25)^T$ using the Kalman filter. The first row indicates the number of data pairs N contained in the data set. The first column refers to the variance σ_z^2 of the normal distribution centred around 0, from which the noise term z was drawn and added to the true data y to obtain the noisy observations y^*. The "state cost" results from the average L_1-loss between the true values y and forecast values y^F over the entire data set. The L_1-loss between true and estimated parameters is called "param cost" in the table.

5.3 Application: forecasting COVID-19 epidemic

In this section we will examine whether PINNs and the Kalman filter are also suitable for differential equations other than the Lorenz system. In addition, the application to real data will be checked. For this purpose, the SIR model is introduced, which is a system of differential equations used to model epidemics. Synthetic data are generated using the classical Runge-Kutta method. Subsequently, both PINNs and the Kalman filter are tested on them. Finally, data from the current COVID-19 epidemic in Germany are used to estimate parameters with both methods.

5.3.1 SIR model

The SIR model was introduced by [Kermack and McKendrick, 1927] to model an epidemic mathematically. In this process, a total population $N \in \mathbb{N}$ is divided into three groups: Susceptible $S \in \mathbb{N}$, infectious $I \in \mathbb{N}$ and recovered $R \in \mathbb{N}$. In general, $N = S + I + R$. The ratio of the individual groups to each other changes with time $t \in \mathbb{R}^+$. These individual groups can thus be regarded as a function over time. $S(t)$ indicates how many people are infectable at the time t. The number of infected persons at this time is given by $I(t)$. Persons who were already infected and can no longer be infected are considered recovered. How many people are recovered at the time t corresponds to $R(t)$. The change with time can be described by the following system of differential equations:

$$\begin{aligned} \frac{dS}{dt} &= -\frac{\beta SI}{N}, \\ \frac{dI}{dt} &= \frac{\beta SI}{N} - \gamma I, \\ \frac{dR}{dt} &= \gamma I. \end{aligned} \tag{5.7}$$

The parameter β is described in [Grimm et al., 2022] as contact rate. This indicates how many contacts an infected person has in a unit of time in which the disease is passed on. The parameter γ is defined there as follows: $\gamma = \frac{1}{\tau}$. τ indicates how long a person is potentially infectious, given in time units.
The SIR model is a simple model to forecast an epidemic. It is limited to the three groups susceptible, infected and removed. Special features that occur in the current COVID-19 epidemic, such as quarantine, infectious people who are asymptomatic or vaccinated people who have a lower infection rate are not modelled. How to deal with such more complex properties is described in [Tomochi and Kono, 2021].
To test the suitability of the two methods presented in this thesis for parameter estimation in an epidemic situation, synthetic data were first generated. For this, analogous to the Lorenz system, the classical Runge-Kutta method was used. A population size of 1000 was assumed. The further initial conditions are: $I(0) = 1$, $R(0) = 0$ and $S(0) = 999$. The parameters are $\beta = 0.3$ and $\gamma = \frac{1}{7}$. So each infected person passes

on the disease to 0.3 people on average every day and is infected for about 7 days. The data were simulated for a time horizon of 100 days and are shown in Figure 16.
A noise term z_k was now added to this data set similar to the Lorenz system. This is normally distributed, i.e. $z_k \sim \mathcal{N}(0, 5)$. On these observations $\{S^*, I^*, R^*\}$ first PINNs and then the Kalman filter can be tested.

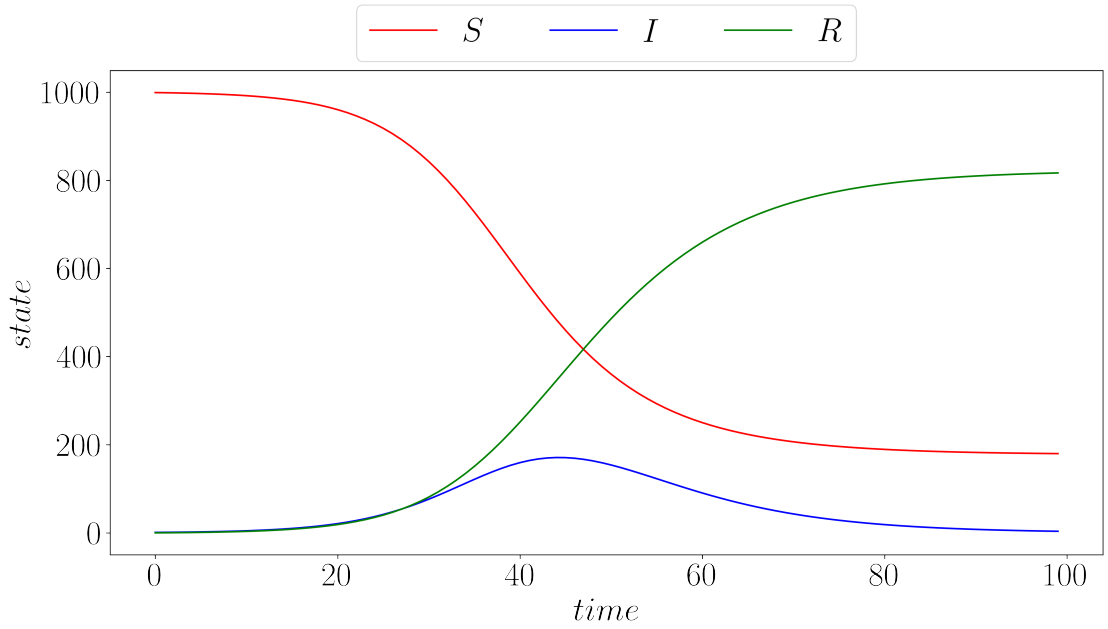

Figure 16: This figure shows the time course of the individual states of the SIR model over the Intervall $[0, 100]$. S stands for susceptible persons and is drawn as a red line. In addition, the infectious persons I are shown as a blue line and the recovered persons R as a green line. For this example, a population size of $N = 1,000$ was chosen. Initially, only one person was infected and none recovered, so the initial conditions were $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$. In this simulation, an infected person is contagious for an average of 7 days and has an average of 0.3 contacts per time step by which the disease is passed on. The parameters are therefore $\beta = 0.3$ and $\gamma = \frac{1}{7}$.

5.3.2 Physics-informed neural networks for the SIR model

To adapt a PINN to the SIR model, only a few changes are necessary compared to the Lorenz system. Firstly, the system of differential equations has to be implemented, so that DeepXDE ([Lu et al., 2020]) can access it. Secondly, the structure of the NN became more complex. The number of hidden layers was increased to 5, each containing 128 neurons. On the other hand, adjustments had to be made to the weights of the cost function defined in equation (3.37). The reason for this are the different value ranges of the various states. While the number of infected I in the example of figure 16 is

relatively small under 10 at the beginning, the value of susceptible S is in the region around $1,000$. With larger populations, the difference can become even greater. To control these differences, the weights w_a, w_b and w_c are adjusted so that the resulting cost is in the interval $(0, 1)$, as this is where the optimisation algorithms work best.
The results of the state estimation are shown in Figure 17. Here you can see that the states for I and R were estimated well, S somewhat worse. Overall, the deviation between prediction and true value is on average 19.29 over the entire time interval $[0, 100]$.

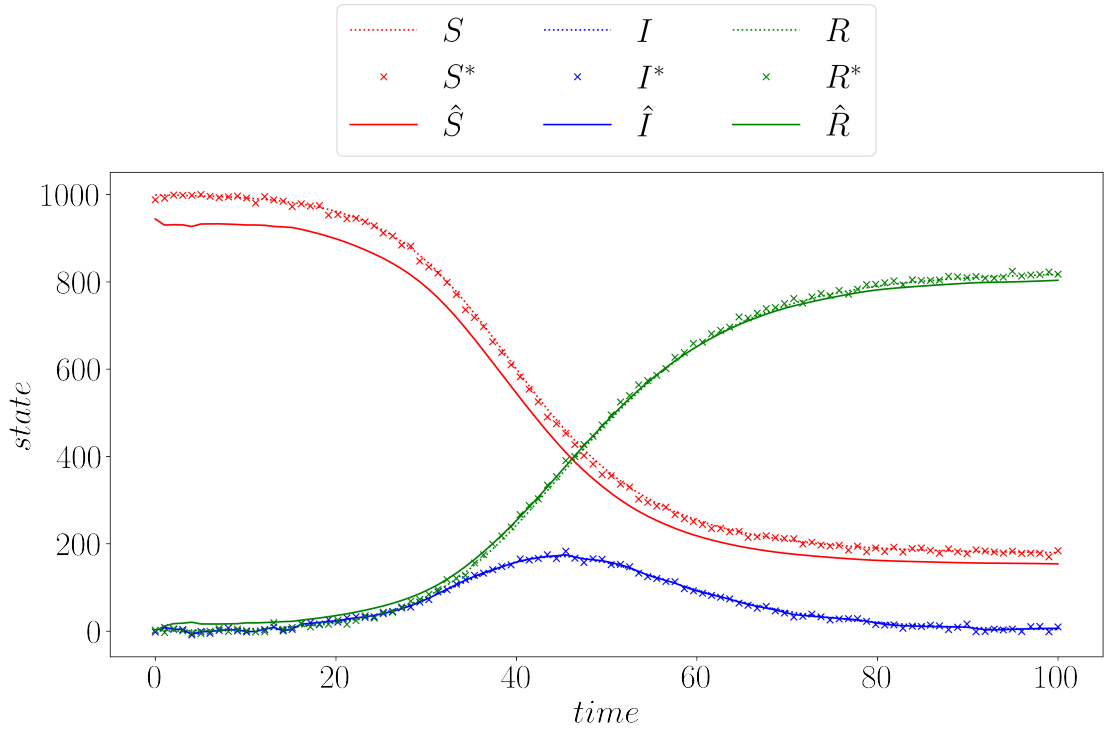

Figure 17: Here is the state prediction of a PINN for the SIR model with initial condition $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$ and parameters $\beta = 0.3$ and $\gamma = \frac{1}{7}$. The true values $\{S, I, R\}$ are shown as dashed lines. There were 100 observations $\{S^*, I^*, R^*\}$, marked as crosses. A random noise term $z \sim \mathcal{N}(0, 5)$ was added to the true observation. The distance between two observation times t_j and t_{j+1} was one time unit. The predictions $\{\hat{S}, \hat{I}, \hat{R}\}$ are plotted as solid lines. Just at the beginning, the observations of the infected $\hat{I}$ deviate from the true value I. Thus, due to the correlation between S, I and R in the differential equation, the PINN also wrongly estimates the susceptible S. This error occurs over the entire time interval.

Very good results were obtained in the parameter estimation. Figure 18 shows the parameter adjustment over the different optimisation iterations. In summary, it can be said that the estimates of both parameters after the last epoch deviate only slightly from the true value. For β the difference is 0.012, for γ it is even about 0.007.

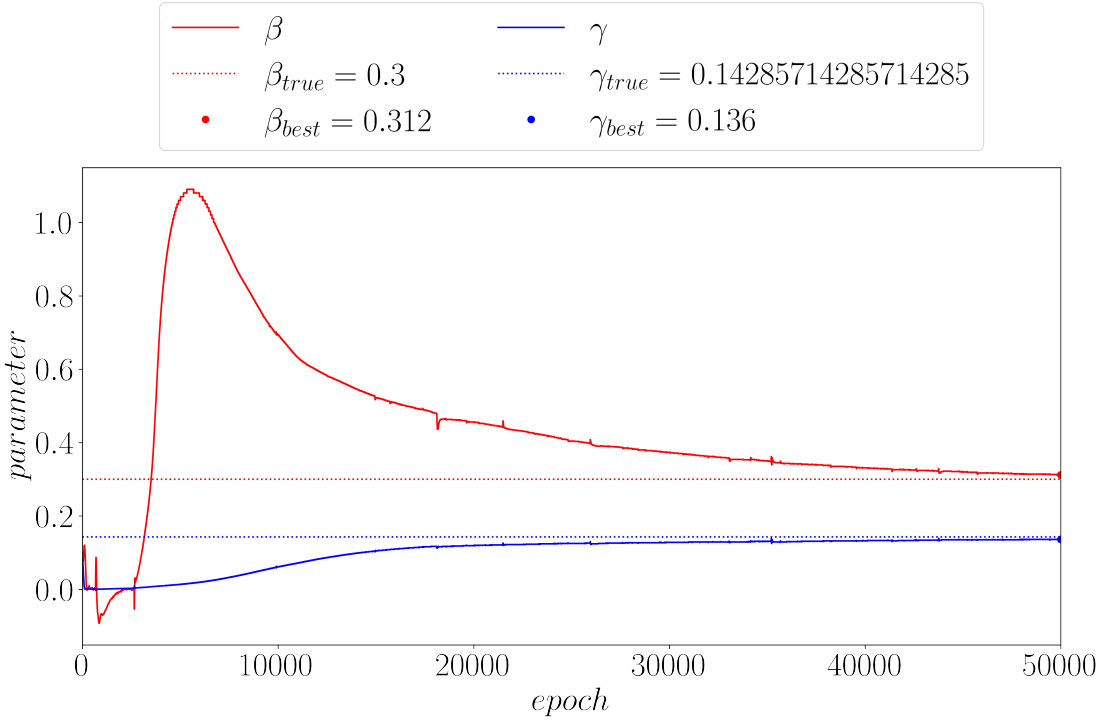

Figure 18: This figure shows the parameter estimate of a PINN for the SIR model over the training epochs. The SIR model with initial condition $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$ and parameters $\beta = 0.3$ and $\gamma = \frac{1}{7}$ was simulated by the classical Runge-Kutta method. The true parameters β and γ are shown as dashed lines in the graph. It can be seen that the estimates (solid lines) approximate the true parameters from about epoch $7,000$ onwards. However, γ is already well approximated from the $20,000$ epoch on. For β it takes almost until the last optimisation step.

Now we will test the Kalman filter on the same data. What adjustments need to be made for this will be considered in the next section. Furthermore, the predictions and estimates are evaluated and compared with the values of the PINN.

5.3.3 Kalman filter for the SIR model

To implement the Kalman filter for the SIR model, it is necessary to determine the forecast operator $\mathcal{M}_j$, or more precisely its linearisation M_j. As with the Lorenz system, the explicit Euler method is used for this. The following then applies

$$
\begin{aligned}
y(t_{j+1}) &= y(t_j) + h \cdot F(t_j, y(t_j)) \\
&= \begin{pmatrix} S(t_j) \\ I(t_j) \\ R(t_j) \end{pmatrix} + h \cdot \begin{pmatrix} -\frac{\beta S(t_j) I(t_j)}{N} \\ \frac{\beta S(t_j) I(t_j)}{N} - \gamma I(t_j) \\ \gamma I(t_j) \end{pmatrix} \\
&= \begin{pmatrix} S(t_j) - h\frac{\beta S(t_j) I(t_j)}{N} \\ I(t_j) + h(\frac{\beta S(t_j) I(t_j)}{N} - \gamma I(t_j)) \\ R(t_j) + h\gamma I(t_j) \end{pmatrix} \\
&= \underbrace{\begin{pmatrix} 1 & h\frac{\beta S(t_j)}{N} & 0 \\ h\frac{\beta I(t_j)}{N} & 1 - h\gamma & 0 \\ 0 & h\gamma & 1\beta \end{pmatrix}}_{=:M_j} \cdot \underbrace{\begin{pmatrix} S(t_j) \\ I(t_j) \\ R(t_j) \end{pmatrix}}_{y(t_j)}.
\end{aligned} \tag{5.8}
$$

The rest of the algorithm remains almost unchanged compared to the Lorenz system. A Monte Carlo method is again used to determine the optimal parameters.
As can be seen in figure 19, the state estimation is better than that of PINN. This can also be expressed in numbers, the deviation of the true states from the forecast state of the Kalman filter is on average 4.27.
The parameter estimation also works very well. This can be seen in figure 20. For β the distance between estimate and true value is 0.016, for γ about 0.008. Thus, the deviations are larger than those of the PINN, but only minimal.
Thus, both methods are also well suited for the simulated data of the SIR model. Finally, we check the generalisability to real data. For this purpose, we use data on the COVID-19 epidemic from Germany. These can be modelled at least approximately by the SIR model. Of essential interest is which parameters are estimated by the two methods.

5.3.4 Coronavirus disease 2019 in Germany

In order to be able to transfer the COVID-19 pandemic figures for the SIR model, the first step was to research the total population. According to [Bundesamt, 2022], $83,200,000$ people currently live in Germany, i.e. $N = 83,200,000$. Tables could be downloaded from [Robert-Koch-Institute, 2022] containing both the current active cases and the cumulative number of all cases.
The number of currently infected persons I can be extracted from the two linked tables. The numbers from 06/05/2020 to 10/09/2021 can be found here: `https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Daten/Fallzahlen_Kum_Tab_Archiv.xlsx`. The subsequent values up to the current day can be taken from this linked table: `https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Daten/Fallzahlen_Kum_Tab_aktuell.xlsx`.
The number of cumulative cases from 25/02/2020 to present can be seen in the following table: `https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Daten/`

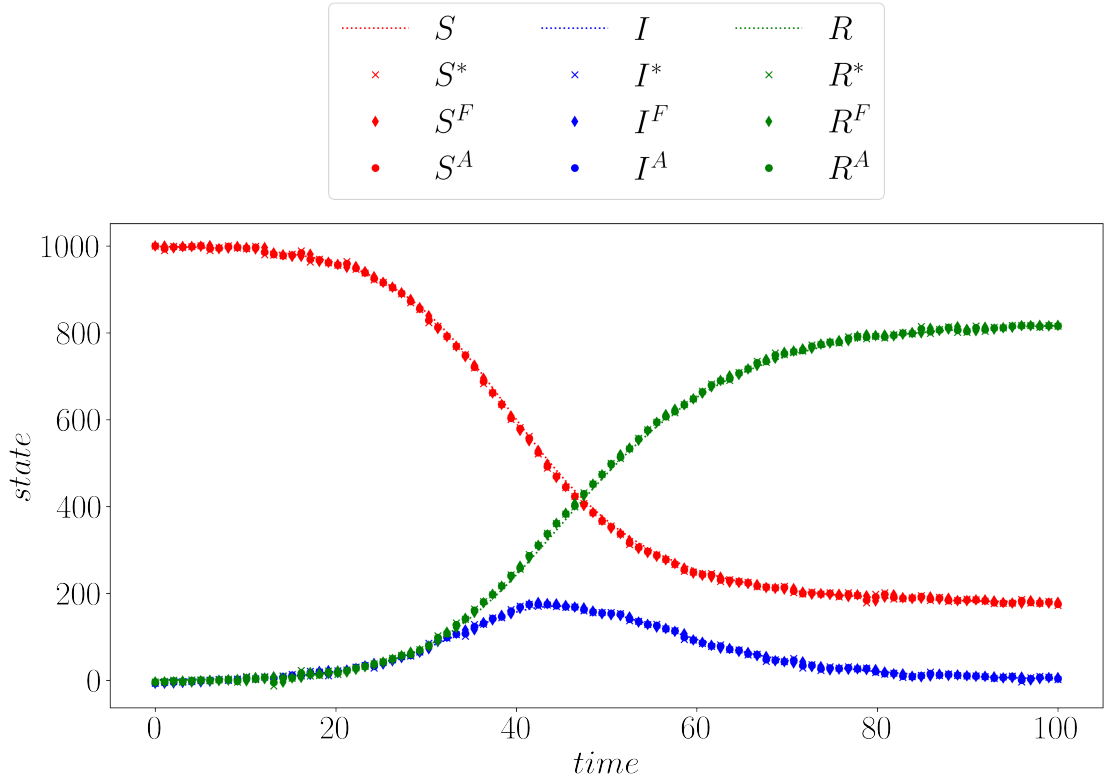

Figure 19: Here is the state estimate of the Kalman filter for the SIR model with initial condition $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$ and parameters $\beta = 0.3$ and $\gamma = \frac{1}{7}$. The true values $\{S, I, R\}$ are shown as dashed lines. There were 100 observations $\{S^*, I^*, R^*\}$, marked as crosses. A random noise term $z \sim \mathcal{N}(0, 5)$ was added to the true observation. The distance between two observation times t_j and t_{j+1} was one time unit. The forecast states $\{S^F, I^F, R^F\}$ are plotted as diamonds and the analysis states $\{S^A, I^A, R^A\}$ as dots. It can be seen that at no time do large deviations occur between the forecast and the true state.

`Fallzahlen_Gesamtuebersicht.xlsx`. The number of recovered people R on a day can be calculated by cumulative case number minus current case number on that day. Afterwards, only the number of susceptible people S has to be determined by $S = N - I - R$. The data for this thesis was downloaded on 12/04/2022, so only values from the COVID-19 epidemic up to 11/04/2022 were used in the computations. The time course from 06/05/2020 to 11/04/2022 of the individual groups S, I and R of the COVID-19 epidemic in Germany can be seen in Figure 21.
Training a PINN on this data with the SIR model as the underlying differential equation was not successful. The PINN could neither predict the states nor estimate the parameters. More precisely, the predictions were the same for each time step and for

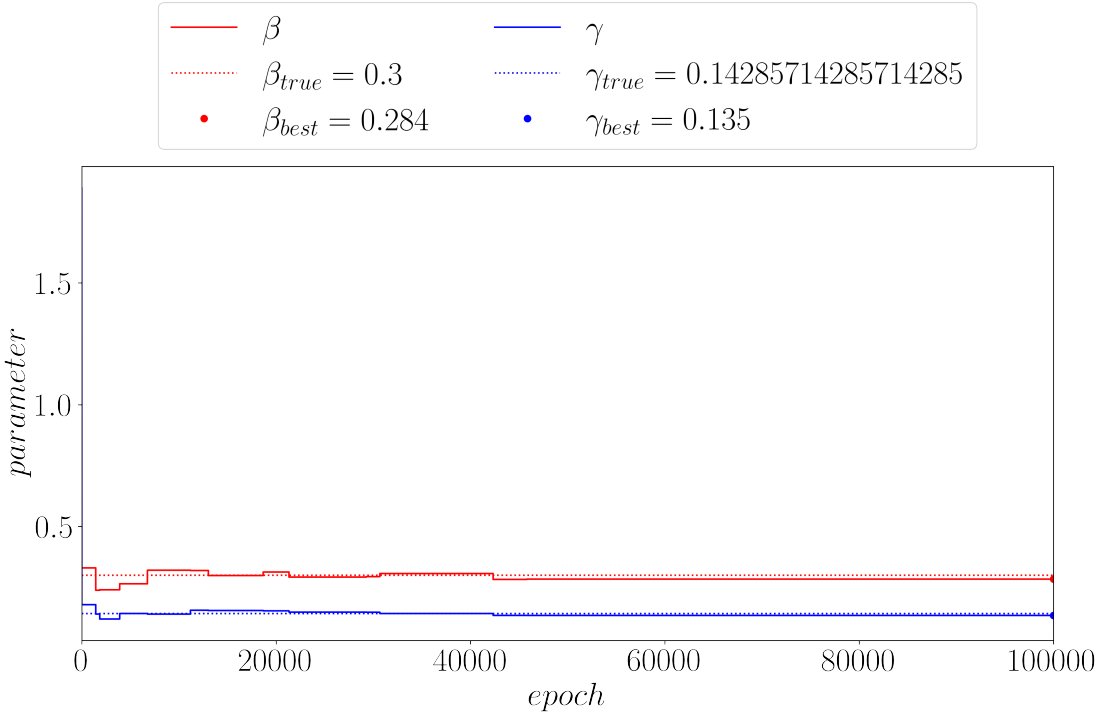

Figure 20: This figure shows the parameter estimate of the Kalman filter for the SIR model over the Monte Carlo epochs. The SIR model with initial condition $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$ and parameters $\beta = 0.3$ and $\gamma = \frac{1}{7}$ was simulated by the classical Runge-Kutta method. The true parameters β and γ are shown as dashed lines in the graph. It can be seen that the estimates (solid lines) approximate the true parameters very well even in the first iterations.

the parameters the estimates approached 0. On the one hand, this may be due to the far-flung value ranges of the individual variables. On the other hand, it is likely that the data cannot be modelled well enough by the SIR model. As a result it is not possible to train a NN that can approximate the solution of the differential equation.
Looking at the Kalman filters, it can be seen that with this method both the states can be predicted and the parameters can be plausibly estimated. The parameter estimation is shown in Figure 22. The parameter β of the SIR model was estimated here to be approximately 0.2. This means that in the COVID-19 epidemic, each infected person had 0.2 contacts per day where the disease was passed on. The estimate for γ was 0.16, i.e. after an infection one is infectious for about 6.25 days.

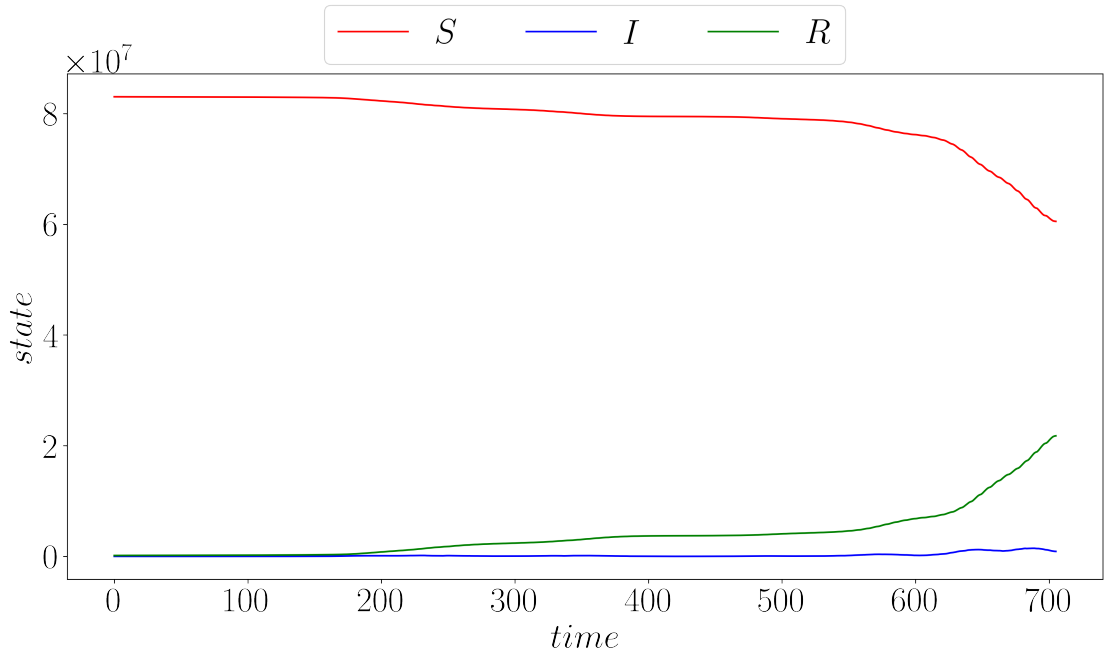

Figure 21: Here the real data from the COVID-19 epidemic in Germany in the period from 06/05/2020 to 11/04/2022 is visualised. The data was obtained from [Robert-Koch-Institute, 2022].

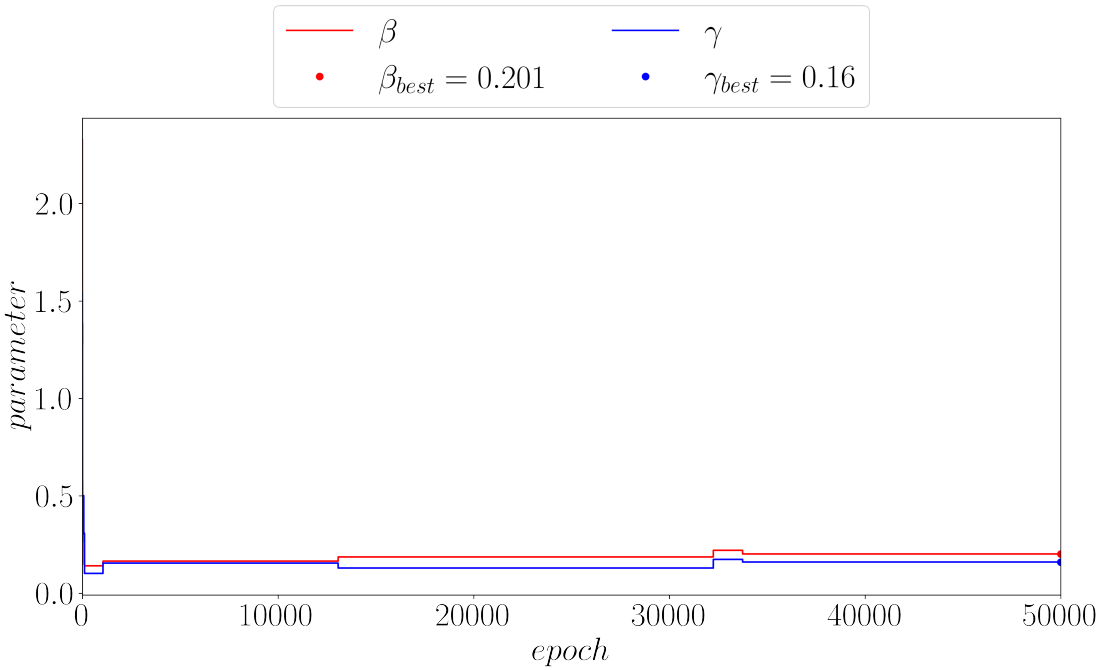

Figure 22: This figure shows the parameter estimation for the SIR model using the Kalman filter and the COVID-19 data from Germany. The course of the best parameters over the optimisation epochs is plotted here. The red line is assigned to β and the blue one to γ.

6 Conclusion

In this thesis we studied parameter estimation for differential equations and compare a DL method with a data assimilation technique. A differential equation is a mathematical problem of the type

$$y' = F(x, y), \quad \text{with } y' = \frac{dy}{dx} \in \mathbb{R}^n \text{ and } F : I \times \mathbb{R}^n \to \mathbb{R}^n.$$

If an initial condition $y(x_0) = y_0$ is known in addition to this system of equations, it is an initial value problem. Using Runge Kutta methods, numerical approximations of a solution for such problems can be found. In the context of this thesis, it is not the solving of differential equations that is of interest, but the estimation of parameters $\lambda \in \mathbb{R}^{n_{param}}$. This can be interpreted as an inverse problem, i.e. using data to find the parameters that best fit. To show how such data can look like, the Lorenz system was introduced. This is a system of differential equations defined in equation (2.10). An example of possible observations from which parameters are to be estimated is given in figure 3.
One way to do this is to use a PINN, a DL method. PINNs are based on FNNs, the simplest NNs. They consist of different layers that are connected with each other. Each FNN f has exactly one input and one output layer. The number of hidden layers can be adapted to the problem, as the number of neurons per layer. The connections between two subsequent layers can also be written as matrix multiplication; the following applies

$$f^l(x) := \boldsymbol{\sigma}(\mathbf{W}^l x + \mathbf{b}^l).$$

Here, x can be interpreted as the neurons of the previous layer. $\mathbf{W}_l$ stands for the weight matrix and $\mathbf{b}_l$ for the bias of the current layer. The function $\boldsymbol{\sigma}$ that is applied to the result of the matrix operations is called the activation function. This provides non-linearity in the FNNs so that even complicated problems can be solved. In the context of PINNs, the hyperbolic tangent is usually chosen as the activation function. To optimise the FNN, a cost function $J(\boldsymbol{\theta})$ is established. As parameter $\boldsymbol{\theta}$ of a FNN all entries of the weight matrices of all layers are summarised. In the cost function, the prediction of data is compared with the true label. The aim is to find the parameters that minimise the cost function. To achieve this, optimisation algorithms are needed. These are usually based on gradient methods. Therefore, it is necessary to be able to efficiently compute the derivatives of a NNs. The concept that makes this possible is called AD. Furthermore, it is useful not to compute the gradient from all data individually or at once, therefore mini-batches have been developed. Here, an optimisation iteration is performed for a smaller data set. From this, some first-order optimisation algorithms can be created, such as gradient descent, SGD and ADAM. It is also possible to use the information of the second derivative (Hessian matrix) in addition to the first derivative. However, this is very complex to compute. To avoid this, an approximation of the Hessian matrix can be used alternatively like in the BFGS algorithm. In order to consider differential equations in this optimisation, PINNs were developed. The basic idea is to add two more loss terms to the cost function, one representing the differential equation L_a and

one for the initial and boundary conditions L_b. The resulting cost function is given by

$$J_D(\boldsymbol{\theta}, \lambda) = w_a \frac{1}{|D_a|} \sum_{t \in D_a} L_a(\boldsymbol{\theta}, \lambda, t) + w_b \frac{1}{|D_b|} \sum_{t \in D_b} L_b(\boldsymbol{\theta}, \lambda, t) + w_c \frac{1}{|D_c|} \sum_{t \in D_c} L_c(\boldsymbol{\theta}, \lambda, t).$$

Here L_c represents the data. Now it is possible to optimise both the parameters of the NN $\boldsymbol{\theta}$ and the differential equation λ simultaneously. This optimisation of λ can be interpreted as parameter estimation.

Data assimilation techniques can also be used to estimate parameters of differential equations. A frequently used method is the Kalman filter, where the model is compared with the observation in regular time steps, to obtain a more accurate model. More precisely, the Kalman filter determines the forecast from a given state using a forecast operator $\mathcal{M}$. Afterwards forecast and observation are combined to the analysis state, so that the expected error is minimal. To achieve this, the Kalman gain is needed. From the just computed analysis state a new forecast is determined by $\mathcal{M}$ and the procedure starts again. The Monte Carlo method is used to estimate the parameters. Different parameter combinations are tried out and the one that best fits the data is selected.

Compared to PINNs, the modification of the Kalman filter to a different system of differential equations is more complicated, since the forecast operator $\mathcal{M}$ has to be determined anew each time. For PINNs, only the system of differential equations itself needs to be implemented for the DeepXDE package.

Comparing the two methods on identical data from the Lorenz system, it was seen that the Kalman filter was more robust to variations in the data set size and the accuracy of the observations. In addition, the Kalman filter estimates the parameters much faster. However, if sufficient data are available, the estimated parameters of the PINN are somewhat more accurate.

Similar results were also observed in the simulations of the SIR model. The SIR model from equation (5.7) can be used to forecast epidemics. The PINN could estimate the parameters of the SIR model very well, the states not quite so well. The Kalman filter could do both.

Using real data, such as the COVID-19 epidemic in Germany, to estimate parameters, only the Kalman filter was able to determine plausible values. However, the estimates cannot be evaluated accurately because the true parameters are unknown. But the estimates of the PINN approach 0, while with the Kalman filter the results were in the expected range and their interpretation was comprehensible.

In conclusion, both methods use data and the underlying model to gain insights. However, PINNs is more sensitive to the observations in order to produce good results.

References

[Bottou et al., 2018] Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. arXiv: 1606.04838.

[Bundesamt, 2022] Bundesamt, S. (12.04.2022). Bevölkerungsstand: Amtliche einwohnerzahl deutschlands 2021. `https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Bevoelkerungsstand/_inhalt.html`.

[Farlow, 2006] Farlow, S. J. (2006). *An Introduction to Differential Equations and Their Applications*. Dover Books on Mathematics. Dover Publications.

[Freitag, 2020] Freitag, M. A. (2020). Numerical linear algebra in data assimilation. *GAMM Mitteilungen*, 43(3).

[Freitag and Potthast, 2013] Freitag, M. A. and Potthast, R. W. E. (2013). Synergy of inverse problems and data assimilation techniques. *Large Scale Inverse Problems*.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[Grewal and Andrews, 2001] Grewal, M. S. and Andrews, A. (2001). Kalman filtering: theory and practice using matlab. *John Wiley and Sons, Inc.*

[Grimm et al., 2022] Grimm, V., Heinlein, A., Klawonn, A., Lanser, M., and Weber, J. (2022). Estimating the time-dependent contact rate of sir and seir models in mathematical epidemology using physics-informed neural networks. *Electronic Transactions on Numerical Analysis*, 56:1–27.

[Hadamard, 1923] Hadamard, J. (1923). *Lectures on the Cauchy Problem in Linear Partial Differential Equations*. Yale University Press,.

[Hairer et al., 1993] Hairer, E., Wanner, G., and Nørsett, S. P. (1993). *Solving Ordinary Differential Equations I*. Springer Series in Computational Mathematics. Springer Berlin, Heidelberg, 2 edition.

[Higham and Higham, 2019] Higham, C. F. and Higham, D. J. (2019). Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860 – 891.

[Humpherys et al., 2012] Humpherys, J., Redd, P., and West, J. (2012). A fresh look at the kalman filter. *SIAM Review*, 54.

[Janocha and Czarnecki, 2017] Janocha, K. and Czarnecki, W. M. (2017). On loss functions for deep neural networks in classification. *Schedae Informaticae*, 25:49–59.

[Kalman, 1960] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82:35–45.

[Kermack and McKendrick, 1927] Kermack, W. O. and McKendrick, A. G. (1927). A contributions to the mathematical theory of epidemics. *Proc. R. Soc. Lond. A*, 115:700–721.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. L. (2017). Adam: A method for stochastic optimization. *arXiv*.

[Kirsch, 2021] Kirsch, A. (2021). *An Introduction to the Mathematical Theory of Inverse Problems*, volume 120 of *Applied Mathematical Sciences*. Springer Nature Switzerland AG, 3rd edition.

[Law et al., 2015] Law, K., Stuart, A., and Zygalakis, K. (2015). *Data Assimilation - A Mathematical Introduction*, volume 62 of *Texts in Applied Mathematics*. Springer Cham.

[Lorenz, 1963] Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20:130–141.

[Lu et al., 2020] Lu, L., Meng, X., Mao, Z., and Karniadakis, G. E. (2020). Deepxde: A deep learning library for solving differential equations. *arXiv*.

[Luengo et al., 2020] Luengo, D., Martino, L., Bugallo, M., Elvira, V., and Särkkä, S. (2020). A survey of monte carlo methods for parameter estimation. *EURASIP Journal on Advances in Signal Processing*, 25.

[Margossian, 2019] Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. *Wiley: WIREs Data Mining and Knowledge Discovery*, 9(4).

[Markidis, 2021] Markidis, S. (2021). The old and the new: Can physics-informed deep-learning replace traditional linear solvers? arXiv.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization Numerical Optimization Numerical Optimization*. Springer Science+Business Media, 2nd edition.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[Raissi et al., 2019] Raissi, M., Perdikaris, P., and Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, pages 686–707.

[Reich and Cotter, 2015] Reich, S. and Cotter, C. (2015). *Probabilistic Forecasting and Bayesian Data Assimilation*. Cambridge University Press.

[Robert-Koch-Institute, 2022] Robert-Koch-Institute (12.04.2022). Neuartiges coronavirus. `https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/nCoV.html`.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533 – 536.

[Sharma et al., 2020] Sharma, S., Sharma, S., and Athaiya, A. (2020). Activation function in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316.

[Strehmel et al., 2012] Strehmel, K., Weiner, R., and Podhaisky, H. (2012). *Numerik gewöhnlicher Differentialgleichungen*, volume 2. Springer Spektrum.

[Tomochi and Kono, 2021] Tomochi, M. and Kono, M. (2021). A mathematical model for covid-19 pandemic - siir model: Effects of asymptomatic individuals. *Journal of General and Family Medicine*, 22:5–14.

[Welch and Bishop, 1995] Welch, G. and Bishop, G. (1995). An introduction to the kalman filter. *Chapel Hill, NC, USA*.

Appendix

The source code of the python programs implemented for this thesis can be found here: `https://github.com/JanikRaue/parameter-estimation-for-differential-equations`. Basically there are 4 python files in the /src folder:

- **data.py** with the classes *DifferentialEquation* and subclasses *LorenzSystem*:

  ```
  from src.data import LorenzSystem
  lorenz = LorenzSystem(param, initial_condition, number_of_data, end_time)
  lorenz.set_data()
  lorenz.plot_data()
  ```

  and *SirModel*:

  ```
  from src.data import SirModel
  sir = SirModel(param, initial_condition, number_of_data, end_time)
  sir.set_data()
  sir.plot_data()
  ```

- **pinn.py** with the class *PhysicsInformedNeuralNet*:

  ```
  from src.pinn import PhysicsInformedNeuralNet
  pinn = PhysicsInformedNeuralNet(differential_equation)
  loss_history = pinn.train(epochs)
  prediction = pinn.predict()
  ```

- **kalman.py** with the class *KalmanFilter*:

  ```
  from src.kalmanfilter import KalmanFilter
  kf = KalmanFilter(differential_equation)
  train_param = kf.train(epochs)
  analysis, forecast = kf.predict()
  ```

- **visualisation.py** to plot the results:

  ```
  from src.visualisation import plot_progress, plot_param_process
  plot_progress(time, observation, true, prediction, analysis)
  plot_param_process(true_params, param_progres)
  ```

In the /notebooks folder are jupyter-notebook files as usage examples of PINNs and the Kalman filter for the Lorenz system, the SIR Model, as well as the COVID-19 data. There is also a file comparing the two methods on the Lorenz system and one plotting the results of this comparison.