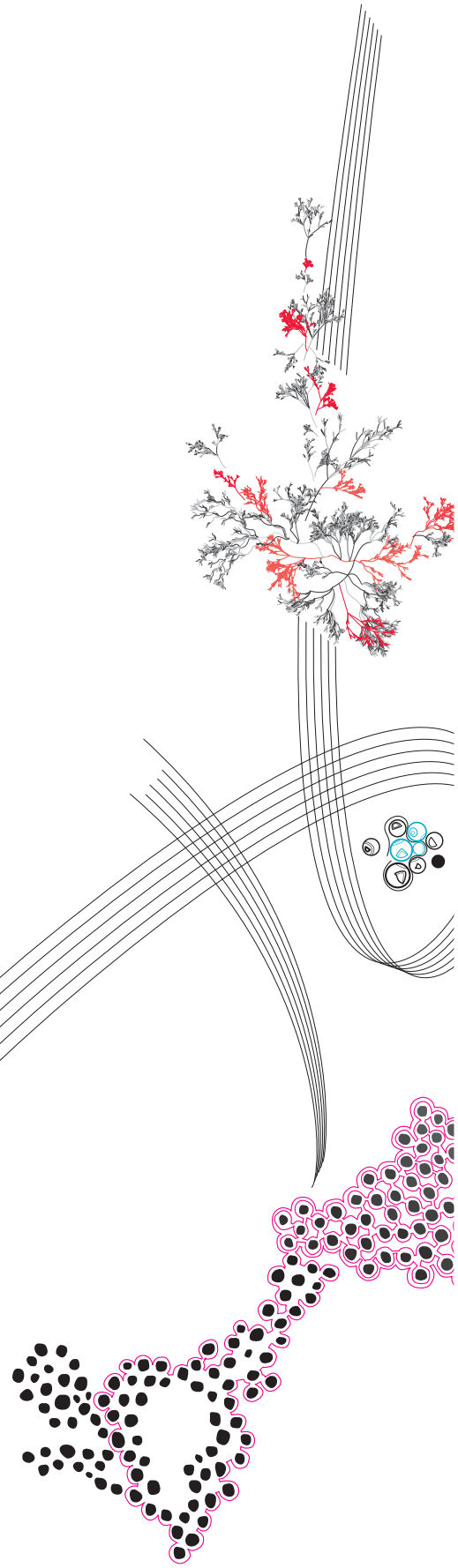

BSc Thesis Applied Mathematics

Approximating differential equations using neural ODEs

L. R. ten Klooster

Supervisors: MSc Len Spek, dr. Christoph Brune

July, 2021

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This thesis has been written as a final part of my bachelor Applied Mathematics at the University of Twente. I have enjoyed doing research and writing an article about deep learning, a subject that was completely new to me. Working on this project gave me fresh insights and helped me in my learning process.

There are however some people I could not have achieved this without. Most important of all are my supervisors, Len Spek and Christoph Brune, who were always there to guide me, steer me in the right direction, or simply give me some encouraging advice. I thank them for the time and effort they spend in helping me.

Besides this, I would like to thank my friends, housemates, and family for supporting me when I needed it. The distractions they gave me were also very much welcomed. Especially Luka deserves a thank-you for her endless support and understanding.

Finally, I hope you, the reader, will enjoy reading this article and find it, if applicable, helpful.

Linda ten Klooster
Enschede, 2021

Approximating differential equations using neural ODEs

Linda ten Klooster

July, 2021

Abstract

Differential equations play an important role in modelling all kinds of phenomena in many disciplines. An example is the prey-predator model, also known as the Lotka-Volterra equations. However, sometimes it is the case that there is not enough information known to construct an explicit model for a problem. This study focuses on approximating differential equations using neural ordinary differential equations (neural ODEs), such that data can still be used in models without having to construct an explicit system of ODEs. Neural ODEs are a recent development that combine deep learning with the structure of differential equations. We train our network for various different sets of initial conditions, after which we see how well our network performs during testing on other initial conditions. We do this for the Lotka-Volterra equations and the Van der Pol oscillator. The presented results focus on the performance of the model dependent on the amount of training points and the amount of epochs used during training. We conclude that a neural ODE can make for an accurate approximation of the differential equations, but there are some uncertainties of the influence of the training set and the kind of differential equations that are used for the data set.

Keywords: ordinary differential equations (ODEs), deep learning, neural ODEs

Contents

1 Introduction

Differential equations are of importance in many fields of science. They came into play when Isaac Newton and Gottfried Wilhelm Leibniz became involved in the invention of calculus. Since then, many disciplines have adopted differential equations as a part of their mathematical toolbox. An example is the predator–prey model, also known as the Lotka-Volterra equations. This system of differential equations describes the dynamics within a biological system and can be used by biologists to understand the changes in populations for different species [1]. Another example is the van der Pol oscillator, a common differential equation used in many fields of science, amongst which physics [2], electrical engineering [3] and biology [4]. It is clear that differential equations are part of several fields of science, and therefore it is beneficial to study their behaviour within neural networks. However, there is not always enough information to construct an explicit model for a problem at hand. Without a model, it is not possible to do any extrapolation or interpolation of the data available. This thesis focuses on tackling this problem with the help of deep learning. There are already some methods that try to approach this problem from different angles. An example is SINDY [5], an algorithm that is developed to investigate nonlinear dynamical systems using regression methods. Another recently published method takes a look at stochastic differential equations, and uses the Euler-Lagrange equation to estimate the diffusion coefficient of these equations [6].

We introduce a method that makes use of an artificial neural network. An artificial neural network is inspired by the biological neural networks. Deep learning is a part of machine learning that depends on this structure of a neural network, using layers and neurons. A technical difficulty in training these neural networks is performing backpropagation. Backpropagation is an important part of the process of training the network. It computes the gradient of an objective function with respect to the parameters of the network. It is effective, but costs a lot of memory and can introduce additional numerical error. A recently developed method is the neural ordinary differential equation (neural ODE), which can train a model with fewer parameters, and which is a lot more memory-efficient and less complex than residual neural networks are [7]. This thesis will focus on neural differential equations and how they can be used to make an approximation of differential equations. We will look into the effect of training the neural ODE and testing it with different initial conditions.

First, some theory related to this thesis is discussed. This includes material concerning ODEs, deep learning, and their connection to neural ODEs. Next, a description of our method that approximates differential equations will follow, with analysis of the performance of the model when it is tested with other initial conditions. Finally, the results are presented and discussed.

2 Ordinary Differential Equations

ODEs can be used to describe many different phenomena, appearing in many fields of science, as we saw in the introduction. An ODE is an equation containing both one or more functions of a variable and its derivative [8]. We will look at numerical solutions, not analytical, since these are of bigger use for our study. An ODE can be of any order; an n^{th} order differential equation will include the n^{th} derivative of a variable. For now we will stick to first-order, nonlinear, autonomous differential equations, which are of the form

$$\frac{\partial x(t)}{\partial t} = f(x(t)). \tag{1}$$

We will use the Euler method [8], an explicit method for numerical integration of an ODE, to make a numerical approximation of (1). Let's say we have initial value $x(t_0) = x_0$. The idea behind the Euler method is that starting from the initial condition, we can find the tangent line at $t_0 \in \mathbb{R}$. By taking a very small step along the tangent line, we can find take a next point, and define a new tangent line along which we will take another small step. Continuing this process, we will get a polygonal curve that will not diverge too far from the actual curve. A key element of this method is taking the step sizes small enough. Now, one step of the Euler method is defined by

$$x_{n+1} = x_n + hf\left(t_n, x_n\right). \tag{2}$$

Here h defines the step size and $t_n = t_0 + nh$, and (2) shows us how to get from t_n to $t_n + 1 = t_n + h$. The value of y_n is approximately the solution of our ODE (1) at time t_n.

3 Deep learning

Deep learning is a method that makes use of artificial neural networks. Artificial neural networks are computational systems consisting of many different layers, and they are inspired by biological neural networks. A artificial neural network tries to mimic a biological neural network through inputs, weights, and biases. In this section we will discuss how these matters play a role in deep learning. The rest of this section is for a large part based on [9]. We will represent a neural network in the following form:

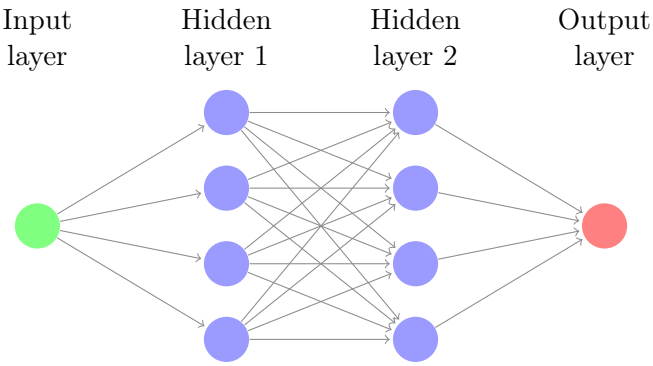

FIGURE 1: Neural network.

The neural network in Figure 1 has four layers: an input layer and an output layer (which are always present in a neural net) and two hidden layers. The hidden layers exist of five neurons, which is taken randomly. A neural network mostly has many more layers and many more neurons per layer. The amount of neurons does not have to be equal in all hidden layers.

3.1 Forward pass

As input one takes the data that we want our neural network to train, let us call this data $x \in \mathbb{R}^{n_1}$, where n_1 denotes the amount of neurons in the input layer. Dependent on this input, each neuron in every layer l will output a real number. We then put these numbers together in a vector, which we call $a^{[l]}$, of dimension n_l, where n_l is the amount of neurons of layer l. We let $a_j^{[l]}$ denote the output, or activation, from neuron j at layer l. At the

next layer, $l+1$, a will be multiplied by a weight matrix W and a bias vector b, which will be put into an activation function. The weight matrix has dimensions $n_{l+1} \times n_l$, so the number of rows in W matches the number of neurons at the current layer, whereas the number of columns matches the number of neurons at the previous layer which output the vector a. The bias vector b has dimension $n_l + 1$, and together this gives us $Wa + b$. After the bias is added, we use this as an input for a non-linear activation function. An activation function mimics what a neuron would behave like in the brain: it either fires if the input is large enough, or it remains inactive if the output is too small. For now we will use a sigmoid function as our activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{3}$$

which will give an output of one if the input is large enough, and an output of zero otherwise. There are several choices possible as activation functions, but for now we will stick to the sigmoid function. If the reader is interested in more background about different activation functions and their performances, see [10] or [11] for example. Now, we define

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \quad \text{for } l = 2, 3, \ldots, L, \tag{4}$$

to be called the weighted input at layer l for neuron j, such that in general the output of layer l becomes

$$\begin{aligned} a^{[1]} &= x, \\ a^{[l]} &= \sigma(z^{[l]}), \text{ for } l = 2, 3, ..., L. \end{aligned} \tag{5}$$

Let us look at how this unfolds in the example presented in Figure 1. The input data for layer one is of the form $x \in \mathbb{R}$, since layer one has one neuron. The weights for layer two are presented by $W^{[2]} \in \mathbb{R}^{4\times 1}$ and the bias for layer two is presented by $b^{[2]} \in \mathbb{R}^4$, since layer two has four neurons. Then the output of layer two has the form

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^4. \tag{6}$$

Layer three has four neurons and an input in $\mathbb{R}^4$, so its weights and biases are presented by $W^{[3]} \in \mathbb{R}^{4\times 4}$ and $b^{[3]} \in \mathbb{R}^4$, respectively. Then the output of layer three has the form

$$\sigma(W^{[3]}(\sigma(W^{[2]}x + b^{[2]})) + b^{[3]}) \in \mathbb{R}^4. \tag{7}$$

Since the output layer has one neuron and an input in $\mathbb{R}^4$, its weight and bias are presented by $W^{[4]} \in \mathbb{R}^{1\times 4}$ and $b^{[4]} \in \mathbb{R}$, respectively, and the output of this layer becomes

$$\sigma(W^{[4]}(\sigma(W^{[3]}(\sigma(W^{[2]}x + b^{[2]})) + b^{[3]})) + b^{[4]}) \in \mathbb{R}. \tag{8}$$

This is also the overall output of the neural network depicted in Figure 1. We will call (8) $F(x)$, which is the learning function of this specific neural network with input x.
Now in order to train our network, we will need a loss function which will adjust the parameters in our network, which are the weights and biases. This loss function is dependent on the amount of training points. Say we have N training points in our data, $\{x^{\{i\}}\}_{i=1}^N$ in $\mathbb{R}^{n_1}$, for which we have target outputs $\{y(x^{\{i\}})\}_{i=1}^N$ in $\mathbb{R}^{n_L}$. This gives us the following loss function

$$\mathcal{L}(w, b) = \frac{1}{N} \sum_1^N \frac{1}{2} \left\| y(x^{\{i\}}) - a^{[L]}(x^{\{i\}}) \right\|_2^2, \tag{9}$$

and this is a function of the weights and biases. Training the network now means that every iteration this loss function is evaluated, with the goal to minimize it, and then the parameters are updated accordingly. One iteration exists out of one forward pass - traversing through all neurons - , calculating the loss, to be followed by a backward pass, which is counting the changes in all parameters, starting at the last layer.

3.2 Backward pass

The second part of a complete iteration of the network is the backward pass, also called backpropagation. As mentioned before, the goal is to minimize our loss as a function of the weights and biases. This is done with a method called gradient descent, which is an optimization method that has an iterative process. For this section we will see our weights and biases as one single vector, called $p \in \mathcal{R}^s$, where s is the total amount of weights and biases. The loss function is $\mathcal{L}(p)$. The aim of gradient descent is to find the parameters that minimize $\mathcal{L}(p)$. We will let $\frac{\partial \mathcal{L}(p)}{\partial p_r}$ denote the partial derivative of the loss function with regard to the r^{th} parameter. $\nabla \mathcal{L}(p)$ is the vector of partial derivatives, or the gradient, and

$$(\nabla \mathcal{L}(p))_r = \frac{\partial \mathcal{L}(p)}{\partial p_r}. \tag{10}$$

If we start with our current vector p, and want our next vector, $p + \delta p$, improve our loss function, we use the Taylor series expansion to give us

$$\mathcal{L}(p + \Delta p) \approx \mathcal{L}(p) + \sum_r \frac{\partial \mathcal{L}(p)}{\partial p_r} \Delta p_r. \tag{11}$$

Combined with (10) this gives us

$$\mathcal{L}(p + \Delta p) \approx \mathcal{L}(p) + \nabla \mathcal{L}(p)^T \Delta p. \tag{12}$$

Since our goal is to minimize the loss function, (12) tells us we have to choose Δp such that $\nabla \mathcal{L}(p)^T \Delta p$ is as negative as possible. That will give us a derivative as close as possible to zero, which indicates a minimum. The Cauchy-Schwarz inequality states that for any $f, g \in \mathcal{R}^s$ we have $\left| f^T g \right| \leq \|f\|_2 \|g\|_2$. In order to get $\left| f^T g \right|$ as negative as possible, we aim for $f = -g$. In order words, we choose Δp in the opposite direction of $\nabla \mathcal{L}(p)$. This gives us the update

$$p \to p - \eta \nabla \mathcal{L}(p), \tag{13}$$

where η is a small stepsize that is known as the learning rate within machine learning. A random initial vector p is chosen, and by iterating (13) the loss function is minimized. Now, if we let

$$\mathcal{L}_{x\{i\}} = \frac{1}{2} \left\| y\left(x^{\{i\}}\right) - a^{[L]}\left(x^{\{i\}}\right) \right\|_2^2, \tag{14}$$

then from the loss function it follows that the gradient vector is

$$\nabla \mathcal{L}(p) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_{x^{\{i\}}}(p). \tag{15}$$

Computing the gradient vector at every iteration of the steepest descent method (13) can be costly if p or N is very large. It can be beneficial if the gradient of a random training

point is chosen instead of the mean of the gradients of all training points. The benefit of this is that, by the law of large numbers, by picking points at random the average value of these numbers will converge to the mean of the numbers. This is called the stochastic gradient descent method, which we will now use for the backward pass in training our network. The task at hand is to compute the partial derivatives of our loss function with respect to each individual weight and bias parameter. For an individual training point we see $\mathcal{L}_{x^{\{i\}}}$ as a function of weights and biases, so we can drop the dependence on $x^{\{i\}}$ and simply write

$$\mathcal{L} = \frac{1}{2} \left\| y - a^{[L]} \right\|_2^2. \tag{16}$$

Next, we introduce the error of the j^{th} neuron at layer l, to be defined as

$$\delta_j^{[l]} = \frac{\partial \mathcal{L}}{\partial z_j^{[l]}} \quad \text{for } 1 \leq j \leq n_l \quad \text{and} \quad 2 \leq l \leq L. \tag{17}$$

The error is a measure of sensitivity of our loss function to the weighted input. With these concepts we can get the following results using the chain rule. For the interested reader, the full proof can be found in [9].

$$\delta^{[L]} = \sigma' \left(z^{[L]} \right) \circ \left(a^{[L]} - y \right), \tag{18}$$

$$\delta^{[l]} = \sigma' \left(z^{[l]} \right) \circ \left(W^{[l+1]} \right)^T \delta^{[l+1]} \qquad \text{for } 2 \leq l \leq L-1, \tag{19}$$

$$\frac{\partial \mathcal{L}}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for } 2 \leq l \leq L, \tag{20}$$

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for } 2 \leq l \leq L. \tag{21}$$

Here $x \circ y$, the Hadamard product, is defined by $(x \circ y)_i = x_i y_i$, so the pairwise multiplication of the corresponding components.

Since the forward pass computes $a^{[L]}$, (18) immediately lets us compute $\delta^{[L]}$. Then, using (19), in the backward pass we can compute $\delta^{[L-1]}, \delta^{[L-2]}, ..., \delta^{[2]}$. Finally, using (20) and (21), we have access to the partial derivatives, which was what we wanted to retrieve. This way of computing gradients is called backpropagation, a widely known term in the field of deep learning.

3.3 Adam optimizer

The Adam optimizer - derived from Adaptive Moment Estimation - is another optimization algorithm [12]. It combines the method of gradient descent with momentum with the RMSP - Root Mean Square Propagation - method [13]. The gradient descent method with momentum is similar to the stochastic gradient descent method mentioned in section 3.2, but it takes the weighted average of the gradients into account. So we get

$$W^{[l+1]} = W^{[l]} - \eta m_l, \tag{22}$$

where m_l denotes the total of the gradients at layer l, or the estimate of the first momentum,

$$m_l = \kappa m_{l-1} + (1 - \kappa) \left[\frac{\delta L}{\delta W^{[l]}} \right]. \tag{23}$$

Here κ denotes the moving average parameter. The RMSP is a variation of this method. It takes the exponential average of moving gradients,

$$W^{[l+1]} = W^{[l]} - \frac{\eta t}{\left(v^{[l]}\right)^{1/2}} \left[\frac{\delta L}{\delta W^{[l]}}\right], \tag{24}$$

where $v^{[l]}$ is the sum of the square of past gradients at layer l, or the estimate of the second momentum,

$$v^{[l]} = \kappa v^{[l-1]} + (1-\kappa)\left[\frac{\delta L}{\delta W^{[l]}}\right]^2. \tag{25}$$

Now, Adam combines these two method such that it uses both its strengths and makes for an optimized gradient descent. It looks as follows

$$m_l = \kappa m_{l-1} + (1-\kappa)\left[\frac{\delta L}{\delta W^{[l]}}\right] v^{[l]}. \tag{26}$$

Studies show that the Adam method is a very good optimization algorithm and outperforms most other algorithms including the stochastic gradient descent method, both with and without momentum, and the RMSP method [13][14].

3.4 Residual networks

A special type of neural networks that we would like to highlight are residual neural networks (ResNets)[15]. In many types of neural networks only the information from the previous layer is passed on to the next layer, but in ResNets the data can be passed through one or more layers, after which the input data is added to the transformed data. This is called a residual block, which is portrayed in Figure 2.

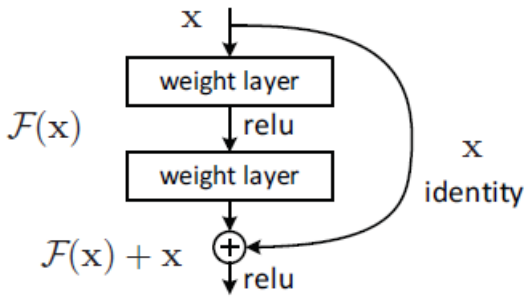

FIGURE 2: A residual building block of a residual neural network [15].

Here $\mathcal{F}$ is the transformation that is applied to the data, $\mathcal{F} : \mathbb{R}^{n_l} \to \mathbb{R}^{n_{l+1}}$, and if x is the data of the previous layer, then

$$x_{t+1} = x_t + \mathcal{F}(x_t, p). \tag{27}$$

Again, p is our vector of parameters. As is the case for all neural networks, the idea behind ResNets is that they improve as more layers are added. We will come back to ResNets in the next section.

4 Neural ODEs

The neural ordinary differential equation is a fairly newly-developed method to combine ODEs and deep learning[7]. Instead of learning a transformation directly, as is done with deep learning in general, we try to learn the structures of the transformation. Let us take (27) as the output of some hidden state. Now we take more and more hidden layer, and our timesteps smaller. If we take t to be infinitely small, we can change our system into a continuous process. Instead of a discrete number of layers between the input layer and the output layer, we allow the progression of the hidden states to become continuous. This gives us the following

$$\frac{\partial x(t)}{\partial t} = \mathcal{F}(x(t), t, p). \tag{28}$$

This is an ODE specified by a neural network. Now using our input layer $x(0)$, which can be specified by an initial condition, we get the output layer $x(T)$ as the solution to our problem 28. Like mentioned, ResNets become more powerful when more layers are added. But rather than adding more layers and taking the smaller steps each time, we can now parameterize the derivative using an ODE, and solve it. By choosing the discretization in our neural network we can get as many layers as we want. Figure 3 shows the ResNets and ODE networks, where the pictures clearly indicate the discrete series and the continuous series.

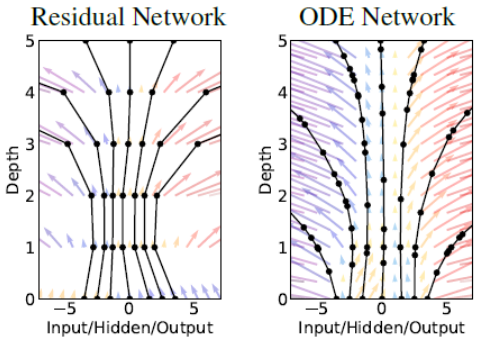

FIGURE 3: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations. Retrieved from [7].

5 Method

The method we use is based on a paper by Rackauckas et al. [16]. It uses the idea of neural ODEs [7] and its main concern is demonstrating the use of differential equation solvers with neural networks in the programming language JULIA. They demonstrate the use of their newly developed JULIA package called DiffEqFlux, which is also used in our study. The purpose of our thesis is to approximate a differential equation with the use of a neural ODE. We therefore have to start by setting up a differential equation as to generate our data. Let us take the following differential equation

$$\frac{\partial x(t)}{\partial t} = f(x(t)), \tag{29}$$

like mentioned before in section 2. We need to start by generating data from this equation. Let us say we sample t uniformly on a certain span, such that we get time samples $t_i, i = 0, 1, 2, ..., n$, where n denotes the amount of time samples we want, $t_i \in [0, T], T \in \mathbb{R}$. We also set initial condition $x(0) = x_0$.

Thereafter we start setting up our neural network. To do this, we have to implement a prediction function. We start by defining a multilayer perceptron with one hidden layer and a ReLU (Rectified Linear Unit) activation function [17],

$$\sigma(x) = \max(0, x). \tag{30}$$

Our deep learning model is the multilayer perceptron as presented in Figure 4.

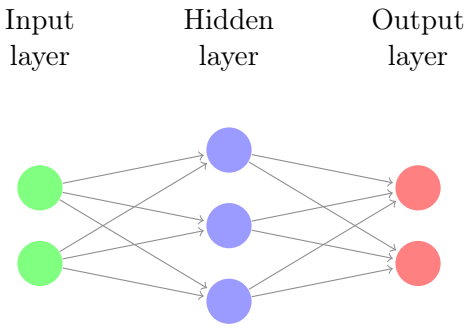

FIGURE 4: Multilayer perceptron.

For simplicity we draw the hidden layer with three neurons only. We create layers with a forward pass given by

$$\sigma(W^{[l]}a^{[l-1]} + b^{[l]}). \tag{31}$$

Now it is time to implement our neural ODE layer in our network. Like mentioned this is an ODE defined by the neural network itself. If the input of our network is x the neural ODE layer is simply

$$\frac{\partial x(t)}{\partial t} = \mathcal{F}(x(t), t), \tag{32}$$

with $\mathcal{F}$ like mentioned in section 4. Our prediction function, $\mathcal{P}$, is then the solution to this ODE. Now it is time to define our loss function, which is

$$\mathcal{L} = \sum_{t_i} \|\mathcal{P}(t_i) - x(t_i)\|. \tag{33}$$

The loss function is minimized and as a consequence the weights and biases are adjusted accordingly. In order to do this, the gradients need to be computed. Since this takes unnecessary effort to do by hand, we let a computer program evaluate these by means of automatic differentiation [18]. Automatic differentiation uses the chain rule from differential calculus to compute derivatives. There are two main types of automatic differentiation, namely forward mode and reverse mode. The package DiffEqFlux from JULIA allows us to switch between those two gradient methods very easily, such that our method does not have to be adapted to either one.

Since our goal is to train our model for several initial conditions, we need to make sure our prediction function depends on the initial condition. This then becomes $\mathcal{P}(x_j(0))$, with

$x_j(0) \in \mathcal{R}$ the initial condition for the j^{th} training point, $j = 1, 2, ..., J, J \in \mathbb{N}$. Our loss function also needs to take the several initial conditions into account, so

$$\mathcal{L} = \sum_{x_j(0)} \sum_{t_i} \|\mathcal{P}(t_i, x_j(0)) - x_j(t_i)\|. \tag{34}$$

After training, we still need to compute the test loss. For this, let us take initial condition $x_t(0) = x_0$, where $x_0 \notin x_j(0)$. Again we generate data from 29, now using initial condition $x_t(0)$. Our loss prediction has been optimized during training for the weights and biases, so all that is left is to compute

$$\mathcal{L} = \sum_{t_i} \|\mathcal{P}(t_i, x_t(0)) - x(t_i)\|. \tag{35}$$

This gives us the test loss, which we can then compare to the training loss.

6 Results

Now our method is set, we can start training our model. We implemented our method in the programming language Julia for this, and tried to train and test our model for two different systems: the Lotka-Volterra equations [19] and the Van der Pol oscillator [20]. Since our goal was to see how our model would react if we tested it on a different set of initial conditions after training it for certain other conditions, we first need to actually train our model.

6.1 Lotka-Volterra

The Lotka-Volterra equations are defined as follows

$$\begin{aligned} \frac{\partial x_1}{\partial t} &= \alpha x_1 - \beta x_1 x_2, \\ \frac{\partial x_2}{\partial t} &= \delta x_1 x_2 - \gamma x_2. \end{aligned} \tag{36}$$

where x_1 defines the number of prey and x_2 the number of predators. α is the prey population growth parameter, β is the first species interaction parameter, γ is the second species interaction parameter and δ is the predator population extinction parameter.

For training our model we chose the settings depicted in Table 1.

Table 1: Values parameters of the model for (36).

Parameter	Description	Value
α	Prey population growth parameter	1.5
β	First species interaction parameter	1.0
γ	Second species interaction parameter	3.0
δ	Predator population extinction parameter	1.0
η	Learning rate	0.01
epochs	Amount of full cycles over the whole training set	200
Δt	Step size	0.05
T	End timespan	2.5

In order to get a qualitative approximation of the prey-predator model using our method it is important to carefully consider the learning rate and amount of epochs. If the number

of epochs were to be too large there is a possibility of overfitting, meaning the model will perform well during training, but poorly during testing. However, recent studies suggest that some deep learning models can exhibit the double-descent phenomenon [21]. This means that the models performance does worsen as the amount of epochs increases at first, but later performs better if the amount of epochs are increased even more. In order to study this possible effect some large amounts of epochs are added as well. The learning rate determines how much the weights are updated for each iteration, and thus a rate too large will cause for unstable training. If the rate is chosen too small, its consequence will be a long computation time as well as the possible development of local minima, meaning the model gets stuck in the process and the loss function cannot be minimized any further. In Figures 5 and 6 the amount of epochs is graphed against the training and test loss. It is visible that after 200 epochs it is not valuable anymore to train the model any longer, since the test loss stays steady and only the training loss decreases slightly. The losses are also shown in Table 2.

TABLE 2: Training and test loss.

Amount of epochs	100	150	200	250	300	1000	2000
Training loss	9.26	2.32	1.58	1.02	0.93	0.46	0.28
Test loss	8.45	1.59	1.26	1.37	1.48	7.01	7.05

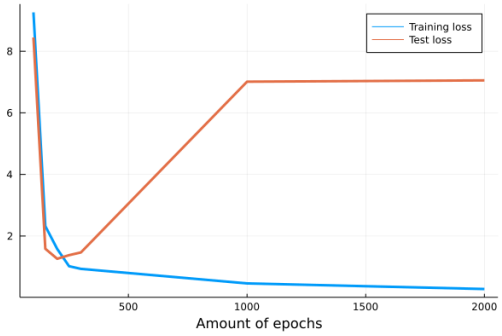

FIGURE 5: The amount of epochs against the training and test loss in Table 2.

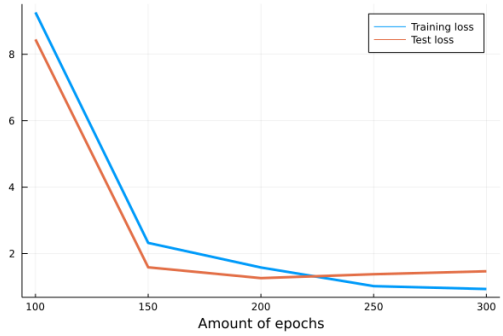

FIGURE 6: Epochs 100-300 against the training and test loss in Table 2.

In JULIA we also have to determine the ODE-solver, for which we chose Tsit5, which has a performance similar to the ODE-solver ode45 in MATLAB [16]. Tsit5 is in general a suitable solver for non-stiff equations in JULIA, therefore a good solver for (36). Finally, the optimizer needs to be determined. We decided to use the optimizer Adam [12], an efficient algorithm for first-order gradient-based optimization of stochastic objective functions, suitable for networks of this size.

In Figures 7 and 8 the approximation of the Lotka-Volterra equations can be seen after our network has been trained for several different initial conditions. The figures depict the prediction the trained model made for initial conditions of $x = 2$ and $y = 5$. In Table 3 the values for the initial conditions during training can be seen. The testing value differs from the training values.

We also ran the model for a longer timespan. In Figures 9 and 10 the approximation is visible for $T = 5$. The training loss is in this case 48.04 and the test loss is 17.31. The amount of epochs used is again 200, with a learning rate of 0.01.

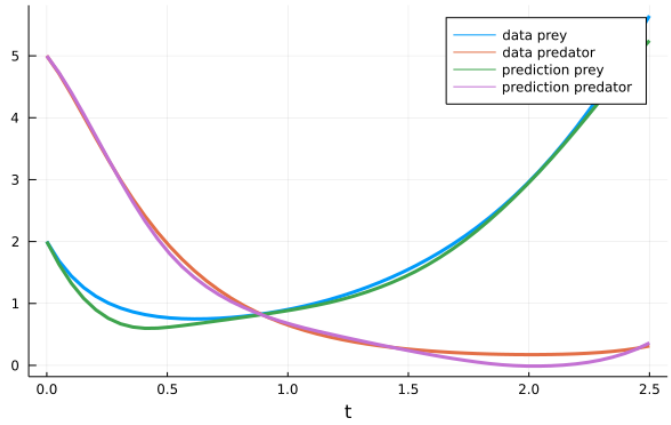

FIGURE 7: Approximation of the Lotka-Volterra equations (36).

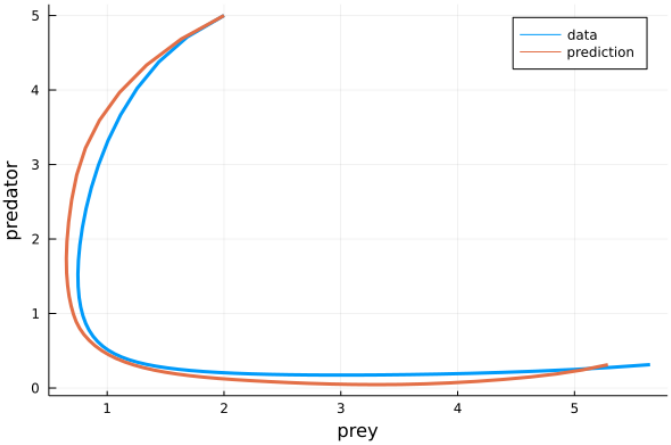

FIGURE 8: Approximation of the Lotka-Volterra equations (36) in 2D.

TABLE 3: Values initial conditions for the different training points for (36).

$[x_1; x_2]$	[2; 3]	[1; 5]	[5; 9]	[5; 1]	[3; 2]	[4; 3]

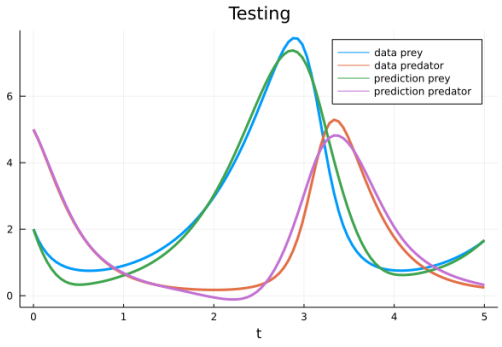

FIGURE 9: Approximation of the Lotka-Volterra equations (36) for a larger timespan.

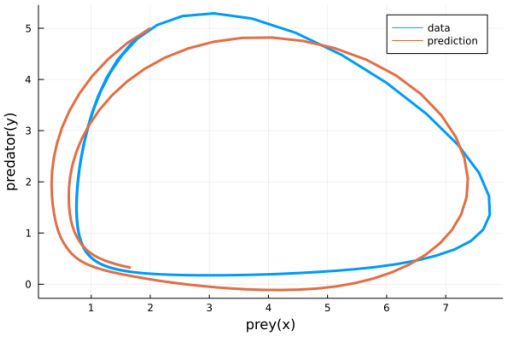

FIGURE 10: Approximation of the Lotka-Volterra equations (36) for a larger timespan in 2D.

6.2 Van der Pol oscillator

The Van der Pol oscillator can be described as follows

$$\frac{d^2x}{dt^2} - \mu\left(1 - x^2\right)\frac{dx}{dt} + x = 0, \tag{37}$$

where u is the parameter indicating the strength of the damping. We would however prefer not to have a second order differential equation, but a first order. By applying transformation $x_2 = x - x^3/3 - \dot{x}/\mu$, and setting $x = x_1$ we get to the following two-dimensional result

$$\begin{aligned} \frac{\partial x_1}{\partial t} &= \mu(x_1 - \frac{1}{3}x_1^3 - x_2), \\ \frac{\partial x_2}{\partial t} &= \frac{1}{\mu}x_1. \end{aligned} \tag{38}$$

In Table 4 the values of the parameters of our model can be seen. Again the Adam-optimizer an the Tsit5 ODE-solver are used.

Table 4: Values parameters of the model for (38).

Parameter	Description	Value
μ	Strength of damping parameter	0.5
η	Learning rate	0.01
epochs	Amount of full cycles over the whole training set	200
Δt	Step size	0.05
T	End timespan	5.0

In Figures 11 and 12 it is again visible what the test and training loss is as the amount of epochs grows larger. Table 5 shows the exact amount of training and test loss.

Table 5: Training and test losses.

Amount of epochs	100	150	200	250	300	1000	2000
Training loss	1.19	1.13	0.12	0.13	0.31	0.06	0.01
Test loss	0.96	1.33	0.10	0.16	0.20	0.10	0.01

In Figures 13 and 14 the approximation of the van der Pol system can be seen after the network has been trained for several different initial conditions. The figures depict the prediction the trained model made for initial conditions of $x = 0.3$ and $x_2 = -0.5$, which are different initial conditions than used in the training set. Table 6 shows the values for the initial conditions of the training points.

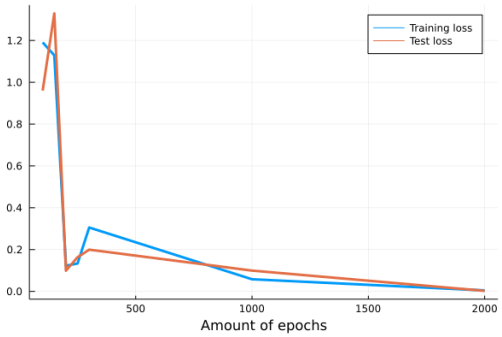

FIGURE 11: The amount of epochs against the training and test loss in Table 5.

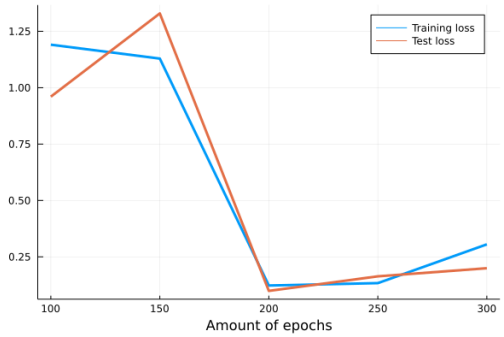

FIGURE 12: Epochs 100-300 against the training and test loss in Table 5.

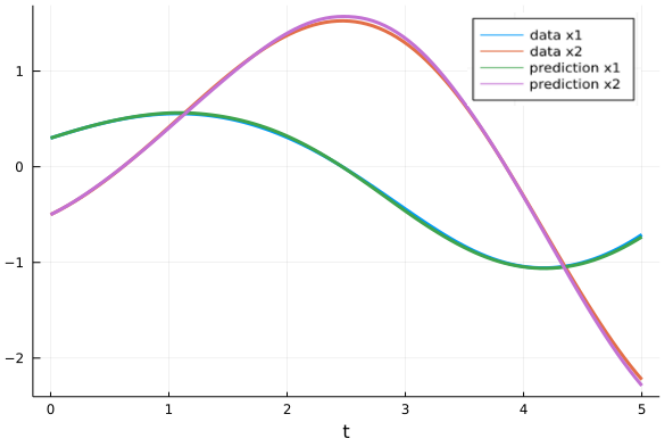

FIGURE 13: Approximation of the Van der Pol oscillator (38).

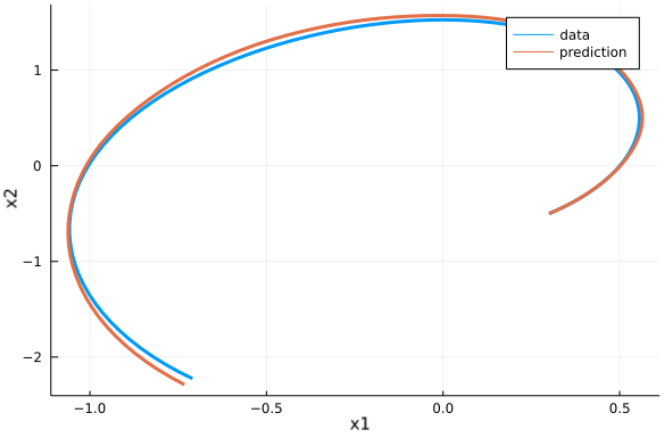

FIGURE 14: Approximation of the Van der Pol oscillator (38) in 2D.

We also ran the model for a longer timespan. In Figures 15 and 16 the approximation is visible for $T = 10$. The training loss is in this case 63.80 and the test loss is 87.85. The amount of epochs used is again 200, with a learning rate of 0.01.

TABLE 6: Values initial conditions for the different training points for (38).

$[x_1; x_2]$	[0.5; -0.5]	[0.3; -0.1]	[-0.5; 0.5]	[-0.2; 0.0]	[0.2; -0.7]

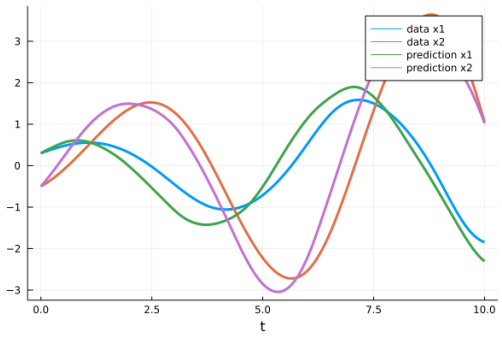

FIGURE 15: Approximation of the Van der Pol oscillator (38) for a larger timespan.

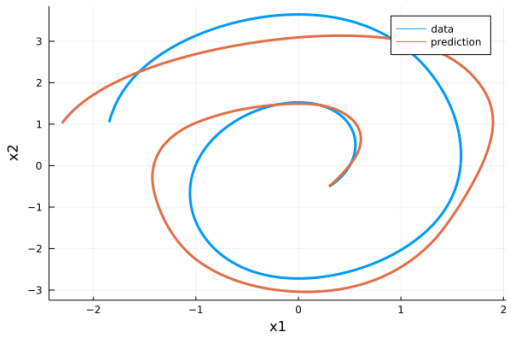

FIGURE 16: Approximation of the Van der Pol oscillator (38) for a larger timespan in 2D.

6.3 Amounts of training points

During training we noticed that the quality of the approximation depended heavily on the amount of training points used. Figure 17 and Table 7 show the loss for an amount of training points ranging from one to six for the Lotka-Volterra equations (36). The actual values of the initial conditions used for each of these training points are the same as in Table 3. The first training point is only the first-depicted value, the training point that was added afterwards is the second-depicted value, and so-on.

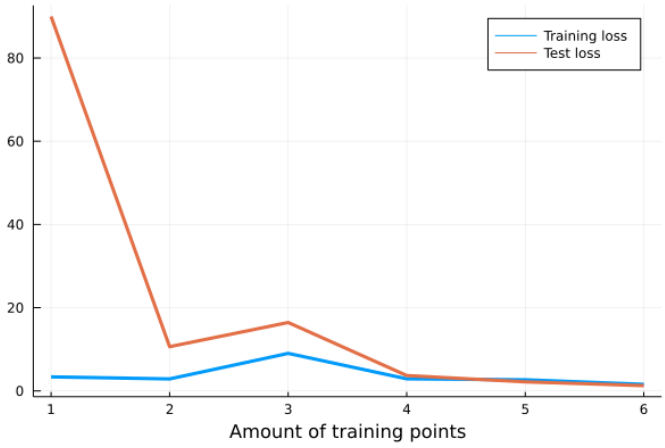

FIGURE 17: The amount of training points against the training and test loss for (36).

TABLE 7: Training and test losses for different amounts of training points for (36).

Amount of training points	1	2	3	4	5	6
Training loss	3.36	2.88	9.02	2.90	2.67	1.58
Test loss	89.97	10.63	16.43	3.68	2.17	1.26

The same procedure was used for the Van der Pol oscillator (38). Figure 18 and Table 8 show the loss dependent on the amount of training points. The values of the training points are depicted in table 6, shown in the same way as for the Lotka-Volterra equations.

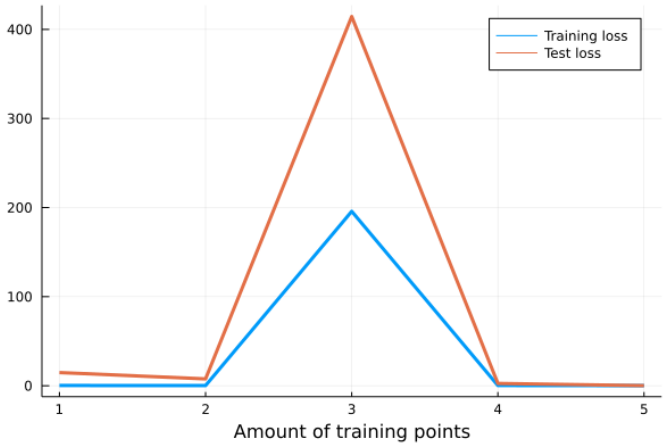

FIGURE 18: The amount of training points against the training and test loss for (38).

TABLE 8: Training and test losses for different amounts of training points for (38).

Amount of training points	1	2	3	4	5
Training loss	0.27	0.20	195.76	0.28	0.12
Test loss	14.77	7.67	414.46	2.55	0.10

7 Discussion & Conclusion

In this paper we have looked at neural ODEs and how they can be used to approximate the Lotka-Volterra equations and the Van der Pol system. As can be seen in the figures in the result section, it is possible to use neural ODEs to set up a network that learns the structure of a system of equations. By training the model for different sets of initial conditions, it learns how the equations behave with different starting points, so it is not dependent on one single system of differential equations to make a prediction for a new set of initial conditions. The parameters were trained to be used for several training points, thus learning the behaviour of the system in general.

However, there are many parameters and variables that need to be taken into account. During training we noticed that it matters a lot what the values are for the learning rate and the amount of epochs. The effect on the quality of the approximation of the amount of epochs is shown in Figures 5 and 11. It is interesting that those figures do not depict the exact same behaviour. Figures 5 and 6 shows a more steady decrease in loss as the amount of epochs increases, after which the test loss increases again a lot for the larger epochs. Figures 11 and 12 show a very sudden descent in loss at 200 epochs, after which both the test and training loss slowly increase again, to be followed by a decrease for larger epochs. We do not have an explanation for this, it is thus an interesting topic for further research.

The learning rate has not been closely looked at during this research, so this could be a factor that could be studied more intensely. Besides this, the training process in this

research was relatively short. The Lotka-Volterra equations were trained on six different sets of initial conditions, and the Van der Pol oscillator was trained on five different sets of initial conditions. Figure 17 shows that at least five training points were necessary to get a test loss lower than the training loss for the Lotka-Volterra equations. After adding the third training point, both the training and test loss increased again. Our hypothesis is that this is caused by the third training point having relatively large initial conditions compared to the other training points, as can be seen in Table 3. As for the Van der Pol oscillator, the model needed five training points before the test loss got smaller than the training loss. Figure 18 interestingly shows that the model was not capable of accurate testing, or training for that matter, when the third training point was added. A possible explanation for this is that the third training point is very different from the first two training points, since x_1 is negative and x_2 is positive, which was the other way around for the first two training points. It implies the effect of the value of the initial conditions of the Van der Pol oscillator is of a large influence on the structure of the system, making it harder to adjust the parameters such that it can minimize the loss function using all those training points. Once a fourth training point is added, the model seems to perform much better. It is thus clear that the performance of the model is highly dependent on the choice of the amounts and values of the initial conditions. However, we do not know how well the model will improve once it is trained on a larger and more diverse dataset. We thus recommend to study this aspect of the model in more detail, as there are still many unknowns.

There was unfortunately not enough time to study the model and its capability of approximating differential equations for a longer timespan. Figures 9, 10, 15 and 16 show that it is possible to make an approximation, and that the network can adjust the parameters such that the general structure of the system is preserved. However, the loss becomes a lot bigger both during training and testing. We thus recommend that more experiments are done with this, since it could be that a larger amount of epochs solves this problem.

On a positive note, our network turned out to work quite well on our datasets, and surprisingly enough managed to make decent approximations for a relative small number of training points. We do recommend to broaden this research to more different kind of systems of differential equations. There are more complicated systems, which could have a very different behaviour, and therefore more difficult to learn for a neural network.

References

[1] L. K. Mühlbauer, M. Schulze, W. S. Harpole, and A. T. Clark, "gauseR: Simple methods for fitting Lotka-Volterra models describing Gause's "Struggle for Existence"," *Ecology and Evolution*, vol. 10, pp. 13275–13283, Dec. 2020.

[2] W. Mansour, "Quenching of limit cycles of a Van der Pol oscillator," *Journal of Sound and Vibration*, vol. 25, pp. 395–405, Dec. 1972.

[3] V. Sundarapandian, "Output regulation of Van der Pol oscillator," *Journal of the Institution of Engineers (India): Electrical Engineering Division*, vol. 88, no. DEC., pp. 20–24, 2007.

[4] P. Veskos and Y. Demiris, "Experimental comparison of the van der Pol and Rayleigh nonlinear oscillators for a robotic swinging task," vol. 1, pp. 197–202, 2006.

[5] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Sparse Identification of Nonlinear Dynamics with Control (SINDYc)**SLB acknowledges support from the U.S. Air Force Center of Excellence on Nature Inspired Flight Technologies and Ideas (FA9550-14-1-0398). JLP thanks Bill and Melinda Gates for their active support of the Institute of Disease Modeling and their sponsorship through the Global Good Fund. JNK acknowledges support from the U.S. Air Force Office of Scientific Research (FA9550-09-0174).," *IFAC-PapersOnLine*, vol. 49, pp. 710–715, Jan. 2016.

[6] J. Ren and J. Duan, "Identifying stochastic governing equations from data of the most probable transition trajectories," *arXiv:2002.10251 [physics, stat]*, Aug. 2020. arXiv: 2002.10251.

[7] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural Ordinary Differential Equations," *arXiv:1806.07366 [cs, stat]*, Dec. 2019. arXiv: 1806.07366.

[8] J. C. Polking, A. Boggess, and D. Arnold, *Differential equations with boundary value problems*. 2014. OCLC: 1024125263.

[9] C. F. Higham and D. J. Higham, "Deep Learning: An Introduction for Applied Mathematicians," *SIAM Review*, vol. 61, pp. 860–891, Jan. 2019.

[10] A. Farzad, H. Mashayekhi, and H. Hassanpour, "A comparative performance analysis of different activation functions in LSTM networks for classification," *Neural Computing and Applications*, vol. 31, pp. 2507–2521, July 2019.

[11] D. B. Mehta, P. A. Barot, and S. G. Langhnoja, "Effect of Different Activation Functions on EEG Signal Classification based on Neural Networks," in *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, (Erode, India), pp. 132–135, IEEE, Mar. 2020.

[12] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017. arXiv: 1412.6980.

[13] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv:1609.04747 [cs]*, June 2017. arXiv: 1609.04747.

[14] R. Zaheer and H. Shaziya, "A Study of the Optimization Algorithms in Deep Learning," in *2019 Third International Conference on Inventive Systems and Control (ICISC)*, (Coimbatore, India), pp. 536–539, IEEE, Jan. 2019.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NV, USA), pp. 770–778, IEEE, June 2016.

[16] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, "DiffEqFlux.jl - A Julia Library for Neural Differential Equations," *arXiv:1902.02376 [cs, stat]*, Feb. 2019. arXiv: 1902.02376.

[17] D. Yarotsky, "Error bounds for approximations with deep ReLU networks," *Neural Networks*, vol. 94, pp. 103–114, Oct. 2017.

[18] C. H. Bischof, P. D. Hovland, and B. Norris, "On the implementation of automatic differentiation tools," *Higher-Order and Symbolic Computation*, vol. 21, pp. 311–331, Sept. 2008.

[19] A. J. Lotka, "Contribution to the Theory of Periodic Reactions," *The Journal of Physical Chemistry*, vol. 14, pp. 271–274, Mar. 1910.

[20] B. van der Pol, "LXXXVIII. *On "relaxation-oscillations"*," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, pp. 978–992, Nov. 1926.

[21] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, "Deep Double Descent: Where Bigger Models and More Data Hurt," *arXiv:1912.02292 [cs, stat]*, Dec. 2019. arXiv: 1912.02292.