Master 2 Machine Learning - Research Internship

Automatic learning of temporal series for chaotic systems

Author :
Robin Duraz

Supervisor :
Onofrio Semeraro
Lionel Mathelin
Alexandre Allauzen
Michele Alessandro Bucci

At
LIMSI - CNRS

Secretariat - tel : 01 69 15 81 58
Email : alexandre.verrecchia@u-psud.fr

Contents

Contents i

1 Introduction 1

2 Work on data from physics 2
- 2.1 Lorenz system . . . 2
- 2.2 Methodological tools . . . 4
- 2.3 Evaluation with Lyapunov Exponents . . . 6

3 Experiments and results 12
- 3.1 First experiments . . . 12
- 3.2 Added information . . . 14
- 3.3 Manipulating data . . . 17
- 3.4 Resnet-like models . . . 18
- 3.5 Analysis of intermediate layers . . . 19

4 Conclusion and perspectives 21

Bibliography 23

Appendices 24

A Lyapunov exponents computation method comparison 24

B QR Decomposition 25

C Singular Value Decomposition 26

D MLP training 27

E MGD training 36

F Sampling 38

Abstract

I present a method to compute Lyapunov exponents for Machine Learning models, a way to evaluate the chaoticity of a system, in order to evaluate the quality of the learning process of those models. Those methods and models are used on time series of the Lorenz system, a simple mathematical system exhibiting chaotical behaviours.
After that, I explore different Machine Learning methods susceptible to correctly get Lyapunov exponents, motivated by both Machine Learning and physical knowledge.

Keywords

Time series, chaotic systems, future prediction, model evaluation, Lyapunov exponents.

CHAPTER 1

Introduction

Machine Learning makes use of algorithms to "learn" on its own how to perform specific tasks, relying on detecting patterns and performing inferences to accomplish it. But how can we make sure that it actually learn what we intended it to ?
We often hear about overfitting, being one the things we have to avoid while training a model. It happens when our model is learning too much about its training data, and becomes unable to correctly handle unknown cases, to generalize.
When we are training our model, we make use of a "loss", a measure of how much did the model's prediction deviate from the solution we had. Those losses are often pretty generic to work with any kind of data, but can also become a limitation.
When we try to use Machine Learning on real world problems, those generic methods often fall short if applied directly. Data scientists often use methods like feature engeneering to help models "pick on" the underlying rules behind the data.
Using an "informed" loss, i.e. a loss making use of knowledge we have about our data could prevent overfitting, as well as potentially having better results and better training speed.
This raises the question of how to correctly evaluate this learning phase, what type of loss we could use, to make sure that our system effectively learns what it was supposed to, or at least, is learning in the right direction.

This work is part of a bigger project called FlowCon, aiming at using Machine Learning as well as knowledge of Fluid Mechanics to control turbulent flows by reducing turbulences. Those methods could, for example, be applied on planes to reduce the drag produced by turbulent flows on the wings.

Those kinds of physical systems can be described by strongly non-linear equations, which can show chaotic behaviours. Our work will be focused on a "simple" (it is only three dimensional) chaotic system, called the Lorenz system. This system, while simple, can already exhibit chaotic behaviour. This allows us to work on a system having the same useful properties as turbulent flows (mainly chaoticity), while also being easier to learn because of the low dimensionality.
The method used to measure the quality of the model predicted will be to compute the Lyapunov exponents, a physical measure of the chaoticity of a system.

We will then have to, first, implement the computation of Lyapunov exponents, and then make sure the model's predictions have the same exponents as the truth data. After that, we will explore different methods likely to get correct exponents more easily, as well as do some experiments to validate or refute ideas coming from knowledge about chaotic systems, and more generally physics.

CHAPTER 2

Work on data from physics

Applying machine learning on data from physics, can and should take advantage of physical knowledge to make the learning process easier. But while we can sometimes manipulate features to get more useful ones, it isn't always possible.
Another solution is to use physical measures to evaluate from the physical point of view the quality of the model's predictions. While it can help evaluate the model, actually using it to improve the learning process faces several limitations.

2.1 Lorenz system

The Lorenz system is a simple mathematical model, originally developed as an imitation of atmospheric convection by Edward Lorenz in [6], and can also be used in simplified models of lasers, motors, electric circuits, etc. It is a dynamical system, so there is a map defining the time dependence of a point, i.e. there exist a function f such that we have $\dot{v}(t) = f(v(t))$ at any time t, where $\dot{v} := \frac{\partial v}{\partial t}$ [1], and $v = (x, y, z)$. The Lorenz system is a three dimensional system and is thus defined by three differential equations, with parameters σ, ρ, β, called Lorenz equations :

$$\begin{cases} \dot{x} = \sigma(-x + y) \\ \dot{y} = x(\rho - z) - y \\ \dot{z} = xy - \beta z \end{cases}$$

Although it is relatively simple, it can exhibit chaotic behaviour when the parameters' values are well chosen. Depending on the values of ρ, this system can have either one or three critical points. For $\rho < 1$, there is one equilibrium point at the origin (0, 0, 0), to which all trajectories converge.
For $\rho > 1$, two other critical points appear at $(\sqrt{\beta(\rho-1)}, \sqrt{\beta(\rho-1)}, \rho-1)$ and $(-\sqrt{\beta(\rho-1)}, -\sqrt{\beta(\rho-1)}, \rho-1)$. Those two points are stable, i.e. all trajectories coming in a neighborhood close enough to those points will converge, if $\rho < \sigma\frac{\sigma+\beta+3}{\sigma-\beta-1}$.

In our case, we will use for our Lorenz system $\sigma = 10, \beta = 8/3, \rho = 28$, and we will keep those parameters for every experiment. The two critical points are unstable since $\sigma\frac{\sigma+\beta+3}{\sigma-\beta-1} \approx 24.74 < \rho$. In that case, beginning from an initial condition[2], it will form what is called a Lorenz attractor (Figure 2.1).

[1] := means definition
[2] The initial point must be different from the the origin point.

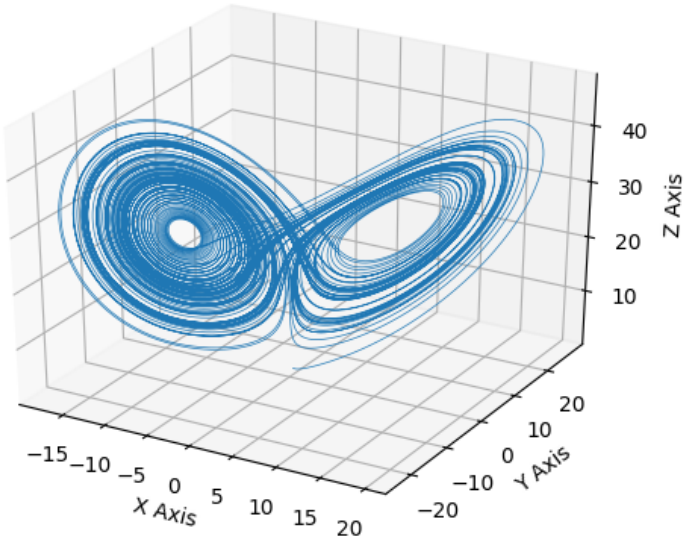


Figure 2.1: Lorenz attractor, initial position [0, 1, 1.05] and $\sigma = 10, \beta = 8/3, \rho = 28$, 100000 points with $\Delta t = 0.001$

I generated myself the data used for the experiments. While it may be easier to use Euler's method to generate it, we can face some problems if we don't take precautions. Firstly, taking a unitary step, i.e. adding $\dot{x}, \dot{y}, \dot{z}$ to respectively x, y and z, can quickly diverge from a Lorenz attractor because steps might be too big. This problem can be easily avoided by making steps smaller than unitary steps, for example using $v_{m+1} = 0.01 \times \dot{v} + v_m$ instead of $v_{n+1} = \dot{v} + v_n$, where $v_{m+100} = v_{n+1}$.
Nevertheless, the main problem comes from discretizing a continuous system in order to get our data. This discretization can create errors in values, which overall can add up to a significant difference in the discrete data for different time steps. For example, if we take time steps of 0.1 and 0.001 with the same initial condition, the 10th and 1000th (each equal to a unitary step) can be relatively different. For example, the end position in the first case is [-635.33, 54364.51, 48892.99], while it is [-9.32, -9.32, 28.11] in the second. Of those two cases, the first one is really far from what should be the correct value, while the second case is really close.

Therefore, to create the data, I used an ODE (Ordinary Differential Equations) solver using the Runge-Kutta method of order 5 to compute the points. This method using adaptive time steps remains much more accurate for different values of the time-step. To this end, I used the function solve_ivp[3], from the library scipy.integrate, which numerically integrates a system of ordinary differential equations.

Note that Δt and t will often be used to refer to different aspects of time of the Lorenz system. t will be used to refer to the global time of the system, the variable by which the system advances. Δt, instead, is the sampling rate used for creating data.

[3]https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

2.2 Methodological tools

This section will introduce all relevant physical concepts, as well as Machine Learning components that will be experimented on.

2.2.1 Jacobian matrix

This will be extensively used for many things, from computing Lyapunov exponents to being used for training a neural network. The Jacobian matrix here is the matrix of partial derivatives. For our case with the Lorenz system, it will be $\begin{pmatrix} \frac{\dot{x}}{\partial x} & \frac{\dot{x}}{\partial y} & \frac{\dot{x}}{\partial z} \\ \frac{\dot{y}}{\partial x} & \frac{\dot{y}}{\partial y} & \frac{\dot{y}}{\partial z} \\ \frac{\dot{z}}{\partial x} & \frac{\dot{z}}{\partial y} & \frac{\dot{z}}{\partial z} \end{pmatrix}$. I will use three different methods to compute the Jacobian matrix, depending on the case.

- To get the Jacobian matrix of the data, I use the Lorenz system's Jacobian matrix, which is easy to compute knowing the Lorenz equations as well as the parameters' values.
- To get the Jacobian matrix of a neural network, two different methods can be used, again, depending on the case.
  - Using PyTorch's autograd package[4]. We need a point v, F the output of v by the neural network, as well as unitary vectors for each dimension. Getting the gradient of F with regard to v with unitary vector $(1, 0, 0)$ will give $(\frac{\dot{x}}{\partial x}, \frac{\dot{x}}{\partial y}, \frac{\dot{x}}{\partial z})$.
  - By differentiation, the mathematical method to compute derivatives. To compute each value of the Jacobian matrix, we use $\frac{f(v+h)-f(v)}{h}$, where h is a vector having ϵ, a very small value, typically 0.001, in the dimension of interest, and zero elsewhere.
    For example, if we take $forward(v)$ as the output of v by the neural network, $(\frac{\dot{x}}{\partial x}, \frac{\dot{y}}{\partial x}, \frac{\dot{z}}{\partial x}) \approx \frac{forward(v+(\epsilon,0,0))-forward(v)}{\epsilon}$.

2.2.2 Type of neural network

I will describe here which type of neural network were used, their abbreviation (for future references), what are their specificities, as well as parameters they have for which I changed values for experiments.

Multi-layer perceptron [MLP]

It is a basic multi-layer neural network, composed of linear layers, and activation layers between linear layers. Those activation layers are generally Swish activation ($Swish(x) = x \times sigmoid(x)$, where sigmoid is the sigmoid activation function. This is similar to ReLU but without the non differentiability in 0.)
The parameters that I changed for experiments are the number of layers as well as their size (number of weights per layer). They were mainly trained with a L_2 loss (on predicted trajectory compared to target data), and PyTorch's l-BFGS optimizer.

> L_2 loss is a Least Squared error loss. It is defined by :
> $$L_2 loss = \sum_{i=1}^{n} (y_i^{true} - y_i^{predicted})^2$$

[4] https://pytorch.org/docs/stable/autograd.html

Multi-gradient descent network [MGD]

This type of network is the same as MLP in its architecture, and the optimizer was also PyTorch's l-BFGS. The difference between both comes from the way they are trained. This network is trained while optimizing two losses at the same time. The first loss remained the same while I tried two different possibilities for the second loss :

- Penalizing the Frobenius norm of the Jacobians.
- L_2 loss on the model's Jacobian (compared to Jacobian of target data)

These two losses will be detailed more later (cf. 3.2.1). The parameters I changed are also the number and sizes of layers.

Long-Short Term Memory network [LSTM]

This is a type of recurrent neural network that has feedback connection and can keep a kind of memory, although we don't have any control on what he will keep in his memory. The optimizer was, again, PyTorch's l-BFGS.
The parameters I changed are the number of layers, the number of cells per layers, the bidirectionality, the dropout rate, as well as the presence of linear layers after the LSTM layers.

Convolutional neural network [CNN]

In my case, it is composed of 1-D convolutional layers, with convolutions on the temporal dimension, and a linear layer at the end. In the same way as the MLPs, there are activation functions between each layer, and the optimizer is l-BFGS.
The parameters I changed are the number of convolutional layers, the size of layers, as well as the size of the convolutional kernels. Furthermore, convolutions are usually done in a non-temporal domain, hence there is no problem with the past and the future. In our case, it wouldn't be acceptable if convolutions use information from the future, so there was a need to make convolutions causal, i.e. only convolving with information from the past.

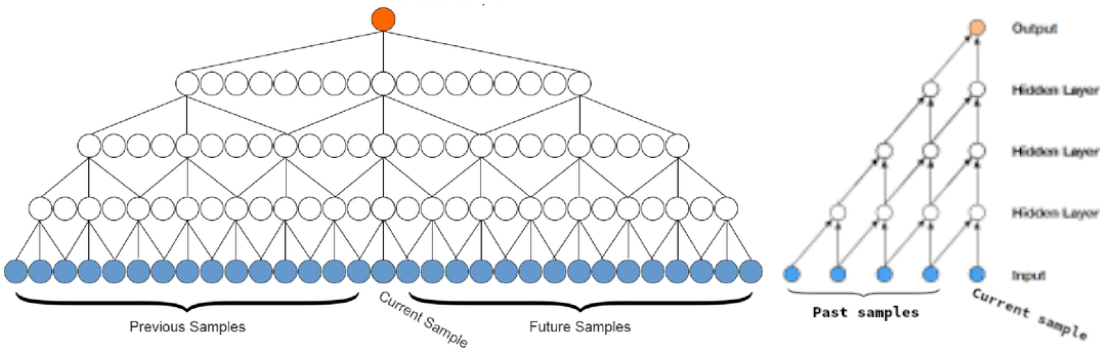


Figure 2.2: Non causal convolutions, with dilated hidden layers (left, found and modified from [8]), and causal convolutions (right, found and modified from [5])

To this end, it was necessary to add a padding on the beginning of the data to have the required number of outputs. The amount of padding needed is equal to the size of the kernel minus 1,

[5] https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_conv/

multiplied by the number of convolutional layers. For example, in Figure 2.2 (right), there are 4 convolutional layers, each having a kernel of size 2. In that case, we need to add a padding of $(2-1) \times 4 = 4$.

Resnet-like network

The architecture of the networks are the same as for MLPs. The optimizer is PyTorch's Adam. The idea is that when training, the model tries to learn a transformation matrix T such that $v_{n+1} = T \cdot v_n$. If we do it like that, the matrix T looks like $\begin{pmatrix} \mathcal{O}(1) & \mathcal{O}(\epsilon) & ... \\ \mathcal{O}(\epsilon) & ... & \mathcal{O}(\epsilon) \\ ... & \mathcal{O}(\epsilon) & \mathcal{O}(1) \end{pmatrix}$, where diagonal values are 1, and the rest are much smaller values. The problem is that depending on which Δt we choose for data creation, those ϵ values can become too small, hence making the learning process much harder.
In that case, a Resnet-like model, a simple implementation of the idea depicted in [5] may be able to solve this problem. Our model, instead of learning T such that $v_{n+1} = T \cdot v_n$ learns T such that $v_{n+1} = v_n + T \cdot v_n$. The matrix T then becomes $\begin{pmatrix} \mathcal{O}(0) & \mathcal{O}(\epsilon) & ... \\ \mathcal{O}(\epsilon) & ... & \mathcal{O}(\epsilon) \\ ... & \mathcal{O}(\epsilon) & \mathcal{O}(0) \end{pmatrix}$. In this case, there is no more problems of ϵ being much smaller than 1.

2.2.3 Sampling

This consists in, after data creation, sampling the data to keep only part of it. We keep only data according to a sampling period, for example, one every two points, which amounts to multiplying Δt, the sampling frequency, by two.

2.3 Evaluation with Lyapunov Exponents

Lyapunov exponents is a measure quantifying the rate of separation of close trajectories. Two trajectories having their origin point v_0 separated by ϵ will be separated by $e^{\lambda t} \times \epsilon$ (2.1) at time t, where λ corresponds to Lyapunov exponents.
Figure 2.4 shows in two dimensions how we can comprehend the relation between divergence of trajectories and Lyapunov exponents.

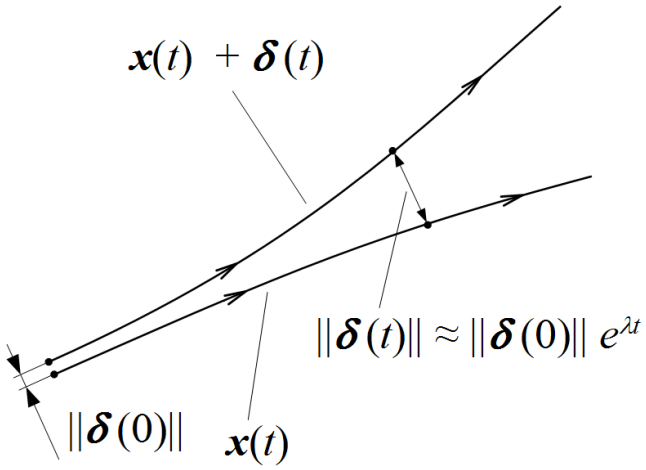


Figure 2.3: Example of the divergence rate

As visible with formula 2.1, positive exponents usually mean that the system is chaotic, whereas negative exponents makes the system predictable. It is important to note that their is an exponent for every dimension of the system. For example, the Lorenz system having three coordinates will have three Lyapunov exponents.

Also, they are originally defined for continuous systems, and while it can be easier to compute them with discrete systems, there are some necessary adjustments to make, visible in line 15 of Algorithm 1. This specific line allows the transition from a discrete system to a continuous system.

There exists multiple methods to compute Lyapunov exponents that characterize the chaotic evolution of dynamical systems. Among those methods, we selected one based on QR-decomposition, because it is relatively general and can be applied to many dynamical systems. This method relies on the computation of the Jacobian matrix at any given time for the system.
The general algorithm, detailed in Algorithm 1, consists in :
Starting with $Q_0 = I$ (the initial perturbation on the trajectory), we have
$Q_{n+1}R_{n+1} = QR\_Decomposition(J_nQ_n)$, where J_n is the Jacobian matrix of the n_{th} point.
The QR Decomposition (cf. Annexe B) helps compute Lyapunov exponents because the Q matrices are orthogonal and span the same space as the Lyapunov vectors, whereas the R matrices describe the changes from one Q matrix to the next. The diagonal entries of R are local growth factors in the direction of Lyapunov vectors, which are averaged to get the Lyapunov exponents (as demonstrate Figure 2.4).

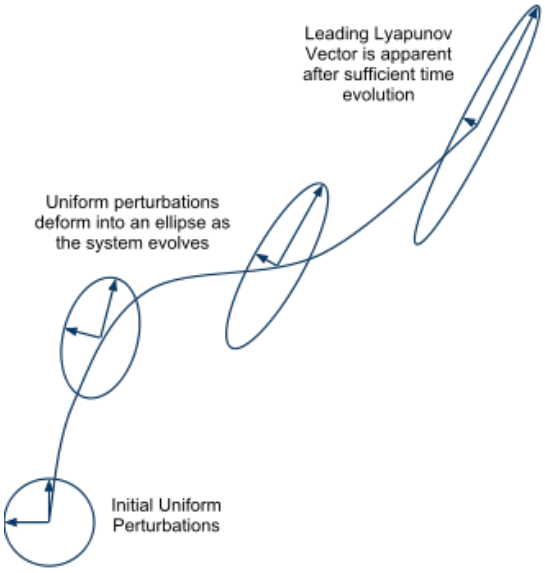


Figure 2.4: Evolution of the frames described by the Q matrices along the trajectory

Algorithm 1 Generic Lyapunov exponents computation

Input:

$\mathcal{J}$ a function used to compute the Jacobian of the system
v the initial position of the Lorenz attractor
`step` a function giving v_{n+1} from v_n
$sizeTraj$ the number of points used for computation
Δt the time-step between v_{n+1} and v_n

Output:

λ the Lyapunov exponents of the system

function LyapunovExponents
    Initialize Q with an identity matrix of dimension (`dim`(v), `dim`(v))
    Initialize $Rdiags$ with an empty array
    for 0 to $sizeTraj$ **do**
        $J_v \leftarrow \mathcal{J}(v)$
        $Q \leftarrow e^{J_v \times \Delta t} . Q$
        Q, $R \leftarrow$`qr`(Q)
        Q, $R \leftarrow$ `positive`(Q, R)
        Append `diag`(R) to $Rdiags$
        $v \leftarrow$ `step(v)`
    end for
    $\lambda \leftarrow$ `mean`(`log`($Rdiags$)) $\div \Delta t$
end function

In this algorithm, several other things are needed : `qr()`, a function performing QR-decomposition, `dim()` gives the dimension, $\cdot$ means matrix or vector multiplication, `mean()` returns the mean vector

of a list of vectors, `log()` returns the log of each element in a list, `diag()` returns the diagonal of a matrix, and `positive()` changes Q and R for R to have positive values in the diagonal, with$Q \cdot R$ having same result.
This last thing is needed because the algorithm used to perform QR-decomposition[6] uses Householder reflections, allowing negative values in R, which isn't possible for our case (because we take their logarithm). We just have to change signs of both matrices, Q and R, to have only positive values in R's diagonal while keeping the same result for their multiplication.

2.3.1 Evaluation on Lorenz analytical system

For testing purposes, I took a Lorenz system with parameters $\sigma = 10, \beta = 8/3, \rho = 28$, its Lyapunov exponents being 0.905 ± 0.005, 0.0 and -14.57 ± 0.01 found in the book [1].
While Algorithm 1 can compute Lyapunov exponents, two steps, lines 14 and 18 can greatly change the results, depending on how they are performed. For each of those two operations, there are two possibilities. Line 14 can be performed either as written, by using the matrix exponential, or by using an interpolation method, doing $(I + J_n \times \Delta t) \cdot Q$, where I is the identity matrix. Line 18 can be performed either by having a function computing the next step from the current one, or providing in the beginning the whole trajectory and moving one point further.
This allows us four different combinations. Of those four, the best one, and the most stable is using matrix exponential and providing the whole trajectory (tests with all four methods in Annexe A). While being a bit slower in computation time, of the different possibilities, it generally gives better results and converges the fastest. Another important point for future experiments is that it is the most stable, by far, when I change the Δt used for Lorenz' data. The difference can be seen in Figure 2.5. In this Figure, each graph represents values of Lyapunov exponents computed at each point, meaning computed from the mean of all points before the one represented on the x-axis.

[6] https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.qr.html

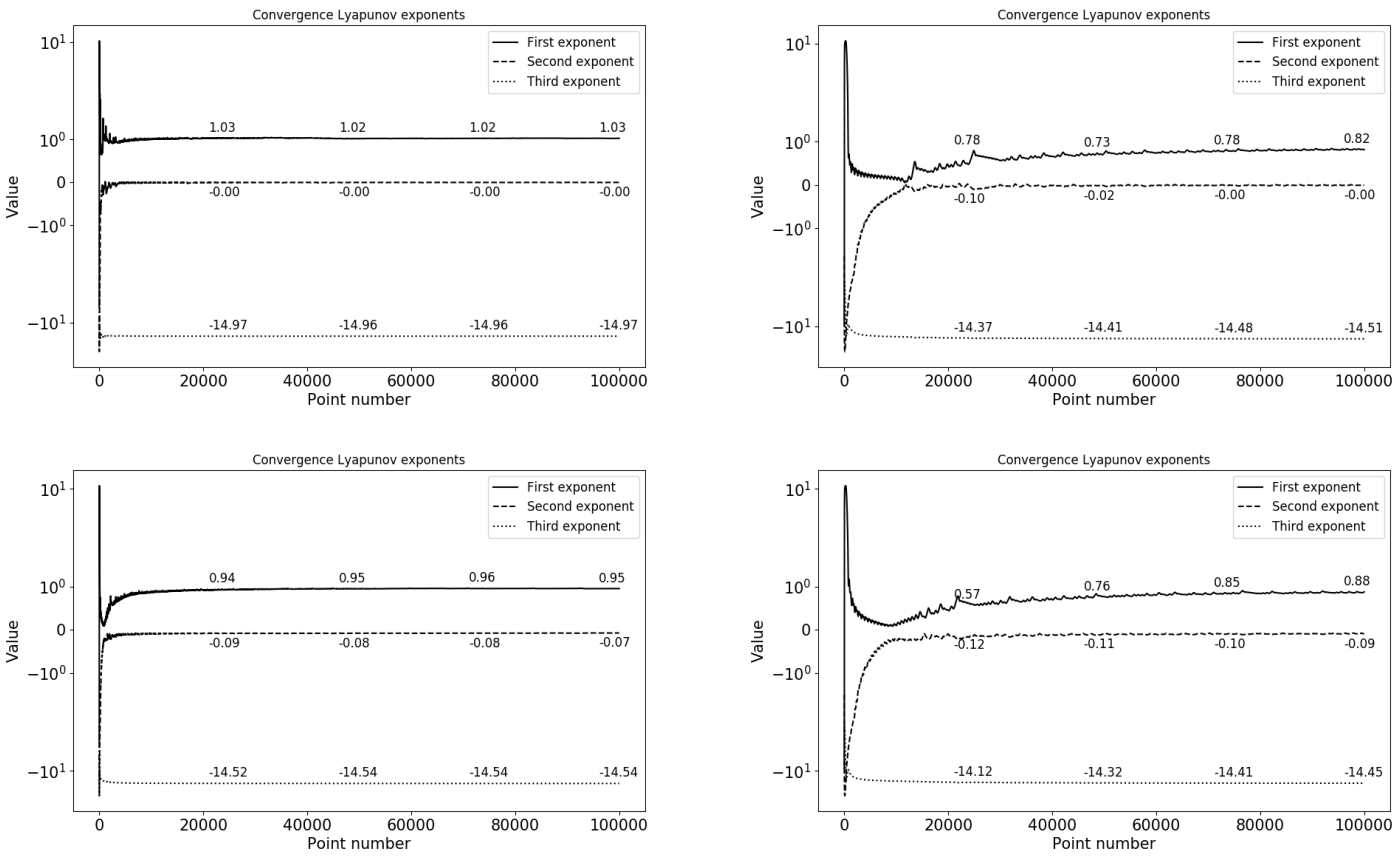


Figure 2.5: Convergence of Lyapunov exponents over a 100000 points trajectory. On top is using a step function with the interpolation method, and on the bottom is with matrix exponential and providing the whole trajectory. On the left is with $\Delta t = 0.01$, and on the right with $\Delta t = 0.001$. Lyapunov exponent values are shown at x = [25000, 50000, 75000, 100000]

While it appears to be relatively similar for $\Delta t = 0.001$, the second option seems more stable when we take a bigger Δt.

2.3.2 Evaluation on model's predictions

What Machine Learning models are learning on Lorenz data is how to perform one step from v_n to v_{n+1}. In that case, the Δt used to create the data is integrated in the model, and hence, when we compute the model's jacobian, it also is correlated to this Δt. The model, contrary to Lorenz equations, also doesn't need to be linearized.
Considering this, we have to change line 14 in Algorithm 1 to become $Q = J_v \cdot Q$.

Following this change, we should have a working algorithm to compute Lyapunov exponents of a model's predictions. Unfortunately, we couldn't know if it worked as intended until successfully getting the correct Lyapunov exponents.
In order to try getting a good model, I first tried different architectures of neural networks, mainly

changing the number of layers as well as the size of those layers. Results where midly disapointing. While networks were sometimes close to getting correctly the first exponents, they were generally worse for the two other exponents.
The first conclusive result I had was after using another method, performing a multi-gradient descent (optimizing two gradients at the same time) for optimizing weights of the neural network. In order to perform optimization with multi-gradient descent, I used the method described in [9]. Instead of only using the distance between predicted and real data, I also used the distance between the model's jacobian and the true data's jacobian.
The difference in computation of Lyapunov exponents for two trained models can be seen in Figure 2.6. It is clearly visible that while the first exponent is not that far for the MLP model, the two others are a lot worse, whereas the MGD model gets almost perfectly all exponents.

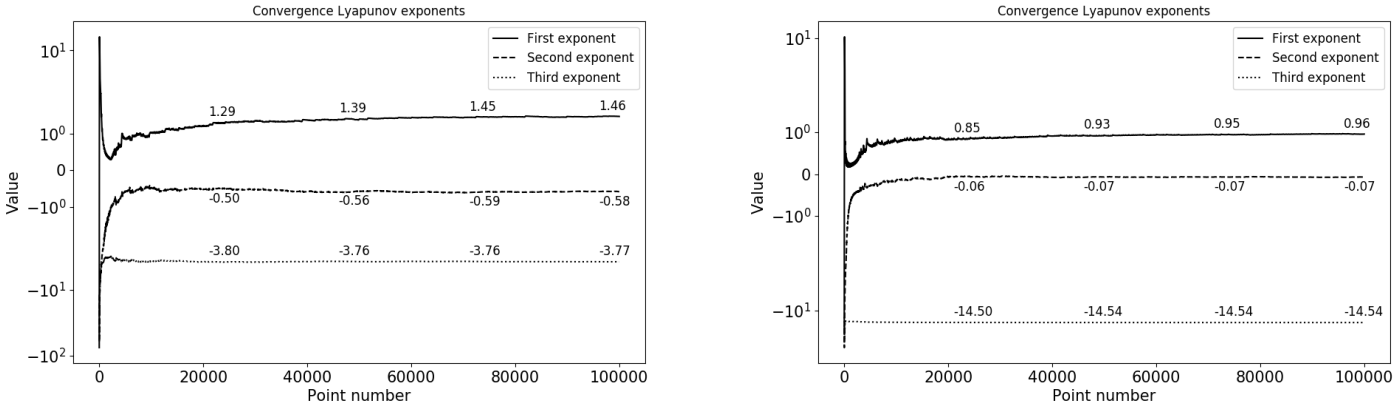


Figure 2.6: Convergence of Lyapunov exponents over a 100000 points trajectory. On the left is a good result for a MLP, on the right is a good result for a MGD. Lyapunov exponent values are shown at x = [25000, 50000, 75000, 100000]

CHAPTER 3

Experiments and results

All experiments were performed on a Lorenz attractor, with initial position $(0, 1, 1.05)$ and parameters $\sigma = 10,\ \ \beta = 8/3,\ \ \rho = 28$.
Since this work has as a final goal to work on controling flows in real life situations, I decided to work with data structured as a single long trajectory of a Lorenz system. Furthermore, a previous study on Lorenz's correlation dimension [7] showed that a single trajectory worked better than multiple smaller trajectories.

For the size of this trajectory, theoretical studies showed the minimum data needed to converge on the value of the correlation dimension of a chaotic system. For the Lorenz system I use with a $\Delta = 0.01$, 27000 points is enough [7]. Since the Δt is chose is smaller ($\Delta t = 0.005$), I chose to take 54000 points to keep the same global time of the system. The first goal, that I kept until the end, was to be able to train a model to correctly predict one timestep ahead (equal to Δt) of the data.

3.1 First experiments

The first experiments were to try training a model to accurately predict one step ahead, and were evaluated with a L_2 loss. The first models I trained were LSTMs and computing it's jacobians for computation of Lyapunov exponents wasn't implemented, hence the evaluation with L_2 loss. The first models I tried training are LSTMs, because from a physical point of view, knowing the past is really important if we want to calculate the future of a physical system. LSTMs are a kind of network possessing a "memory", hence the choice of trying it first.
I tried training different LSTM architectures by using data created with different $\Delta t(0.01, 0.005, 0.001)$. I chose to continue with $\Delta t = 0.005$ for future experiments because it gave the best results on the LSTM's training.

Since computing Lyapunov exponents with LSTMs involved more complications in how to compute its jacobian matrix, I switched to train basic neural networks composed of linear and activation layers. I was first surprised by their losses, which were better than those of the LSTM architectures, while also being much faster to train (L_2 losses are $\mathcal{O}(10^{-3})$ against $\mathcal{O}(10^{-2})$ for LSTMs). The L_2 loss seems relatively good and predictions, and as evidence in Figures 3.1 and 3.2, are visually almost the same.

All figures shown in this section were obtained with a MLP composed of two layers of size (64, 32)
Most of the MLP architectures tested with their results are visible in Annexe D

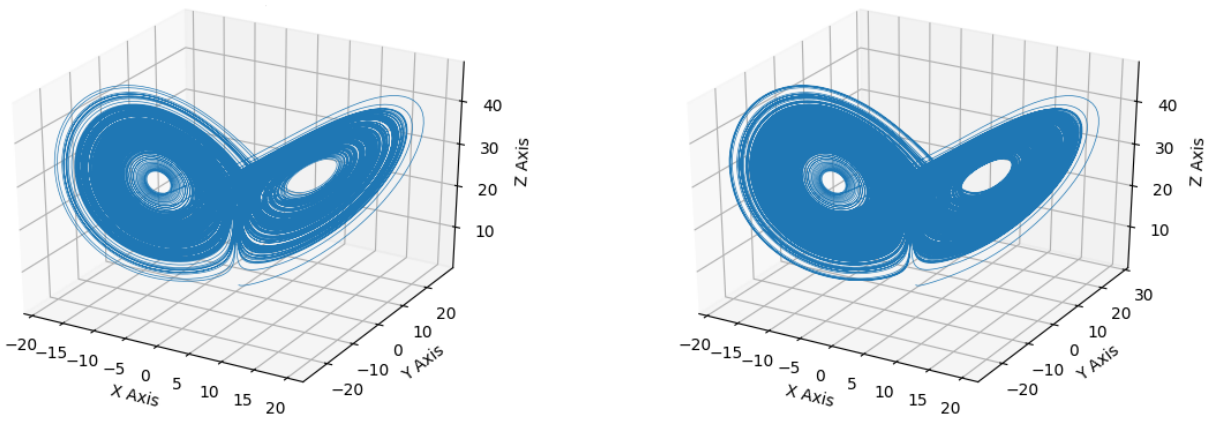


Figure 3.1: 3D visualization truth (left) & prediction (right)

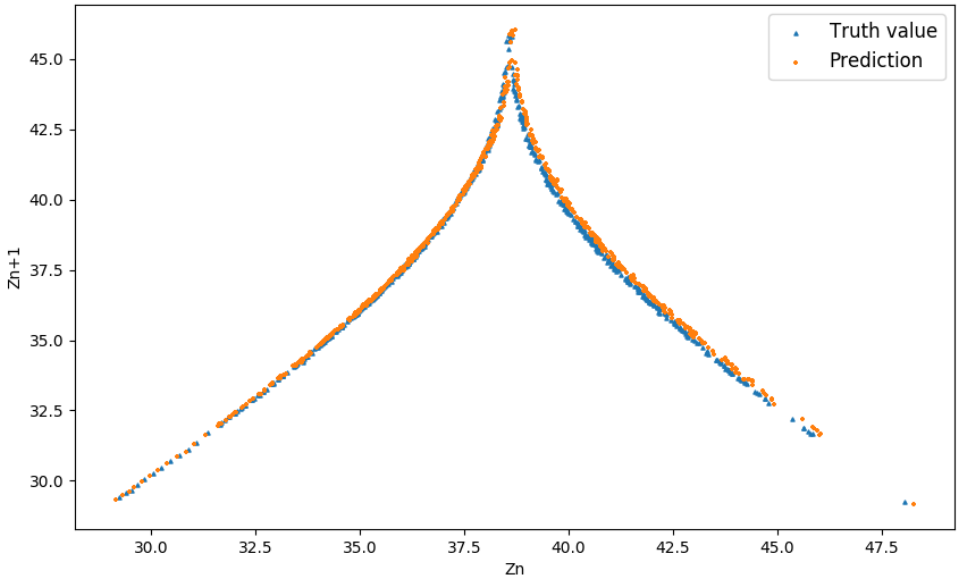


Figure 3.2: Visualization of Z_{n+1} with regard to Z_n

However, although it visually looks like the neural network managed to reproduce the Lorenz system quite well, computed Lyapunov exponents are often quite far from the correct values. Not

getting the correct Lyapunov exponents reflects that the model, although able to predict quite accurately one step ahead, doesn't necessarily learn the underlying principles behind the Lorenz system. It for example is translated in the model's inability to predict alone, i.e. no using the truth input at every time step, as is shown in Figure 3.3.

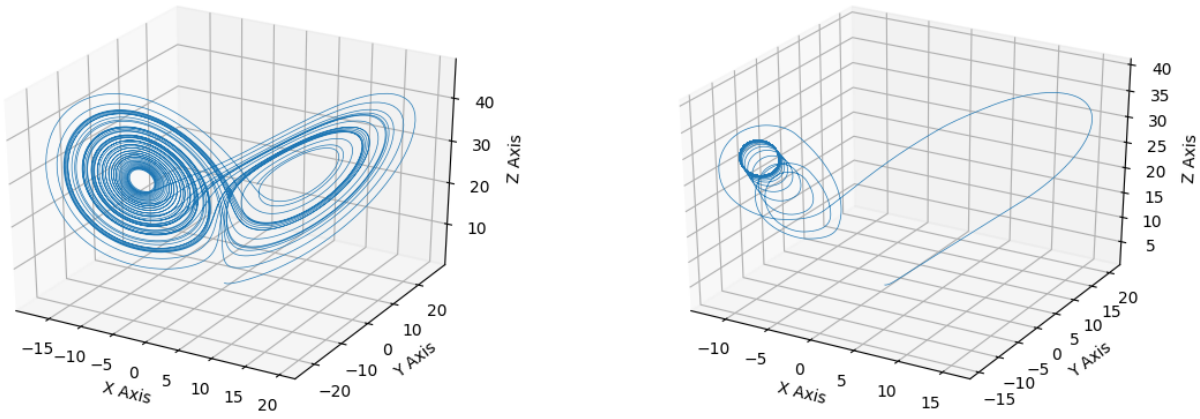


Figure 3.3: Prediction of the model alone, with target (left) and predictions (right)

Along with this, since I wasn't able to get the correct Lyapunov exponents, I didn't have proof that the implemented method could correctly get Lyapunov exponents with a neural network. There was hence a need to find a better method, or a better model to get those Lyapunov exponents.

3.2 Added information

Since the previous methods and models weren't enough, there was a need for something more. During the previous experiments, along with the loss and Lyapunov exponents, I made graphics of singular values (using the Jacobian matrix differentiating the output with regards to the input) along the predicted trajectories. It is informative because singular values (cf Annexe C) represent the amplitude of the deformation of an object submitted to a transformation. It is hence similar to Lyapunov exponents. From a validation point of view, getting correct singular values is equal to getting correct Lyapunov exponents.
As is shown in Figure 3.4, the pattern of singular values for the model seems to match, but its amplitude is bigger than necessary.

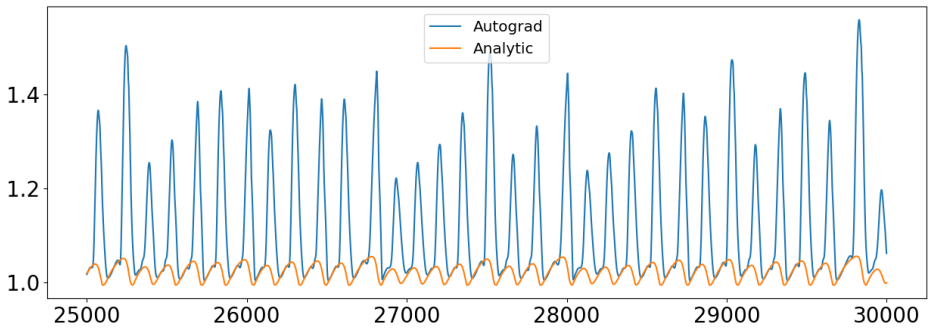


Figure 3.4: Singular values computed along points 25000 to 30000 of data (Analytic) and the model's predicitions (Autograd), obtained with a MLP with two hidden layers (64, 32)

Since it was the case, I researched a way to train the model while also trying to correctly get the singular values. The first solution was using a second loss and optimize the model with it, while still keeping the previous L_2 loss on the trajectory.

3.2.1 Multi-gradient

When adding a second loss, there is a need to decide how to use the two losses to correctly optimize the model's weights. The method described in [9] allows to optimize the model using simultaneously the two losses, by finding a direction optimizing the two losses' gradients.
The too big amplitude visible in Figure 3.4 probably meant that the model's Jacobians matrices were sometimes too big. My first try for a second loss was hence penalizing the sum of the Frobenius norms of the model's Jacobian matrices. The goal was to help reduce the values of those matrices and hence making the model closer to the correct behaviour.

Using this penalization with Frobenius norms was, although a bit better on the loss side, not better, and sometimes even worse with regards to singular values (Figure 3.5) as well as Lyapunov exponents.

Most of the architectures tested with their results are visible in Annexe E

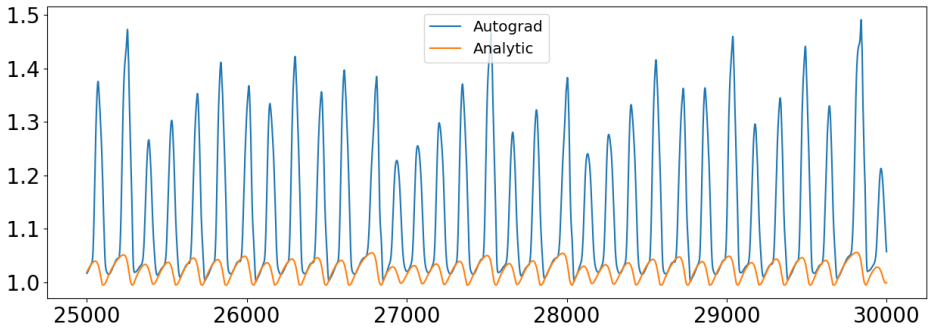


Figure 3.5: Singular values computed along points 25000 to 30000 of data (Analytic) and the model's predicitions (Autograd), obtained with a MGD with two hidden layers (64, 16), trained with Frobenius norm penalization

My second try was using a L_2 loss between the data's Jacobian matrices and the predictions' Jacobian matrices. Although we don't generally know the Jacobian matrix of physical system in real life applications, it was interesting as a study case, to know if using it was sufficient in our case. This time, results were exactly what we hoped for. In Figure 3.6, we can see that the model's behaviour seems to match really well the data. Moreover, as stated in Chapter 2, I finally got really close to the correct Lyapunov exponents, meaning the method used to compute them with neural networks seems to be correct.

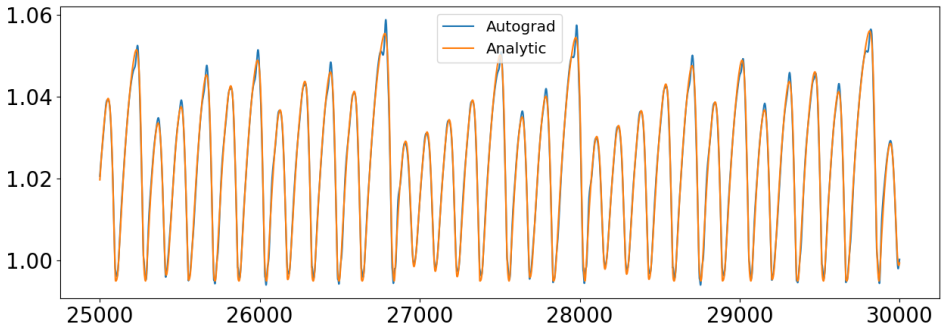


Figure 3.6: Singular values computed along points 25000 to 30000 of data (Analytic) and the model's predicitions (Autograd), obtained with a MGD with two hidden layers (64, 32), trained with L_2 loss on predictions' Jacobian matrix

3.2.2 Information from the past (LSTM/CNN)

A well-know theorem [10] for chaotic dynamical systems states that chaotic dynamical systems can be reproduced from a sequence of observations of its state, depending on some conditions. Using models that use the current observation and past observations to predict could then possibly be better than other kinds of models.

LSTMs can keep some information from past observations, but we don't and can't know which information the model kept. Unfortunately, results were also worse than simple MLPs and computing the LSTMs Jacobian matrices could be subject to problems depending on how we use its memory. CNNs convolve the current observation with only a few past observations, depending on its convolutions' kernel sizes. This create a higher dimensional data probably containing more information that can help the model to predict. We settled on a method (by differentiation, only adding ϵ to the current observation) to compute its Jacobian matrix, and looking at the singular values of a CNN (Figure 3.7), the amplitude of the value are respected, so the method should be correct. Unfortunately, possibly from the fact that Lyapunov exponents aren't really computed on an observation but on the convolution of observations, Lyapunov exponents tend to all be negative. The CNN used for Figure 3.7 has as Lyapunov exponents $(-0.49, -6.04, -54.61)$. t

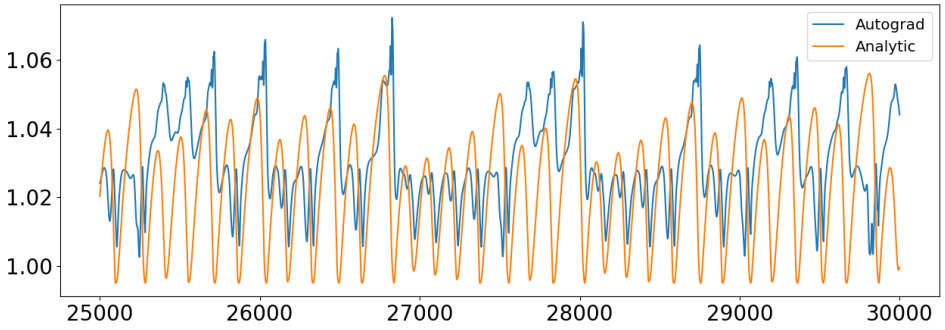


Figure 3.7: Singular values computed along points 25000 to 30000 of data (Analytic) and the model's predicitions (Autograd), obtained with a CNN with convolutional layers' output size (64, 16) and kernel size of 15

3.3 Manipulating data

I managed to train models able to correctly get lyapunov exponents of the Lorenz system. Unfortunately, training the model required knowledge of the system's jacobian which for a complex chaotic system is generally unknown, hence reducing the practicality of this method. Instead of trying to change the way I train the models, or the models themselves, we had the idea of manipulating the data to see whether it could improve performance. As stated in Chapter 2, I did sampling, i.e. I kept only one every n points (values are n = [1, 3, 5, 10, 50, 100]).

Most of the architectures tested with their results are visible in Annexe F

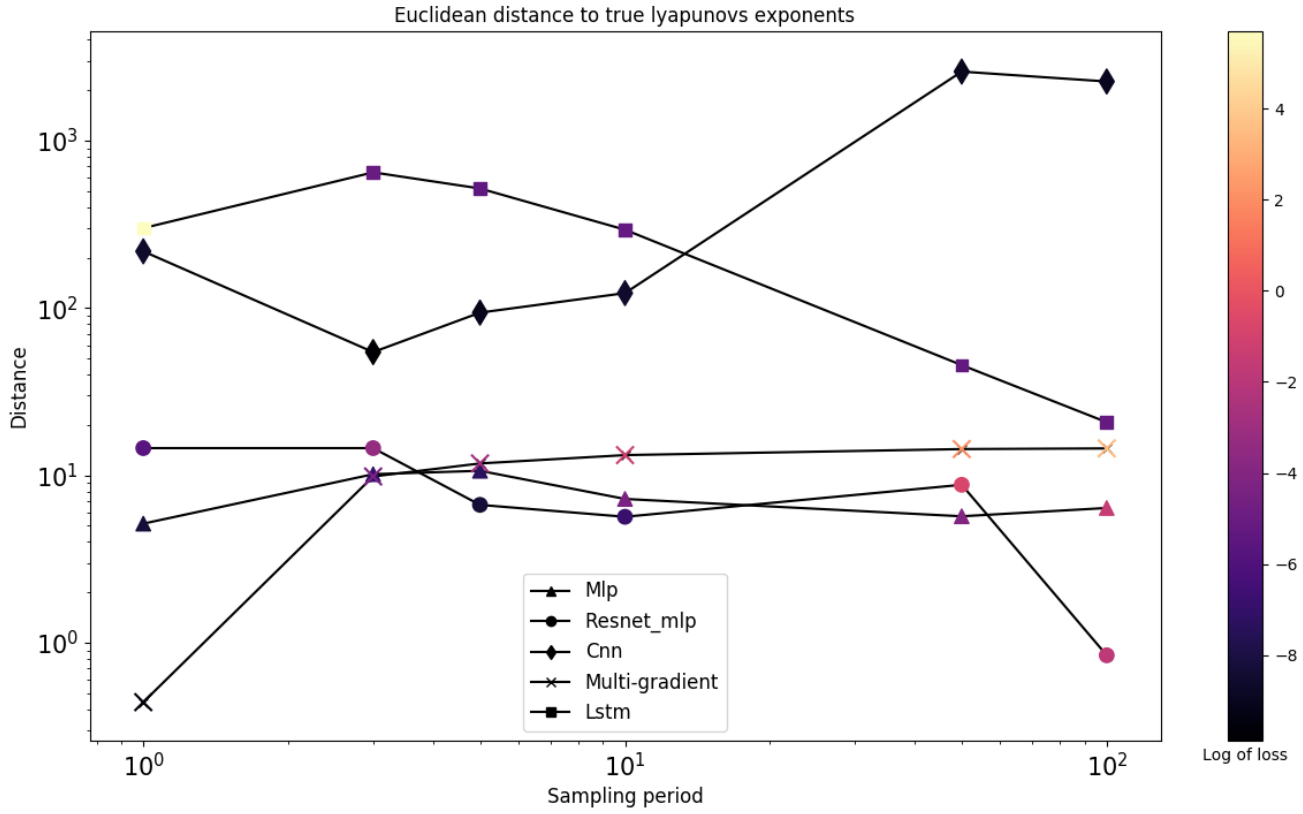


Figure 3.8: Distance of computed Lyapunov exponents compared to the true Lyapunov exponents, depending on type on neural network. $\Delta t = 0.001$ for data creation

This experiment was performed with $\Delta t \in [0.01, 0.005, 0.001, 0.0005, 0.0001]$ for data creation, and then with a sampling performed on these data (sampling period of 1, 3, 5, 10, 50, 100). With a sampling period of 1, MGDs are always better than other types of networks, regardless of the Δt, although the difference diminishes when it becomes smaller. LSTMs tend to improve the bigger the sampling period, and hence the bigger the global Δt (sampling period times Δt for data creation), and sometimes become better than other types, mainly because the other types end up with poorer performance for high sampling period.
The smaller the Δt, the better the performance of Resnets MLPs (MLPs architectures trained like Resnets) compared to other networks.

3.4 Resnet-like models

As introduced in Chapter 2, those models only have to learn f such that $f(v_n) = \frac{v_{n+1}-v_n}{\Delta t}$ instead of $f(v_n) = v_{n+1}$. The models I trained possess the same architecture as MLPs, i.e. only linear layers and activation layers. Only the method to get the output differs.

This allows those kinds of models to become almost independant of Δt, and are really stable when its value varies, as shown in Figure C.1.

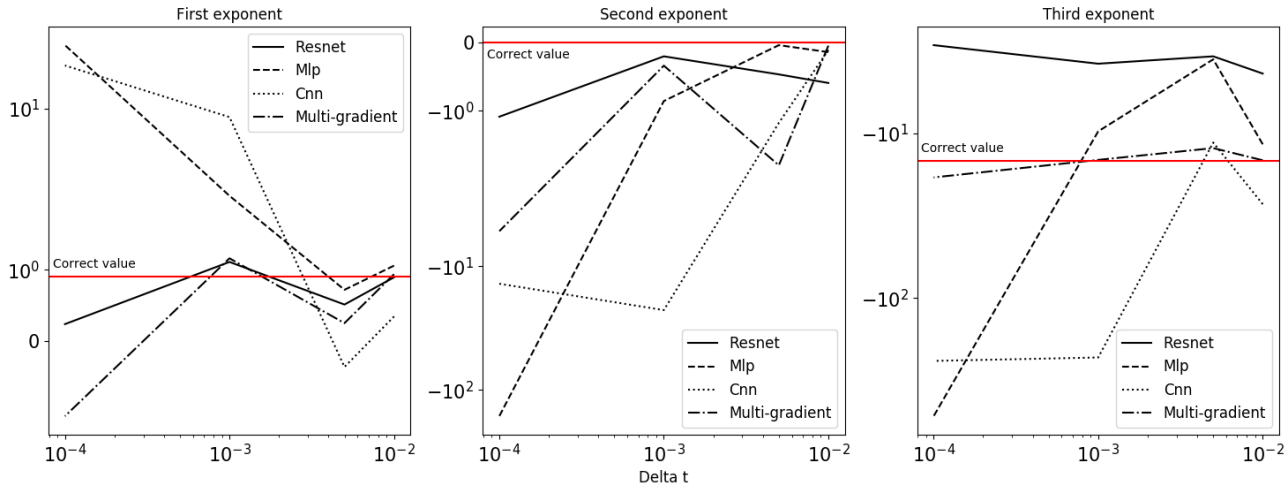


Figure 3.9: Distance of computed Lyapunov exponents compared to the true Lyapunov exponents, depending on Δt for different types of neural networks

Since they are more stable, it might be possible to learn in a globally smaller time, if the data sampling is smaller. Also, it is important to note that results were obtained for Resnet-like models with the same architectures and same training method as for MLPs. It is possible to train them in the same way as Multi-gradient descent networks, and it would be interesting to see if it is possible to always get the correct Lyapunov exponents, regardless of the Δt used.

3.5 Analysis of intermediate layers

When we apply Machine Learning techniques on physical systems, the models that are trained are often a lot more predictable than those systems, particularly when the size of their layers increases. In chaotic dynamical systems, Lyapunov exponents are a good way to know the predictability of a system, since negative exponents means that two trajectories with a small initial separation will only grow closer. The inverse of the biggest Lyapunov exponent is know as the Lyapunov time, time taken for the distance between two trajectories to increase by a factor of e.
Studying what happens in the intermediate layers of a neural networks can give us clues on what happens to make those networks more predictable than actual physical systems.
In order to study Lyapunov exponents on intermediate layers, they have to be of the same size, since we need square matrices for Jacobian matrices to perform QR Decomposition.

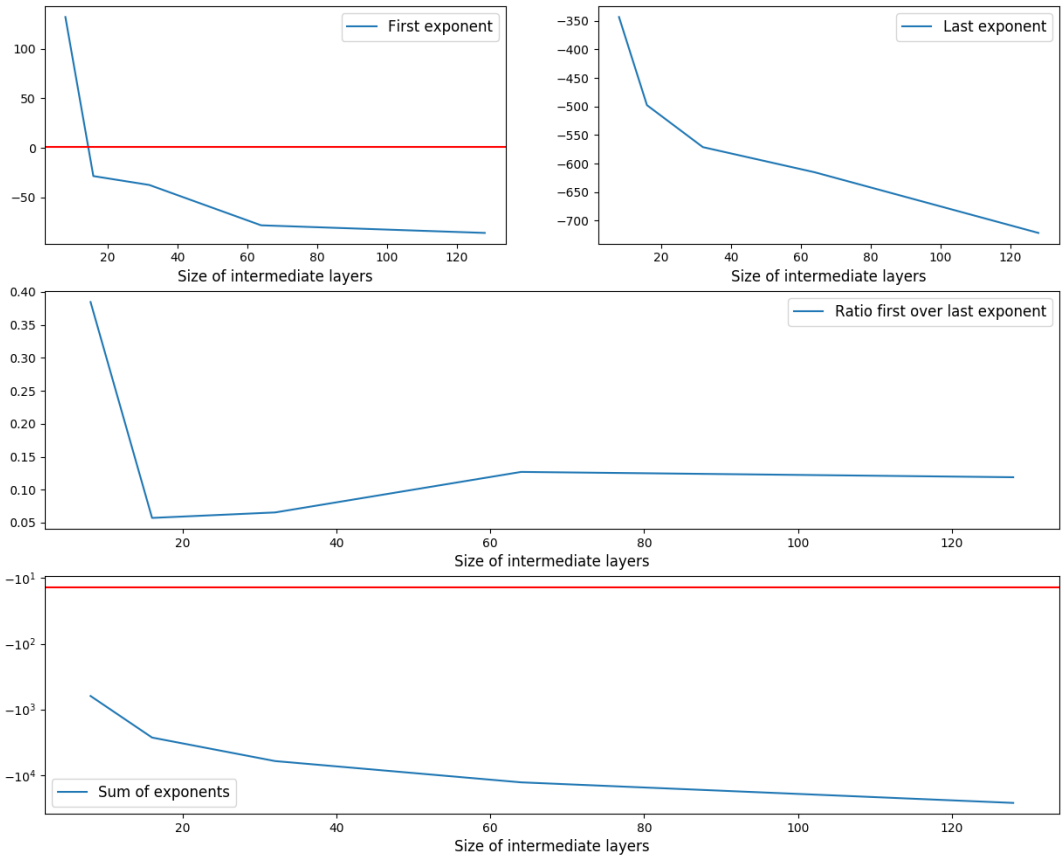


Figure 3.10: Lyapunov Exponents computed between intermediate layers (First and Last, Ratio, and sum of all three) depending on the size of the layers

Figure 3.10 shows that the bigger the size of the intermediate layers, the more negative are the exponents, and hence the more predictable are the networks, from a physical point of view. It also seems that the values will stabilize at some point, after which it will be useless to further increase the size of the layers.
Another important point is that those results were obtained with MLPs composed of two linear layers, and results might be different, maybe values will be even more negative, with more layers.

CHAPTER 4

Conclusion and perspectives

Applying Machine Learning to concrete problems often suffer from its generic methods and there is a need for more specific "tools". Computing Lyapunov exponents can be one of those tools, useful when working on many problems involving chaotic dynamical systems, and more generally physical systems.
Through many experiments, I showed that although it is possible to compute Lyapunov exponents for Machine Learning models, we also often encounter trouble when using some complex models. I also tested other methods, different than changing for a better model, that sometimes have interesting results. For example, trying to train MGD models the way we train a Resnet could probably give good results, much more resistant to sampling than traditional MGDs, and hence potentially faster "in real time". For example, it could be interesting in real world applications by getting correct results in a shorter time just by increasing the rate of collecting information.
Experiments with intermediate layers also showed a possible explanation of the predictability of Machine Learning models through using Lyapunov exponents.

Also, although the experiments showed give some conclusive results, it doesn't mean that every method can work. Among those I have tested, some failed and can also give us information on what could work and what will probably not work when using Machine Learning methods.
Among those failed experiments, as mentioned before, networks often have poor results when asked to predict multiple steps ahead (multiple times one step ahead) without truth input. I tried GANs (Generative Adversarial networks), models generally used for generation, to see if they could be better than other models to predict on multiple time steps. Unfortunately, GAN training often takes a long time, and there was no telling if the learning process was going in the right direction.
I also tried training MLPs to predict multiple time steps ahead, and although possible for a few steps, it often hits a limit around 5, 10, and it becomes increasingly harder to train the higher the number of time steps we try to predict alone.
Last but not least, I tried training a kind of MLP with memory. Its method was to handle data in a way that it transforms it to have a memory. Instead of performing linear combinations on one input, it performs it on one input, plus its previous few inputs, and gives only one ouput. Although the idea was interesting, it performed worse than simple MLPs.
Last but not least, any method tested was, regardless of Lyapunov exponents, unable to predict correctly on a longer horizon than a few time steps. Although getting correctly the Lyapunov exponents shows that the models I trained are still lacking in some way to reproduce a chaotic dynamical system.

Some interesting directions to pursue on would be : Exploring Resnets with MGDs, further researching why models are unable to predict multiple time steps ahead, and use methods such as computing Lyapunov exponents to not only evaluate the learning process, but also help models learn better and faster with it.

To conclude, this work presents a method to compute Lyapunov exponents of neural networks, experiments showing what we can expect of Machine Learning models with regard to those Lyapunov exponents, and more than that, ideas about how to use Machine Learning methods on physics and other practical problems.

Acknowledgements

I would like to thank my supervisors, Onofrio, Lionel and Alessandro for the help they provided to understand some physical knowledge I originally didn't have. I would like to also thank them for making me see another aspect of Machine Learning which I discovered.
I would like to thank all of my supervisors for the help they provided in writing this report.

ANR/DGA Flowcon project, ANR-17-ASTR-0022.

Bibliography

[1] Kathleen T. Alligood, Tim D. Sauer, and James A. Yorke. *Chaos: An Introduction to Dynamical Systems (Textbooks in Mathematical Sciences)*. Springer, 2000.

[2] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. 2016.

[3] Ronald R. Coifman and Stéphane Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30, July 2006.

[4] Russell A. Edson, J. E. Bunder, Trent W. Mattner, and A. J. Roberts. Lyapunov exponents of the kuramoto-sivashinsky pde, 2019.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[6] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, March 1963.

[7] Alexandre Allauzen Laurent Cordier Guillaume Wisniewski Lionel Mathelin Michele Alessandro Bucci, Onofrio Semeraro. Control-oriented model learning with a recurrent neural network. 11 2018.

[8] Dario Rethage, Jordi Pons, and Xavier Serra. A wavenet for speech denoising. 06 2017.

[9] Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. 2018.

[10] Floris Takens. *Detecting Strange Attractors in Turbulence. Lecture Notes in Mathematics*, volume 898, pages 366–381. 11 2006.

[11] Ronen Talmon, Stephane Mallat, Hitten Zaveri, and Ronald R. Coifman. Manifold learning for latent variable inference in dynamical systems. *IEEE Transactions on Signal Processing*, 63(15):3843–3856, August 2015.

[12] Lloyd N. Trefethen and David Bau. *Numerical linear algebra.* SIAM, 1997.

[13] Kailiang Wu and Dongbin Xiu. Numerical aspects for approximating governing equations using data. 2018.

APPENDIX A

Lyapunov exponents computation method comparison

I tried four different methods to compute Lyapunov exponents on the analytical system, combinations of two choices :

- The next point is computed with the Euler formula / all the points computed with an integration method beforehand
- With first order estimation ($I + J \times \Delta t$) / Directly using matrix exponential ($e^{J \times \Delta t}$)

Of those 4 possibilities, I tried them for 10000, 100000, and 1000000 points, and for delta t of 1e-3, 5e-3, and 1e-2.

Table A.1: Test of the four different computations, with correct values being 0.905, 0 and -14.57, everything rounded to 10^{-2}

10000 points			
	Δt		
Method	1e-3	5e-3	1e-2
Euler formula First order estimation	0.15, -0.21, -13.61	**0.89, -0.02, -14.69**	1.00, -6.49, -14.94
Euler formula Matrix exponential	0.11, -0.27, -13.51	0.72, -0.28, -14.10	0.66, -0.59, -13.74
Trajectory by integration First order estimation	0.16, -0.17, -13.66	0.96, 0.13, -14.91	1.15, 0.46, -15.73
Trajectory by integration Matrix exponential	0.10, -0.21, -13.56	0.76, -0.11, -14.32	**0.87, -0.09, -14.45**

100000 points			
Euler formula First order estimation	**0.82, 0, -14.51**	0.95, 0, -14.78	1.02, 0, -14.97
Euler formula Matrix exponential	0.76, -0.04, -14.40	0.76, -0.24, -14.19	0.67, -0.55, -13.78
Trajectory by integration First order estimation	**0.92, -0.06, -14.56**	1.13, 0.15, -15.15	1.22, 0.46, -15.84
Trajectory by integration Matrix exponential	**0.88, -0.09, -14.45**	**0.95, -0.08, -14.54**	**0.95, -0.07, -14.54**

1000000 points			
Euler formula First order estimation	**0.91, 0, -14.61**	0.95, 0, -14.78	1.04, 0, -14.98
Euler formula Matrix exponential	0.87, -0.04, -14.50	0.76, -0.23, -14.19	0.68, -0.55, -13.80
Trajectory by integration First order estimation	1.00, -0.04, -14.66	1.14, 0.15, -15.17	1.23, 0.46, -15.86
Trajectory by integration Matrix exponential	**0.95, -0.07, -14.54**	**0.96, -0.08, -14.55**	**0.97, -0.08, -14.56**

APPENDIX B

QR Decomposition

We often find ourselves interested in the columns spaces of a matrix A. Note the plural: these are the successive spaces spanned by the columns a_1, a_2,... of A:

$$\langle a_1 \rangle \subseteq \langle a_1, a_2 \rangle \subseteq \langle a_1, a_2, a_3 \rangle \subseteq \quad ...$$

The notation $\langle ... \rangle$ indicates the subspace spanned by whatever vectors are included in the brackets. The idea of QR factorization is the construction of a sequence of orthonormal vectors q_1, q_2,... that span these successive spaces.
To be precise, assume for the moment that $A \in \mathbb{C}^{m \times n}$ $(m \geqslant n)$ has full rank n. We want the sequence q_1, q_2,... to have the property

$$\langle q_1, q_2, ..., q_j \rangle = \langle a_1, a_2, ..., a_j \rangle, \quad j = 1, ..., n.$$

This amounts to the condition

$$\left[\begin{array}{c|c|c|c} & & & \\ a_1 & a_2 & ... & a_n \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ a_1 & a_2 & ... & a_n \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & ... & r_{1n} \\ & r_{22} & & \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}$$

where the diagonal entries r_{kk} are nonzero–for if the condition holds, then $a_1, ..., a_k$ can be expressed as linear combinations of $q_1, ..., q_k$, and the invertbility of the upper-left $k \times k$ block of the triangular matrix implies that, conversely, $q_1, ..., q_k$ can be expressed as linear combinations of $a_1, ..., a_k$. Written out, these equations take the form

$$\begin{aligned} a_1 &= r_{11} q_1, \\ a_2 &= r_{12} q_1 + r_{22} q_2, \\ a_3 &= r_{13} q_1 + r_{23} q_2 + r_{33} q_3, \\ &\vdots \\ a_n &= r_{1n} q_1 + r_{2n} q_2 + ... + r_{nn} q_n. \end{aligned}$$

As a matrix formula, we have

$$A = \hat{Q} \hat{R},$$

where $\hat{Q}$ is $m \times n$ with orthonormal columns and $\hat{R}$ is $n \times n$ and upper triangular.

Definition copied from [12]

APPENDIX C

Singular Value Decomposition

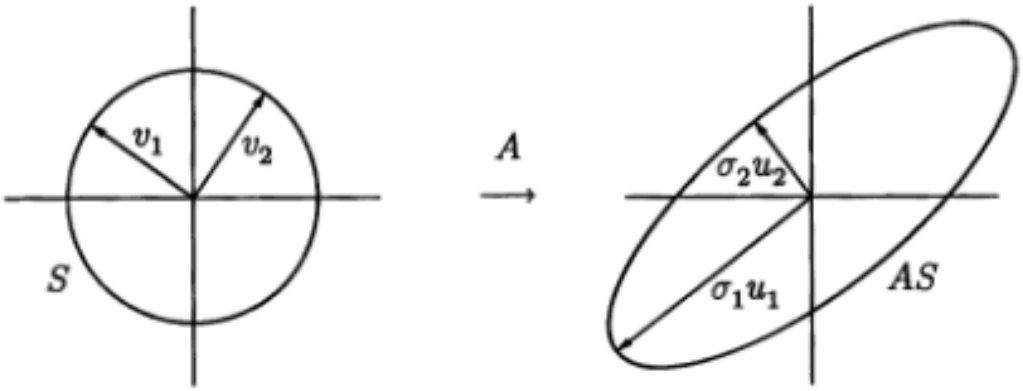


Figure C.1: SVD of a 2×2 matrix

Let m and n be arbitrary; we do not require $m \geqslant n$. Given $A \in \mathbb{C}^{m \times n}$, not necessarily of full rank, a singular value decomposition of A is a factorization

$$A = U\Sigma V^* \tag{C.1}$$

where

$$\begin{gathered} U \in \mathbb{C}^{m \times m} \text{ is unitary,} \\ V \in \mathbb{C}^{n \times n} \text{ is unitary,} \\ \Sigma \in \mathbb{R}^{m \times n} \text{ is diagonal.} \end{gathered}$$

In addition, it is assumed that the diagonal entries σ_j of Σ are nonnegative and in nonincreasing order; that is $\sigma_1 \geqslant \sigma_2 \geqslant ... \geqslant \sigma_p \geqslant 0$, where $p = min(m, n)$.
Note that the diagonal matrix Σ has the same shape as A even when A is not square, but U and V are always square unitary matrices.
It is clear that the image of the unit sphere in $\mathbb{R}^n$ under a map $A = U\Sigma V^*$ must be a hyperellipse in $\mathbb{R}^m$. The unitary map V^* preserves the sphere, the diagonal matrix Σ stretches the sphere into a hyperellipse aligned with the canonical basis, and the final unitary map U rotates or reflects the hyperellipse without changing its shape.

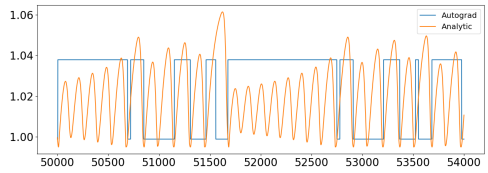
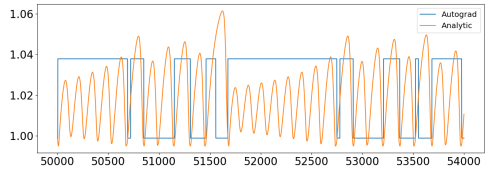
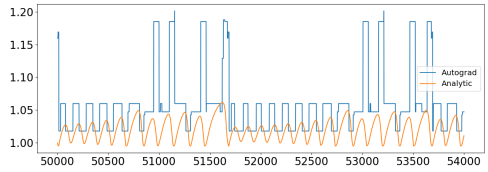
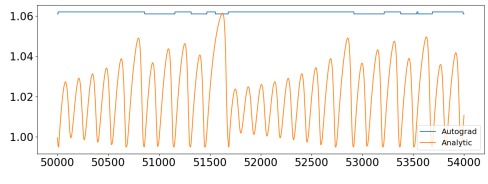
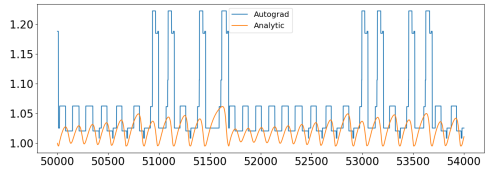
Definition copied from [12]

APPENDIX D

MLP training

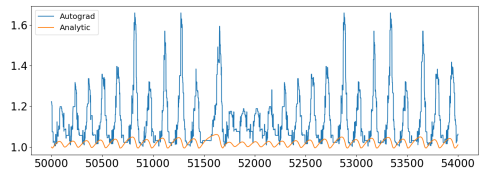
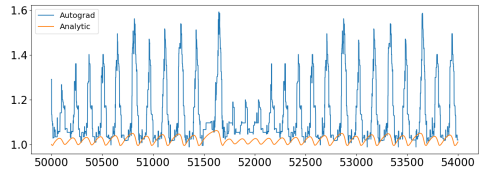
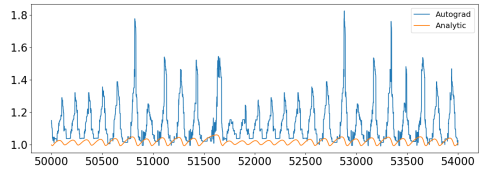
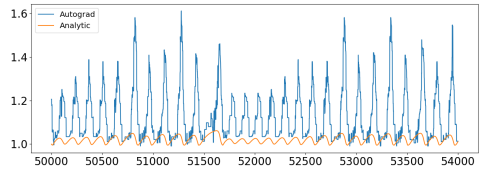
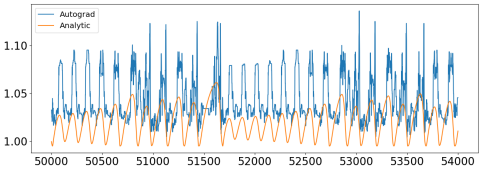
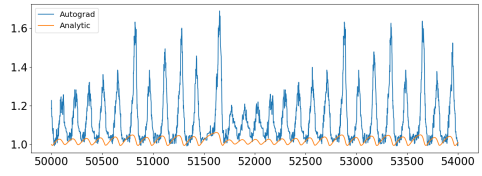
Test on different architectures, with ReLU or Swish activation, with or without bias.
Data is created by using Runge-Kutta, with a delta t of $5e^{-3}$, used as is by the model.
Models tested are all different.

Table D.1: Results per trained model

Model		Loss	Singular Values	Lyapunov exponents
With bias, ReLU activation				
No hidden layer	Training 1	35.87		[1.63, inf, inf]
	Training 2	35.87		[1.63, inf, inf]
8	Training 1	0.0047		[3.2, -1.16, -5.54]
	Training 2	0.02		[-0.12, -0.21, -1.7]
16, 8	Training 1	0.003		[1.23, -1.18, -4.77]
	Training 2	Diverged		

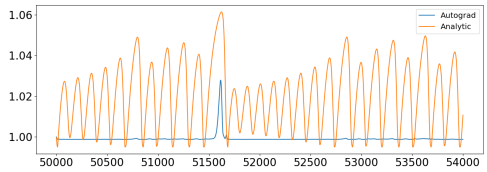
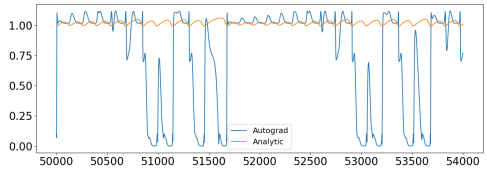
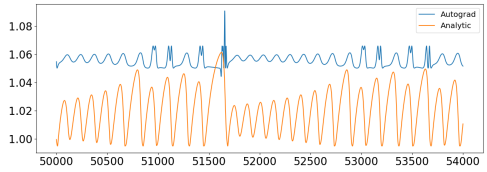
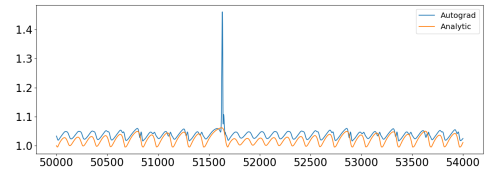
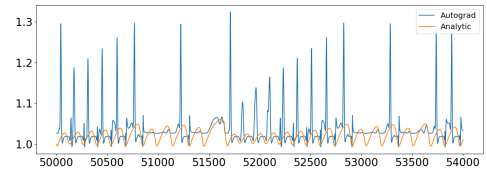
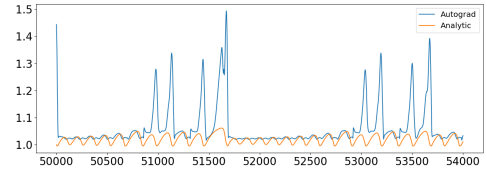
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
128, 64	Training 1	0.00015		[1.17, -0.21, -4.53]
	Training 2	0.00024		[0.93, -0.33, -4.5]
128, 256, 128	Training 1	0.00018		[0.74, -0.28, -5.37]
	Training 2	0.00021		[0.68, -0.29, -4,24]
256, 1024, 1024, 256	Training 1	0.018		[0.54, -1.51, -10-46]
	Training 2	0.0001		[0.75, -0.15, -4.21]

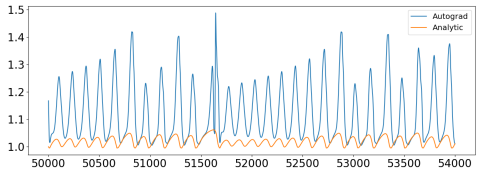
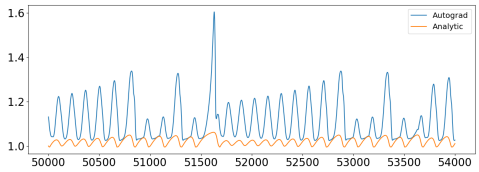
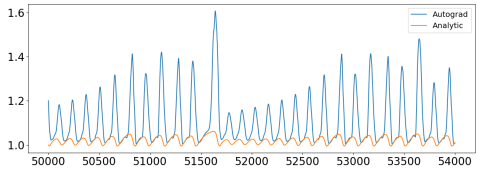
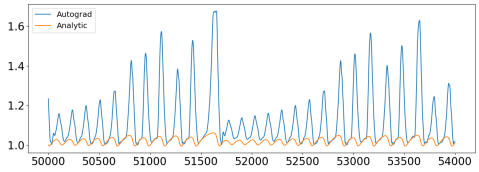
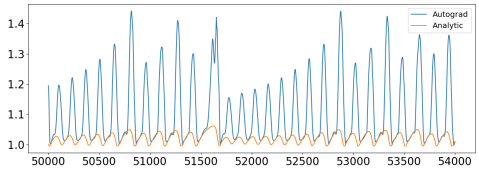
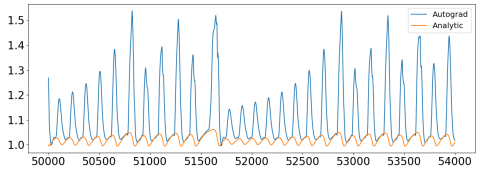
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
With bias, Swish activation				
No hidden layer	Training 1	47.12		[-0.24, -841, -1394]
	Training 2	238		[278, -98, -323]
8	Training 1	0.018		[-0.13, -0.16, -2.86]
	Training 2	0.0095		[0.49, -1.05, -4.96]
16, 8	Training 1	2.71		[0.82, -2.66, -1774]
	Training 2	0.005		[3.69, 0.49, -7.74]

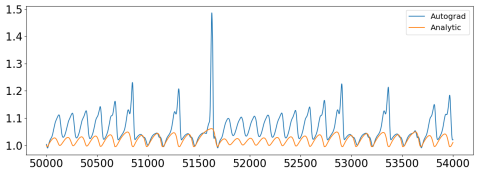
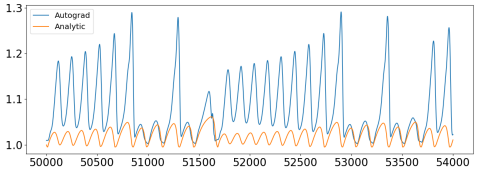
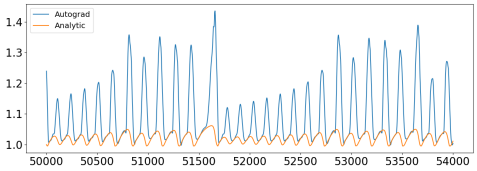
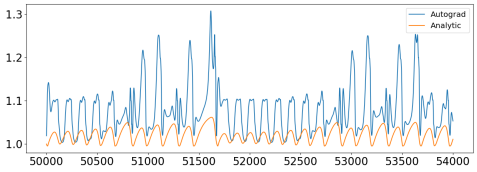
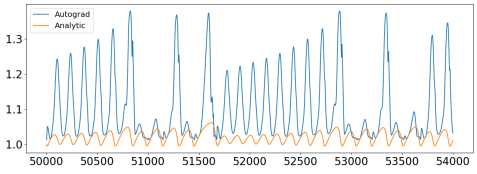
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
32, 16	Training 1	0.0003		[2.62, 0.05, -4.26]
	Training 2	0.001		[2.29, -0.06, -7.19]
64, 32	Training 1	0.0001		[1.46, -0.58, -3.77]
	Training 2	0.00012		[2.8, 0.53, -3.16]
128, 64	Training 1	$7.8e^{-5}$		[0.98, -0.37, -4.54]
	Training 2	$8.6e^{-5}$		[1.13, -0.12, -5.75]

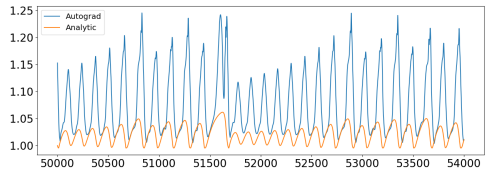
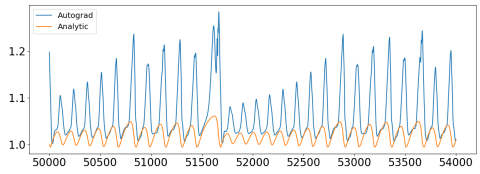
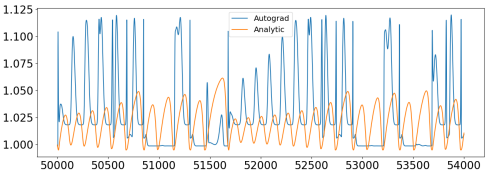
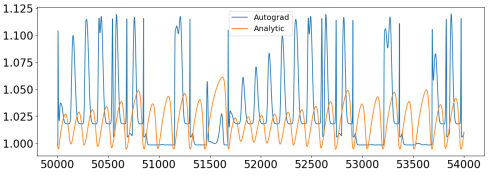
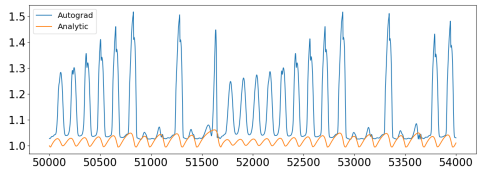
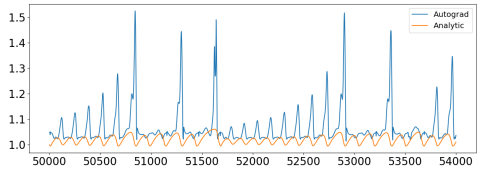
Continued on next page

Table D.1 – *Continued from previous page*

Model		**Loss**	**Singular Values**	**Lyapunov exponents**
8, 16, 8	Training 1	0.00097		[2.51, -0.5, -5.23]
	Training 2	0.00092		[2.63, 0.39, -6.06]
128, 256, 128	Training 1	$6.81e^{-5}$		[1.15, -0.14, -3.51]
	Training 2	0.0057		[2.31, -1.28, -5.28]
8, 16, 16, 8	Training 1		Diverged	
	Training 2	0.0043		[4.44, -0.28, -4.52]

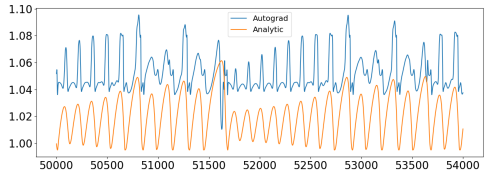
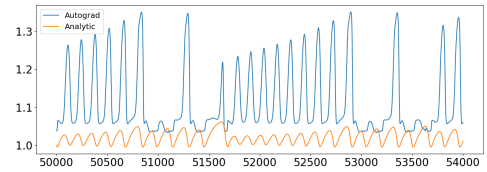
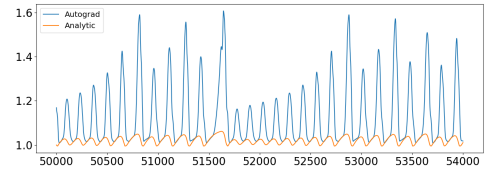
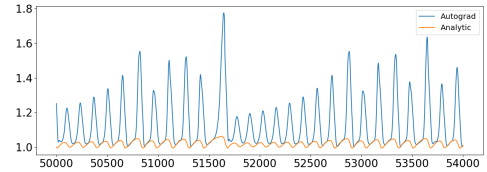
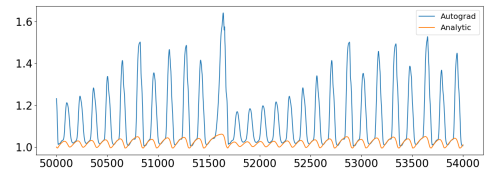
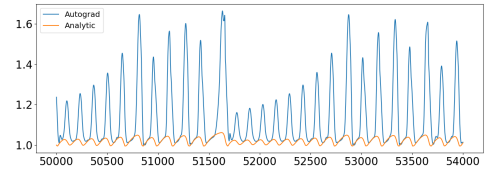
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
256, 1024, 1024, 256	Training 1	0.00016		[0.63, 0.02, -7.15]
	Training 2	0.00014		[0.83, 0.18, -6.74]
Without bias, Swish activation				
No hidden layer	Training 1	26.21		[0.13, -40.72, -94.62]
	Training 2	26.21		[0.14, -40.73, -94.62]
8	Training 1	0.005		[3.95, -0.23, -4.17]
	Training 2	0.01		[1.76, -0.28, -8.15]

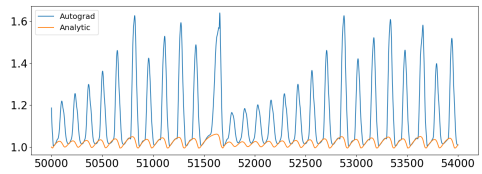
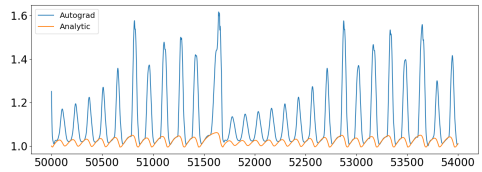
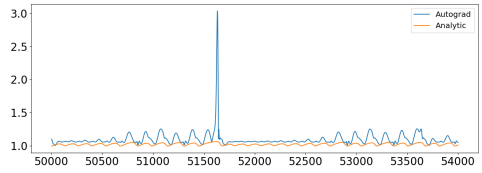
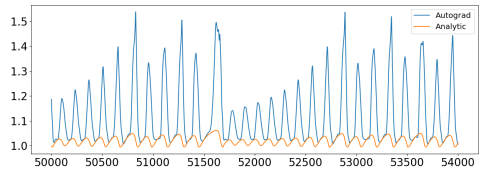
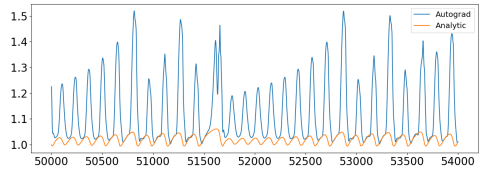
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
16, 8	Training 1	0.018		[0.75, 0.11, -4.85]
	Training 2	0.011		[3.72, 0.23, -6.81]
32, 16	Training 1	0.0002		[2.61, 0.65, -1.38]
	Training 2	0.00016		[2.02, 0.31, -3.9]
64, 32	Training 1	0.00018		[2.07, -0.17, -1.59]
	Training 2	0.00016		[2.74, -0.05, -2.9]

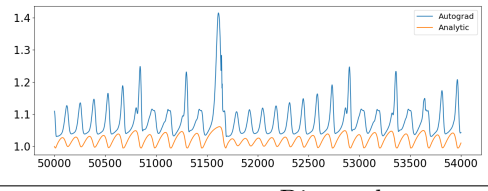
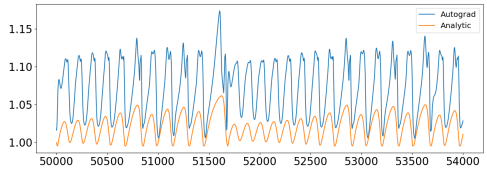
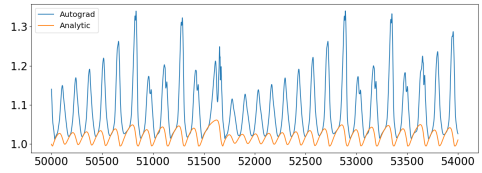
Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
128, 64	Training 1	$8.85e^{-5}$		[2.84, -0.36, -2.84]
	Training 2	0.00011		[1.49, -0.04, -3.6]
8, 16, 8	Training 1		Diverged	
	Training 2	0.05		[6.17, 0.65, -8.69]
128, 256, 128	Training 1	0.00012		[1.75, -0.09, -1.55]
	Training 2	0.00014		[2.42, 0.02, -3.47]

Continued on next page

Table D.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
8, 16, 16, 8	Training 1	0.007		[5.65, -0.38, -3.09]
	Training 2	Diverged		
256, 1024, 1024, 256	Training 1	0.004		[2.17, -0.56, -4.54]
	Training 2	0.00026		[1.52, -0.48, -2.97]

APPENDIX E

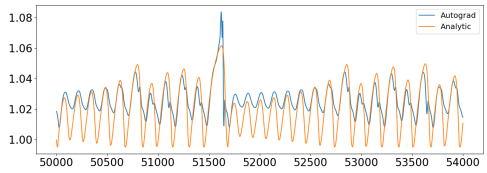
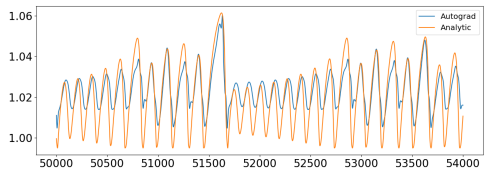
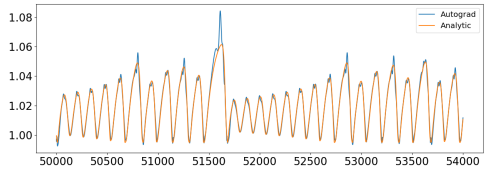
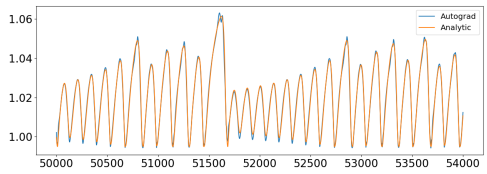
MGD training

Test on different architectures, two layers, but different number of neurons per layer, optimized with multi-gradient descent on :

- Loss on the distance between trajectories.
- Loss on the distance between jacobians.

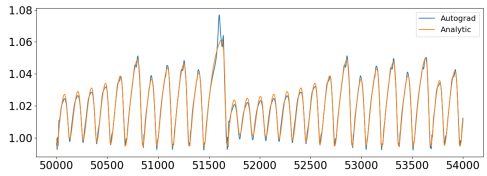
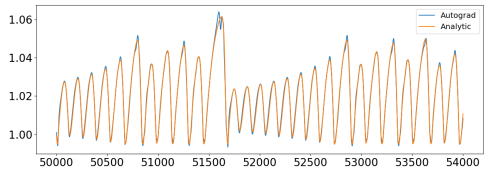
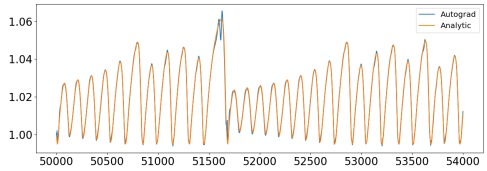
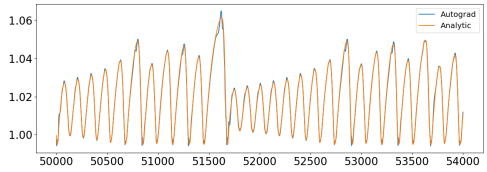
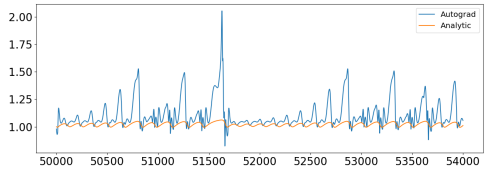
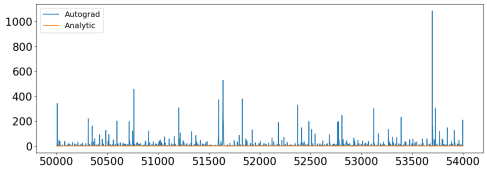
Data is created by using Runge-Kutta, with a delta t of $5e^{-3}$, used as is by the model.
Also one model where activation function is tanh instead of swish.

Table E.1: Results per trained model, loss is the loss on trajectory

Model		Loss	Singular Values	Lyapunov exponents
8, 8	Training 1	0.09		[1.17, -3.39, -11.67]
	Training 2	0.15		[1.53, -0.58, -14.81]
32, 16	Training 1	Diverged		
	Training 2	0.00044		[0.89, -0.12, -14.51]
64, 16	Training 1	0.0015		[1.04, -0.19, -14.52]
	Training 2	Diverged		

Continued on next page

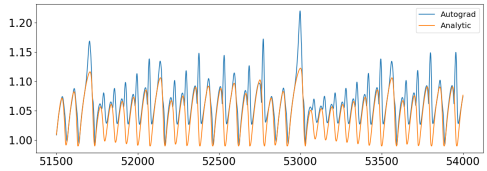
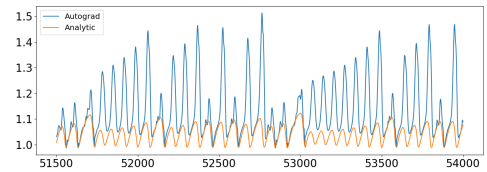
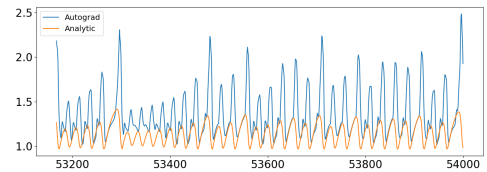
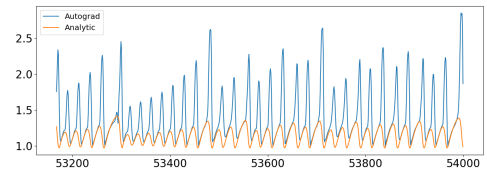
Table E.1 – *Continued from previous page*

Model		Loss	Singular values	Lyapunov exponents
64, 32	Training 1	0.00014		[1.16, -0.22, -14.48]
	Training 2	$6.99e^{-5}$		[0.96, -0.07, -14.54]
128, 64	Training 1	0.00015		[0.91, -0.04, -14.54]
	Training 2	0.0003		[1.02, -0.14, -14.55]
64, 16, tanh activation	Training 1	11.04		[30.5, -50.86, -121.3]
	Training 2	3.76		[-73.6, -1094.2, -2399.2]

Sampling

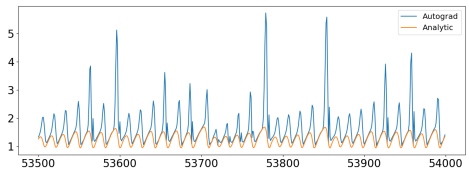
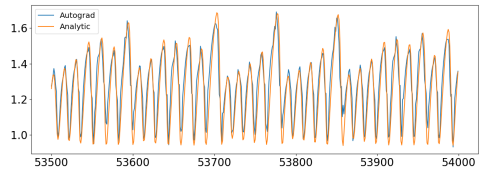
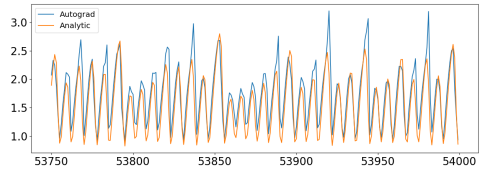
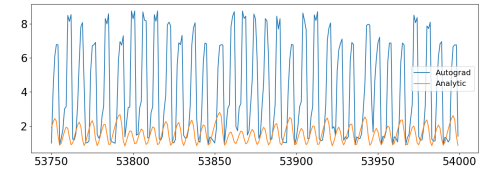
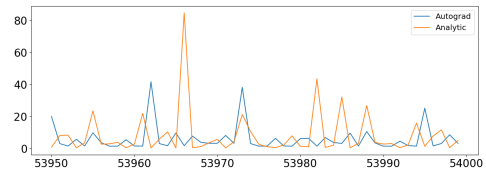
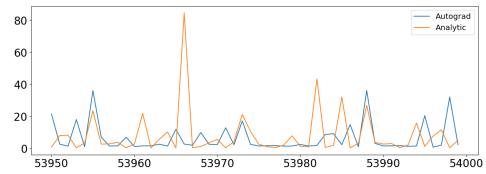
The same model is used every time, a simple NN with two layers of 16 and 8 (which isn't the best one).
Data was created using Runge-Kutta with a given Δt, but data used can be only one every n times. In any case, there are 54000 points in the training data.

Table F.1: Results per sampling period

Model		Loss	Singular Values	Lyapunov exponents
$\Delta t = 1e^{-2}$				
1	Training 1	0.001		[1.12, -0.19, -11.8]
	Training 2	0.0015		[0.99, -0.1, -11.45]
3	Training 1	0.009		[0.773, 0.09, -10.93]
	Training 2	0.0032		[0.71, 0.11, -8.18]

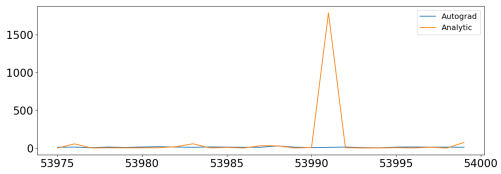
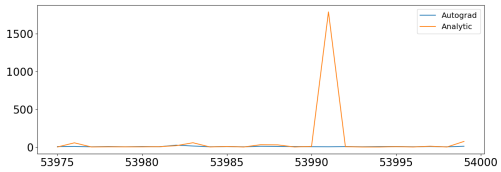
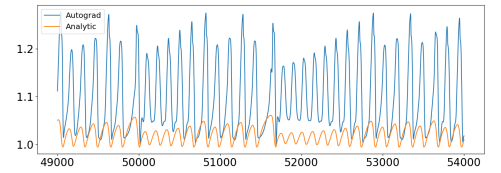
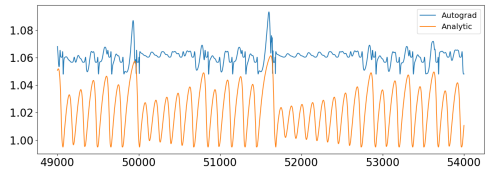
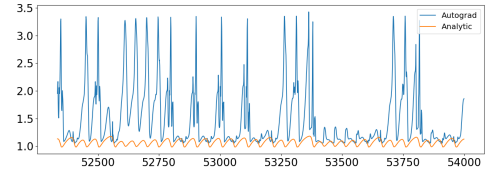
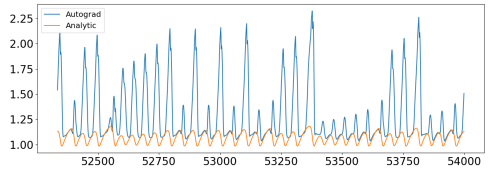
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
5	Training 1	0.0093		[0.97, -0.07, -13.91]
	Training 2	0.037		[1.08, -0.03, -12.93]
10	Training 1	0.026		[0.94, -0.04, -11.53]
	Training 2	2.48		[1.09, -2.27, -11.3]
50	Training 1	25.35		[0.67, -1.05, -8.21]
	Training 2	23.99		[0.78, inf, inf]

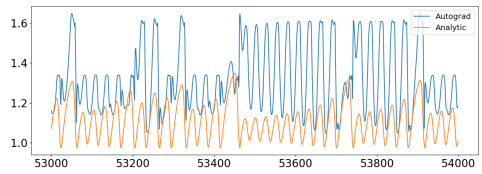
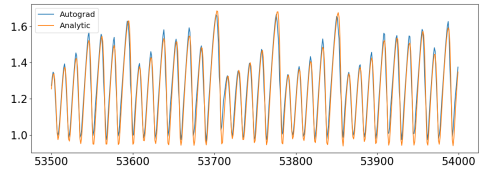
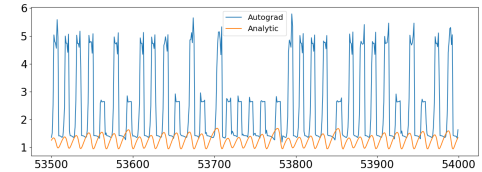
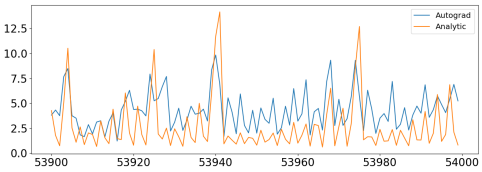
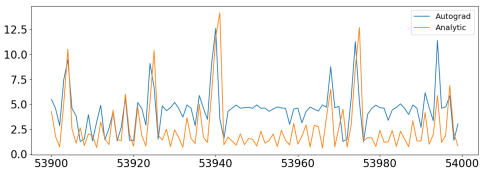
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
100	Training 1	27.33		[1.14, -0.53, -1.89]
	Training 2	33.95		[0.67, -6.15, inf]
$\Delta t = 5e^{-3}$				
1	Training 1	0.0012		[0.83, -0.28, -4.27]
	Training 2	0.019		[0.6, 0.2, -2.81]
3	Training 1	0.15		[12.19, 1.66, -14.92]
	Training 2	0.008		[3.07, -0.62, -6.11]

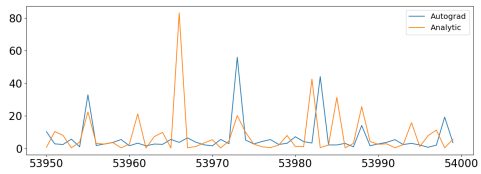
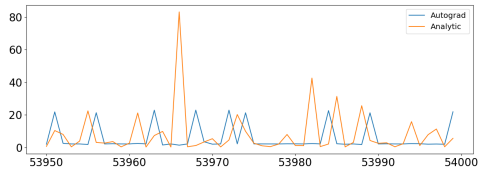
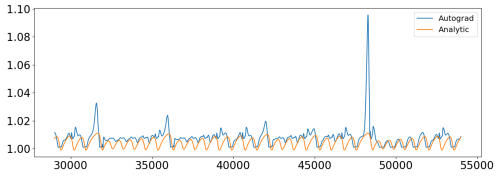
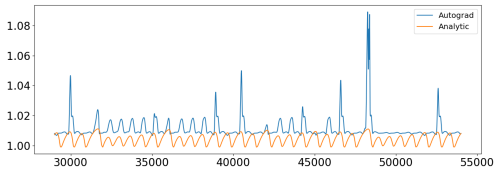
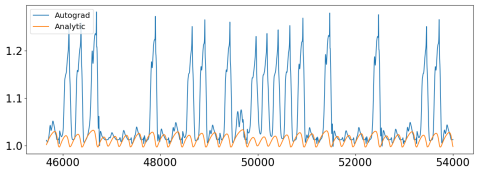
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
5	Training 1	0.38		[0.91, -0.34, -3.37]
	Training 2	Diverged		
10	Training 1	0.013		[1.24, -0.2, -15.99]
	Training 2	1.24		[1.7, 0.39, -29.53]
50	Training 1	1.03		[1.28, -0.82,-6.82]
	Training 2	1.91		[2.3, -1.91, -10.83]

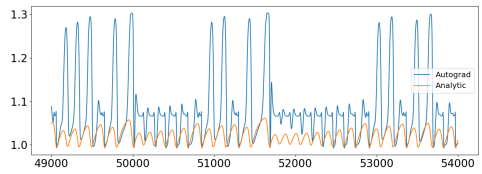
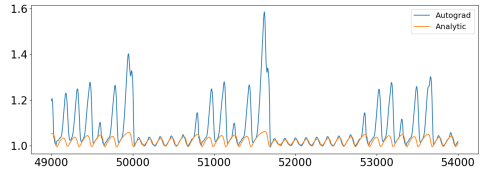
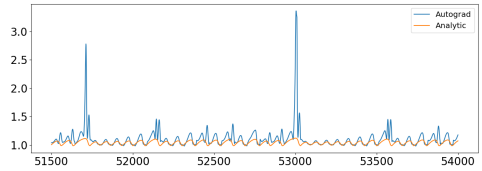
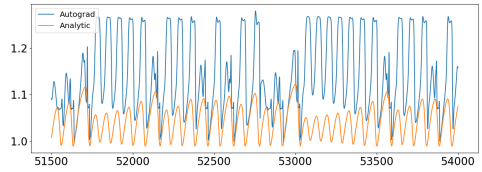
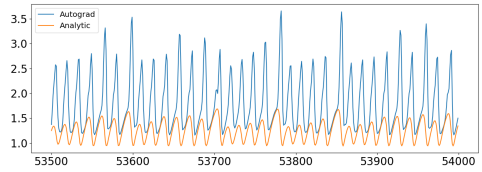
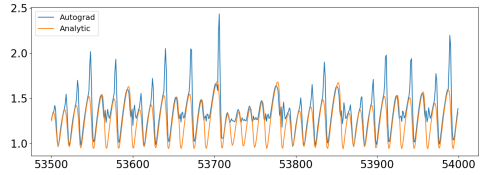
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
100	Training 1	20.58		[0.5, inf, inf]
	Training 2	56.11		[-1.24, -2.76, -8.01]
$\Delta t = 1e^{-3}$				
1	Training 1	0.00025		[2.48, -0.18, -9.93]
	Training 2	0.00047		[1.7, -1.53, -9.46]
3	Training 1	0.001		[4.59, 0.27, -5.11]
	Training 2	Diverged		

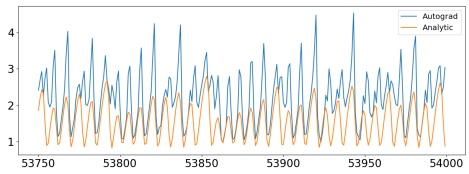
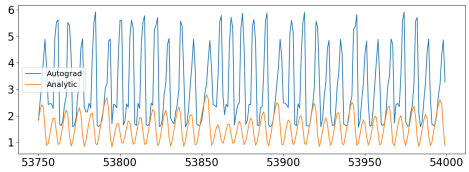
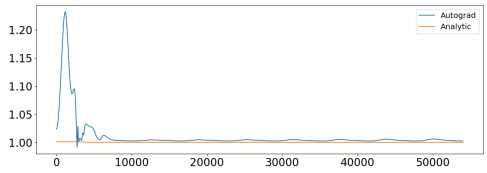
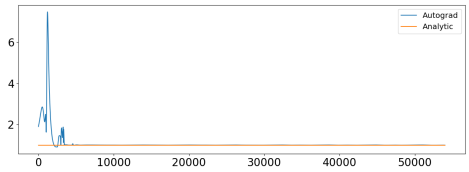
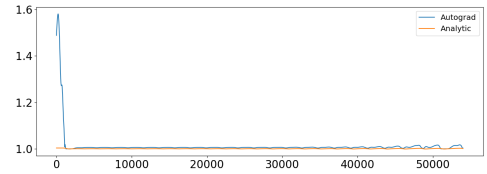
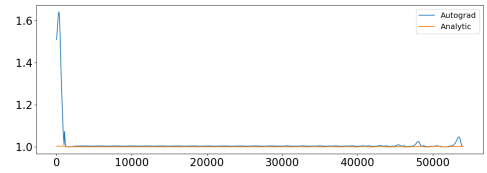
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
5	Training 1	0.0093		[2.3, -0.73, -3.2]
	Training 2	0.037		[3.85, 0.1, -5.21]
10	Training 1	0.0047		[2.31, -0.73, -3.2]
	Training 2	0.00066		[3.85, 0.1, -5.21]
50	Training 1	0.014		[1.93, -0.24, -19.95]
	Training 2	0.033		[2.6, 0.41, -5.77]

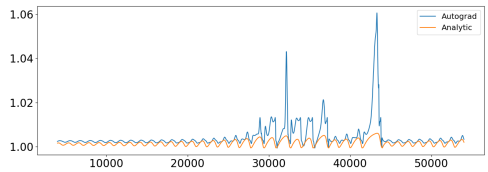
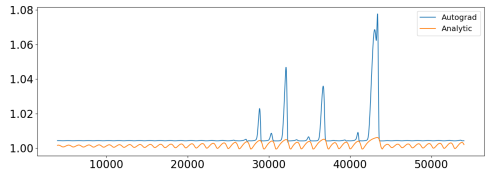
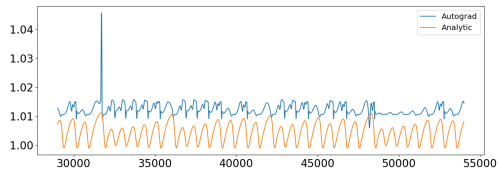
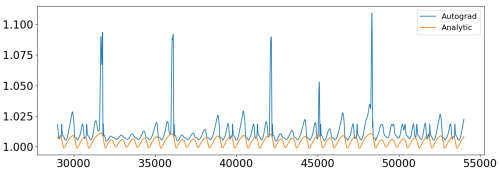
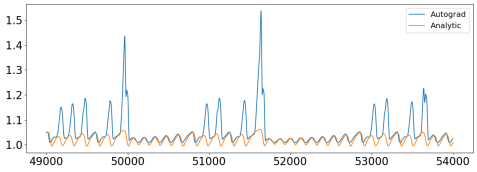
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
100	Training 1	0.10		[0.57, -0.21, -6.18]
	Training 2	1.07		[2.11, -2.6, -9.15]
$\Delta t = 1e^{-4}$				
1	Training 1	$6.82e^{-5}$		[16.8, -141.95, -242.59]
	Training 2	0.007		[45.46, -183.99, -803.22]
3	Training 1	0.00012		[9.56, -2.38, -5.56]
	Training 2	0.00012		[9.98, -10.2, -18.61]

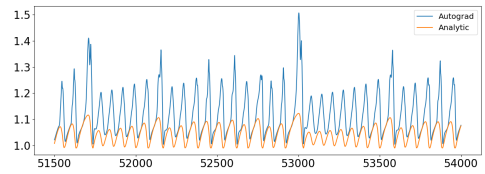
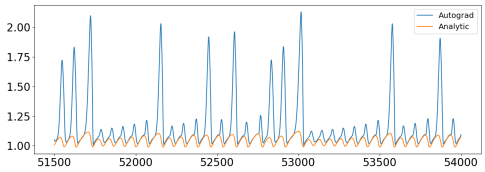
Continued on next page

Table F.1 – *Continued from previous page*

Model		Loss	Singular Values	Lyapunov exponents
5	Training 1	0.00012		[3.43, 0.62, -8.99]
	Training 2	0.00045		[5.61, 0.91, -3.69]
10	Training 1	0.00018		[3.43, -1.05, -3.86]
	Training 2	0.00037		[2.20, -0.17, -4.75]
50	Training 1	Diverged		
	Training 2	0.00014		[1.84, -0.42, -6.33]

Continued on next page

Table F.1 – *Continued from previous page*

Model		**Loss**	**Singular Values**	**Lyapunov exponents**
100	Training 1	0.00019		[1.12, 0.01, -8.34]
	Training 2	0.00015		[1.7, 0.59, -6.04]