Report for Final Project from group A

Authors:	MELIH MERT AKSOY (03716847) ATAMERT RAHMA (03711801) ARDA YAZGAN (03710782)
Last compiled:	2022–07–27
Source code:	https://github.com/mlcms-SS22-Group-A/mlcms-SS22-Group-A/tree/FinalProject

The work on tasks was divided in the following way:

MELIH MERT AKSOY (03716847)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
ATAMERT RAHMA (03711801)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
ARDA YAZGAN (03710782)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

Report on task Task 1/5: Summary of the paper contents

Continuous-time nonlinear signal processing: A neural network based approach for gray box identification

The first paper (release date 1994) investigates a gray-box approach for solving Ordinary Differential Equations (ODE). ODEs can also be solved using a usual black-box approach where the derivative descriptor of the ODE is approximated using a Neural Network (NN). To achieve this, the training of the neural network is embedded into a numerical integrator scheme. The authors of this paper (R. Rico-Martinez, et al.) attempt to capture the behaviour of a dynamical system by "hard-wiring" the known parts of an ODE and trying to approximate only the unknown parts using a NN. The different approaches that are taken (black-box vs. gray-box) are also made clear in the following formulations:

$$\dot{\vec{X}} = F(\vec{X}, \vec{p})$$
$$\dot{\vec{X}} = G(\vec{X}, \vec{p}) + F(\vec{X}, \vec{p})$$

To briefly explain the used notations, $\vec{X}$ is the vector state of variables, $\vec{p}$ is vector of parameters and $\dot{\vec{X}}$ is the vector of state variable derivatives w.r.t time. The black box approach corresponds to the first formulation, where we train a neural network to approximate the right hand side of the equation. In the gray box approach (second formulation above), right hand side of the ODE is split into two parts, where $G(\vec{X}, \vec{p})$ represents the part that is known, and $F(\vec{X}, \vec{p})$ is the unknown part (only this part is approximated with a NN). It is to be noted that the gray-box approach is not restricted to models of additive form.

Nonlinear system identification using neural networks: dynamics and instabilities

Same authors from the previous paper discuss the advantages and drawbacks of modelling continuous-time systems using discrete-time delayed-based NN and continuous-time models by comparing the attractors (instabilities, bifurcations etc.) of experimental systems from two chemically reacting systems exhibiting oscillatory behavior with the attractors of the approximated NNs. For long-term predictions, continuous-time models allow better capturing of the dynamical behavior.

Report on task Task 2/5: Setting up the neural network for Euler's method

In this task we want to model a feedforward neural network (FFNN) for modeling an ODE system in the form of $\dot{\vec{X}} = F(\vec{X}, \vec{p})$. We omit the vector symbol in this report for readability.

Problem Statement

First consider a nonlinear dynamical system described by the following ODE:

$$\dot{X}(t) = F(X(t))$$

with initial condition

$$X(0) = x_0$$

Where the state vector $X(t) \in \mathbb{R}^2$ and time $t \in [0, T]$. Our model should be able to predict the states $X(t)$ over the times in $[0, T]$ with high precision. In order to be able to do the training successfully, we assume that the start trajectory belongs to a bounded open domain of $\mathbb{R}^2$, the training set does not contain any noise, and F is time-invariant such that we can use different time-step sizes and later we will explain how we create the training dataset accordingly.

We focus on the dynamical systems resulted from the Andronov-Hopf bifurcation defined in the following equations in lecture.

$$\dot{x_1} = \alpha x_1 - x_2 - x_1(x_1^2 + x_2^2)$$
$$\dot{x_2} = x_1 + \alpha x_2 - x_2(x_1^2 + x_2^2)$$

The Euler Method

One way of approximating ODEs with an initial value is using the explicit Euler method [1]. It is the simplest version of the Runge-Kutta method [2], which we will inspect more deeply in the report of the next task. The Euler method is a first-order method (depends on the derivative and the previous result). Figure 1 shows a graphical visualization of the Euler method. The real unknown curve that we want to approximate is depicted in blue, and its approximation in red. Assuming we know the initial state $X(0) = A_0$ and its slope from the differential equation, then we can approximate the state $X(0 + \Delta t)$ by taking a step along the slope tangent of A_0. Ideally we want the error to be small, so we would want to use a sufficiently small step size Δt. Here we reach $A_1 \approx X(0 + \Delta t)$. We know assume that A_1 is on the blue curve, and repeat the same process to get an approximation for the state at time $0 + 2\Delta t$. In general we have $X(t + \Delta t) \approx A_{n+1} = A_n + \Delta t * F(A_n)$ with its computational graph depicted in figure 2.

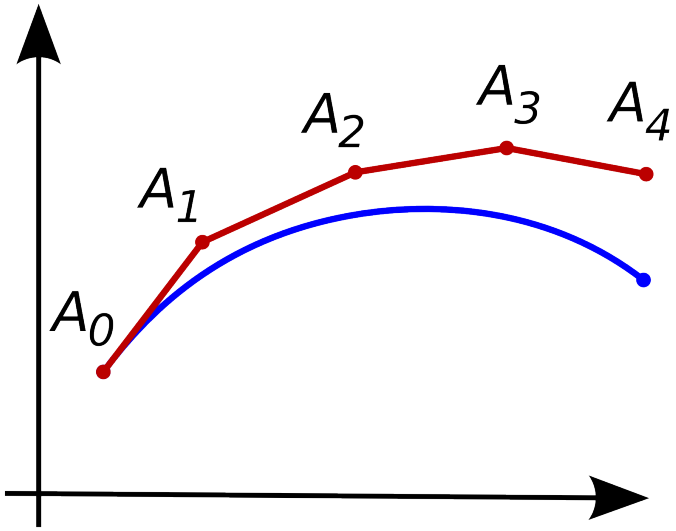


Figure 1: Euler method graphical visualization from wikipedia. The blue line is the unknown function, the red one is its approximation.

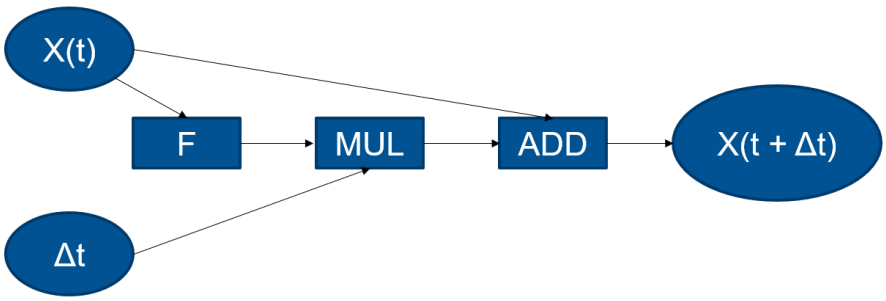


Figure 2: Computational graph of the Euler method.

The local error of the Euler method is the error between the approximated solution at time $t_{n+1} = t_n + \Delta t$ and the exact solution at time t_{n+1}. It is proportional to the square of the step size Δt (we omit the derivation), so we can reduce this error arbitrary small by choosing a sufficiently small Δt. The global error is the error at a fixed time t. There are $(t - t_0)/\Delta t$ steps until the state at time t is reached, and at every time step the local error is proportional to $(\Delta t)^2$. The global error is therefore proportional to the step size Δt.

Proposal

We have worked with NNs in this practical course before, and we know the universal approximation theory of NNs, that is, *"[..] operator NNs of bounded width and arbitrary depth are universal approximators for continuous nonlinear operators."* [3]. Therefore if we can formulate a dynamical system that can define its states, then we can also approximate any nonlinear function within this formulation, and can approximate the transition function of the dynamical system – in our case: its time derivative. The aim of this task is to construct such a NN where we approximate the function F and consequently approximate the ODE with an initial value using the Euler method explained in the previous subsection:

$$X(t + \Delta t) \approx \hat{X}(t) + \Delta t * NN(\hat{X}(t))$$

where $\hat{X}(t)$ is the previously approximated value, and NN is the approximated function. Here we want to train a FFNN with this given Euler method formulation above.

Dataset Creation

For the training of the NN we need a training dataset. Our dataset consists of trajectories from random initial positions. The states through time are computed using the `solve_ivp` function of the `scipy` module. The initial points are generated within the set $[\texttt{low}, \texttt{high}]$. We have generated M initial points therefore M different trajectories. The states that are computed from the initial states are computed for $t \in [\texttt{t\_start}, \texttt{t\_end})$ with a time step of Δt. So our values in the training set has the dimension $(M \times 2 \times (((\texttt{t\_end} - \texttt{t\_start})/\Delta t) - 1)$ since the points are 2-dimensional and we neglect the last datapoint. The targets that we want to calculate are the direct neighbors (since Euler method only considers the previous state) so the targets have also the same shape as the training value set. For example for a state $X(t)$ its target would be $X(t + \Delta t)$. We follow the same procedure in order to create the validation set, with the exception that for the validation set we a smaller set. In the evaluation we will specify the exact ranges and the values for the parameters.

Model Architecture

We have used FFNN with 3 hidden layers. The architecture follows the following layers:

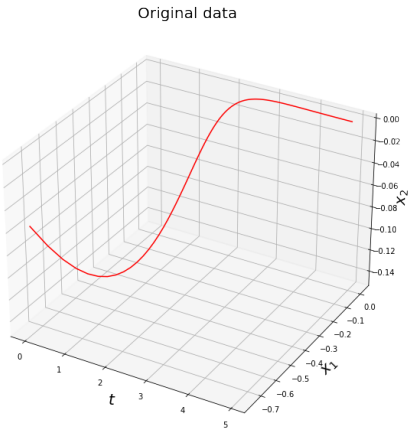
- Input Layer: outputs $(x, y) \in \mathbb{R}^2$.
- First Hidden Layer: takes $x \in \mathbb{R}^2$ and outputs $z \in \mathbb{R}^{512}$
- Second Hidden Layer: takes $x \in \mathbb{R}^{512}$ and outputs $z \in \mathbb{R}^{512}$
- Third Hidden Layer: takes $x \in \mathbb{R}^{512}$ and outputs $z \in \mathbb{R}^{512}$
- Output Layer: takes $x \in \mathbb{R}^{512}$ and outputs $z \in \mathbb{R}^2$

We use the `Lightning` module for the class that defines our model. Therefore we have to implement the required functions such as `forward`, `training_step` and `configure_optimizers`, which specify the model architecture, loss computation and the optimizer that is used to update the model weights. Using the `Sequential` torch module we can easily setup the model architecture specified above. For the loss function we use the `MSELoss` in the torch module which measures the mean squared error (squared L2 norm) between each element of the original datapoints and the predictions with the default 'mean' reduction, which can be computed as in the following.

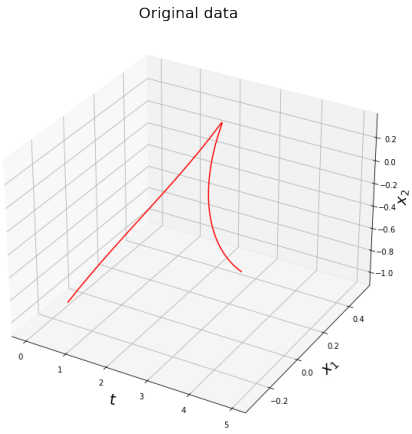
$$\frac{1}{n}\sum_{i=1}^{N}(D[i]) - \hat{D}[i])^2$$

Evaluation

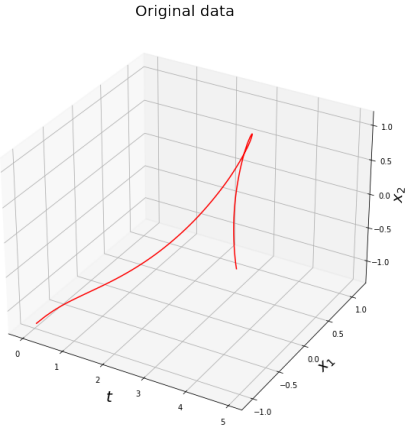
To test our network, we have created three different training and validation datasets from the Andronov-Hopf equation for $\alpha \in -1.8, 0, 1.3$ with the parameters `low = -2`, `high = 2`, $M = 1000$, `t_start = 0`, `t_end = 5`, and trained 3 different NN models to predict the derivative function for each dataset. In this evaluation section we use the same time step $\Delta t = 0.05$ value for the dataset generation and for the model training. Since we use a very small time step and a very high number of datapoints for a small field range, we expect our basic model



(a) $\alpha = -1.8$



(b) $\alpha = 0$



(c) $\alpha = 1.3$

Figure 3: Example trajectories with different alpha values (original).

to capture the structure of the trajectories. In this example we use 200 datapoints for the validation dataset. An example trajectory resulted from the original data creation with $\alpha = -1.8$ is depicted in figure 3.

We see the predictions of our model for $\alpha = -1.8$ before and after the training in figure 4. As you can see, our model could successfully capture the trajectory. After the training, we can plot the phase portraits as well. In order to plot them we predict the next point in our system using our predicted FFNN model and we subtract the points and divide the resulting vector with Δt to get the derivative. The resulting phase portrait form for $\alpha = -1.8$ can be seen in figure 5. The predictions are very accurate since we use the same time step in the model predictions as in the original data. And since the time step is sufficiently small and there are sufficiently many datapoints, we could 'overfit' and the resulting phase portrait looks very similar to the original portrait from the lecture.

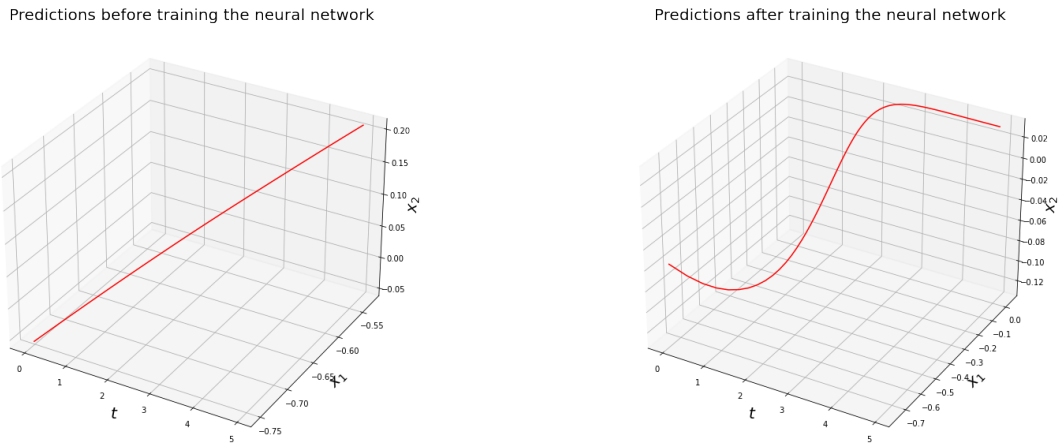


Figure 4: Example trajectory predicted by model for $\alpha = -1.8$. Before training and after training.

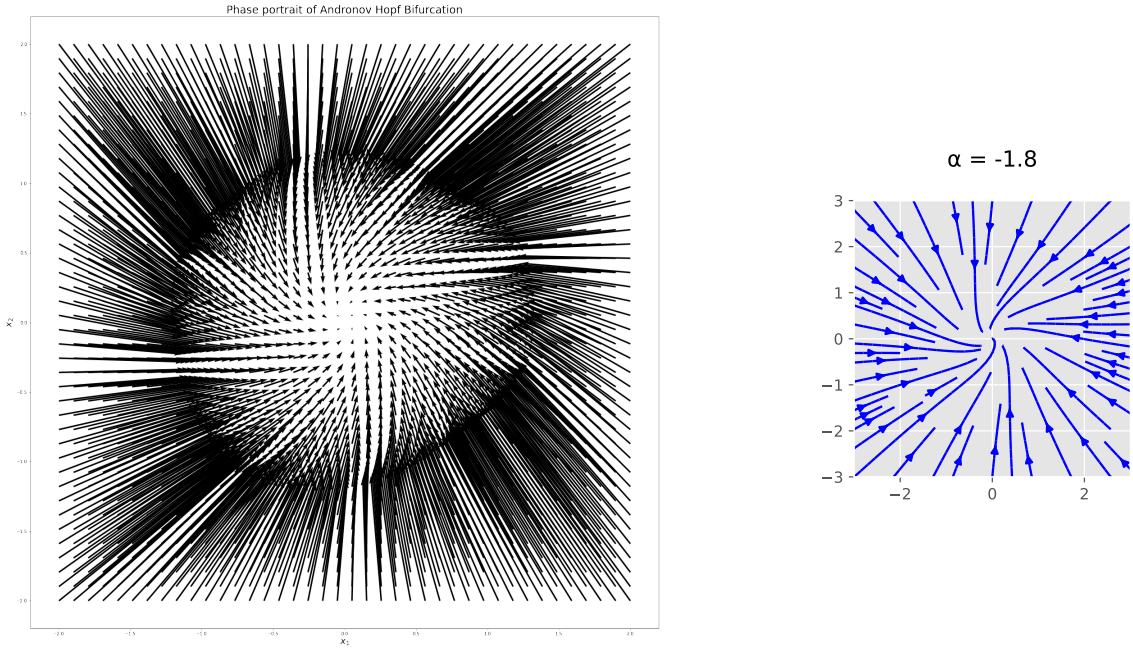


Figure 5: Phase portrait generated from our trained model for $\alpha = -1.8$ on the left. On the right we see the original portrait taken from our previous report paper 3.

Similarly for $\alpha = 0$ and $\alpha = 1.3$ we see the results on the next two pages in figures 6, 7 and 8, 9. For the figures for $\alpha = 1.3$, one can say that even the structure of the limit cycle can be successfully captured by our trained model.

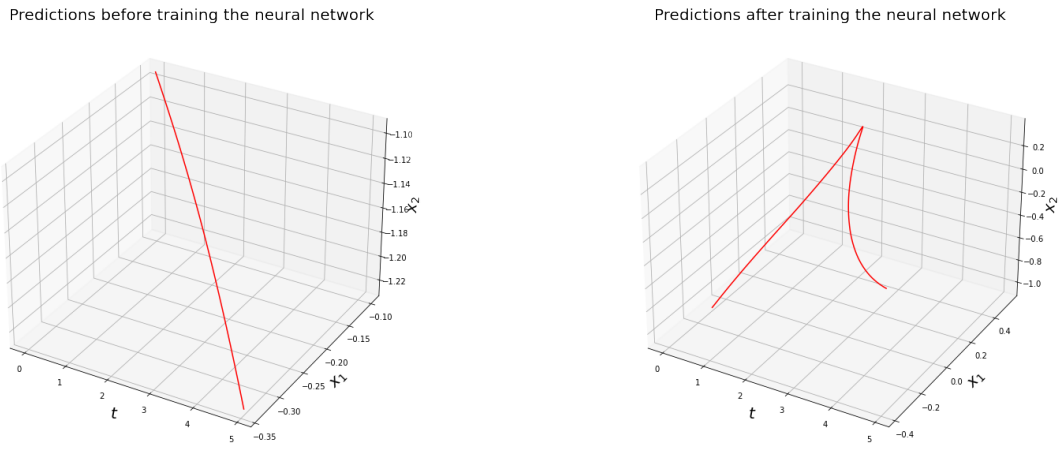


Figure 6: Example trajectory predicted by model. Before training and after training.

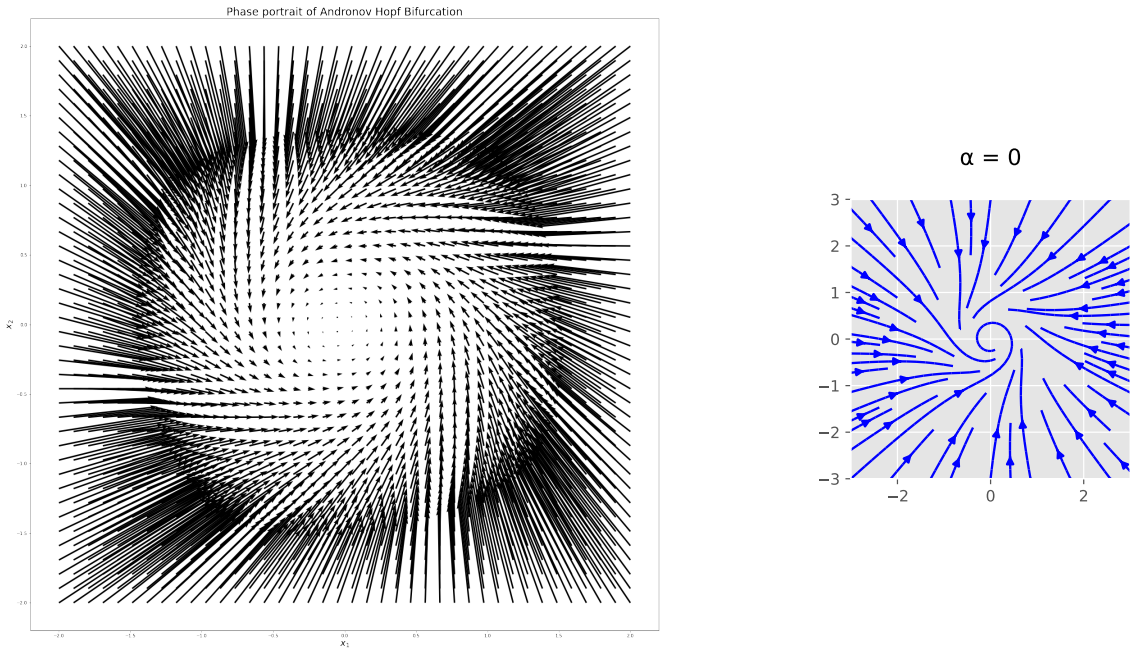


Figure 7: Phase portrait generated from our trained model for $\alpha = 0$ on the left. On the right we see the original portrait taken from our previous report paper 3.

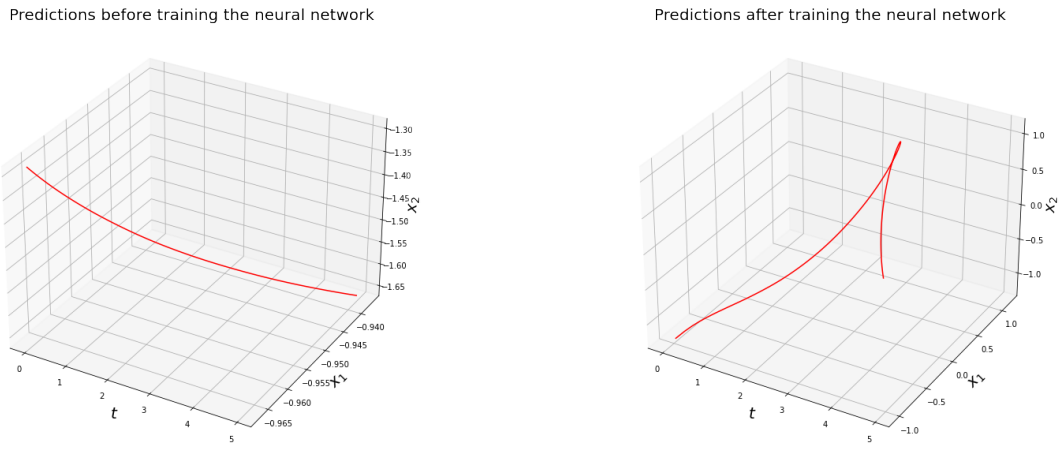


Figure 8: Example trajectory predicted by model. Before training and after training.

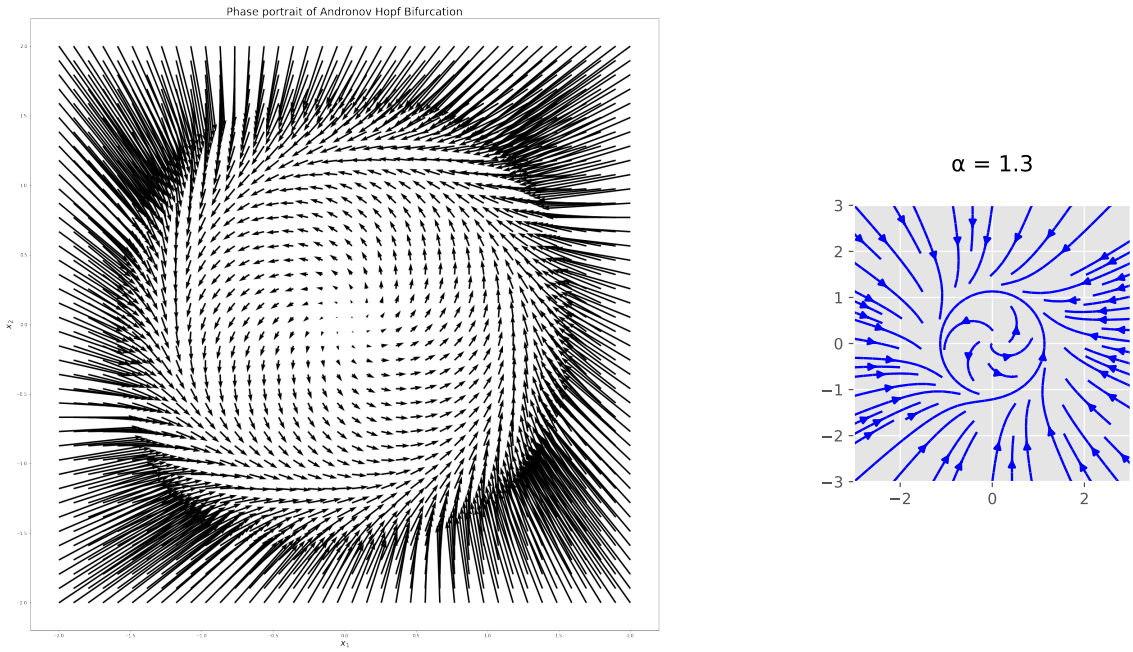


Figure 9: Phase portrait generated from our trained model for $\alpha = 1.3$ on the left. On the right we see the original portrait taken from our previous report paper 3.

Report on task Task 3/5: Setting up the neural network for the Runge-Kutta method

Now in this task we again want to model a FFNN for modelling an ODE system in the same form as we did in task 2. We again consider the systems resulted from the Andronov-Hopf bifurcation with $\alpha \in -1.8, 0, 1.3$.

The Runge-Kutta Method

Another technique that is used for approximating ODEs with an intial value is the Runge-Kutta method [2]. It is an extension of the Euler method and is again a first-order approximation method. Its formulation is as follows:

$$X(t + \Delta t) = X(t) + \frac{1}{6}\Delta t(k_0 + 2k_1 + 2k_2 + k_3)$$

where we define the k values as follows:

- $k_0 = F(X(t))$
- $k_1 = F(X(t) + \frac{1}{2}\ \Delta t\ k_0)$
- $k_2 = F(X(t) + \frac{1}{2}\ \Delta t\ k_1)$
- $k_3 = F(X(t) + \Delta t\ k_2)$

This formulation predicts the next system state at time $t + \Delta t$ by taking a weighted average of the resulting derivative values k_0, k_1, k_2 and k_3. The method weights more on the values k_1 and k_2. These values are computed at different system states. First value k_0 is actually equal to the derivative one uses in the Euler method, which is taken at the time state at time t. Second value k_1 is the derivative taken from the system state $X(t) + (1/2)\ \Delta t\ k_0$ which is reached by going in the direction of k_0 with $(1/2)\ \Delta t$ from the original starting state $X(t)$. Similarly, for the third value k_2 the method takes the derivative at the system state that is reached by going in the direction of k_1 with $(1/2)\ \Delta t$ from the original system state $X(t)$. For the last value k_3 the computation is similar but this time we go in the direction of k_2 with a full time step Δt from the original system state and take the derivative there. By using these different derivatives at different system state estimates, we expect this method to produce better predictions than the Euler's method in some cases, where we have a large time step Δt. As a downside the Runge-Kutta is more complex than the Euler's method. Figure 10 depicts the more complex computational graph of the Runge-Kutta method.

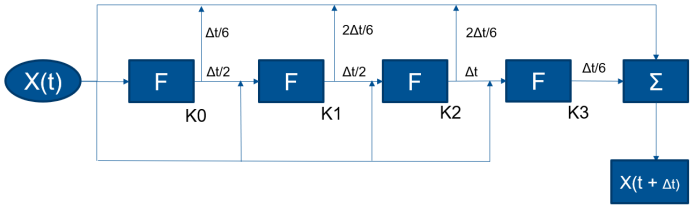


Figure 10: Computational graph of the Runge-Kutta method.

One can see how the Runge-Kutta works graphically in figure 11. In this figure the k values start with a shifted index! And we have a time step of $\Delta t = h$. The starting state is at time $t = t_0$. It is relatively easy to see how the Runge-Kutta makes the approximation for the next system state at time $t = t_0 + h$ by combining different k values.

Evaluation

The model architecture and the procedure in the evaluation is identical to what we have used in the previous task. Since the results are also identical we do not include any trajectory or phase portrait figures in this task. In the end we could show that the model is working without any problems by overfitting it using a small time step.

Report on task Task 4/5: Testing the neural networks in a two-dimensional dynamical system example

In task 4 we test our NNs which are trained according to different approximation methods, namely, the Runge-Kutta and the Euler's method. To create the training set, we use the approach described in the subsection Dataset Creation. While creating our data we use three different α parameters: -1.8, 0, 1.3

In the training step, our first approach was to use many datapoints with a small time step to obtain almost perfect (overfitted) results to make sure our models can be trained as expected. For this we have created

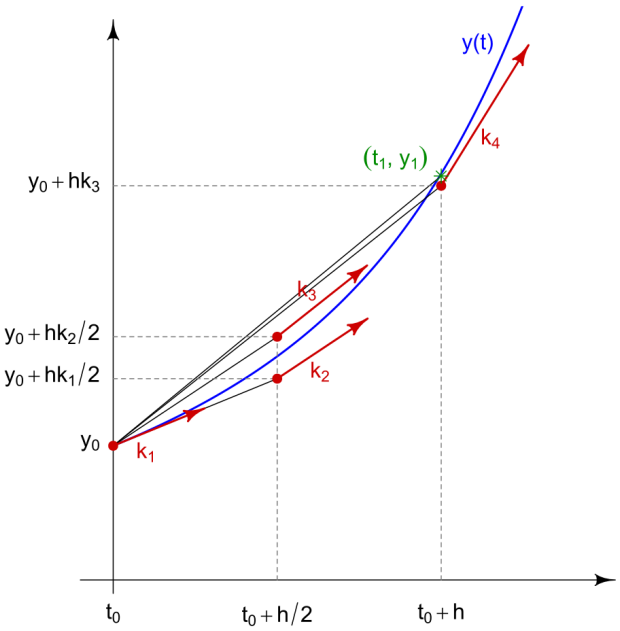


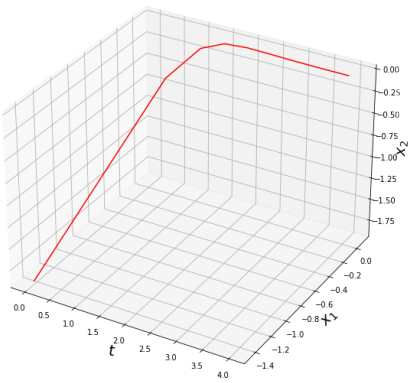
Figure 11: Graphical visualization of the Runge-Kutta method.

datapoints with a $\Delta t = 0.05$ with 1000 sample trajectories in the space $(-2, 2)$, which is a quite large number for this given space. With this setup the results were almost perfect when we were comparing the original trajectory with the estimated one and the phase portraits constructed by the estimations were very similar to the original phase portraits with the given α parameters of Androv Hopf equation.

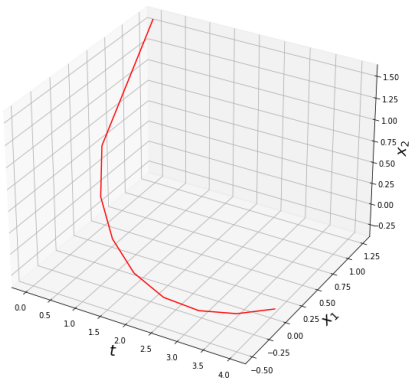
As finding so many datapoints in practice is not that easy, we have decided to train our networks with more sparse data and test it with the trajectories which are again constructed with the approach described in our Dataset Creation subsection. The dataset we use for the training contains 50 trajectories (compare it to 1000 in previous version). We test our networks with the two approximation techniques and with the created trajectories. From the created trajectories we give the initial points to the networks to predict the trajectories and get the derivatives in certain states. After that we compare the trajectories and the phase portraits with the original ones. As our training data is more sparse and the Euler's method does not consider the points between the current state and the next state while approximating the next state, the Runge-Kutta outperforms the Euler's method in all cases, as it also considers the derivatives in between.

Figure 12 visualizes the original trajectories that we compare with their approximated values for $\alpha = -1.8, 0, 1.3$ using the Euler's method in figures 13, 15 and 17 respectively. The resulting phase portrait approximations for $\alpha = -1.8, 0, 1.3$ can be seen in figures 14, 16 and 18 respectively. Similarly for the Runge-Kutta method we compare the original trajectories depicted in figures 19 with their approximations in figures 20, 22 and 24. The resulting phase portraits are in figures 21, 23 and 25. One can notice that the original trajectories are not as 'smooth' as the trajectories from the previous sections since here we use a much larger time step $\Delta t = 0.5$.

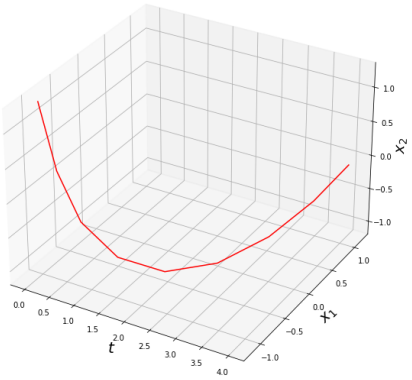
While deciding our architecture and hyperparameters, we have used common choices. As a result of that our neural network has performed well while approximating the trajectories in the test phase without modifications of the architecture and hyperparameters. Therefore we did not put a lot of effort into hyperparameter tuning.



(a) $\alpha = -1.8$



(b) $\alpha = 0$



(c) $\alpha = 1.3$

Figure 12: Example trajectories with different alpha values for Euler's method (orginal). Comparisons for Euler's method can be found in following

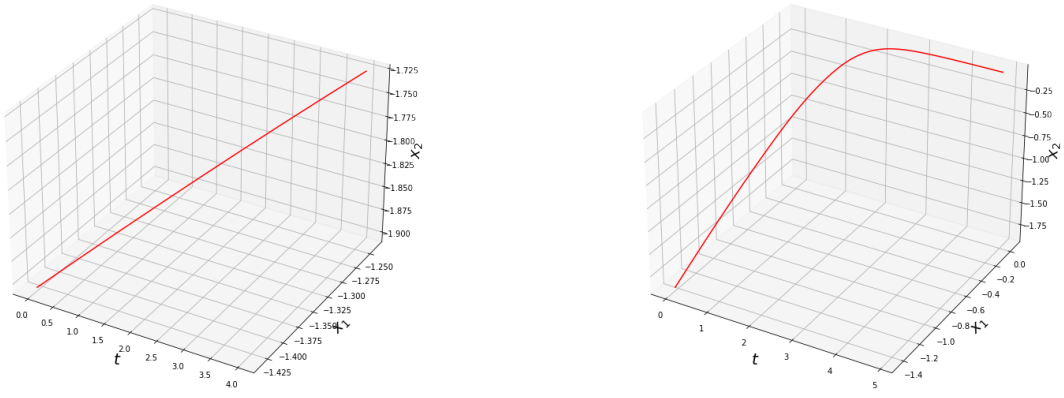


Figure 13: Example trajectory predicted by model for $\alpha = -1.8$. Before training and after training.

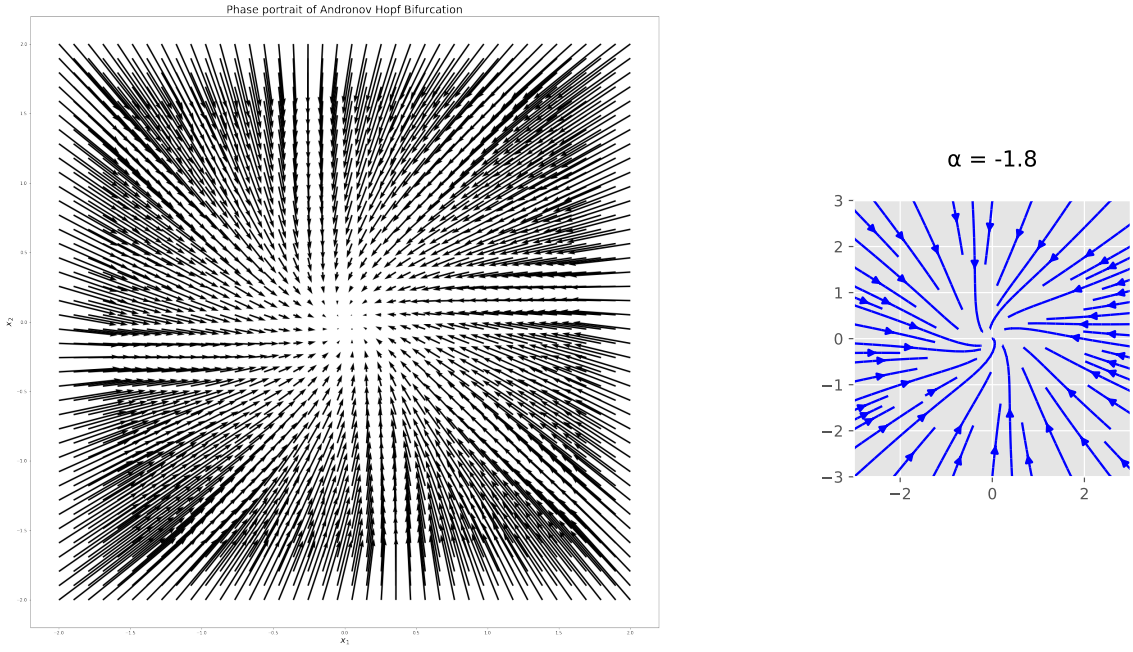


Figure 14: Phase portrait generated from our trained model for $\alpha = -1.8$ on the left. On the right we see the original portrait taken from our previous report paper 3.

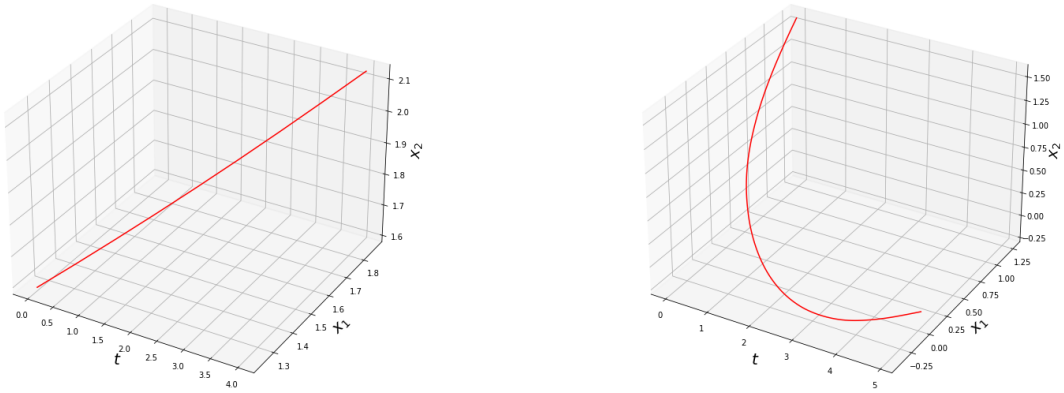


Figure 15: Example trajectory predicted by model for $\alpha = 0$. Before training and after training.

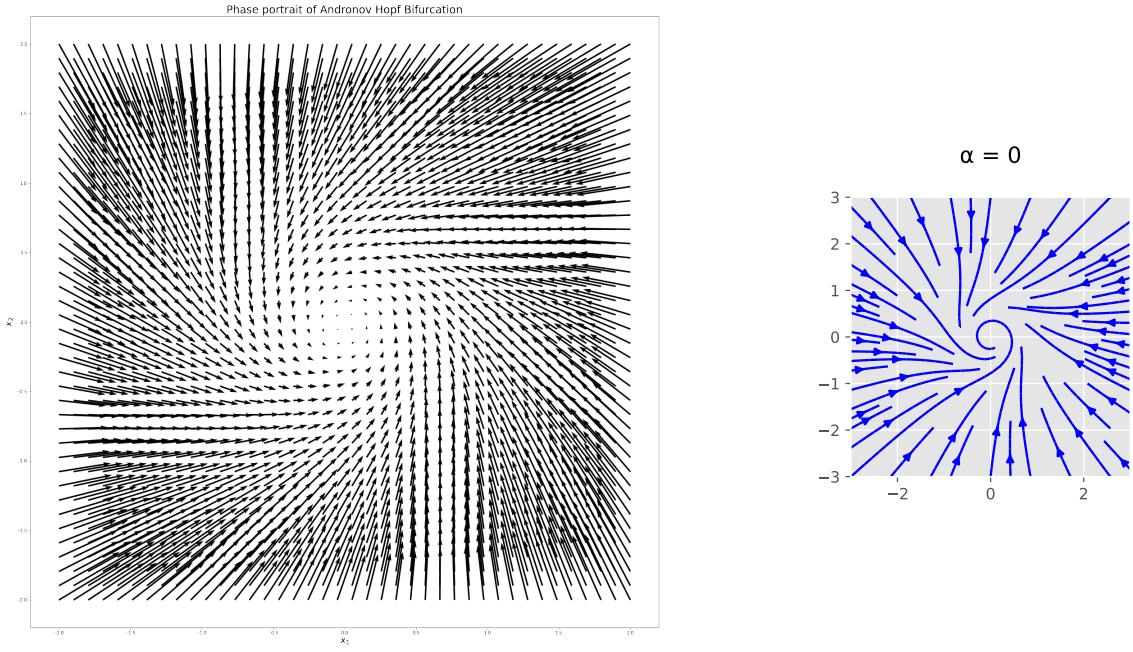


Figure 16: Phase portrait generated from our trained model for $\alpha = 0$ on the left. On the right we see the original portrait taken from our previous report paper 3.

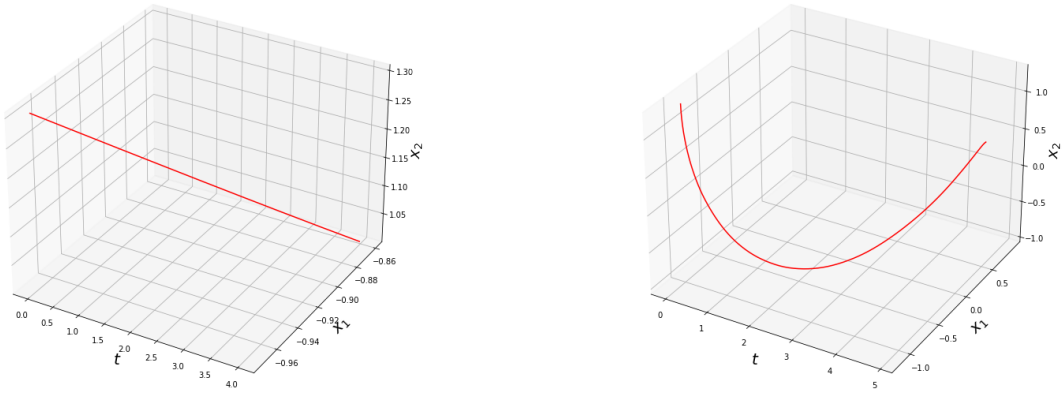


Figure 17: Example trajectory predicted by model for $\alpha = 1, 3$. Before training and after training.

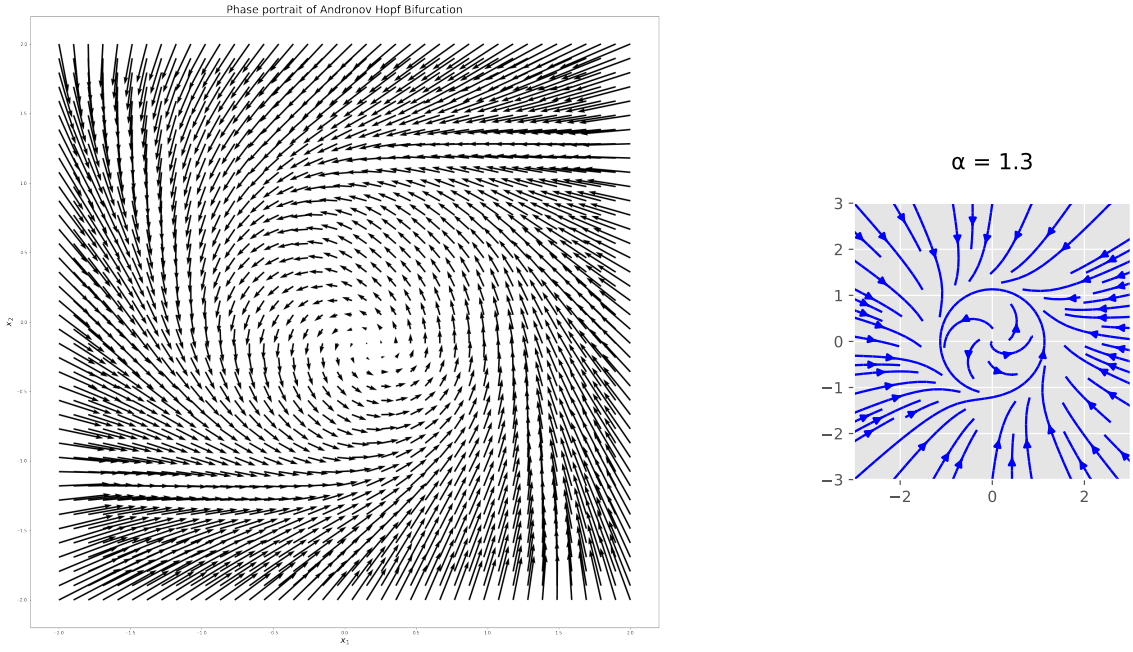
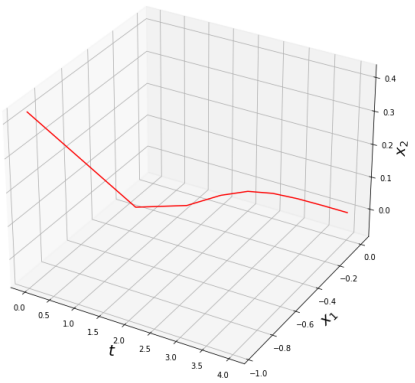
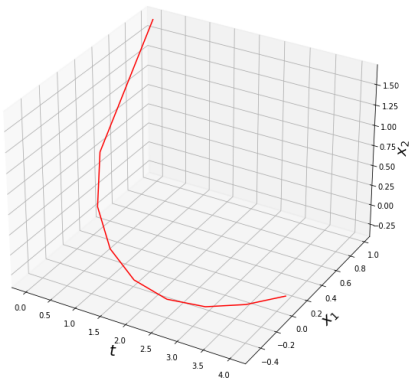


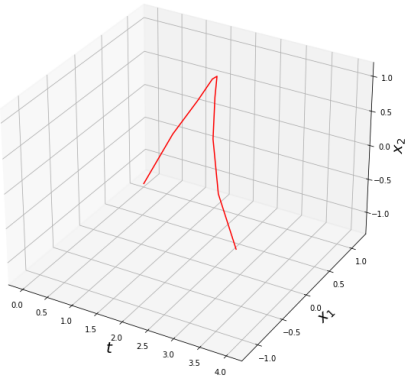
Figure 18: Phase portrait generated from our trained model for $\alpha = 1, 3$ on the left. On the right we see the original portrait taken from our previous report paper 3.



(a) $\alpha = -1.8$



(b) $\alpha = 0$



(c) $\alpha = 1.3$

Figure 19: Example trajectories with different alpha values for Runge-Kutta's method (orginal). Comparsions for Runge Kutta's method can be found in following.

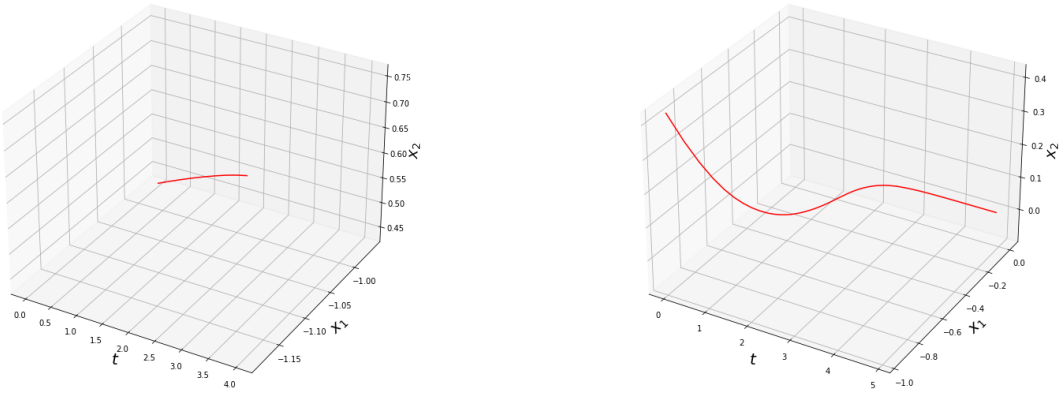


Figure 20: Example trajectory predicted by model for $\alpha = -1,8$. Before training and after training.

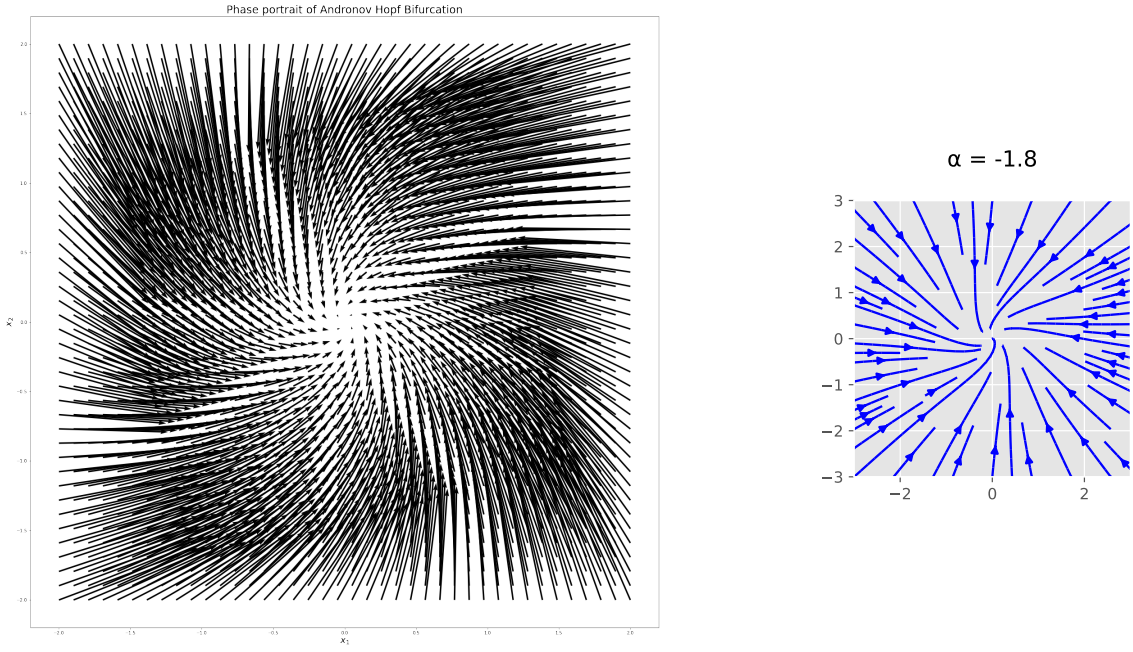


Figure 21: Phase portrait generated from our trained model for $\alpha = -1,8$ on the left. On the right we see the original portrait taken from our previous report paper 3.

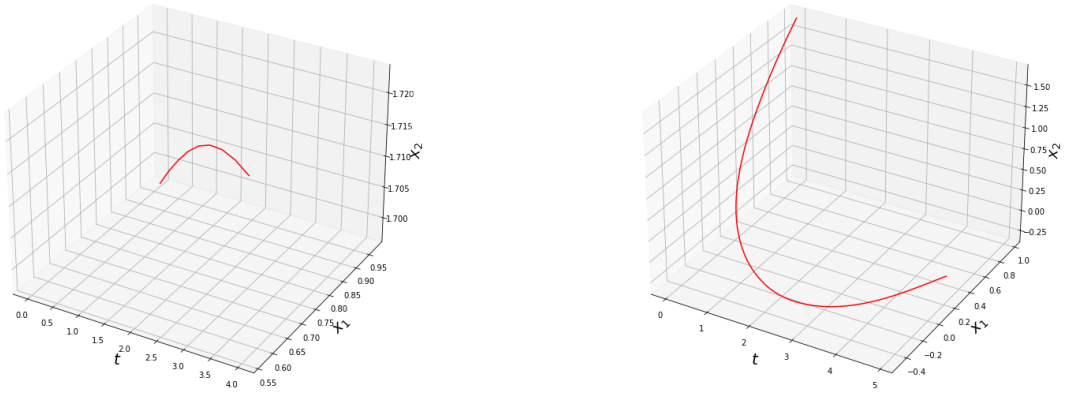


Figure 22: Example trajectory predicted by model for $\alpha = 0$. Before training and after training.

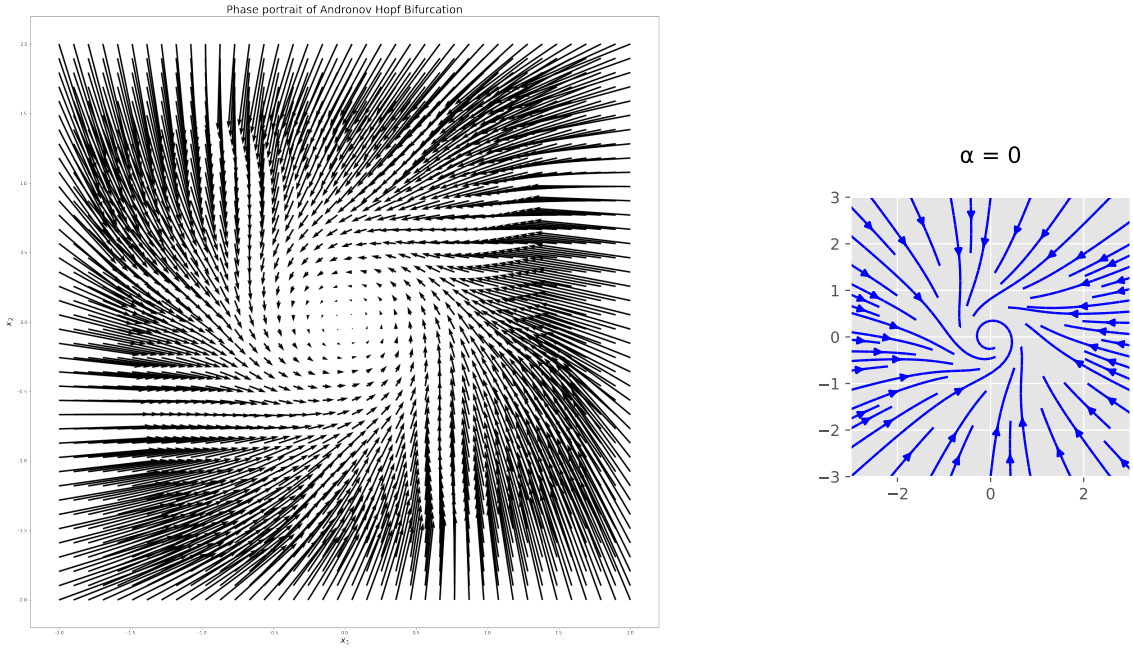


Figure 23: Phase portrait generated from our trained model for $\alpha = 0$ on the left. On the right we see the original portrait taken from our previous report paper 3.

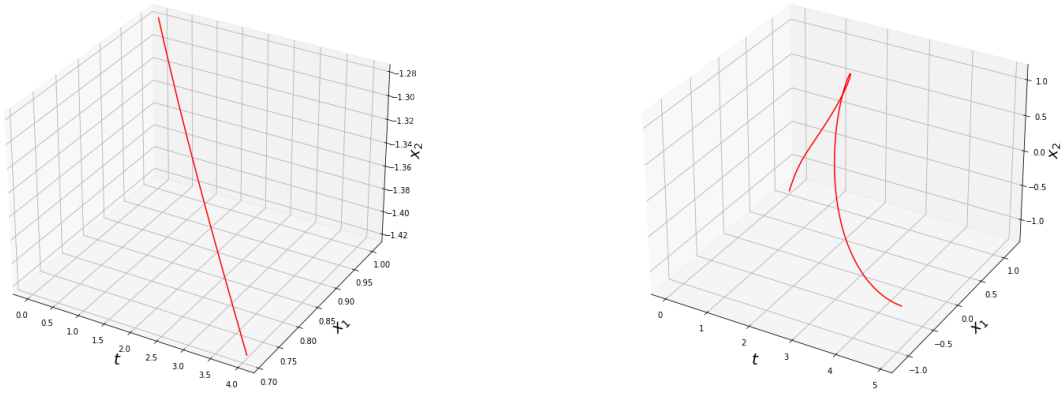


Figure 24: Example trajectory predicted by model for $\alpha = 1, 3$. Before training and after training.

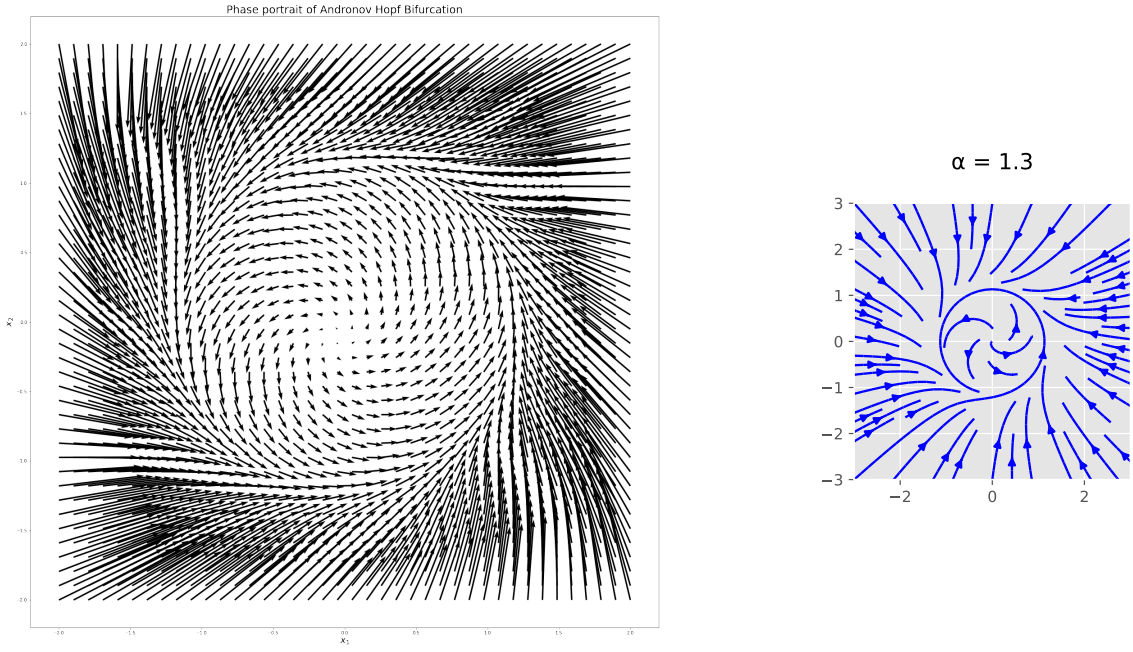


Figure 25: Phase portrait generated from our trained model for $\alpha = 1.3$ on the left. On the right we see the original portrait taken from our previous report paper 3.

Integrating (α) parameter into the model

In the previous part we have trained our model always with fixed values of (α), but it maybe also interesting to train our model along with the (α) parameter. For this purpose we have appended the (α) values into the training set. The previous model has taken as input x coordinate and the y coordinate. As we also want to feed (α) value to the network, the new model takes as input the x coordinate, the y coordinate and the (α) parameter and outputs the derivative according to these given inputs. In the following figures 26, 27, 28, 29, 30, 31, 32, 33, 34 and 35 one can find the phase portraits for different (α) values (with Euler and Runge-Kutta method) obtained via new version of our model.

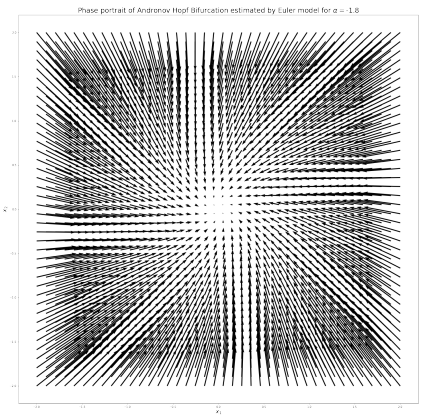


Figure 26: Phase portrait for $\alpha = -1.8$. Created by NN using Euler.

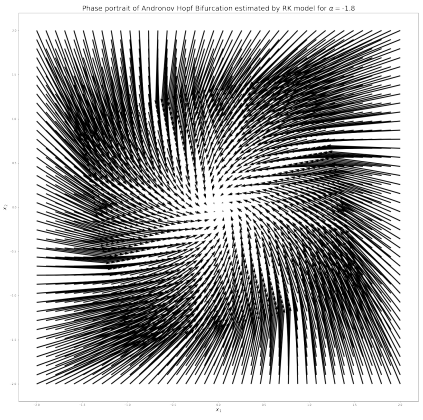


Figure 27: Phase portrait for $\alpha = -1.8$. Created by NN using Runge Kutta.

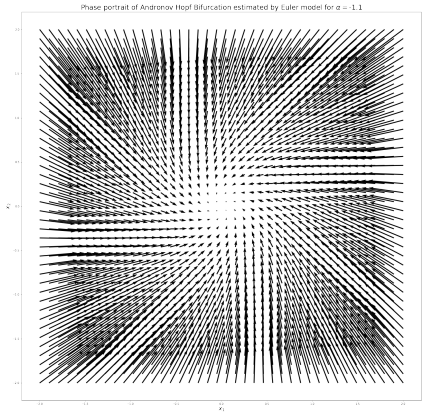


Figure 28: Phase portrait for $\alpha = -1.1$. Created by NN using Euler.

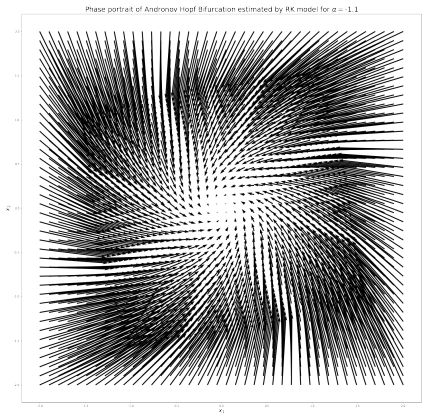


Figure 29: Phase portrait for $\alpha = -1.1$. Created by NN using Runge Kutta.

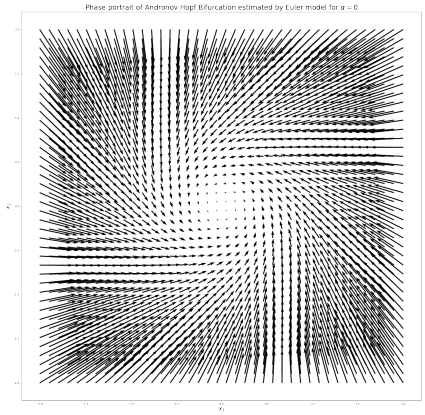


Figure 30: Phase portrait for $\alpha = 0$. Created by NN using Euler.

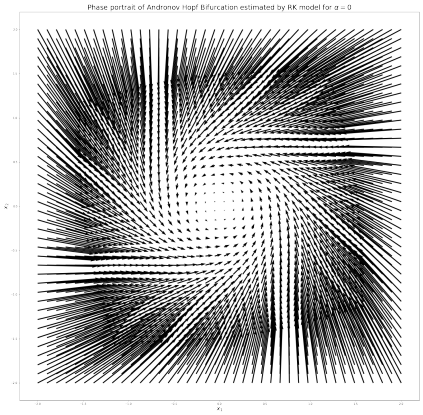


Figure 31: Phase portrait for $\alpha = 0$. Created by NN using Runge Kutta.

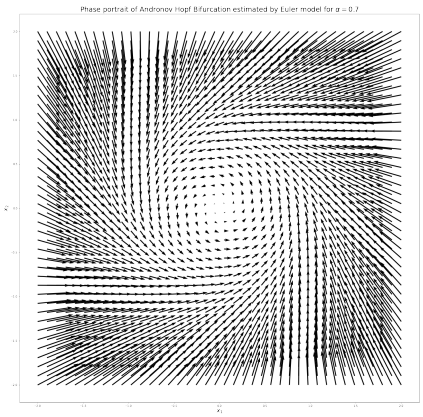


Figure 32: Phase portrait for $\alpha = 0.7$. Created by NN using Euler.

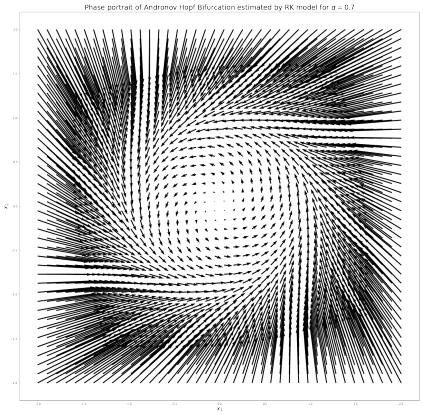


Figure 33: Phase portrait for $\alpha = 0.7$. Created by NN using Runge Kutta.

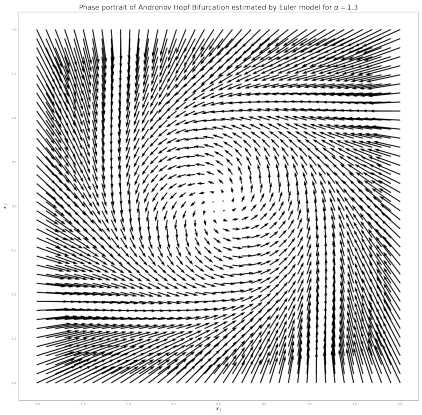


Figure 34: Phase portrait for $\alpha = 1.3$. Created by NN using Euler.

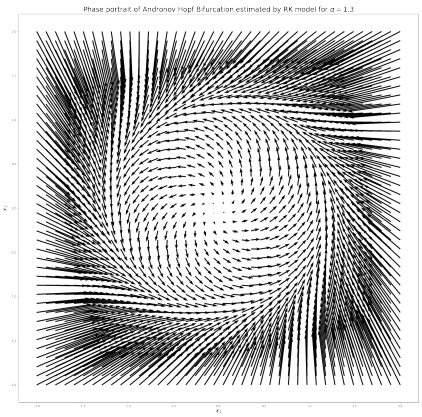


Figure 35: Phase portrait for $\alpha = 1.3$. Created by NN using Runge Kutta.

Report on task Task 5/5: Construction bifurcation diagrams using the extracted dynamics

In this task we were supposed to construct bifurcation diagrams from the dynamics extracted using the NN models. Since our model also integrates the parameter of the dynamical system (Andronov Hopf bifurcation) α, we are able to create the state of the dynamical system at different values of this parameter and find the steady states at specific parameter values. To achieve this, we have first created a range of alpha values where we want to exactly calculate the steady states of the system, discretesizing the range $(-2, 2)$ into 41 values gave us the requested values. Then we have selected 100 starting points again in the range $(-2, 2)$. We then integrated these starting positions using our trained model until they converge to a steady state. Convergence check is made by two assessments in the implementation: There is a counter for the number of integration steps, when it reaches 100, integration is stopped and the last position of integration is saved as the steady state, there is also a check on the prediction of the NN model, where a threshold value of 0.001 is chosen, so if the prediction, to be exact the derivative is smaller in norm than 0.001, it is assumed that the steady state is reached and the integration is stopped. Steady states for a specific alpha value is stored in a list, this list is than appended into a larger list containing steady states for different alpha values. The range of alpha values and their corresponding steady states lists are then used to plot the bifurcation diagrams (see Figure 36 for example).

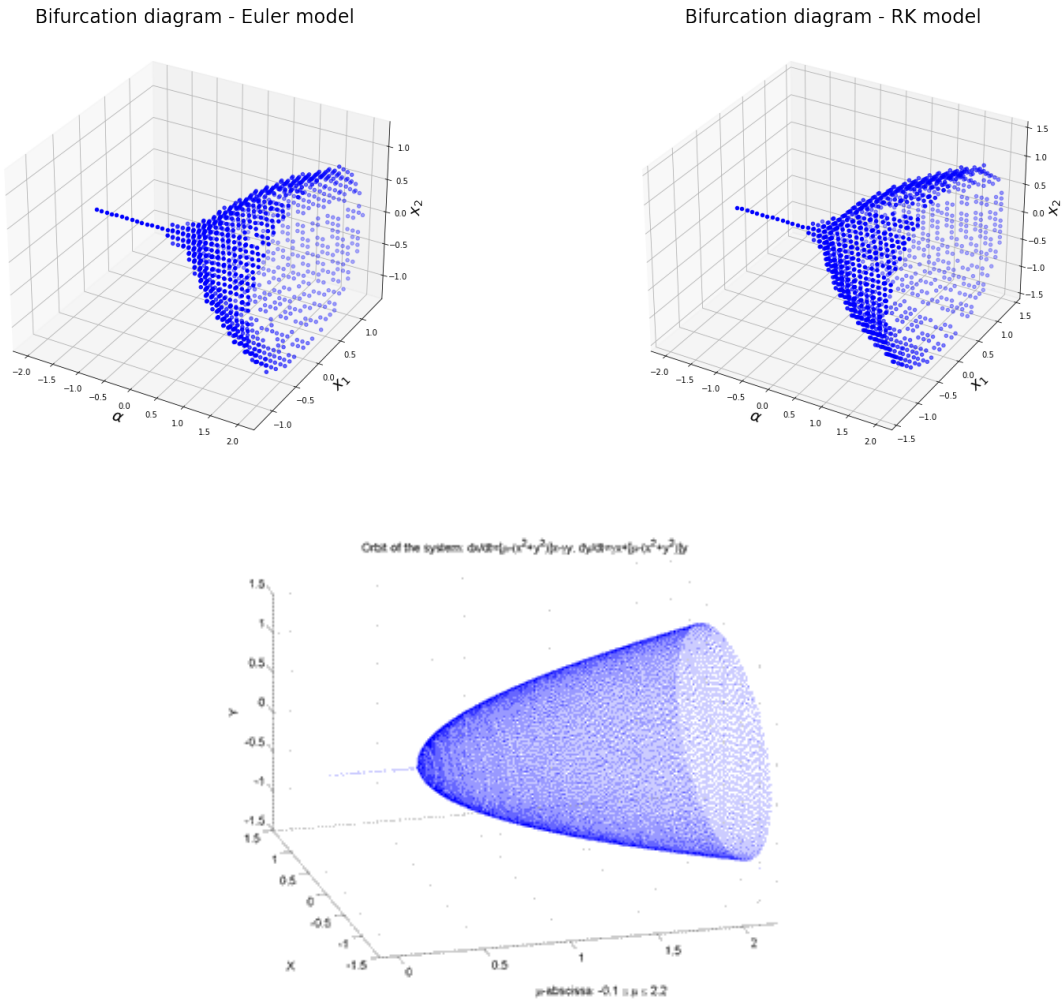


Figure 36: Bifurcation diagrams computed according to the predictions of the Euler (left), RK model (right). Bifurcation diagram using original data on the right. Source (middle): https://www.isnld.com/indexD2.html

References

[1] B N Biswas et al. “A DISCUSSION ON EULER METHOD: A REVIEW”. In: *Electronic Journal of Mathematical Analysis and Applications* 1 (June 2013), pp. 294–317.

[2] J.C. Butcher. “A history of Runge-Kutta methods”. In: *Applied Numerical Mathematics* 20.3 (1996), pp. 247–260.

[3] Annan Yu et al. “Arbitrary-Depth Universal Approximation Theorems for Operator Neural Networks”. In: *CoRR* abs/2109.11354 (2021).