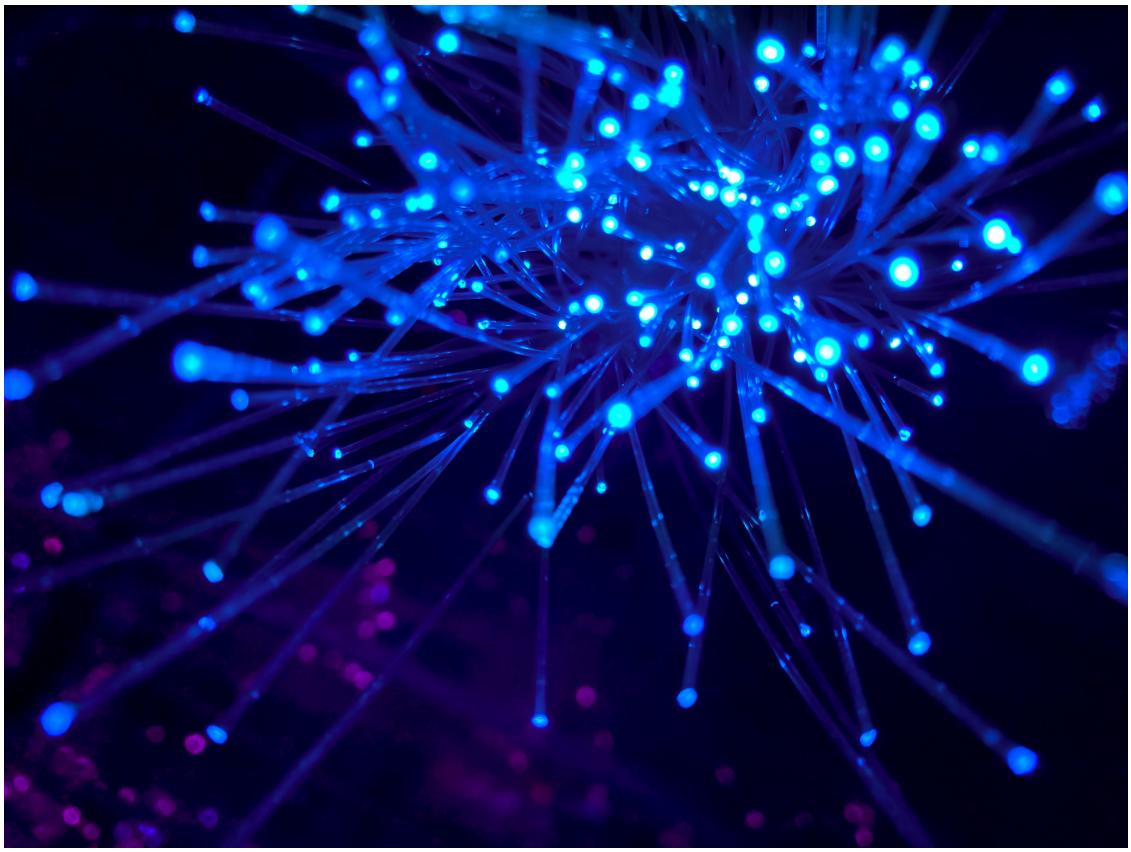

Chalmers University of Technology

Centiro Solutions AB

Analysis of numerical methods for data driven regression problems in neural networks

Master's thesis in Systems, Control and Mechatronics

Apostolos Anastasiadis

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Analysis of numerical methods for data driven regression problems in neural networks

Apostolos Anastasiadis

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Automatic Control research group
Chalmers University of Technology
Gothenburg, Sweden 2023

Analysis of numerical methods for data driven regression problems in neural networks.
A modeling of artificial neural networks using stohastic gradient descent for regression problems, analysed and compared with already existing methodologies for discretising and solving the update functions.
Apostolos Anastasiadis

Supervisor: Balázs Adam Kulcsár, Department of Electrical Engineering
Examiner: Viktor Andersson, Centiro Solutions AB

Master's Thesis 2023
Department of Electrical Engineering
Division of Systems and Control
Automatic control research group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Human brain neural network visualization and the correlation with the artificial neural networks[12].

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Modeling and analysis of artificial neural network convergence.
Linear and non-linear neural network models implementation analyses, followed by discretization methods and the comparison between them and already existing built-in Matlab functions.
Apostolos Anastasiadis
Department of Electrical Engineering
Chalmers University of Technology

Abstract

This thesis project is a research endeavor to analyse artificial neural network model training convergence, regarding the input and output data in numerous forms of implementation. The update function for the ANN (artificial neural network) model is generated using SGD - stochastic gradient descent method. At first, some simple linear single input - single output neural network simulations are evaluated. Afterwards, various types of activation functions (like for instance linear or sigmoid) are applied to the existing single input - single output model. The resulting non-linear differential equation behaviour and convergence, in accordance to the data input - output label, is tested. We then start investigating multiple input - multiple output systems, from mega to mini batches. The second part of the thesis focuses on solving the numerical integration problem arising in the MIMO nonlinear differential equations. As such, we implemented Crack Nicolson and Euler methods, in order to be compared them with already existing built-in Matlab functions. Accuracy, runtime, and convergence properties are analyzed.

Keywords: artificial, neural network, convergence, data, linear, non-linear, discretization, method.

Acknowledgements

This research was a team endeavor, not just mine. Hence, I would like to offer my special thanks to Professor Balázs Kulcsár for giving me guidance and support throughout this project. In addition, I want to acknowledge all the advice and aid I was given by PhD candidate Viktor Andersson, from the beginning till the very end. Both of them have been very supportive and understanding and I cannot express enough, how grateful I feel. Last but not least, I feel greately thankful for the opportunity that **Centiro Solutions AB** and **Chalmers University of Technology** jointly gave me.

Apostolos Anastasiadis, Gothenburg, February 2023

Contents

List of Figures

List of Tables

1
Introduction

Artificial neural networks(ANNs) and their background were first formulated in the late nineteenth century. This consisted primarily of interdisciplinary work in physics, psychology and neurophysiology. This early step emphasized general theories of vision, learning, condition and etc., without including any mathematical model of neuron operation. However, this step invigorated the specific field and during the last two decades, a great amount of papers have been published, resulting to the investigation of a significant amount of ANN models. Since then, neural networks have been applied to various and sometimes, completely diverse fields, including aerospace, banking, automotive, medical, telecommunications, transportation and many others. [1]

In this research project, an attempt to experiment and analyse the efficiency of artificial neural network's training convergence, in accordance with a plethora of data types, specific each time, and their corresponding output, is made. The research question behind this endeavor, is to further investigate whether neural networks can perform efficiently, in terms of convergence, for different kinds of data sets and activation functions, or not. It is the search for the appropriate methodology, for the corresponding regression problem and neural network model, that has driven this specific scientific research. It is the need for analysing the behaviour of a neural network on multiple linear and non-linear scenarios, using multiple kinds of input data and feeding them in different ways to the analogous model that precedes this thesis, in order to determine what the possibilities are, regarding future applications.

During this project, many existing theorems and implementations are mentioned, analysed and taken into account as the scientific background and groundwork to work with. Furthermore, various forms of researching actions take place, such as forming methodologies and testing them on directly, or as in its final part, comparing them with existing, already tested, implementations and analysing their results and capabilities, for present and future purposes.

Specifically, different methods for updating the learning method (it is later referred as update function $\mathbf{H}$) where applied, leading to diverse learning behaviours, providing us necessary information regarding the model's competence. In addition, an obstacle and a scientific question at the same time, that showed up later on during this research, is whether it is possible or not, to discretise and solve, using existing discretisation techniques, the non linear systems of the differential equations that constitute the update function of the corresponding, constructed model. This step is taken so that, in turn, the neural network models of such non linear nature can be

explored on their optimisation and robustness potential, by either keeping, adjusting or replacing pre existing methods for solving these systems.

2

Theory

2.1 Artificial Neural Networks

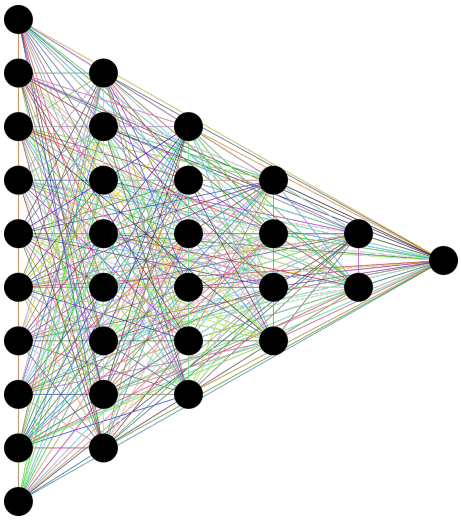

Figure 2.1: Artificial Neural Network model visualisation, [13]

Artificial neural networks(ANNs) are biologically inspired computational networks. These networks emulate a biological neural network but they use a reduced number of concepts from biological neural networks. Specifically, ANN models stimulate the electrical activity that takes place in a brain and a nervous system. They consist of elements that carry out all the required processes, called neurons. A neuron can be connected to all the other neurons of the model, or a subset of those. This kind of connection simulates the synaptic connections of the brain. Commonly, the neurons are arranged in a vector, called layer, with the output of one layer serving as input to the next layer and, possibly, other layers. Data signals, most of the times weighted, while entering a neuron, are in turn, simulating the electrical excitation of a nerve cell and therefore, the information transfer within the brain or a general, biological neural network.
The input values to a neuron, or a layer of neurons, with a data structure of $1xN$, are multiplied by the weight $w_{i,j}$, that simulates the strengthening or weakening of the neural pathways in the brain.Then the bias $b_{i,j}$ is added to this value.

$$\omega = w_{i,j}x_{i,j} + b_{i,j} \tag{2.1}$$

Where ω is the output of this procedure, where, if no activation function is applied is equal to $x_{i+1,j+1}$. **i** is the neuron index and **j** the layer one.
It is by adjusting and re-adjusting over time, the corresponding connection weights, that learning through an ANN, is achieved. All the weight,bias-adjusted input of values to the corresponding neuron or layer of neurons, are then aggregated using a vector to a scalar function such as averaging, summation, input maximum or others, to produce a single input value for the upcoming neuron or layer. Afterwards, as soon as the input is calculated, it is then fed to the analogous *activation function* to produce its output, which of course is the input signal for the next neuron or layer. The activation function transforms the neuron's or layer's input value. It is therefore a pivotal tool that guides the model's performance and nature. Typically, the activation function involves the use of a sigmoid, ReLu, Heaviside, or other non linear function. This procedure is repeated between the layers of neurons until a final output value or vector of values, is calculated by the model.

$$x_{i+1,j+1} = O(\omega) \tag{2.2}$$

Where O defines the implementation of the corresponding activation function and ω refers to 2.1.
Theoretically, to simulate the asynchronous activity of the human nervous system, the processing elements - neurons, should behave and act in an asynchronous manner as well. However, most software and hardware implementations of ANNs enforce a discretized method, to ensure that each neuron is activated once and only, for each set of inputs. [2]

2.2 Continuous Gradient Descent - Gradient Flow

The most vital method applied to the artificial neural network in order for it to be taught and led to convergence, is the gradient descent method. Gradient descent is an optimization algorithm used to minimise some function (in most cases, a loss function) by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient. In artificial neural networks, machine learning and in this project, gradient descent is used to update the parameters of the respective model.

To make it more clear, a simple single input - single layer neural model, without an activation function, has the following attributes,

$$\omega = wx + b \tag{2.3}$$

$$L(\omega, t) = \frac{1}{2}(t - O(\omega))^2 \tag{2.4}$$

Where, x is the input data, t is the target/output label and L is the quadratic loss function for scalar output and O is the corresponding activation function.
A loss function (or Cost function) is analyzed to depict the model's performance, in terms of making predictions for a specific set of parameters. Hence, the optimization of the Loss function's through stochastic gradient descent is essential for updating

our parameters through time and finding the minimum of it, which will lead to a more accurate model.
As mentioned already, the loss function is to be directed towards its minimum, at each and different scenario, using the gradient descent method. The only thing that is controlled, is the parameters w and b. Since we need to consider how each parameter affects the final prediction made by our model, the partial derivatives of the loss/cost function, with respect to those parameters, are calculated. For the sake of the analysis we are not using any activation function, so $O(\omega) = \omega$.

$$\frac{\partial L}{\partial w} = x(b - t + wx) \tag{2.5}$$

$$\frac{\partial L}{\partial b} = b - t + wx \tag{2.6}$$

What has been calculated above is the gradient we were aiming for. The point of which is to make our loss function, gradually, minimum. The gradient of course, will always point upslope, therefore making our loss function growing bigger over time. The way to avoid that and at the same time, achieve optimization, is by following the negative gradient, through gradient flow.
Gradient flow, by definition, is a curve that depicts the direction of steepest descent of a function. In our case, our function is L and the gradient flow of which, F_L, goes as following,

$$F_L(t) = -\nabla L(w, b) = -[F_{w,t}, F_{b,t}]^T \tag{2.7}$$

Which leads to,

$$F_{w,t} = -x(b - t + wx) \tag{2.8}$$

$$F_{b,t} = -b + t - wx \tag{2.9}$$

2.3 Lipschitz Continuity

The beginning of all our research purposes, for defining and converging to the optimal values for the parameters used in our neural network modeling, is the gradient descent. Calculating the gradients of the loss/energy function is required. So, before moving on with applying our methods, we have to make sure that these gradients follow a continuous, data-related curve that allows logical numerical approximations and therefore, solutions to our updates. If that is not the case, then small changes in the data can cause large changes in the solution, and thus, our calculations may produce meaningless results. Lipschitz continuity is a way to ensure that this is not happening to our corresponding update function.
Let us now start with defining a Lipschitz continuous function. [9]
Given two metric spaces (X, d_X) and (Y, d_Y), where d_X and d_Y denote the metric on the set X and Y respectively, a function f: X $\longrightarrow$ Y is called Lipschitz continuous if there exists a constant M ≥ 0 such that, for all x_1 and x_2 in X,

$$d_Y(f(x_1), f(x_2)) \leq \mathbf{M} d_X(x_1, x_2) \tag{2.10}$$

Any such **M** is referred to as a *Lipschitz constant* for the function *f*. A real valued f:R $\longrightarrow$ R, in particular, is called *Lipschitz continuous* if there exists a positive real constant **M** such that, for all real x_1, x_2,

$$|f(x_1) - f(x_2)| \leq \mathbf{M}|x_1 - x_2| \tag{2.11}$$

This inequality is trivially satisfied when $x_1 = x_2$, but we are interested in all of the rest cases, for all $x_1 \neq x_2$. Therefore, one can define a function to be *Lipschitz continuous* if and only if there exists a constant $\mathbf{M} \geq 0$ such that, for all $x_1 \neq x_2$,

$$\frac{|f(x_1) - f(x_2)|}{|x_1 - x_2|} \leq \mathbf{M} \tag{2.12}$$

2.4 Euler's Method

Since the group of gradient flow equations as appeared in 2.2 is not the one we are working with, because an activation function is always applied, we are therefore in need of a method to help us deal with the ODEs(ordinary differential equations) that are describing each and every neural network model.
Assuming we have a function g and a differential equation,

$$\frac{\partial y}{\partial z} = g(y, z) \tag{2.13}$$

We will now proceed by applying the Euler method for this differential equation. Euler's method assumes that our solution is written in the form of Taylor series,

$$y(z + h) \approx y(z) + h\frac{\partial y}{\partial z} + \frac{h^2}{2!}\frac{\partial^2 y}{\partial z^2} + \frac{h^3}{3!}\frac{\partial^3 y}{\partial z^3} + ... \tag{2.14}$$

For Euler's Method we keep only the first two terms. By also substituting the term $y^{'}$ we result in,

$$y(x + h) = y(x) + hg(y, z) \tag{2.15}$$

We can also replace x+h with new while removing x and we end up with an equation that solves iteratively,

$$y_{new} = y + hg(y, z) \tag{2.16}$$

Where now y_{new} is the estimated next value, y is the current value, h is the interval between the steps(the step size), g is the derivative of function y(z), and z is time

2.5 Phase portraits

To assess the efficacy of all the methodologies applied, one cannot simply run some experimental tests and evaluate the outcome. For situations like this, at first, a phase vector of the specific ODE (ordinary differential equation) of the system, is to be calculated. A phase portrait is a geometric representation of the trajectories of a dynamical system in the phase plane. Each set of initial conditions is represented

by a different curve, or point. [4] For instance, using the equations (2.7,2.8,2.9) for specified x_0, t_0 we can calculate the gradient of the parameters at each and every combination of the parameters and thus, plot the sum of the phase trajectories which is the phase portrait required.
However, what is needed the most is a visualization of the trajectory of our ODE given a certain initial condition. That is exactly what a phase portrait is. The phase portrait is a geometric representation of the trajectories of a dynamical system in the phase plane(i.e. a visual display of certain characteristics of certain kinds of differential equations). It is a display of the trajectory after a period of time/steps, given a specific initial condition, that evaluates in practice, the robustness of the system.

2.6 Average loss function

A different implementation in MATLAB ® is to be made, changing the methodology of our update functions criteria and providing us with an alternative scientific approach. At first with the help of the built in function *ode45.m* and later on with different methods to solve the corresponding non-linear differential system.
At first, the input data(i.e. a vector of length N) are being generated solely, by which, the true label data are also generated as a function of the former. Then, the method for the calculation of the gradients is calculated as followed.
The standard gradient of the loss function is,

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \\ \frac{\partial L}{\partial b} \end{bmatrix}$$

Now, the average gradient of each data point is to be calculated, with the intention of reducing this *average loss function* instead of the regular one.

$$\nabla L_{sum} = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \\ \frac{\partial L}{\partial b} \end{bmatrix}_{x=x_1} + \begin{bmatrix} \frac{\partial L}{\partial w} \\ \\ \frac{\partial L}{\partial b} \end{bmatrix}_{x=x_2} + ... + \begin{bmatrix} \frac{\partial L}{\partial w} \\ \\ \frac{\partial L}{\partial b} \end{bmatrix}_{x=x_N}$$

$$\nabla L_{average} = \frac{\nabla L_{sum}}{N}$$

With all the above in mind, our parameters are being updated like this,

$$\begin{bmatrix} w_{new} \\ b_{new} \end{bmatrix} = \begin{bmatrix} w_{old} \\ b_{old} \end{bmatrix} + H$$

Where **H** is the update function resulting from the negative gradient of the loss function and the learning rate,

$$H = -\eta \nabla L_{average}$$

Afterwards, the average loss function is numerically computed, at each time step **i** of the chosen solver and over all the **k=1,2,..,N** data points, as followed. Setting

$S(\omega)_{i,k}$ as the corresponding update function,

$$L_{i,k} = \frac{1}{2}(y_{i,k} - S(\omega)_{i,k})^2$$

$$\Sigma L = \sum_{k=1}^{N} L_{i,k}$$

And therefore, the average loss function,

$$L_{average} = \frac{\Sigma L}{N}$$

This average loss function is now to be tested and analyzed for many different scenarios, directly showing us the efficiency of the corresponding model. For instance, different number of data points are to be used, different output/true labels, various learning rate values and activation functions.

2.7 Newton's Method

In numerical analysis, Newton's method [11], also known as the Newton–Raphson method, named after Isaac Newton and Joseph Raphson, is a root-finding algorithm which produces successively better approximations to the roots (or zeroes) of a real-valued function. The most basic version starts with a single-variable function f defined for a real variable x, the function's derivative f, and an initial guess x_0 for a root of f. If the function satisfies sufficient assumptions and the initial guess is close, then

$$x_1 = x_0 - \frac{f(x_0)}{f(x_0)'}$$

is a better approximation of the root than x_0. Geometrically, $(x_1, 0)$ is the intersection of the x-axis and the tangent of the graph of f at $(x_0, f(x_0))$: that is, the improved guess is the unique root of the linear approximation at the initial point. The process is repeated as

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n)'}$$

until a sufficiently precise value is reached. The number of correct digits roughly doubles with each step. The method can also be extended to complex functions and to systems of equations.

2.8 Crank-Nicolson

In numerical analysis, the Crank–Nicolson method is a finite difference method used for numerically solving the heat equation and similar partial differential equations. It is a second-order method in time. It is implicit in time, can be written as an implicit Runge–Kutta method, and it is numerically stable. The method was developed by John Crank and Phyllis Nicolson in the mid 20th century. [10]

3
Methods

3.1 Euler Method - Gradient Descent

Now we arrive to a point where we get to introduce an applied method to make use of the continuous gradient descent equations.
Now, expressing our gradient flow equation,(2.8,2.9), as the derivative of our parameters,

$$\dot{w} = -\frac{\partial L}{\partial w} \tag{3.1}$$

$$\dot{b} = -\frac{\partial L}{\partial b} \tag{3.2}$$

Applying the Euler method(2.4) to these two functions we end up with,

$$w_{new} = w - \eta \frac{\partial L}{\partial w} \tag{3.3}$$

$$b_{new} = b - \eta \frac{\partial L}{\partial b} \tag{3.4}$$

Where η is a step size of our choice. This step size is commonly known as the *learning rate* of our neural network. The use of these update functions is to iterate and upgrade the parameters, following the steepest gradient that minimizes the Loss function of our neural network model.
This process is continued until it is no longer possible to move further down, to lower loss values (i.e when a local minimum is found). It is noteworthy to mention that, the value of η determines the way our model parameters go towards the direction specified. The smaller the value of η, the more iterations the model will need to reach the desired outcome. However, a big enough value of η may lead to a "jump" and overshooting of the corresponding minimum that the model is in search of.

3.2 Lipschitz continuity

Now, let's call the loss function $L(w, b)$ (given specific input data and target). For a single neuron network and a sigmoid activation function applied we get the following,

$$O = wx + b$$

$$S = \frac{1}{1 + e^{-O}}$$

$$L(w,b) = \frac{1}{2}(t-S)^2$$

$$L(w,b) = \frac{1}{2}(t - \frac{1}{1+e^{-(wx+b)}})^2$$

With t being the label target.
First of all, we have to ensure that L is Lipschitz continuous(both in direction of w and b) so that it can be differentiated for the use of gradient descent method.
So we have to ensure that, for every w value,

$$\frac{|L(w,b_{i-1}) - L(w,b_i)|}{|b_{i-1} - b_i|} \leq M_1$$

Where M is a real number and greater than 0.
Now, we have to do the same for the direction of w parameters. For every b parameter,

$$\frac{|L(w_{i-1},b) - L(w_i,b)|}{|w_{i-1} - w_i|} \leq M_2$$

Now, in order to achieve the existence and uniqueness of the solution of the gradient descent method, we have to repeat the process, and define the weight and bias upgrade functions as Lipschitz continuous functions in a specified range of values. Therefore, let's denote the upgrade of the weight parameters as $\Delta W(w,b) = -\eta\frac{\partial L}{\partial w}$ and the upgrade of the bias parameters as $\Delta B(w,b) = -\eta\frac{\partial L}{\partial b}$.Then we made sure that the following stood (for every constant value of b),

$$\frac{|\Delta W(w_{i-1},b) - \Delta W(w_i,b)|}{|w_{i-1} - w_i|} \leq M_3$$

While as well, for every constant value of w,

$$\frac{|\Delta B(w,b_{i-1}) - \Delta B(w,b_i)|}{|b_{i-1} - b_i|} \leq M_4$$

Where of course,for all the above, $M_1, M_2, M_3, M_4 > 0$

Algorithm 1 Lipschitz Continuity

```
Loss function and update function=set
for j = 1 : length(data) do
    for i = 2 : length(data) do
        Defining the Lipschitz Continuity constraints for both the Loss
        function and the update function.
        |L_{i-1} − L_i| ≤ M_1|b_{i-1} − b_i| for the direction of b.
        |L_{i-1} − L_i| ≤ M_2|w_{i-1} − w_i| for the direction of w.
        |ΔW_{i-1} − ΔW_i| ≤ M_3|b_{i-1} − b_i| for the direction of w.
        |ΔB_{i-1} − ΔB_i| ≤ M_4|w_{i-1} − w_i| for the direction of b.
    end for
    If all the above are always True then Lipschitz Continuity exists.
end for
```

3.3 Phase portraits

In this section, the way the phase portrait figures were produced, will be analyzed. Firstly, a list of figures of a combined phase vector - phase portrait, for single neuron - single layer neural network, constant input data (labelled 'x') and a sigmoid activation function applied, is to be presented in the "Results" part of this project. The list of the figures that are to be displayed, are calculated using MATLAB® and the built-in function *ode45.m* . It is also of high importance to mention that, in these first experiments, while running the *ode45.m*, the gradient of the input data with respect to time, is fed to the ordinary differential system.

The sigmoid function takes the form of,

$$S(\omega) = \frac{1}{1 + e^{-\omega}}$$

Where ω in our neural network is,

$$\omega = wx + b$$

w, b being the parameters.
The quadratic loss function then comes to be,

$$L = \frac{1}{2}(t - S(\omega))^2$$

$$L = \frac{1}{2}(t - \frac{1}{1 + e^{-wx-b}})^2$$

The gradients of which, with respect to our model parameters,are,

$$\frac{\partial L}{\partial w} = \eta x e^{-wx-b} \frac{t - \frac{1}{(e^{-wx-b}+1)}}{(e^{-wx-b} + 1)^2} \tag{3.5}$$

$$\frac{\partial L}{\partial b} = \eta e^{-wx-b} \frac{t - \frac{1}{(e^{-wx-b}+1)}}{(e^{-wx-b} + 1)^2} \tag{3.6}$$

In that way, we get the equivalent update function,

$$H = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} = \begin{bmatrix} -\eta \frac{\partial L}{\partial w} \\ -\eta \frac{\partial L}{\partial b} \end{bmatrix} \tag{3.7}$$

This way, with a pre-specified step size η , in our case $\eta = 1$, we can use these differential equations for the update of our model and solve iteratively using the MATLAB® built-in functions to produce the phase vector and phase portrait plots.

Algorithm 2 Phase Portraits

```
initial conditions=set
for w0 = 1, 2, ... do
    for b0 = 1, 2, ..., N do
        Define the time span and run ode45.m
        Compute all the values of w,b by iterations and plot them.
    end for
end for
```

3.3.1 Nonlinear trajectories

The method however does seem to be applicable to a lot more complex implementations and forms. To start with, input data can be time variant. In such a case, phase vectors are of low importance, since they are computed for the initial conditions of the ODE system and not for each time step (e.g phase vectors are now a dynamic object and therefore, impossible to display). With that in mind, from now on, the plots and displays of our simulations are going to be in the form of phase portraits, excluding the non important part of the phase vectors.
In the corresponding section, 4.1 in displaying the results from this methodology. It is highly essential and noteworthy, to mention that the use of the built in function *ode45.m* is used for the production of all of the upcoming figures. When, later on, the use of *ode45.m* is to be neglected or altered, it is going to be noted.

3.3.1.1 Sinusoidal trajectories

In this subsubsection, the results of the non linear trajectories subsection 3.3.1, are to be displayed.
The input data used for the results below is

$$x = x_0 \sin 10z$$

With z corresponding to time and x_0 ,t varying.

3.3.1.2 Exponential Trajectories

Another interesting pile of non linear scenarios of input to output relationship, is the exponential input data, and the way the training in our neural network works. In order to get a substantial result that can provide us with scientific importance, it is again needed to evaluate the over time development of the loss function. After all, the loss function is the key factor when it comes to detecting convergence in the training process.
The input data for the below figures is to be given as following,

$$x = x_0 e^{10^{-3}z}$$

As previously, z variable is corresponding to time.

3.3.2 Target/Label related to input

While many of the different above examples are extremely useful to display and analyse, a direct relationship between the target/label and the input needs to be implemented. A function connecting the input x with the label t, that enables us to simulate many "physical" scenarios.

3.3.2.1 Second order functions

The fact that we were able to analyse different scenarios of various linear and non linear functions shaping our neural network and determining the convergence of it was great. However, it is now time to address the convergence of a specific set of functions that "govern" the neural network. What we will be tuning now is the learning rate η and the number of iterations of our ode45 solver.
At first, we are using the non linear set of equations,

$$x = x_0 sin(10z)$$
$$t = f(x) = x^2 - 0.5$$

The step size that is being chosen for our *ode45* solver to run is varying, while using the below initial conditions.

$$x_0 = 1, t_0 = -0.5$$
$$\eta = 0.005 \rightarrow \eta = 0.05$$

3.4 Average loss function

The current implementation takes place in accordance to theory 2.6,by applying a sigmoid activation function to our sinusoidal, here, data.

$$S(\omega)_{i,k} = \frac{1}{1 + e^{-(w_k x_k + b_k)}}$$
$$L_{i,k} = \frac{1}{2}(y_{i,k} - S(\omega)_{i,k})^2$$
$$\Sigma L = \sum_{k=1}^{N} L_{i,k}$$

So, the average loss function to be actually implemented is,

$$L_{average,sigmoid} = \frac{\Sigma L}{N}$$

This average loss function is tested with various data sizes and iterations/simulation durations.
For starters, we begin by creating our input as a vector of N=10 data points.

$$x = [-10, -7.77, -5.55, -3.33, -1.11, 1.11, 3.33, 5.55, 7.77, 10]$$
$$t = sin(x)$$

Where t is the true label data. $\eta = 0.005$ and solver iterations = 5000. The results are being displayed in the corresponding chapter.

3.4.1 Increasing the dataset points

Our next step now involves expanding the dataset and increasing the number of data points from 10 to N=1000, expanding and broadening our view on this analysis. So now we are going to work with the following,

$$x = [-10, -9.98, ..., 9.98, 10]$$

$$t = sin(x), x \in [-10, 10]$$

3.5 Batched gradient descent

Now that a clear implementation of the previous method has been shown, it is time for a next step method, keeping the same structure with the previous one, i.e. the one shown in 3.4. The input and output data are to be divided into small subsets called **batches**, using the results of each batch, as initial parameter data for the next one, using of course *ode45.m* multiple times. The gradient is computed every time for each batch and the point of this method is to lead to faster and more accurate convergence of our training model.
Let us now denote p as index of the number of batches, where $p = 1, 2, ..., q$ and q is the number of batches used. In addition, n is equal to the length of each batch(i.e. the number of data points contained in each batch) and $k = 1, 2, ..., n$.
We then have,

$$w_{1,p} = w_{n,p-1}$$

$$b_{1,p} = b_{n,p-1}$$

And then, after each batch calculation, calculating the corresponding average loss function over the time steps used in *ode45.m* (or any other solver),with respect to, of course, the sum of the all data points of each and every batch used.
The average loss function is computed in the same way that it was in 3.4, with the exception now, that the whole dataset is used for it's calculation at each specific time step.
To make that clear,taking the activation function for granted,

$$L(N)_{p,k} = \frac{1}{2}(y(N)_{p,k} - S(N)_{p,k})^2$$

$$\Sigma L = \sum_{k=1}^{N} L(k)_{p,k}$$

$$L_{average,p} = \frac{\Sigma L}{N}$$

Where now,

$$N = qn$$

Now, for a number of batches, $q = 10$ and data points stored in each batch, $n = 128$ we get the resulting trajectories and corresponding loss functions in Figure 4.14. Here, a sinusoidal dataset is used, with a sigmoid for an activation function and an average loss function for the whole dataset at each batch.

3.6 Discretization of the non linear differential, continuous system

The system of equations that derives from an artificial neural network with an activation function, a sigmoid in that case, is a non linear one with the form of:

$$\frac{\partial y_1}{\partial z} = \eta \cdot x e^{(-y_2 - y_1 x)} \frac{(t - \frac{1}{e^{(-y_2 - y_1 x)} + 1})}{(e^{(-y_2 - y_1 x)} + 1)^2} \tag{3.8}$$

$$\frac{\partial y_2}{\partial z} = \eta e^{(-y_2 - y_1 x)} \frac{(t - \frac{1}{e^{(-y_2 - y_1 x)} + 1})}{(e^{(-y_2 - y_1 x)} + 1)^2} \tag{3.9}$$

With $y_1, y_2 = w, b$ respectively, z being time and t the output target/ true label.

An endeavor of discretizing the corresponding differential system is to be displayed, first by presenting some discretization methods to be used, and secondly by applying those methods and demonstrating their results.

3.6.1 Newton's Method

Newton's method is an algorithm for approximating and then solving iteratively, non linear equations [5].The starting point for Newton's method is the general nonlinear vector $\mathbf{F}(y) = 0$. The idea is that F is approximated around y^- by a linear function $\hat{\mathbf{F}}$, calculated by the first two terms of a Taylor expansion of $\mathbf{F}$. In our multivariate case, those two terms become,

$$F(y^-) + J(y^-)(y - y^-),$$

where J is the *Jacobian* of $\mathbf{F}$, defined by

$$J_{i,j} = \frac{\partial F_i}{\partial y_j}.$$

The original non linear system is then approximated by,

$$\hat{F}(y) = F(y^-) + J(y^-)(y - y^-) = 0,$$

which is linear in y and can be solved in a two-step procedure,
· First solve $J\delta y = -F(y^-)$ with respect to δy.
· Update $y = y^- + \delta y$ until convergence.
A relaxation parameter can also be added and used,

$$y = y^- + \omega \delta y$$

.

3.6.2 Crack Nicolson - Discretization method

The Crack-Nicolson discretization method is a well-known method for discretizing non linear differential systems [5]. The scheme of the non linear system (3.8,3.9) reads,

$$\frac{y_1^{n+1}-y_1^n}{\Delta z}=\eta x e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}\frac{(t-\frac{1}{e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1})}{(e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1)^2} \tag{3.10}$$

$$\frac{y_2^{n+1}-y_2^n}{\Delta z}=\eta e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}\frac{(t-\frac{1}{e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1})}{(e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1)^2} \tag{3.11}$$

Replacing (y_1^{n+1}, y_2^{n+1}) with (y_1, y_2), (y_1^n, y_2^n) with (y_1^-, y_2^-), respectively, multiplying by Δz and moving all terms to the left-hand side, we get,

$$y_1-y_1^- -\Delta z(\eta\cdot x e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}\frac{(t-\frac{1}{e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1})}{(e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1)^2})=0 \tag{3.12}$$

$$y_2-y_2^- -\Delta z(\eta e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}\frac{(t-\frac{1}{e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1})}{(e^{(-y_2^{n+\frac{1}{2}}-y_1^{n+\frac{1}{2}}x)}+1)^2})=0 \tag{3.13}$$

The terms $y_1^{n+\frac{1}{2}}, y_2^{n+\frac{1}{2}}$ are to be computed using the linear interpolation, as followed,

$$y_1^{n+\frac{1}{2}}\approx\frac{1}{2}(y_1^{n+1}+y_1^n) \tag{3.14}$$

$$y_2^{n+\frac{1}{2}}\approx\frac{1}{2}(y_2^{n+1}+y_2^n) \tag{3.15}$$

Now, replacing (3.14) and (3.15) to (3.12) and (3.13) we get,

$$y_1-y_1^- -\Delta z\cdot\eta\cdot x e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}\frac{(t-\frac{1}{e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}+1})}{(e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}+1)^2}=0 \tag{3.16}$$

$$y_2-y_2^- -\Delta z\cdot\eta\cdot e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}\frac{(t-\frac{1}{e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}+1})}{(e^{-\frac{1}{2}(y_2+y_2^-+(y_1+y_1^-)x)}+1)^2}=0 \tag{3.17}$$

Which is of the form,

$$F(y)=0$$

And can be solved numerically using Newton's Method[3] & 3.6.1.

3.6.3 Euler discretization method

The equations (3.8) , (3.9) are now subjected to the Euler's discretization method [6], with the outcome of,

$$y_1^{n+1} = y_1^n + \Delta z f(z_n, y_1^n) \tag{3.18}$$

$$y_2^{n+1} = y_2^n + \Delta z f(z_n, y_2^n) \tag{3.19}$$

Now, from equations (3.8) , (3.9), one can derive function f.

$$f(z_n, y_1^n) = \eta x e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}$$

$$f(z_n, y_2^n) = \eta e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}$$

Replacing now the function f we get to our final system of equations when Euler's discretization method is applied,

$$y_1^{n+1} = y_1^n + \Delta z (\eta \cdot x e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}) \tag{3.20}$$

$$y_2^{n+1} = y_2^n + \Delta z (\eta \cdot e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}) \tag{3.21}$$

Moving all terms to the left hand side, we get the following expression,

$$y_1^{n+1} - y_1^n - \Delta z (\eta \cdot x e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}) = 0 \tag{3.22}$$

$$y_2^{n+1} - y_2^n - \Delta z (\eta \cdot e^{(-y_2^n - y_1^n x)} \frac{(t - \frac{1}{e^{(-y_2^n - y_1^n x)} + 1})}{(e^{(-y_2^n - y_1^n x)} + 1)^2}) = 0 \tag{3.23}$$

Which is, yet again, of the form

$$F(y) = 0$$

And can be solved numerically with the Newton's method (ref).

3.6.4 Single input- single output method

The equations already mentioned in this section can be used as they are for a specific, initial scenario. A specific constant input x and a constant output label t can be fed with the outcome of producing the desirable trajectory of the parameters w,b (here mentioned as y_1, y_2).
The methodology for this is as followed, Providing (3.16), (3.17) or (3.20),(3.21) with the constant input x and constant output t, the y_i^{n+1} is computed for the specified value of Δz in time. Following the same principle, one can loop over the number of steps and calculate the desired trajectory(i.e. the parameters w,b with respect to time).

For this method's application, the mathematics were calculated using Matlab's *Symbolic Math Toolbox* [7].

Algorithm 3 Single Input - Single Output

SET $(y_0 = [w_0, b_0], \eta = 1, \Delta z)$
SET input x and target t
for i=1:steps number **do**
    if i=1 **then**
        $y_{old} = y_0$
    end if
    if $i \neq 1$ **then**
        $y_{old} = y_{new}$
    end if
    Solve numerically using Newton's method
    $y_{new} = solutions$
    Calculate the values of the sigmoid loss function that is used.
end for

A first implementation of the Crack Nicolson, Euler and *ode45.m* solution methods are to be displayed in the corresponding, results' chapter, part, for a single input x and a single output t, a sigmoid activation function, a learning rate $\eta = 1$ and a step size Δz, after i iterations for our discretization methods.

Variants	i	Δz	w_0	b_0
Fig. 4.16	500	0.005	0.5	0
Fig. 4.17	50	0.005	0.5	0
Fig. 4.18	50	0.5	0.5	0

Table 3.1: Single input - Single output simulation parameters

3.6.5 Multiple input - multiple output

In this method, a predefined input vector x is created, with a corresponding function, predefined as well, that generates the output data vector t.
The specific method requires a differentiation from the defined equations (3.16), (3.17) for Crack Nicolson scheme or again, for Euler's (3.20),(3.21).

For a data input vector x=$[x_1, x_2, .., x_{10}]$ of length N and an output vector t=$[t_1, t_2, ..., t_{10}]$ we have,

$$\frac{F(y)|_{x=x_1,t=t_1} + F(y)|_{x=x_2,t=t_2} + ... + F(y)|_{x=x_{10},t=t_{10}}}{.N} = 0 \tag{3.24}$$

Which can of course be simplified to,

$$\frac{\sum_{i=1}^{N} F(y, x_i, t_i)}{N} = 0 \tag{3.25}$$

The equation (3.25) needs to be dealt with, the exact same way, as in the previous subsection 3.6.1, i.e. solving it with Newton's method. Afterwards, looping over a specific amount of time steps will produce the trajectories aimed.
The mathematics in this method were calculated using Matlab's *Function handle* function [8].

Algorithm 4 Multiple Input - Multiple Output

SET $(y_0 = [w_0, b_0], \eta = 1, \Delta z)$
input vector $x = [-10, ..., 10]$ and target vector $t = sinusoidal(x)$
for i=1:steps number **do**
    if i=1 **then**
        $y_{old} = y_0$
    end if
    if $i \neq 1$ **then**
        $y_{old} = y_{new}$
    end if
    Solve numerically using Newton's method
    $y_{new} = solutions$
    Calculate the values of the sigmoid loss function that is used.
end for

In the analogous **Results** section, the results of this method are to be shown, in comparison with, this time, three different continuous time solvers, *ode45*, *ode15s* and *ode23*. The discretatization methods *Crack Nicolson* and *Euler*, are to be compared with all three of the continuous solvers at different variants of the simulation with an input vector **x**, an output vector **t**, a sigmoid activation function, a learning rate $\eta = 1$ and a step size Δz, after i iterations for our discretization methods and only (the continuous built-in functions choose their own number of steps). More details about those variants are described in Table 3.2.

Variants	i	Δz	w_0	b_0
Fig. 4.19	200	0.005	0.5	0
Fig. 4.20	500	0.005	0.5	0
Fig. 4.21	2000	0.005	0.5	0
Fig. 4.22	20	0.5	0.5	0
Fig. 4.23	200	0.5	0.5	0
Fig. 4.24	2000	0.5	0.5	0
Fig. 4.25	50	2	0.5	0

Table 3.2: Multiple input - Multiple output simulation parameters

3.6.6 Experiments motivation

In the next chapter, the figures and plots presented at each section, are directly correlated with the method applied to them and therefore, with the section that this corresponding method is previously mentioned. In that way, everything is to be connected in the exact way that the research and work for this project took place.

4

Results

In this chapter, the results of the project are going to be displayed. The comments made for each and every plot/figure are presented in the following chapter[5].

4.1 Phase portraits

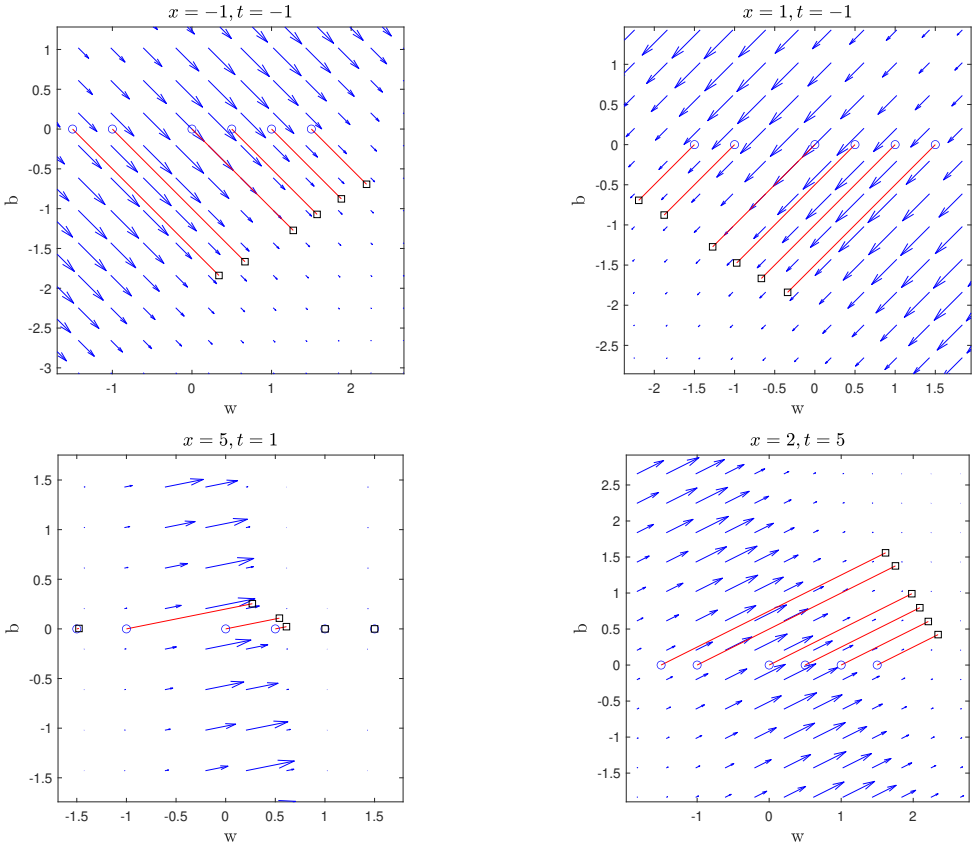

Figure 4.1: Trajectories for different values of input 'x' and target 't' and a sigmoid activation function

The trajectories of five different initial points (the ones circled) are shown in Fig. 4.1, where these points end up after 15 steps in time, in the squared ones. The outcome was expected as the model is linear and the input and output data are single.

4.1.1 Nonlinear trajectories - Sinusoidal trajectories

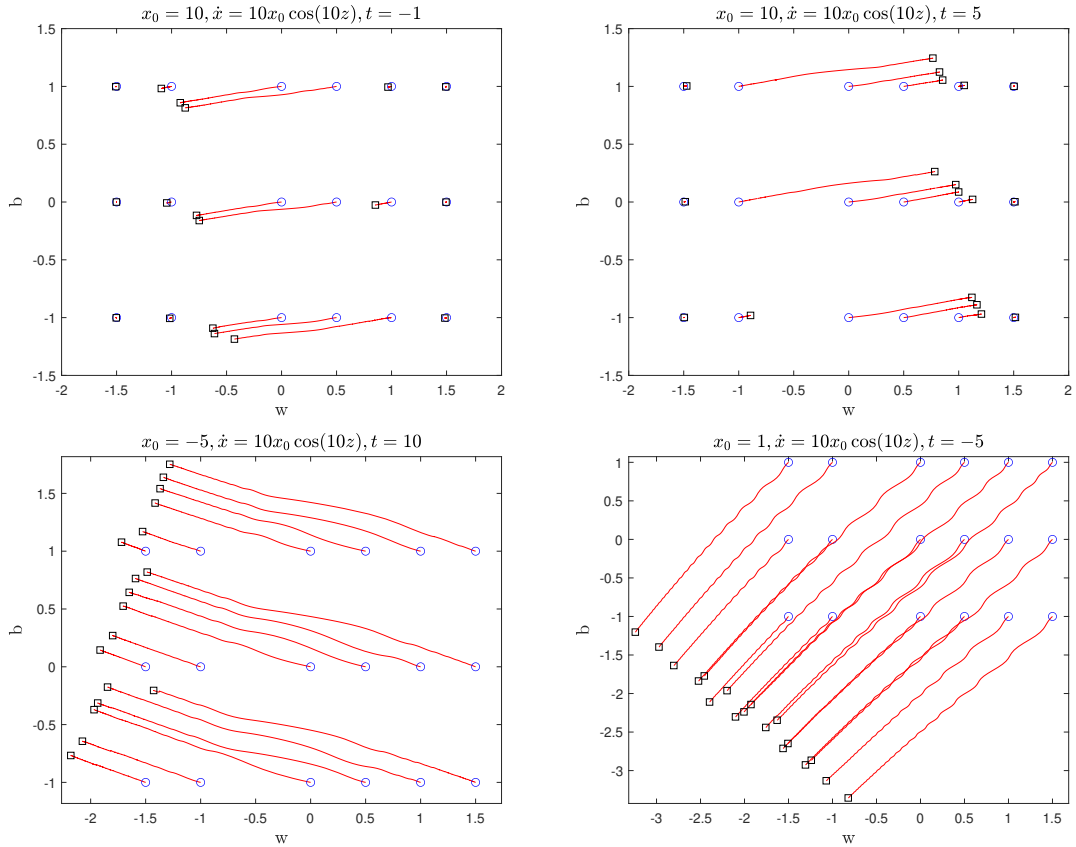

Figure 4.2: Trajectories for sinusoidal input data x and a sigmoid activation function

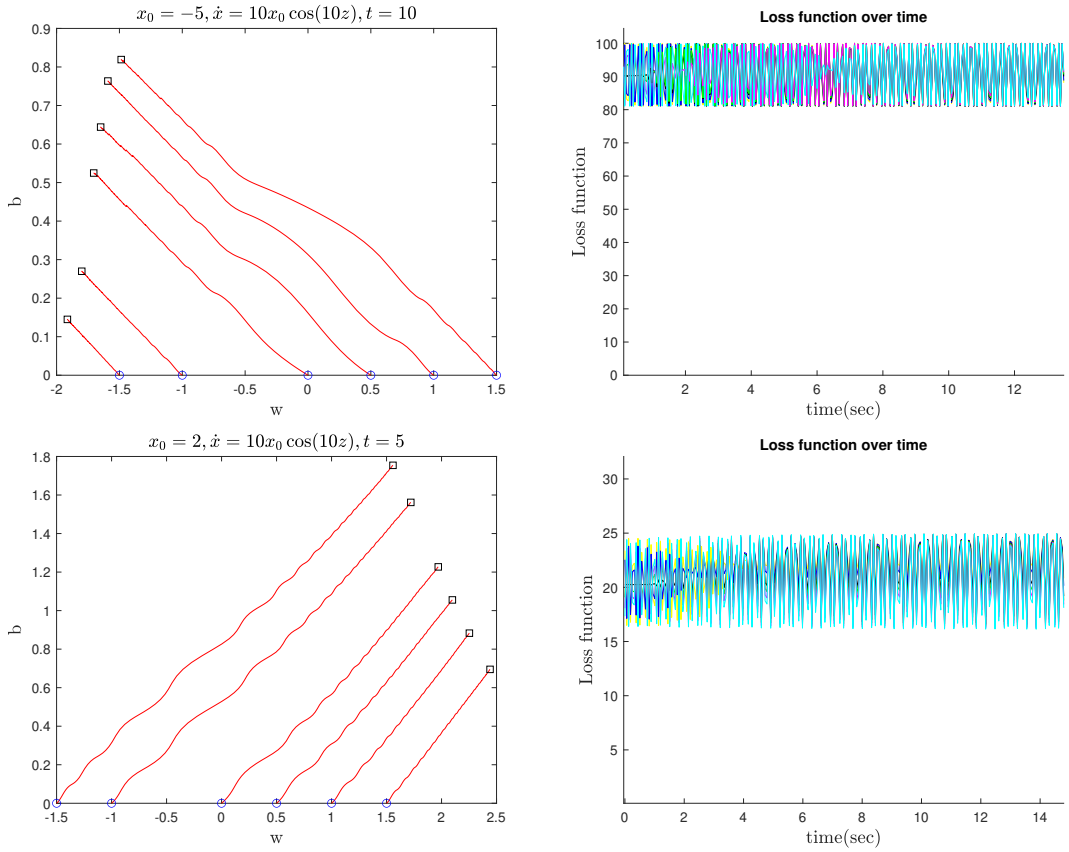

Figure 4.3: Sinusoidal trajectories and corresponding loss functions

In the above figure 4.3, it can easily be concluded that the corresponding loss function is fluctuating and not converging over time, as far as the specified characteristics of the data are taken into account.

4.1.2 Exponential trajectories

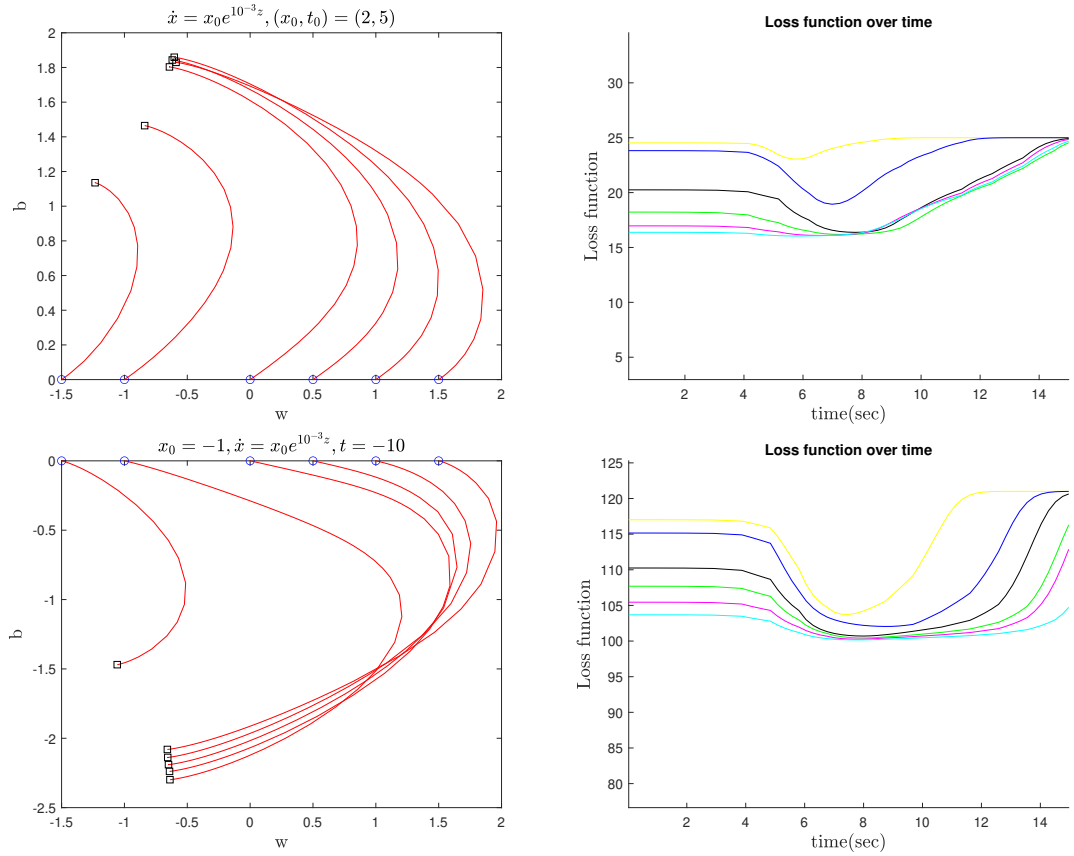

Figure 4.4: Exponential trajectories and corresponding loss functions

Keeping the same format and idea, since the loss functions are having, even in their state space behaviour, pretty high values, we are now displaying the trajectory simulation for a different target, one corresponding to our sigmoid's output range(i.e. from 0 to 1).

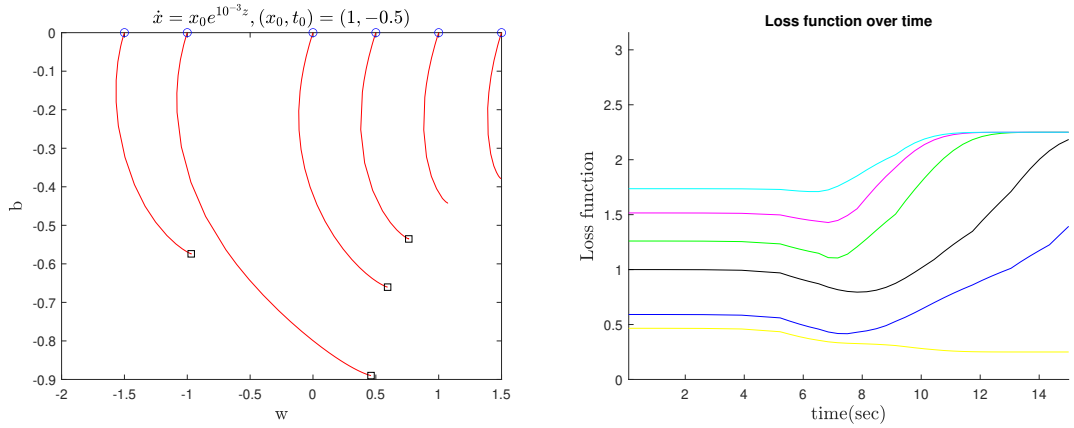

Figure 4.5: Exponential trajectories and corresponding loss functions. $(x_0, t_0) = (1, -0.5)$

4.1.3 Target/Label related to input

4.1.3.1 Second order functions

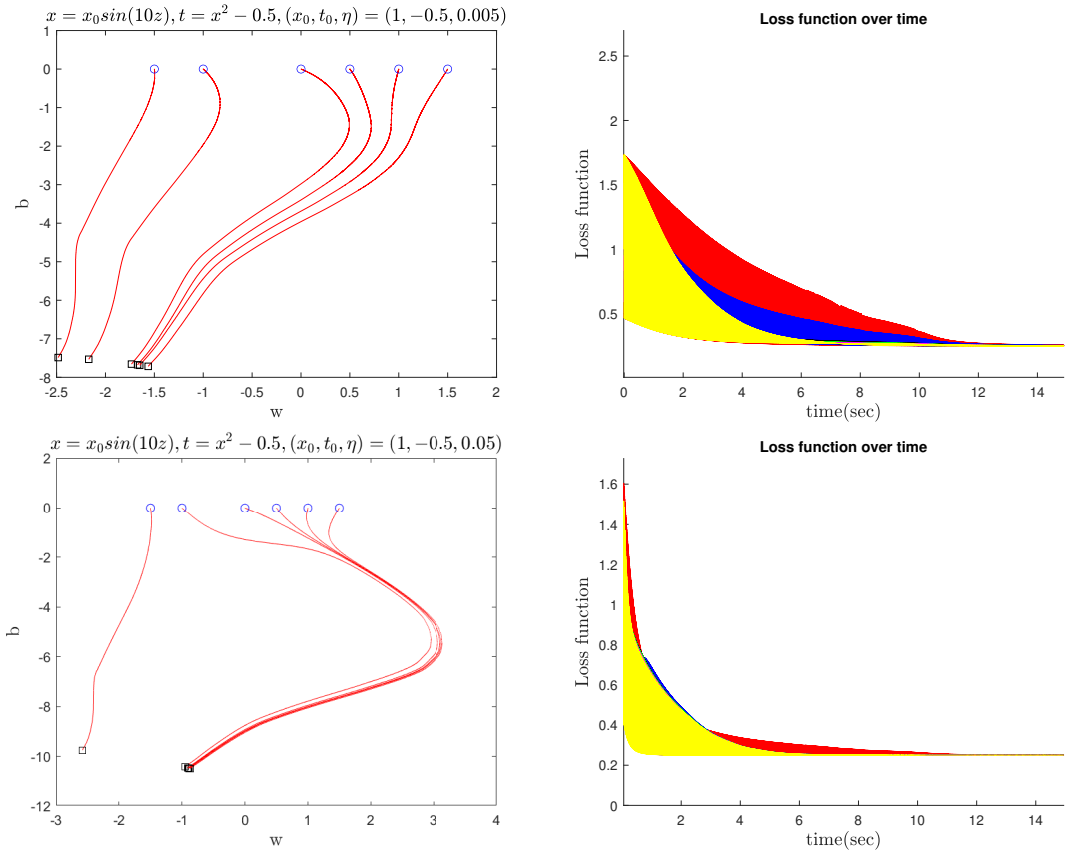

Figure 4.6: Non linear trajectories and their corresponding loss functions. Varying learning rate.

As it is highly logical, now, a different input-output relation is used so that we can guide the output label to belong in the range of [0,1], so that our corresponding sigmoid activation function can cover for it.
So,

$$x = x_0 sin(10z)$$

$$t = f(x) = x^2$$

$$x_0 = 1 \rightarrow t_0 = 0$$

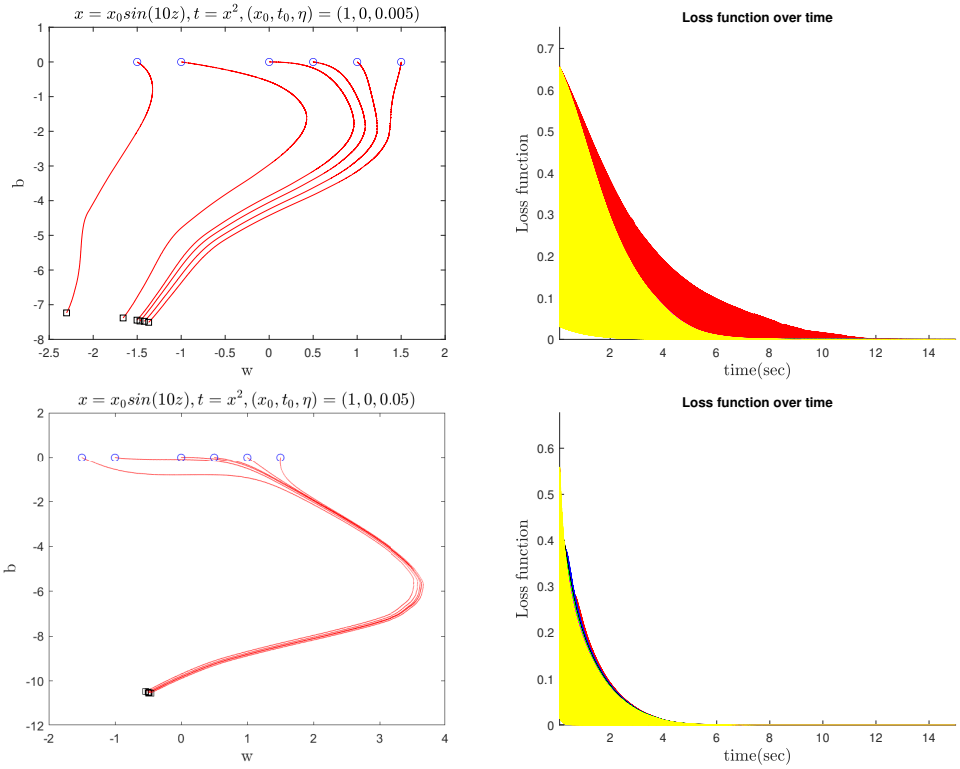

Figure 4.7: Non linear trajectories and their corresponding loss functions. Changed function.

4.2 Average Loss Function

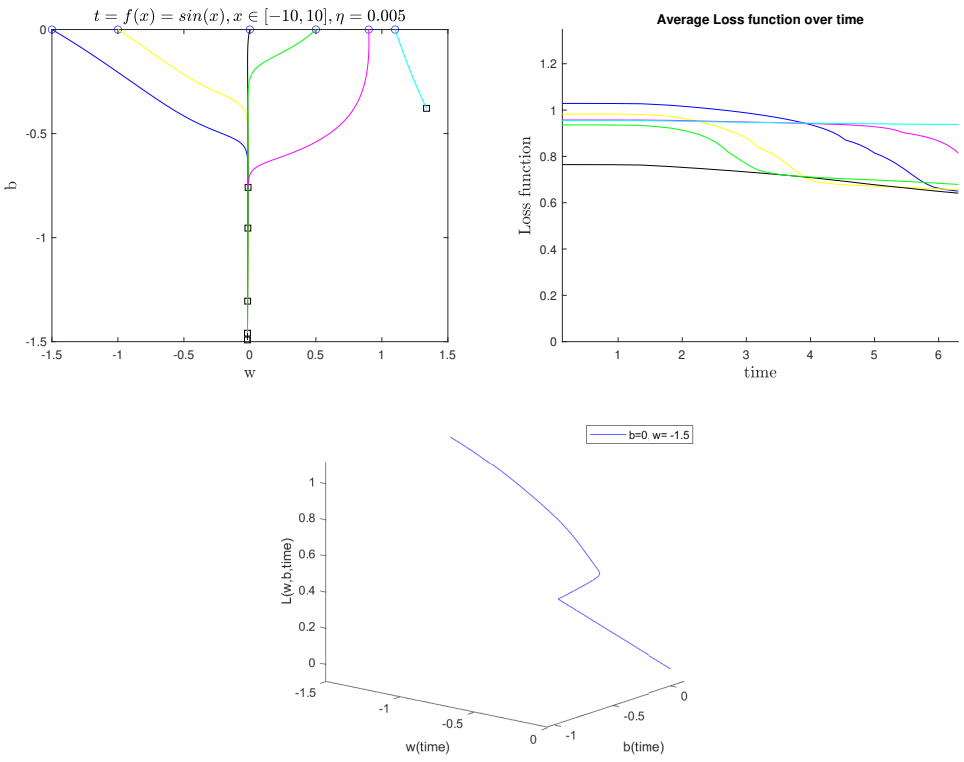

Figure 4.8: Sinusoidal data - Average loss function method. Number of steps = 5000 , N=10

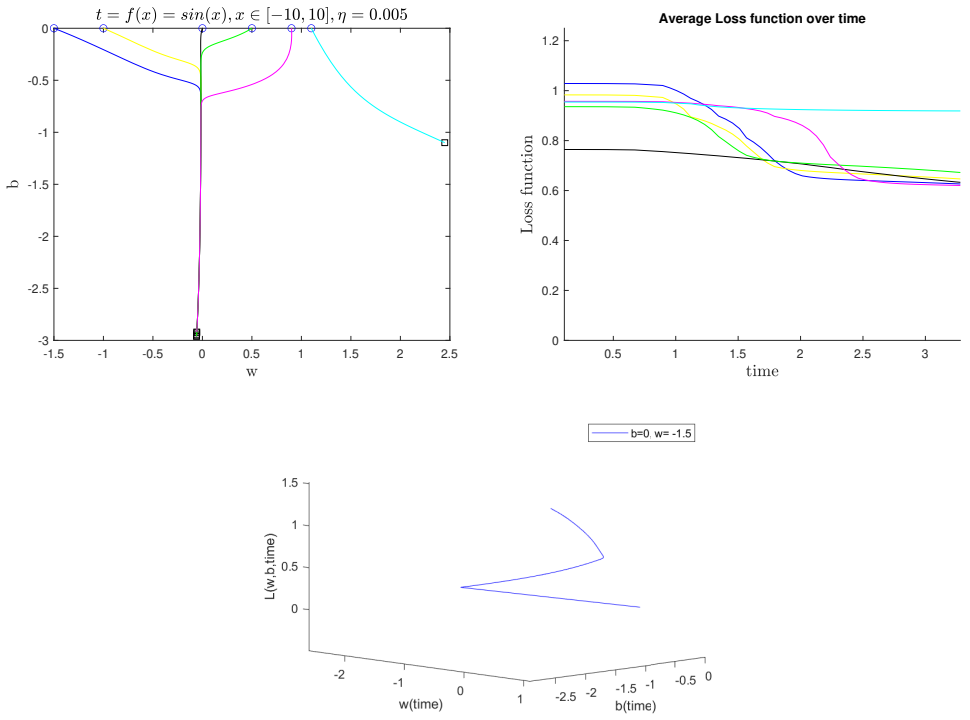

Figure 4.9: Sinusoidal data - Average loss function method. Number of steps = 50.000, N=10

4.2.1 Increasing the dataset points

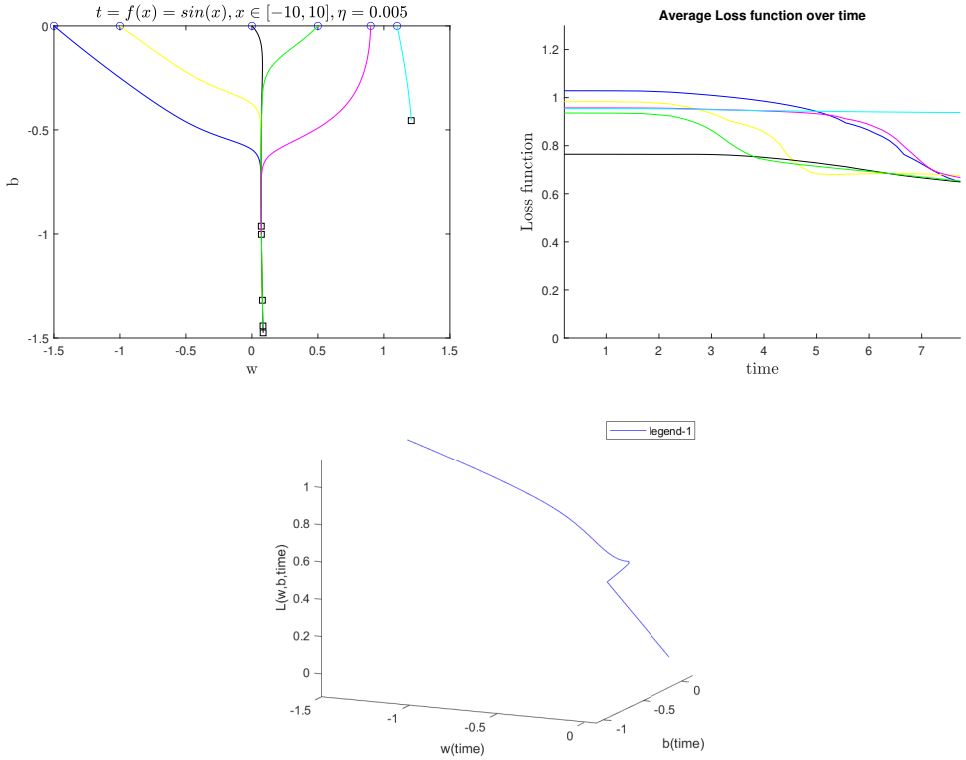

Figure 4.10: Sinusoidal data-Average loss function method. Number of steps/iterations=5000, N=1000

Nothing seems to be drastically changing after the plots in Figure 4.10. We shall continue however, with the same thinking as previously, increasing the iterations to 50000.

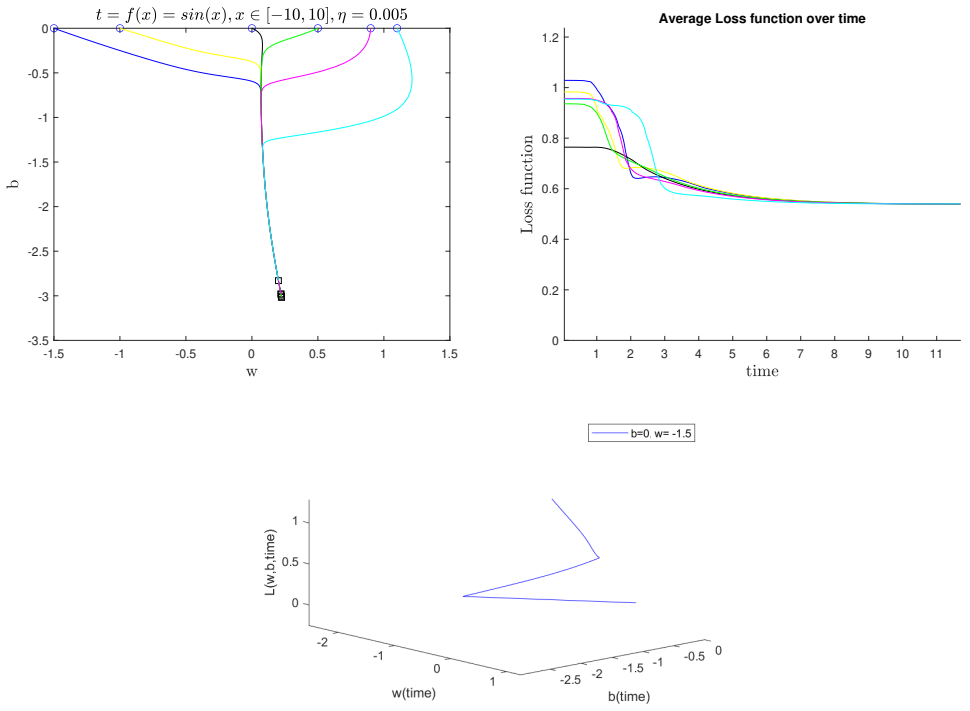

Figure 4.11: Sinusoidal data-Average loss function method. Number of steps/iterations=50000, N=1000

4.2.2 Various initial parameters conditions

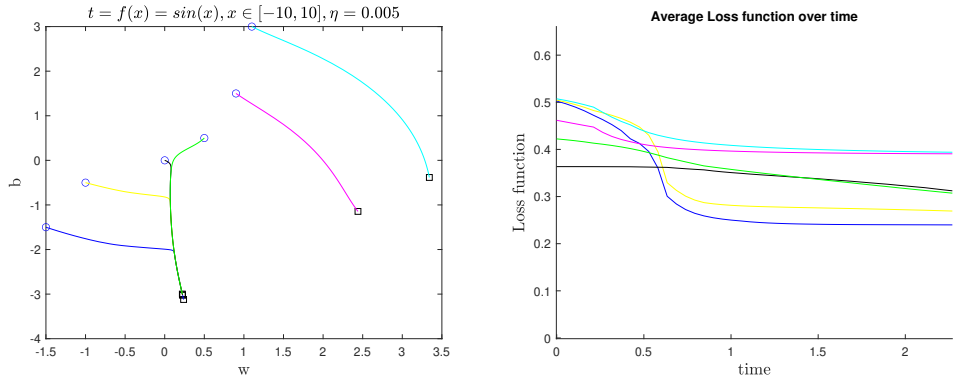

Figure 4.12: Implementation with various initial b paramater values, with sigmoid activation function.

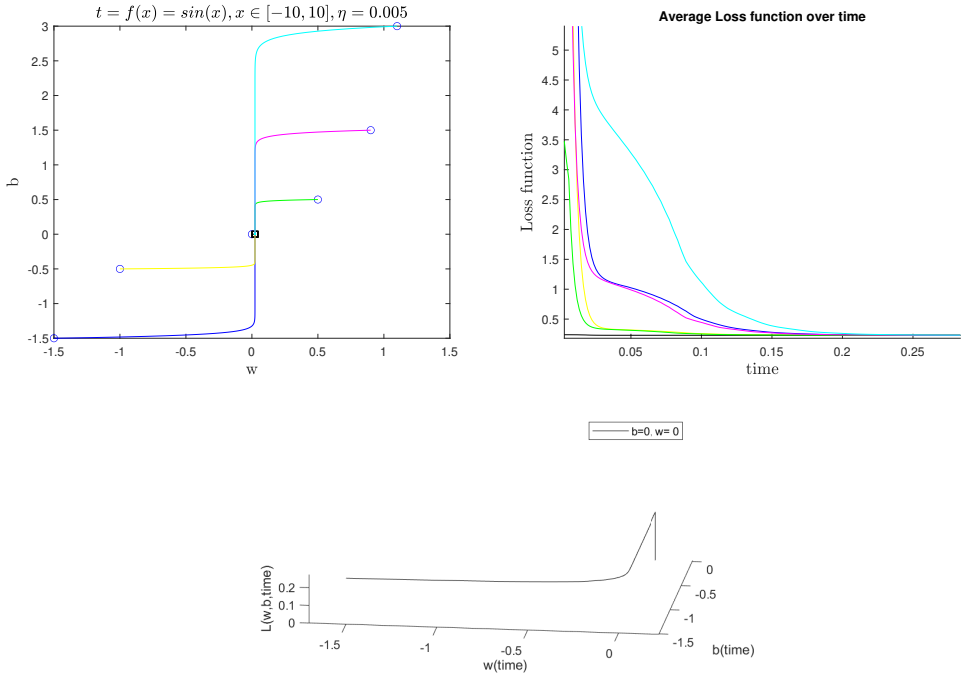

Figure 4.13: Implementation with various initial b parameter values, without any activation function.

4.3 Batches Method

This section is in correspondence to the methods section 3.5.

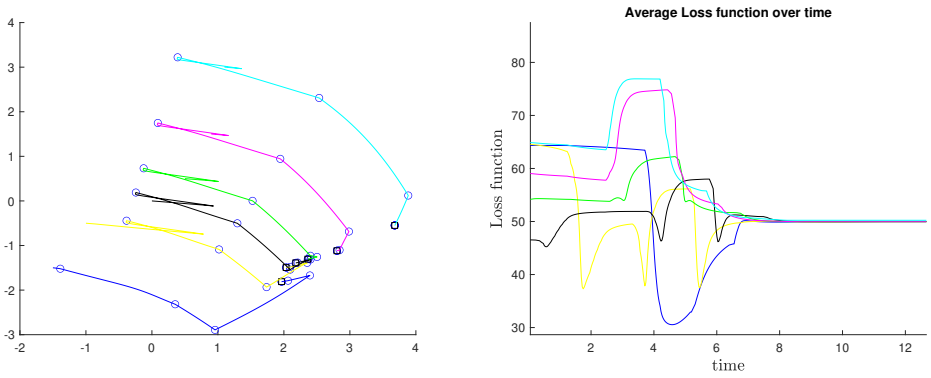

Figure 4.14: Various trajectories, using the batches method, and their corresponding average loss function, for sinusoidal input.

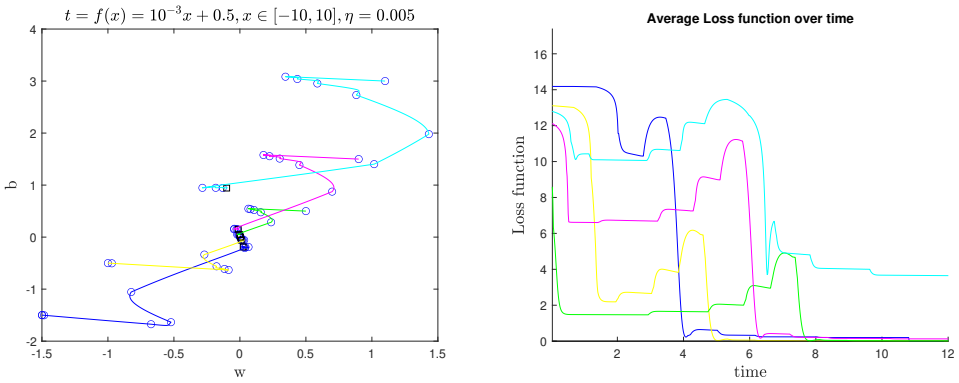

Figure 4.15: Various trajectories, using the batches method, and their corresponding average loss function, for linear input.

4.4 Discretization of the non linear differential, continuous system

4.4.1 Single input - Single output

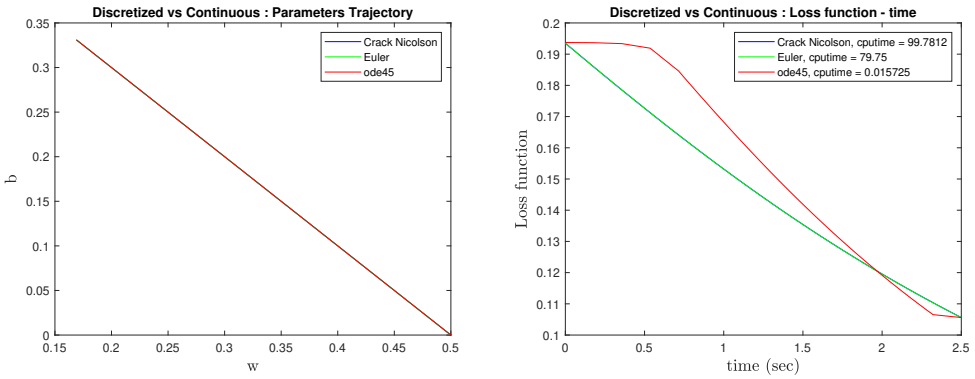

Figure 4.16: Single input - Single output implementation of the discretization methods and the continuous *ode45.m* case. Simulation No.1 .

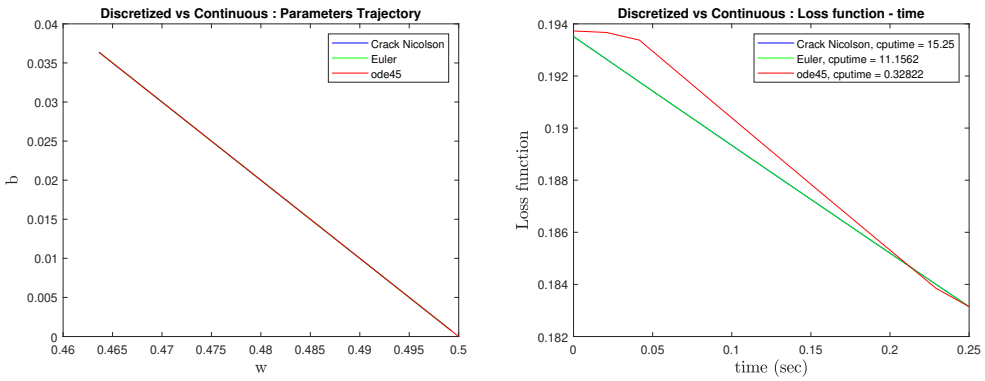

Figure 4.17: Single input - Single output implementation of the discretization methods and the continuous *ode45.m* case. Simulation No.2 .

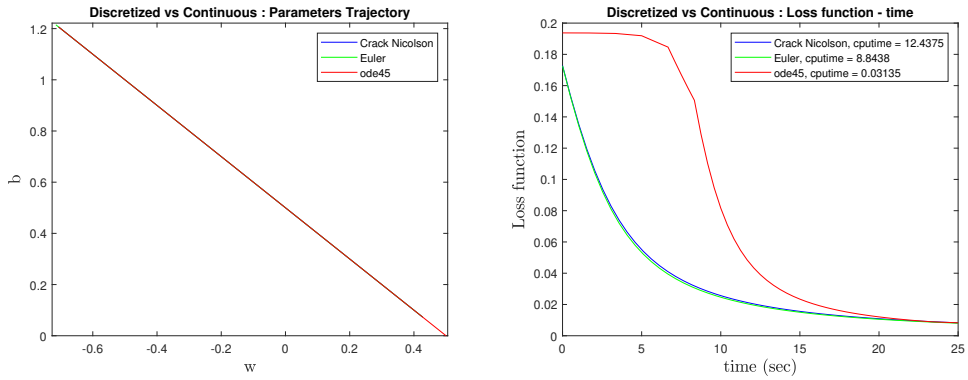

Figure 4.18: Single input - Single output implementation of the discretization methods and the continuous *ode45.m* case. Simulation No.3 .

4.4.2 Multiple input - Multiple output

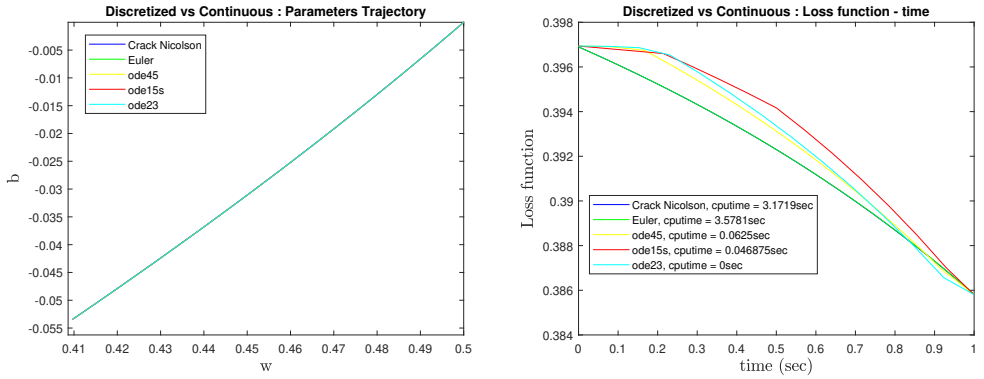

Figure 4.19: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.1 .

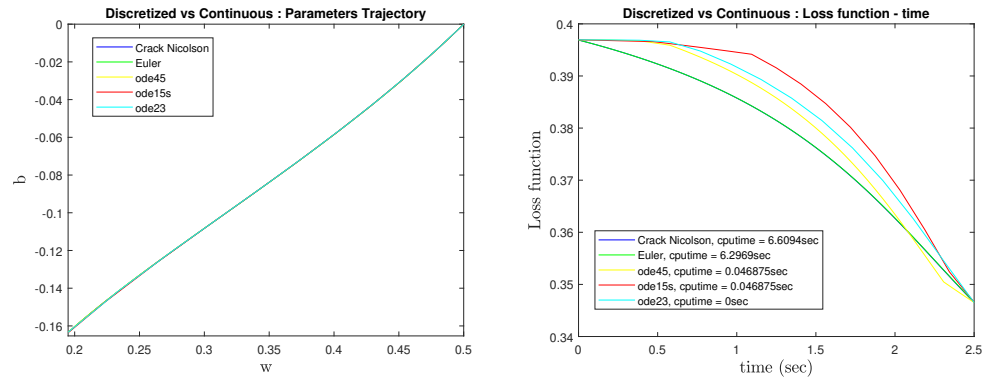

Figure 4.20: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.2 .

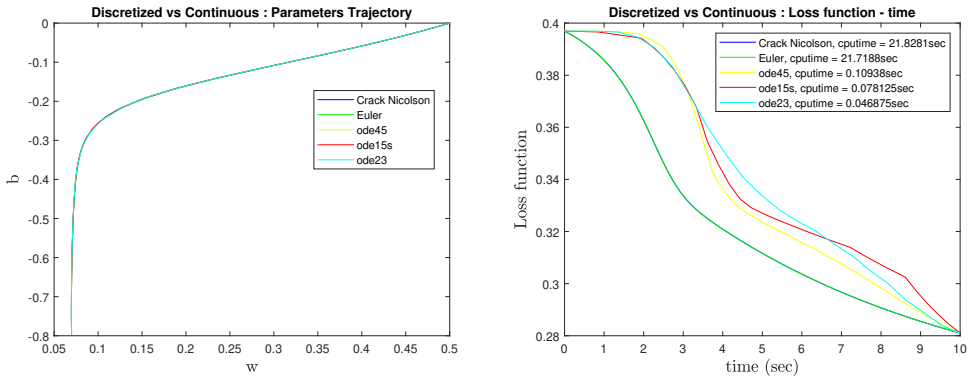

Figure 4.21: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.3 .

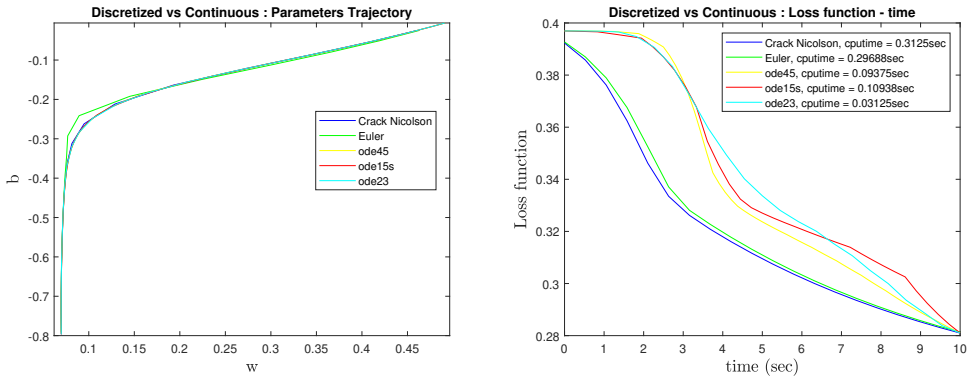

Figure 4.22: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.4 .

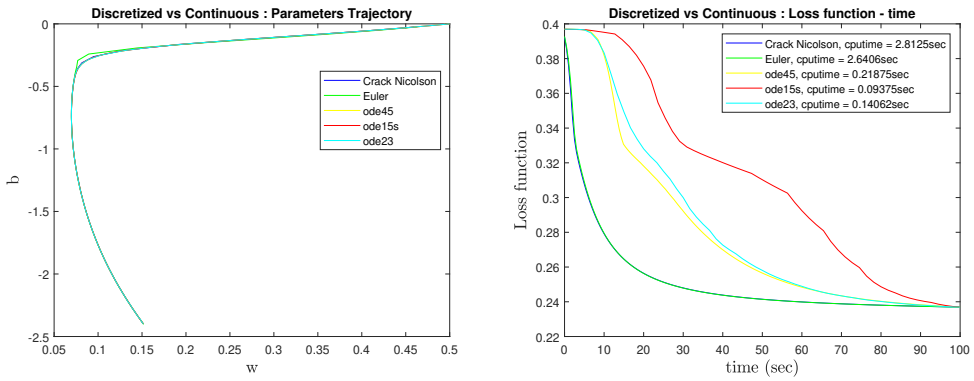

Figure 4.23: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.5 .

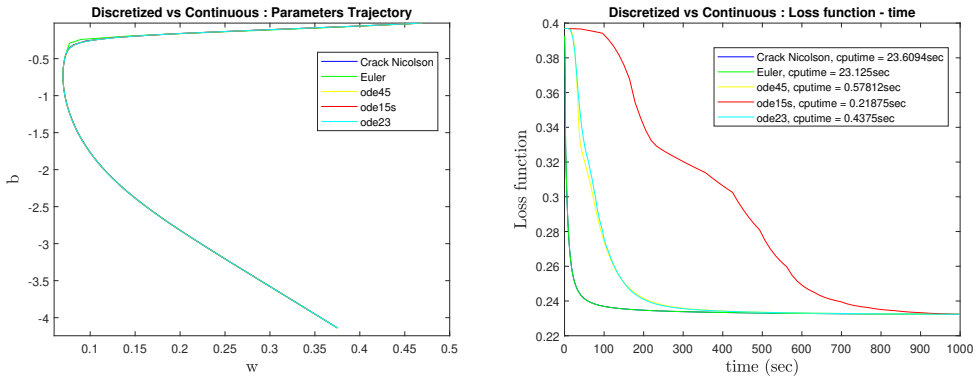

Figure 4.24: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.6 .

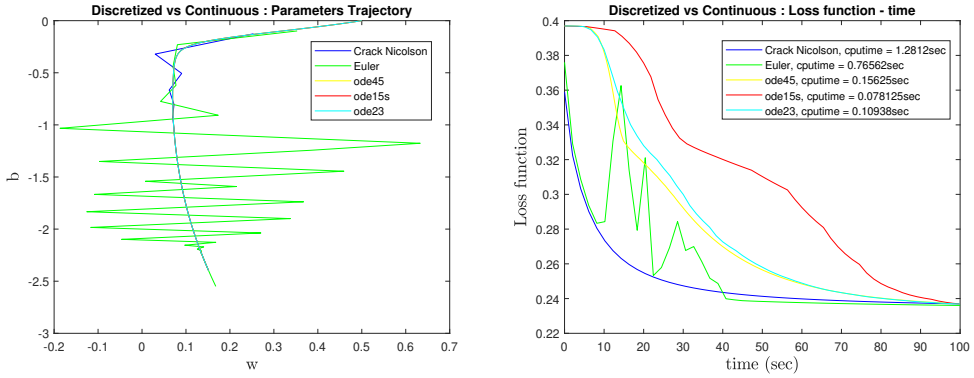

Figure 4.25: Multiple input - Multiple output implementation of the discretization methods and the continuous cases. Simulation No.7 .

5
Discussion

Here, in this chapter, some comments regarding the outcome of the Results chapter[4], are to be made.

As a starting point, one can easily tell from Fig. 4.1 that there is limited information as far as convergence is concerned. In order to move forward, some experimental trajectories accompanied by their loss functions over time, are going to be displayed and shed light on the matter.

With a brief look on Figure 4.2, one cannot directly imply anything about the convergence, before more information is acquired.The model seems to be continuously fluctuating. Analysing the loss fuction figures further on, we can tell that there is no convergence achieved.

Considering the results of 4.1.2, what can be commented with certainty is that after plotting Fig. 4.4 the above exponential trajectories, no matter what the initial conditions and targets are, cannot be subjected to training convergence, at least not after the specific amount of time given at the ode45 Matlab solver. It's also noteworthy to mention, that the loss function plots seem to be ending up at a state space value and not fluctuate, in contrast to what was happening when a sinusoidal input was given. This is probably due to the algorithm being stuck in a local minimum. To sum things up, this behaviour is expected when a single-neuron neural network is to be trained. One cannot expect to get convergence of a complex nonlinear input with a single weight and bias parameter. The technique/method used and the characteristics of our experiment (initial value, gradient of the input data function, etc) may also be to blame.
The same behaviour is now witnessed in Fig. 4.5, with the difference that in this case, the loss functions seem to be converging in a lot lower values than previously(Fig. 4.4).

Moving on with the average loss function section, in the plots of Figure 4.9, when comparing them with the previous from Figure 4.8 what can be concluded is that the behaviour of the loss function does not seem to be changing. The 3 dimensional plot seems to have a little differentiation, which is normal, taking into consideration the fact that the *ode45.m* has been running for a longer period of time. Finally, comparing the trajectories plots we can clearly now see the convergence of five of the trajectories and the certain divergence of the sixth, cyan coloured trajectory.
After increasing the dataset points, what is of great importance after analyzing the

last plots from Figure 4.11, is that the sixth trajectory, the cyan coloured, now joins the others and is led to convergence and at the same time, it's corresponding loss function seems, as expected, to be decreasing significantly, until stabilization.
We can now, with certainty say that this increase in the number of the data points used for the training of our neural network, led to an increase of the range of convergence for the trajectories with different initial values of our weight and bias parameters.

Using the batches method, as it is expected, in Figure 4.14, the average loss function seems to be fluctuating at first, but finally, it converges for every trajectory at the same value, meaning that this method seems to be working in the most desirable of the ways.
Another comment to be made on this figure, is the fact that the convergence value seems to be higher than some loss function values from previous time steps. The reason for this, is that the average loss function used, tries to lower exactly that, the average loss for the whole dataset, and at times, for a specific batch, it might get to lower values, but it's impossible to retain them while moving on to the next batches.

Now on to the discretization part. It is clear from Figures 4.16, 4.17 that the discretization methods both work efficiently in that running times, with low step size Δz, even though the cputime for the *ode45* is vastly smaller. However, when we start testing bigger running times (Fig. 4.18), while trying to retain a similar cputime difference by increasing the step size, we start detecting dissimilarities, which are mostly visible in the *Loss function - time* plot.
It is also very important to mention that the computation time for our methods, is extremely big, as it is expected due to using *symbolic* functions [7]. Hereby, we are moving to the next section to continue and discuss our experiments, in the general, more precise scope of the multiple input - multiple output field, where we will be using *function handle* for our discrete time computations.

In the above figures (Fig. 4.19,4.20,4.21,4.22,4.23,4.24,4.25), what can easily be commented, is that for the specified characteristics, *ode45* seems to be covering the curvature in a more precise manner, since it is using smaller step size, whereas, *ode23* is the one that is a lot faster, missing of course some information. However, in this case *ode15s* is neither fast nor precise.

6
Conclusion

At first, introducing and creating neural networks consisting only of one neuron has been a takeaway that we can say with confidence was a strong stepping stone for the rest to come. The trajectory of the model would align entirely with the phase vectors even though many different input data points were tested.

This is the point were we started testing out the performance of the, single neuron, neural network when it faced more exquisite, non linear scenarios of data progression over time. At the beginning, some simple single input data with sinusoidal gradient over time are being fed to the network. The result of this experiment led us to a conclusion that we cannot really lead this model to convergence (through loss function reduction) with only one neuron in it, due to the periodical nature of the data.

We continued, sticking to the single neuron model, with exponential gradient data that could actually be led to some sort of convergence, but not giving us the required loss function values. Still, the single neuron model could not keep up with this steep change of the input.

We then moved on with attributing a connection between the target and the input data, keeping the single neuron - single layer model as is. The outcome of that was a clear convergence, at first not at the desired values due to the output target but finally, at loss function values around zero.

Next milestone in this research was, introducing the average loss function in our model, in order to produce a more well established and generic training procedure that covers various types of input data with accuracy. At the beginning of this attempt, the results were not satisfying enough to cover for our convergence criteria. Increasing both the data set points(moving away from the single neuron - single layer neural network to a multiple neuron - single layer model) and the training iterations the outcome was clearly improved, leading to convergence although with a loss function not that close to zero.

An extra step to that was including the batches method to our endeavor which sure was a big improvement in our results.

As a final attempt, a discretization of the non linear differential continuous equations for the update function, where calculated. The point of this pursuit was to try

not to rely on the Matlab ODE (ordinary differential equation) solvers and seek a unique and original solution. Hence, the comparison of the discretized solutions and the Matlab solver ones, gave us a huge insight regarding this scientific direction, in this research project. To sum up, the main takeaway of this final part is that the solutions that this research provided were usually equally good at constructing the parameters' trajectories and in some cases even better (Fig. 4.25). The weak point of this try, which is a substantial one of course, is the computing time of our solvers. That is were the Matlab solvers would, mainly because they have a shifting learning rate η, be greatly more competent.

As a future potential research on top of this one, one can directly suggest that there are plenty of options. Although that more neurons have been used in this research, it is highly likely that more layers can provide a lot more robust solutions and convergence competent models. In addition, regarding the discretized solvers at the end, a shifting learning rate could really be of aid, when it comes to computing time. The usage of an alternate η during the training of the network could hugely avoid unnecessary computing of parts of the trajectory where the gradient does not really change.

Bibliography

[1] S. Lek, Y.S. Park, "Artificial Neural Networks", Editor(s): Sven Erik Jørgensen, Brian D. Fath, Encyclopedia of Ecology, Academic Press, 2008,ISBN 9780080454054, https://doi.org/10.1016/B978-008045405-4.00173-7

[2] Robert A. Meyers, "Encyclopedia of Physical Science and Technology" , 2002

[3] C. T. Kelley, "Solving Nonlinear Equations with Newton's Method", "https://doi.org/10.1137/1.9780898718898", *North Carolina State University, Raleigh, North Carolina*

[4] Hassan K. Khalil, "Nonlinear Systems", *Michigan State University*, 1995

[5] Hans Petter Langtangen, "Solving nonlinear ODE and PDE problems", *Center for Biomedical Computing, Simula Research Laboratory*, *Department of Informatics, University of Oslo* , 2016

[6] Michael Zeltkevic, "Forward and Backward Euler Methods", https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html, 1998

[7] MATLAB®, "Symbolic Math Toolbox", https://www.mathworks.com/products/symbolic.html

[8] MATLAB®, "Create Function Handle", https://www.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html

[9] "Lipschitz Continuity Theorem", https://en.wikipedia.org/wiki/Lipschitz_continuity

[10] Crank, J. , Nicolson, P. (1947). ,"A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type"

[11] "Newton's method", https://en.wikipedia.org/wiki/Newton%27s_method

[12] "Neural Network", https://unsplash.com/photos/8bghKxNU1j0

[13] "ANN", https://pixabay.com/vectors/neural-network-thought-mind-mental-3816319/

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY