Deep Learning In Practice: Homework 5

Mahdi Kallel, Noureddine Sedki, Raphael Reme

March 16, 2021

1 Introduction

The goal of this assignment is to learn to predict correctly trajectories of Rössler systems. In such a system, each trajectory $\left\{W_t = (x_t, y_t, z_t)^T\right\}_{t\geq 0}$ follows a simple temporal ODE:

$$\dot{W}_t = f(W_t) = \begin{cases} -y_t - z_t \\ x_t + a y_t \\ b + z_t(x_t - c) \end{cases}$$

With a, b, c some parameters. In this experiment, we will focus on the predefined value $a = 0.2$, $b = 0.2$, and $c = 5.7$.

The trajectories of such system are deterministic but chaotic: A minor deviation leads to a very different trajectory. Our goal is to create a model that is able to predict the trajectory given a starting point.

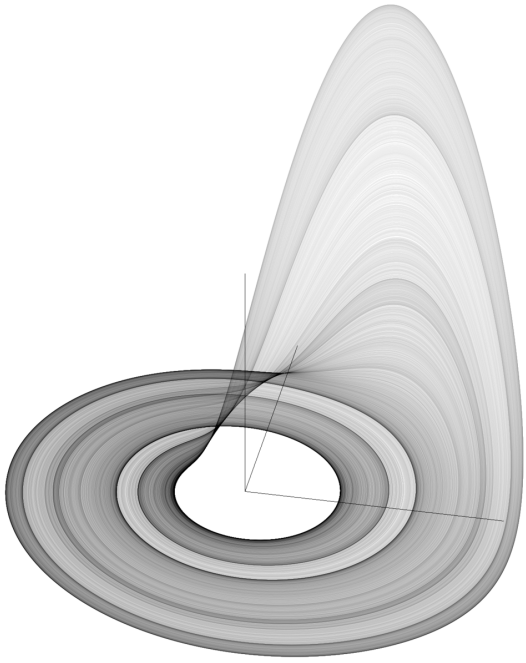

Figure 1: One typical trajectory
(https://en.wikipedia.org/wiki/Rossler_attractor)

2 Chosen Approach

We were proposed several approaches to solve this problem:

1. Continuous system: Use a Neural Network to predict directly the derivative ($\dot{W}_t = \mathrm{NN}(W_t)$).

2. Discrete system without memory: Use a Neural Network to predict the next state ($W_{t+dt} = \text{NN}(W_t)$).

3. Discrete system with memory: Use a Recurrent Neural Network to predict the next state ($W_{t+dt}, H_{t_+dt} = \text{RNN}(W_t, H_t)$).

We have chosen to go with the second approach as it is straightforward to set it up. Indeed we do not need to compute the true $\dot{W}_t$ to regress it, nor compute W_{t+dt} from this regression as it would have been needed for the first approach. Moreover we can use directly the whole state (no embedding) and the jacobian (that is needed later) is easy to compute, which is not the case for the third approach.

2.1 Architecture

With this approach, a simple architecture should be enough. We have decided to use a simple feed-forward architecture with only linear layers and ReLU activation. We found that having at least a width of 100 helped the training to converge quickly, and we used only 4 layers to prevent exploding gradient.

Finally the architecture that we used is a feed-forward network composed of 4 layers:

1. Linear layer of size (3 x 50) + ReLU

2. Linear layer of size (50 x 100) + ReLU

3. Linear layer of size (100 x 200) + ReLU

4. Linear layer of size (200 x 3)

2.2 Dataset

In order to learn correctly the dynamics of the system we have to generate a dataset that represent well the problem. In this state of mind, for the training dataset we have generated several trajectories from different initial points of duration 200s (Enough to perform several circles). We have chosen the initial points either randomly inside the typical distribution of one long trajectory, or near the equilibrium point. (Without it the space near the equilibrium point is unknown for network as no trajectory starting far from it will converge toward it.)

We have used a Δ_t of 0.1s as smaller ones (0.01 and 0.001) did not yield good trajectories. With this large Δ_t we are able to handle longer trajectories.

At training time we have found that using a single trajectory in each batch was more efficient.

Finally we have also used a validation set with different initialization points (sampled as described above) and keep the best model with respect to a Mean Square Error on this validation set between the predicted trajectory and the true trajectory.

2.3 Loss

We focus on a simple MSE loss. But we tried to improve training with more complexe predictions: Instead of only predicting the next sample we tried to predict the N-next samples for each batch. The idea is to let the network learn the real autoregressive task on n-sample and not just focus on the next one. Given W_t we compute $\hat{W}_{t+idt} = \text{NN}^i(W_t)$. And the loss is simply the MSE over each computed $\hat{W}_{t+idt}$.

This approach is not stable for large value of N (As we stack the same network several times, we are facing exploding gradient issues). It gives good results with $N \leq 10$, nonetheless it didn't not really improve the results of $N = 1$.

3 Results analysis

We would like to compute some quantities in order to be sure that our model is able to generate good trajectories. In the training we have only check a MSE criterion. Here we will try to estimate the global quality of the trajectory with other features.

3.1 PDF

The idea here is to check that the distribution over each axis is the same for the predicted and true trajectory. The figure 2 shows clearly that the predicted trajectory has the same distribution than the true one.

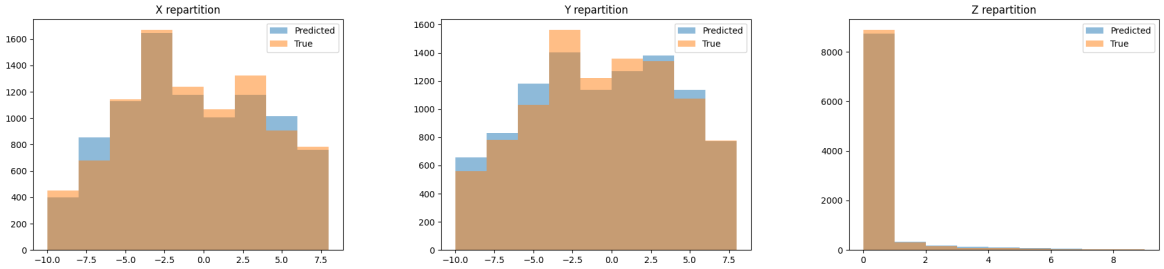

Figure 2: Distribution over each axis

3.2 Time correlation

We have computed the autocorrelation function for each trajectory (only with y_t as it is a sufficient embedding for the whole trajectory). As one can see they are very similar, which indicates that they have similar pseudo periods.

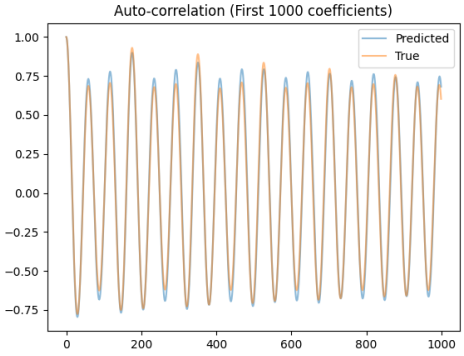

Figure 3: Autocorrelation function for lags betwen 0 and 100s

3.3 Fourier transform

We have compared the main frequencies for each trajectory. They are also very similar which is not a suprise, as we already showed some time correlation.

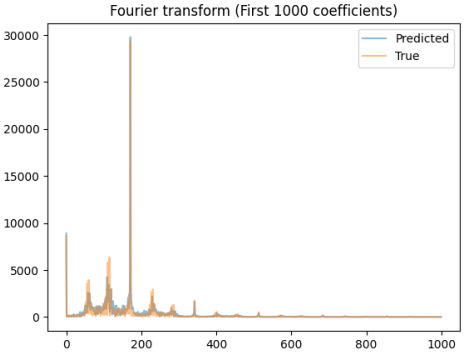

Figure 4: Main fourier coefficients

3.4 Equilibrium point

We tried to find the equilibrium point of our network, and compare it with the real equilibrium point of the system. We have found that the equilibrium point of our network $(0.009, -0.036, 0.034)$ is very close of the true equilibrium point $(0.007, -0.035, 0.035)$.
Nonetheless as there are very litle movements near the equilibrium point, our network has learnt a stable (or more stable) equilibrium. Given a point near the equilibrium the trajectory will stay close to the equilibrium. (whereas the true trajectory diverts slowly).

3.5 Lyapunov exponents

We have finally computed the first Lyapunov exponent of the predicted trajectory that should be around $7e-2$ as for the true trajectory.

We have found a Lyapunov exponent of $6.4e-2$.

4 Reproducing results

In order to reproduce the training of the model, run the command:

```
$ python train.py
```

If you have a trained model, you can run

```
$ python TP.py
```

This command will output all the figures and analysis required.
These commands accept a list of options, but the default one are those that we finally used (use -h to see them)

5 Conclusion

We are able to generate quite good trajectories with a simple neural network that predicts the next point with a time delta of 0.1s. Even though errors stack at each new prediction, the global quality of the trajectory is conserved but it struggles to fit the extreme points.

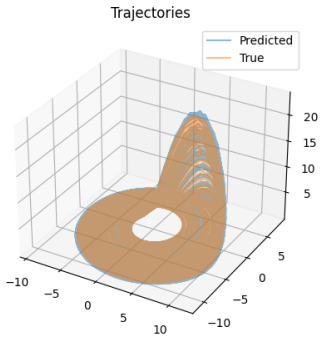

Figure 5: An example of 1000 seconds