Learning chaotic dynamics with RNNs*

Daniel Haas and Keran Chen
University of Oslo, Department of Physics
(Dated: June 9, 2023)

In this study, we delve into the challenge of predicting the dynamics of differential equations by leveraging the power of recurrent neural networks (RNNs). Our primary objective is to investigate how to employ different types of RNNs in dynamic systems while also assessing their performance in two distinct systems - one stable and one chaotic. We show that regular feed-forward neural networks (FFNNs) have difficulty in capturing the time dependency of the temporal data. We then show that RNNs can be used to get results from this time-series analysis, essentially by learning the dynamics of systems. For that, we train the network in different trajectories of Lorenz attractor simulations with different initial conditions. We experiment with some hyperparameter tuning and show that using LSTM can yield better results than the standard RNN. We further motivate the introduction of physically informed loss functions to embed physical constraints to the neural network. Illustrating the difficulty of predicting chaotic dynamics, we also benchmark our methods by predicting stable spiral trajectories.

I. INTRODUCTION

In 1963, the meteorologist and mathematician E. N. Lorenz unveiled an extraordinary system. By significantly reducing the complexity of a model for weather prediction, he came across a three-dimensional differential equation that demonstrated a phenomenon known as "deterministic chaos".

Upon employing certain constants, an extensive collection of solutions gravitate towards a set resembling the shape of a butterfly - hence dubbed the 'Lorenz Attractor'. This particular type of behavior is categorized as "chaos" [1].

The difficulty created by this set among many other sets of coupled differential equations is that they have no analytical solutions. Recently, advancements in machine learning and, more specifically, recurrent neural networks (RNNs), have provided us with a new way through which we can investigate this dynamic system.

In this report, we studied different ways to predict a time sequence with various complexities. Using time sequences generated by standard fourth-order Runge-Kutta (RK4) methods, we started by attempting to predict future steps using a standard Feed Forward Neural Network, then a simple RNN, and thereafter, LSTM RNNs with and without an added physics-informed loss function. We then compared their behaviors and explained why LSTM RNNs are best suited for studying this time series. Lastly, we also explored a less chaotic system, namely the stable spiral. The results proved to be very interesting.

The goal of this report is to provide our findings on how RNNs predict time series and chaotic systems and explain the difficulties and potential solutions, in an attempt to generate new insights.

In section II, we present an overview of RNNs and their basic ingredients, such as recurrent connections, hidden states, activation functions, and backpropagation through time. Subsequently, we present the physical model of Lorenz attractors to explain how we generated the data set via simulations and discuss how chaotic systems are hard to learn for an ML algorithm. We start section III, by showing that a regular FFNN is unable to learn the proper dynamics of the chaotic motion for a satisfactory trajectory extrapolation.

Section IV analyses the obtained results, comparing the implemented methods and their ability to extract data from the model, noting possible future improvements. In Section V, we conclude with a summary of what was learned from the several results and methods.

II. METHODS

A. Data Set: a physics overview

The field of dynamic systems encompasses a broad discipline, and a discussion of such can be approached both with a physics or a purely mathematical scope. A general but simple description is that it is the study of how a system evolves over time. For our purposes and investigations, we shall focus on the mathematical aspects of it, as we are interested in a dynamical system as a solution for a differential equation, which we face as a time-series evolution. Nonetheless, we will make use of physics tools in order to devise better-predicting algorithms.

A dynamical system can be categorized based on how it develops over time, but also how this behavior is affected by initial conditions and other rules. Two

* https://github.com/Daniel-Haas-B/AdvancedMachineLearning

canonical examples of dynamical systems which exhibit completely different time evolution properties are the Lorenz attractor and a stable spiral.

1. *Stable Spiral*

The dynamics of a stable spiral evolve in such a way that the system's trajectory converges to a fixed point while spiraling inward. These oscillations around the fixed point are gradually dampened until the system reaches a steady state at a fixed point. Suppose we have a two-dimensional system of coupled differential equations of the form

$$\begin{aligned} \frac{dx}{dt} &= ax + by, \\ \frac{dy}{dt} &= cx + dy. \end{aligned} \tag{1}$$

The choice of $a, b, c, d \in \mathbb{R}$ completely determines the behavior of the solution, and for some of these values, albeit not all, the system is said to be a stable spiral. This condition is satisfied when the eigenvalues of the matrix formed by the coefficients are complex conjugates with a negative real part [2].

For illustration, Fig. (1) depicts the simulation of a stable spiral trajectory.

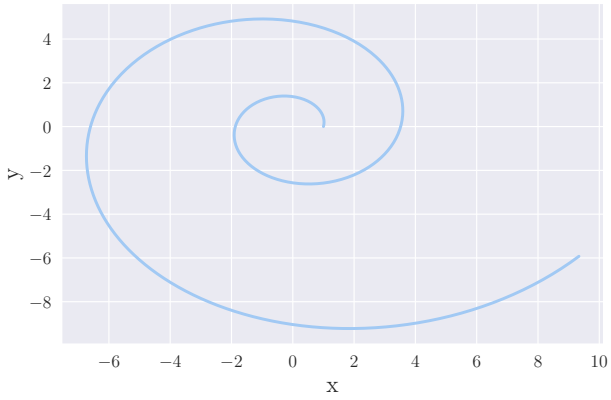

Figure 1: Trajectory of a simple stable spiral with coefficients $a = 0.2$, $b = -1.0$, $c = 1.0$, $d = 0.2$.

2. *Lorenz attractor*

In contrast to the previous example, a Lorenz attractor presents some added complexity. Firstly, it is by definition a three-dimensional system, but more importantly, it exhibits what is called chaotic behavior [3]. While there is no universal consensus on the definition of a chaotic system, all definitions agree chaotic systems are extremely sensitive to initial conditions. This means an arbitrary perturbation of such a set of conditions - or the current trajectory- will lead to vastly distinct asymptotic behavior.

The expression for the Lorenz attractor evolution consists of a set of three coupled nonlinear differential equations given by

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z. \end{aligned} \tag{2}$$

For this problem, x, y, z are the variables that determine the state of the system in the space while σ, ρ and β are, similarly to the constants a, b, c, d of the stable spiral, parameters that influence largely how the system evolves. For illustration, Fig. (2) depicts two particular Lorenz attractor trajectories.

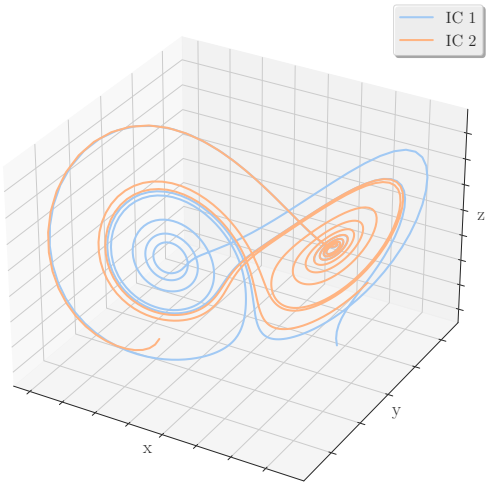

Figure 2: Simulation of two Lorenz attractor trajectories. While both are determined by Eq. (2), they had two different initial conditions, hence the legend IC 1 and IC 2.

3. *Generating the data*

Both of the above-mentioned systems are governed by differential equations, and as such, they can be solved numerically through some integration scheme such as forward-Euler or fourth-order Runge-Kutta. The latter was the algorithm of choice for the generated data sets.

For our attractor, we will use the common choice of parameters $\sigma = 10$, $\rho = 28$, $\beta = 8/3$. Although apparently arbitrary, this is a natural choice. Not only was it used by Lorenz himself [4], but it generates complex and aesthetic trajectories that have been extensively investigated and benchmarked in the literature of numerical simulations. For the stable spiral, we used $a = 0.2$, $b = -1.0$, $c = 1.0$, $d = 0.2$ as it showed a good number of oscillations before reaching a steady state for our simulation time.

B. Feed-forward networks

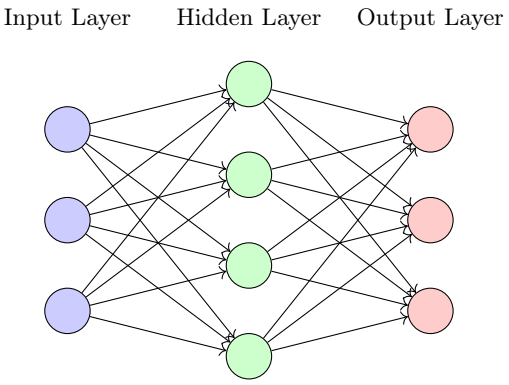

Figure 3: Basic schematic of a feed-forward neural network showing its layers and units.

In its general definition, a feed-forward neural network is a way to organize a set of function compositions in a sequential way. The algorithmic way of doing so is referred to as a network, given that it can be understood as being composed of individual but somehow connected units in a graph-like manner.

The network is said to be of feed-forward type if information flows in one preferential direction in the network. This means each neuron receives inputs from the previous layer, and after multiplying it by a set of weights and adding some bias - and often composing it with some activation function - passes the result to neurons in the next layer. The way in which the units are organized in layers and how they are connected results in different architectures.

The choice of the architecture of a network is often guided by the problem one aims at tackling, but most networks have some overlapping core structure in the form of input layers, hidden layers, output layers, and activation functions, as can be seen in Fig. (3). The network's weights and biases are then iteratively updated via a gradient scheme that tries to converge to a global minimum of a cost-function. This cost function is a measurement of well the model performs, be it in regression, or classification problems.

$$\begin{aligned} &\mathbf{h}_0 = \mathbf{x} \\ &\left.\begin{aligned} &\mathbf{a}_i = W_i * \mathbf{h}_{i-1} + \mathbf{b}_i \\ &\mathbf{h}_i = \sigma_i(\mathbf{a}_i) \end{aligned}\right\} \qquad i \in \{1, ..., L-1\}, \\ &\hat{\mathbf{y}} = \sigma_L(W_y * \mathbf{h}_{L-1} + \mathbf{b}_y) \end{aligned} \tag{3}$$

where W_i and b_i are respectively the weight matrix and bias vector of the i-th layer, h_i represents the output of the i-th hidden layer and we are defining the 0-th hidden layer as the input of the network, $\mathbf{x}$. Similarly, σ_i represents the activation function of each hidden layer and L is the output layer.

There are limitation of FFNNs, one of which being that FFNNs are not designed to handle sequential data (data for which the order matters) effectively because they lack the capabilities of storing information about previous inputs; each input is being treated independently. This is a limitation when dealing with sequential data where past information can be vital to correctly process current and future inputs.

C. Recurrent networks

In contrast to FFNs, recurrent networks introduce feedback connections, meaning the information is allowed to be carried to subsequent nodes across different time steps. These cyclic or feedback connections have the objective of providing the network with some kind of memory, making RNNs particularly suited for time-series data, natural language processing, speech recognition, and several other problems for which the order of the data is crucial.

RNN architectures vary greatly in how they manage information flow and memory in the network. Different architectures often aim at improving some sub-optimal characteristics of the network. The simplest form of recurrent network, commonly called simple or vanilla RNN, for example, is known to suffer from the problem of vanishing gradients. This problem arises due to the nature of backpropagation in time. Gradients of the cost/loss function may get exponentially small (or large) if there are many layers in the network, which is the case of RNN when the sequence gets long. Consequently, the convergence to correctly predicted values in the training process happens very slowly. This is particularly the case when using activation functions like the Sigmoid or hyperbolic tangent, (whose outputs are between 0 and 1 for Sigmoid, and -1 and 1 for tanh), with equally small gradients by design (< 1).

To address that, one natural alternative is to use the Long Short-Term Memory (LSTM) variation. While

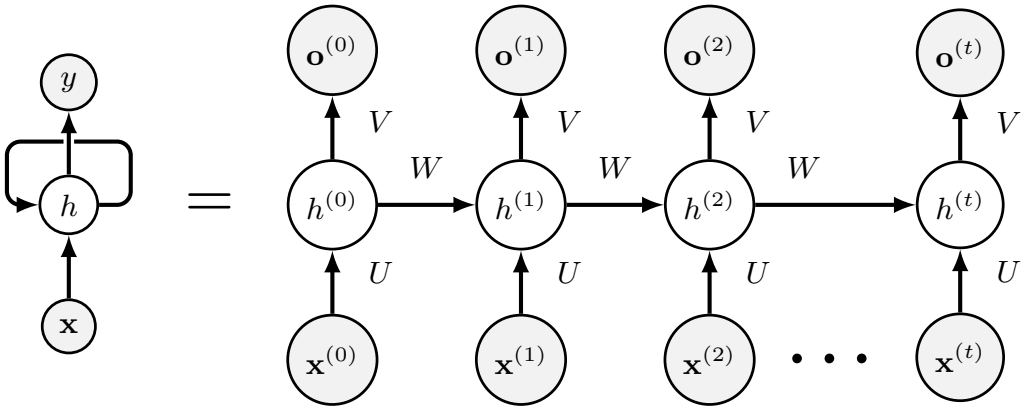

Figure 4: Folded versus unfolded representation of a Vanilla RNN

LSTMs are generally used, they also present the downside of being inefficient for large-scale models. For such problems, Gated recurrent units (GRUs) can be employed as they have a simpler architecture, thus being computationally faster. We will hereafter focus simply on the Vanilla and LSTM variants for their simplicity and popularity.

1. *Vanilla RNN*

The expression for the simplest Recurrent network resembles that of a regular feed-forward neural network of Eq. (3), but now with the concept of temporal dependencies as shown in the equations for the forward pass, Eq. (4),

$$\begin{aligned}
\mathbf{a}^{(t)} &= U * \mathbf{x}^{(t)} + W * \mathbf{h}^{(t-1)} + \mathbf{b}, \\
\mathbf{h}^{(t)} &= \sigma_h(\mathbf{a}^{(t)}), \\
\mathbf{o}^{(t)} &= V * \mathbf{h}^{(t)} + \mathbf{c}, \\
\hat{\mathbf{y}}^{(t)} &= \sigma_y(\mathbf{o}^{(t)}).
\end{aligned} \tag{4}$$

Equation Eq. (4), can be better understood from the schematics of Fig. (4). Here, the superscripts denote the temporal dependency of each pass. As usual, $\mathbf{h}$ represents the output of the hidden layer, after its activation σ_h, while $\mathbf{y}$ and σ_y have the analogous role for the output layer. Similarly to a regular FFN, $\mathbf{b}$ and $\mathbf{c}$ represent the bias of the hidden layers and the output layers respectively. Notice however that we have weight matrices that modulate two inputs to a fixed hidden layer, U and W. Indeed, hidden layers now receive a temporal input of the data $\mathbf{x}^{(t)}$ but also from the output of the same hidden layer in a previous temporal step, $\mathbf{h}^{(t-1)}$. Finally, the output layer's input is also modulated by weights V.

For this breakdown of equation Eq. (4) it is important to note that we are not specifying which layer of the network we are propagating, to avoid carrying indices. As can be seen from Fig. (4), RNNs are often depicted in both a folded and an unfolded visualization, but it is important to note that a unit $h^{(t)}$ can be composed of many layers.

The learning procedure for RNNs also follows the same idea as that of regular FFNs: we need to optimize weights and bias so that the loss function $\mathcal{L}$ which assesses how well the model performs, gets minimized. To construct $\mathcal{L}$, it is paramount it takes into account all the temporal outputs, $\mathbf{y}^{(t)}$, and sum over their respective losses,

$$\mathcal{L} = \sum_t L^{(t)}. \tag{5}$$

The choice of the individual losses $L^{(t)}$ is heavily dependent on the problem at hand, and shall be discussed in the following section II D. This temporal dependency of the loss means the usual back-propagation process to evaluate the gradient of the parameters and update them depends on previous time steps due to the chain rule. For that reason, it is called back-propagation through time (BPTT), but the idea behind it is the same as for regular back-propagation: derive the loss function with respect to the parameters.

2. Backpropagation through time

The subsequent derivation follows closely the explanations of [5], however, we shall not assume any loss function for now. To derive the expression of the gradients of $\mathcal{L}$ for the RNN, we need to start recursively from the nodes closer to the output layer in the temporal unrolling scheme - such as $\mathbf{o}$ and $\mathbf{h}$ at final time $t = \tau$,

$$\begin{aligned}(\nabla_{\mathbf{o}^{(t)}}\mathcal{L})_i &= \frac{\partial \mathcal{L}}{\partial L^{(t)}}\frac{\partial L^{(t)}}{\partial o_i^{(t)}},\\ \nabla_{\mathbf{h}^{(\tau)}}\mathcal{L} &= \mathbf{V}^\mathsf{T}\nabla_{\mathbf{o}^{(\tau)}}\mathcal{L}. \end{aligned} \tag{6}$$

For the following hidden nodes, we have to iterate through time, so by the chain rule,

$$\nabla_{\mathbf{h}^{(t)}}\mathcal{L} = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}}\right)^\mathsf{T}\nabla_{\mathbf{h}^{(t+1)}}\mathcal{L} + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}}\right)^\mathsf{T}\nabla_{\mathbf{o}^{(t)}}\mathcal{L}. \tag{7}$$

Similarly, the gradients of $\mathcal{L}$ with respect to the weights and biases follow,

$$\begin{aligned}\nabla_{\mathbf{c}}\mathcal{L} &= \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}}\right)^\mathsf{T}\nabla_{\mathbf{o}^{(t)}}\mathcal{L}\\ \nabla_{\mathbf{b}}\mathcal{L} &= \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}}\right)^\mathsf{T}\nabla_{\mathbf{h}^{(t)}}\mathcal{L}\\ \nabla_{\mathbf{V}}\mathcal{L} &= \sum_t\sum_i \left(\frac{\partial \mathcal{L}}{\partial o_i^{(t)}}\right)\nabla_{\mathbf{V}^{(t)}}o_i^{(t)}\\ \nabla_{\mathbf{W}}\mathcal{L} &= \sum_t\sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right)\nabla_{\mathbf{w}^{(t)}}h_i^{(t)}\\ \nabla_{\mathbf{U}}\mathcal{L} &= \sum_t\sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right)\nabla_{\mathbf{U}^{(t)}}h_i^{(t)}. \end{aligned} \tag{8}$$

The pseudo-code in 1 illustrates algorithmically how information and the gradients flow in a vanilla RNN based on the previous equations of Eq. (6), Eq. (7) and Eq. (8). Note that it already assumes the bias and weights to be initialized - for such, it is usual to use normal random distribution $\mathcal{N}(0, \sigma)$, where σ is small to prevent exploding gradients.

3. LSTM

The Long Short Term Memory RNN is a type of RNN designed to avoid the vanishing or exploding gradient

Algorithm 1 Single Vanilla RNN epoch

$\boldsymbol{\delta_h}, \boldsymbol{\delta_W}, \boldsymbol{\delta_V}, \boldsymbol{\delta_b}, \boldsymbol{\delta_c} = \mathbf{0}$ ▷ Initialize gradient accumulation states
for each time step t in the sequence **do**
  $\mathbf{o}^{(t)}$, $\mathbf{h}^{(t)} \leftarrow$ **Forward pass**
$\mathcal{L} \leftarrow$ **Compute loss**
Compute gradient of output layer:
$(\nabla_{\mathbf{o}^{(t)}}\mathcal{L})_i$ ▷ Based on specific loss
Compute gradient of hidden layer at $t = \tau$**:**
$\boldsymbol{\delta_h} \leftarrow \nabla_{\mathbf{h}^{(\tau)}}\mathcal{L}$
Backpropagate through time:
for each time step $t = \tau - 1$ to $t = 1$ **do**
  Compute and update hidden state gradient:
  $\boldsymbol{\delta_h} = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}}\right)^\mathsf{T}\boldsymbol{\delta_h} + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}}\right)^\mathsf{T}\nabla_{\mathbf{o}^{(t)}}\mathcal{L}$
  Acumulate parameter gradients:
  $\boldsymbol{\delta_c} += \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}}\right)^\mathsf{T}\nabla_{\mathbf{o}^{(t)}}\mathcal{L}$
  $\boldsymbol{\delta_b} += \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}}\right)^\mathsf{T}\nabla_{\mathbf{h}^{(t)}}\mathcal{L}$
  $\boldsymbol{\delta_V} += \sum_i \left(\frac{\partial \mathcal{L}}{\partial o_i^{(t)}}\right)\nabla_{\mathbf{V}^{(t)}}o_i^{(t)}$
  $\boldsymbol{\delta_W} += \sum_i (\delta_{\mathbf{h}})_i \nabla_{\mathbf{W}^{(t)}}h_i^{(t)}$
  $\boldsymbol{\delta_U} += \sum_i (\delta_{\mathbf{h}})_i \nabla_{\mathbf{U}^{(t)}}h_i^{(t)}$
Weight and bias update:
$\{W, U, V, \mathbf{b}, \mathbf{c}\} -= \alpha \cdot \{\boldsymbol{\delta_W}, \boldsymbol{\delta_U}, \boldsymbol{\delta_V}, \boldsymbol{\delta_b}, \boldsymbol{\delta_c}\}$

Pseudo code for the forward and backward propagations of a simple RNN. Equations for the gradient expressions follow Eq. (6), Eq. (7), Eq. (8)

caused easily by the Vanilla RNN, which makes it difficult to learn long-range dependencies in a sequence. Given the recurrent relation in equation Eq. (7), the long-term gradient can end up having many factors of $\partial \mathbf{h}^{(t+1)}/\partial \mathbf{h}^{(t)}$, which may be close to zero or excessively large, leading to vanishing or exploding gradients respectively. In 1997, in order to address the aforementioned problem, Hochreiter and Schmidhuber introduced LSTM which truncates the gradient where it does not incur significant detriment to performance [6].

The LSTM is a unit cell that is made of three gates: the input gate, the forget gate, and the output gate. It also introduces a cell state c, which can be thought of as the long-term memory, and a hidden state h which can be thought of as the short-term memory. As illustrated in Figure 5.
The first stage is called the forget gate, where we combine the input at (say, time t), and the hidden cell state input at $t-1$, passing it through the Sigmoid activation function and then performing an element-wise multiplication, denoted by $\otimes$. It follows

$$\mathbf{f}^{(t)} = \sigma(W_f\mathbf{x}^{(t)} + U_f\mathbf{h}^{(t-1)} + \mathbf{b}_f)$$

where W and U are the weights respectively. This is called the forget gate since the Sigmoid activation function's outputs are very close to 0 if the argument for the function is very negative, and 1 if the argument is very positive. Hence we can control the amount of information we want to take from the long-term memory.

The next stage is the input gate, which consists of both a Sigmoid function (σ_i), which decide what percentage of the input will be stored in the long-term memory, and the $\tanh_i$ function, which decide what is the full memory that can be stored in the long term memory. When these results are calculated and multiplied together, it is added to the cell state or stored in the long-term memory, denoted as $\oplus$.
It follows

$$\mathbf{i}^{(t)} = \sigma_g(W_i\mathbf{x}^{(t)} + U_i\mathbf{h}^{(t-1)} + \mathbf{b}_i),$$

$$\tilde{\mathbf{c}}^{(t)} = \tanh\Big(W_c\mathbf{x}^{(t)} + U_c\mathbf{h}^{(t-1)} + \mathbf{b}_c\Big),$$

again the W and U are the weights.

The forget gate and the input gate together also update the cell state with the following equation,

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)},$$

where $f^{(t)}$ and $i^{(t)}$ are the outputs of the forget gate and the input gate, respectively.

The final stage of the LSTM is the output gate, and its purpose is to update the short-term memory. To achieve this, we take the newly generated long-term memory and process it through a hyperbolic tangent (tanh) function creating a potential new short-term memory. We then multiply this potential memory by the output of the Sigmoid function (σ_o). This multiplication generates the final output as well as the input for the next hidden cell ($h^{\langle t\rangle}$) within the LSTM cell.

It follows,

$$\mathbf{o}^{(t)} = \sigma_g(W_o\mathbf{x}^{(t)} + U_o\mathbf{h}^{(t-1)} + \mathbf{b}_o),$$
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \sigma_h(\mathbf{c}^{(t)}).$$

where $\mathbf{W_o}, \mathbf{U_o}$ are the weights of the output gate and $\mathbf{b_o}$ is the bias of the output gate.

4. Training and testing trajectories

Training and testing procedures in recurrent neural networks follow what is usual for regular FNNs, but some special consideration needs to be taken into account due to the sequential character of the data. Training and testing batches must not be randomly shuffled for it would clearly decorrelate the time-series points and leak future information into present or past points of the model.

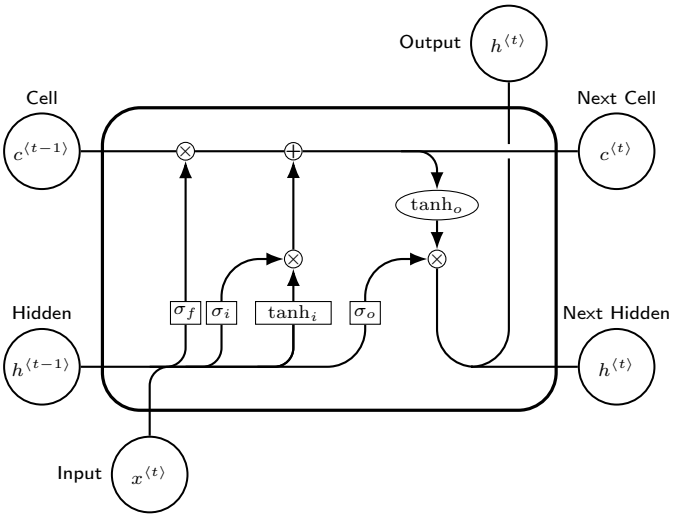

Figure 5: LSTM illustration [a]

[a] source: https://tex.stackexchange.com/questions/432312/how-do-i-draw-an-lstm-cell-in-tikz

Additionally, the training algorithm in Algorithm (1) can become computationally costly, especially if the losses are evaluated for all previous time steps. While other architectures such as that of GRUs can be used to mitigate that, it is also possible to introduce another hyperparameter responsible for controlling how much of the network will be unfolded in the training process, adjusting how much the network will remember from previous points in time. We called this parameter the *look-back*. Similarly, the number of steps the network predicts in the future per iteration greatly influences the assessment of the loss function. we called this parameter the *look-ahead*.

The training and testing batches were separated into whole trajectories. This means that instead of training and testing on different fractions of the same trajectory, all trajectories that were tested had completely new initial conditions. In this sense, from a total of 10 initial conditions (independent trajectories), 9 were used for training and 1 for testing. Each trajectory consisted of 800 points in each space coordinate.

D. On the choice of the loss function

Our problem is essentially a time-series forecasting problem, so, we are free to choose the loss function amongst the big collection of regression losses. Using the mean-squared error of the predicted versus factual

trajectories of the dynamic systems is a natural choice for it is a way to measure their discrepancy in space, and it punished outliers strongly for its quadratic form. Furthermore, it is a convex function, so given sufficient time and appropriate learning rates, it is guaranteed to converge to global minima irrespective of the weight's random initialization. For completeness' sake, the expression for this loss function can be written

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i}^{N}(y(\mathbf{x}_i) - \hat{y}(\mathbf{x}_i, \theta))^2 \tag{9}$$

where we are denoting θ as the set of all parameters of the network, and $\mathbf{x}_i$ is our input.

1. *The "naivety" of data-driven losses*

Simply using a loss function that is based on the observational and predicted data, such as the one in equation Eq. (9) is commonly referred to as a purely data-driven approach. While this is a well-established way of assessing regressions, it does not make use of other intuitions we might have over the problem we are trying to solve. At the same time, it is a well-established fact that neural network models are data-greedy - they need large amounts of data to be able to generalize predictions outside the training set. One way to try to mitigate this is by using physics-informed neural networks (PINNs) when possible.

Trying to improve the performance of our model beyond training sets, PINNs then add physics-informed penalties to the loss function. In essence, this means that we add a worse evaluation score to predictions that do not respect physical laws we think our real data should obey. This procedure often has the advantage of trimming the parameter space without adding bias to the model if the constraints imposed are correct, but the choice of the physical laws can be a delicate one.

2. *Adding physics to the loss function*

A general way of expressing this added penalty to the loss function is shown in Eq. (10) below,

$$\mathcal{L} = w_{MSE}\mathcal{L}_{MSE} + w_{PI}\mathcal{L}_{PI}. \tag{10}$$

Here, the weights w_{MSE} and w_{PI} explicitly mediate how much influence the specific parts of the total loss function should contribute. In our experimentations with the additional physics penalty, we limited results to only have $w_{MSE} = 1.0$ and $w_{PI} = 0.5$.

For one, it is important to note that this procedure is not a universal strategy to predict time-series data. In a general sense, the system need not obey the laws of physics. For instance, the Lorenz attractor, given its attractor behavior, does not conserve momentum.

While we are working with RNNs, the general schematics behind PINNs is illustrated in Fig. (6) using a feed-forward net. The standard approach to construct $\mathcal{L}_{PI}$ is to ensure the outputs of the network satisfies the differential equations at hand. The outputs of the NN are often differentiated via an automatic differentiation software such as JAX [7]. In our case, a natural approach would be

$$\begin{aligned}\mathcal{L}_{PI} = &\frac{d\hat{x}}{dt} - \sigma(\hat{y} - \hat{x}) \\ &+ \frac{d\hat{y}}{dt} - \hat{x}(\rho - \hat{z}) - \hat{y} \\ &+ \frac{d\hat{z}}{dt} - \hat{x}\hat{y} - \beta\hat{z},\end{aligned} \tag{11}$$

where we denoted with hats the neural network outputs. As long as the dynamics of the predictions respect the differential equations, this loss will be minimized. However, since we are working with recurrent neural networks that output one predicted time-step at a time during the training process, taking the derivative in time would not be straightforward. For this reason, we approach the problem of adding physics to the RNN in a similar but subtly different way.

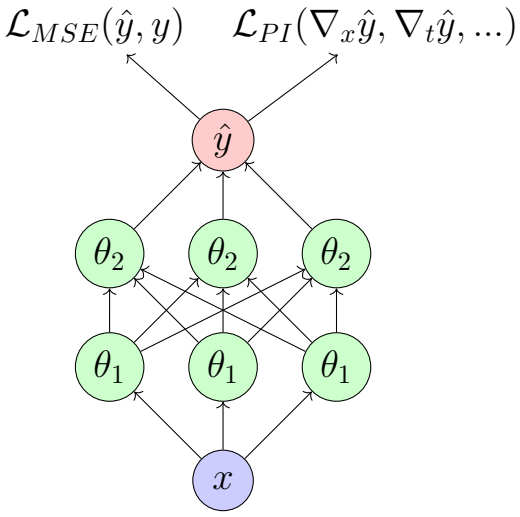

Figure 6: Sketch of the concurrent assessment of both physics-informed and data-driven losses.

As mentioned, the Lorenz attractor does not conserve momentum. Therefore, we cannot simply add a penalty term for when the subsequent temporal values of the

right-hand side (RHS) of the differential equations (DE) (2) differ from the initial one. To still enforce learning of the differential equation we instead construct the RHS of the DE from the positions predictions during training. We then compare those values with the same RHS of the DEs given the true trajectories. More specifically, the added physics loss was given by

$$\begin{aligned}\mathcal{L}_{PI} = & MSE(\sigma(y-x), \sigma(\hat{y}-\hat{x})) \\ & + MSE(x(\rho-z)-y, \hat{x}(\rho-\hat{z})-\hat{y}) \\ & + MSE(xy-\beta z, \hat{x}\hat{y}-\beta\hat{z}) \end{aligned} \tag{12}$$

The physics-informed term in the custom loss function represents the difference between the predicted derivatives of the Lorenz system and the true derivatives, as defined by the Lorenz equations. In other words, it measures how well the predicted trajectories of the system obey the physical laws that govern the Lorenz system.

By incorporating this physics-informed term into the loss function, the LSTM is encouraged to generate predictions that are not only accurate but also physically meaningful. This is important because the Lorenz system is a highly nonlinear and chaotic system, and therefore making predictions, especially long ones, is a hard task.

III. RESULTS

1. *Regular FFNN*

We start by investigating how well a regular feed-forward neural network is able to predict the trajectories of the Lorenz attractor. By training on 9 trajectories with different initial conditions and testing on one, we generate the graph of Fig. (7). This figure was obtained with a three-layer network, where the number of nodes in each was, respectively, 64, 32, and 3.

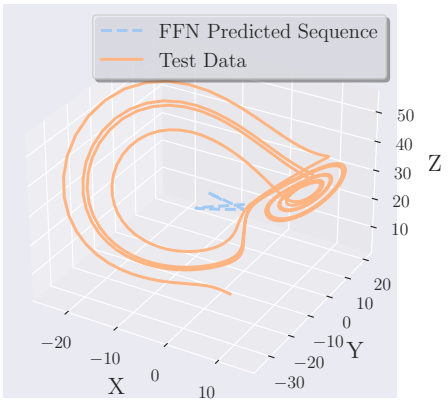

Figure 7: Predicted versus test trajectories for the FFNN on the Lorenz attractor.

For the loss function in this implementation, we used the mean absolute error and an Adam optimizer with a learning rate of 10^{-3}. The training process was of 200 epochs, and Fig. (8) shows how the validation loss evolves in comparison to the training set loss throughout the training. The metric used for training this network was the mean absolute error, but the y-axis is shown as the mean absolute error (MAE) divided by the average of the trajectory's coordinates.

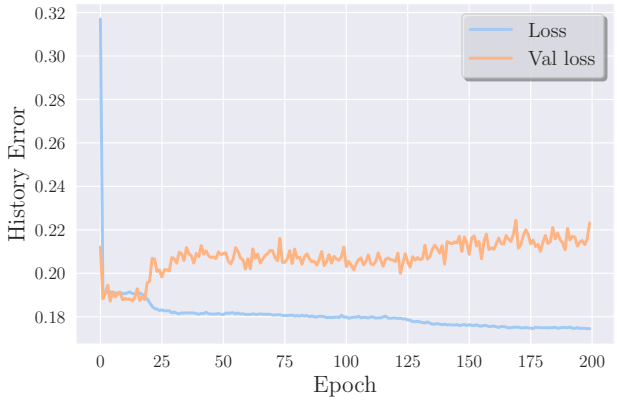

Figure 8: Training history of the FFNN for the prediction of Lorenz attractor. y-axis is shown as the MAE divided by the average of the trajectory's coordinates.

Fig. (9) decomposes the coordinates of the trajectory used for testing the predictions, giving a more clear visualization than the 3D representation.

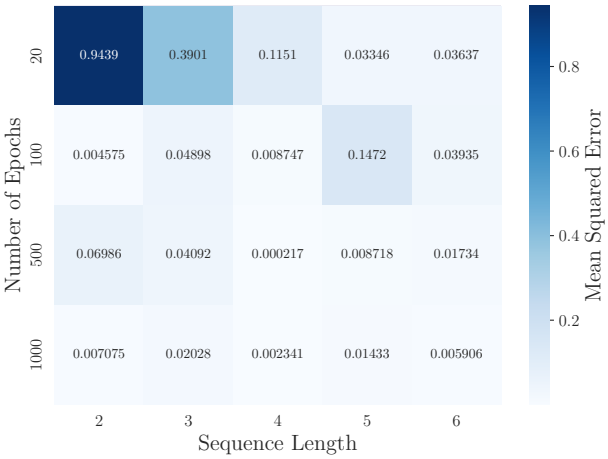

Figure 10: Gridsearch of look-back and epochs for the vanilla RNN trained on the stable spiral trajectory.

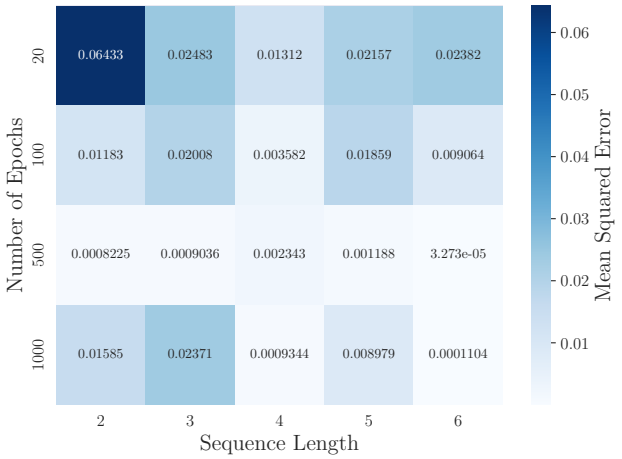

Figure 11: Gridsearch of look-back and epochs for the LSTM trained on the stable spiral trajectory.

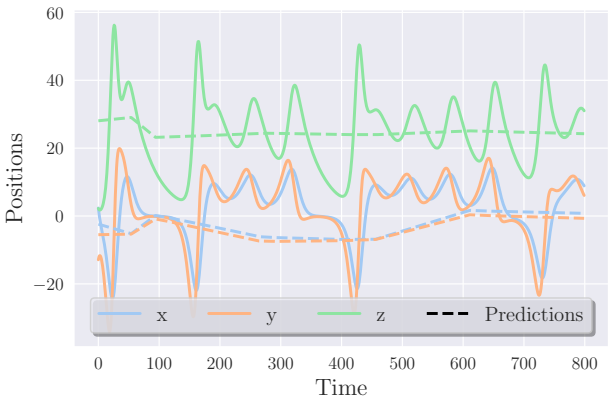

Figure 9: Testing versus predicted trajectories decomposed by coordinates for the FFNN.

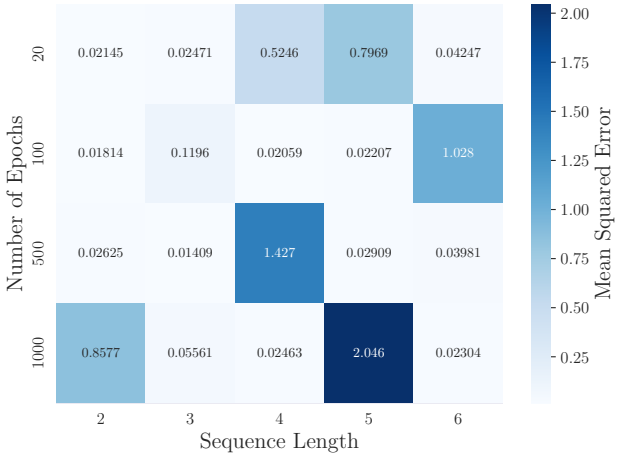

Figure 12: Gridsearch of look-back and epochs for the vanilla RNN trained on the Lorenz attractor trajectory. Average MSE of 0.35 and variance of 0.31

2. *Building an RNN with TensorFlow*

Aiming at investigating the RNN variants at their best, we start a series of basic hyperparameters gridsearch. More specifically, we begin investigating how the accuracy of the stable spiral predictions changes for different hyperparameters. In that sense, Fig. (10) shows the gridsearch of the number of epochs of training versus the length of the sequence the network is retroactively trained on for each step - the *look-back* parameter.

Similarly, we can also visualize, in figure Fig. (11), the accuracy of predictions for the same spiral train and test set, but trained on an LSTM implementation.

Changing now the data set, we display in Fig. (12) and Fig. (13) how the Lorenz attractor predictions performance vary with the look-back and number of epochs trained on. In an analogy to the stable spiral investigation, while the former shows the mean squared error for the vanilla model, the latter shows the loss on the LSTM implementation. Across parameters, This set of values, for the vanilla implementation has an average MSE of 0.35 and variance of 0.31. For the LSTM implementation, the average MSE was of 0.024 with $2.2 \cdot 10^{-5}$ variance.

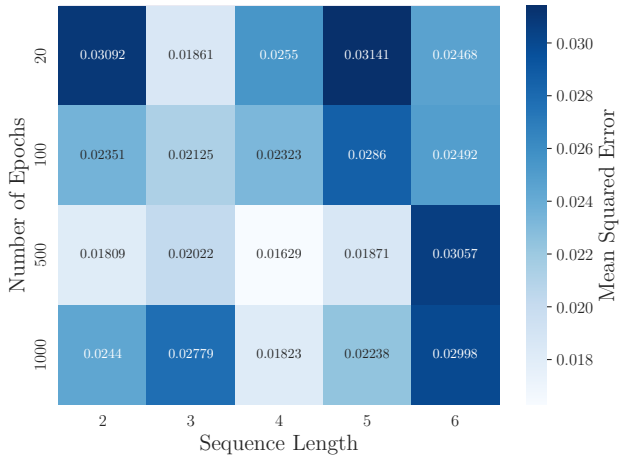

Figure 13: Gridsearch of look-back and epochs for the LSTM trained on the Lorenz attractor trajectory. This set of values has an average of 0.024 and variance of $2.2 \cdot 10^{-5}$

3. *Comparing the networks*

Now that some base parameters were set, we can more fairly compare them. The trajectories for the predictions of the spiral can be seen in Fig. (14), where we display the prediction of both models for the best LSTM parameters. The RMSE for these trajectories of the plot were 0.006 for the Vanilla and 0.0005 for the LSTM.

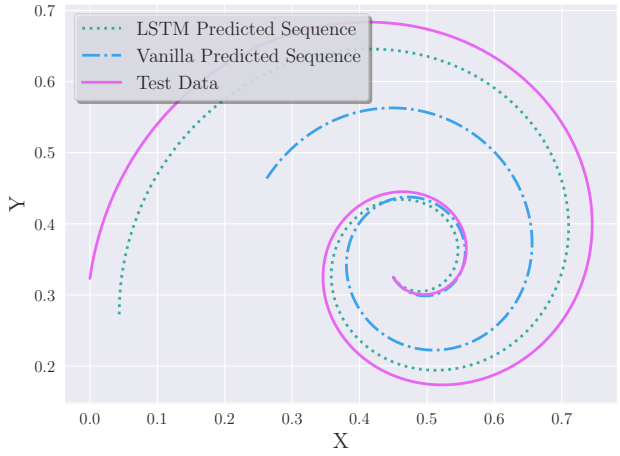

Figure 14: Here we display the predicted trajectories for both LSTM and Vanilla models for the best LSTM hyperparameters.

We now turn to the more intricate problem of predicting the time-series behavior of the Lorenz attractor. Fig. (15) gives a not-so-informative illustration of the predicted trajectories for the LSTM model versus the simple RNN. The mean-squared errors in this case were of 0.02 for the LSTM versus 0.03.

A perhaps mode clear illustration is given by the graph contained in Fig. (16). Here we display the cumulative absolute error of the coordinate's predictions along the time-steps

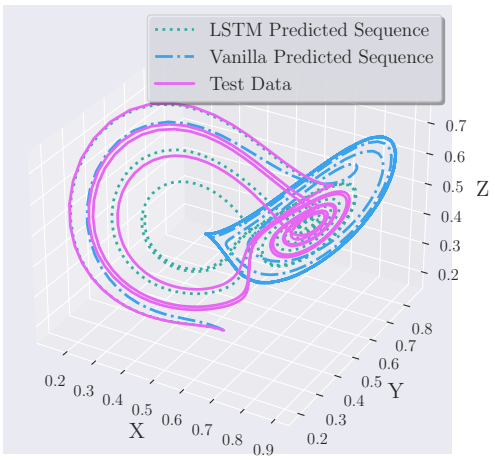

Figure 15: Predictions for the Lorenz attractor trajectories. Comparison between LSTM and Vanilla network implementations.

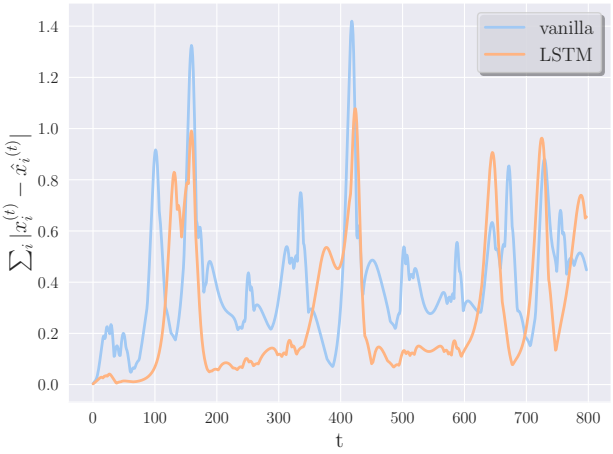

Figure 16: Sum of the coordinate's absolute errors for the LSTM and Vanilla RNNs for the predictions contained in Fig. (15).

4. *Adding physics to the loss function*

The same results obtained for the comparison of the LSTM versus Vanilla implementation of the network

can be repeated, now with the added physics-informed loss function term. The typical side-by-side predictions of the trajectories can be seen in Fig. (17). For this simulation, we had mean squared errors of 0.02 (0.018) for the naive LSTM model versus 0.01 (0.013) for the physics-informed LSTM model.

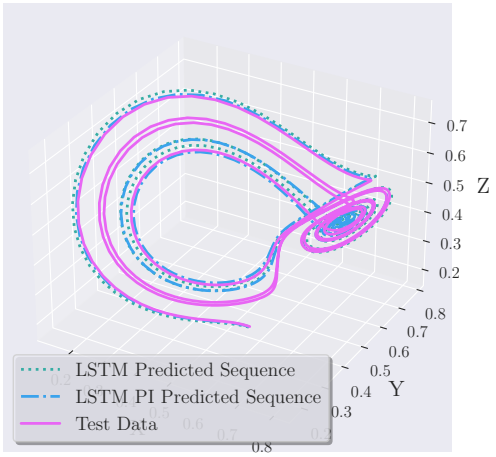

Figure 17: Comparing the predictions of Lorenz trajectories for the LSTM with and without an added physics-informed term to the loss function.

Additionally, we can again study how the sum of the coordinate's errors evolves throughout the time evolution of the system in Fig. (18).

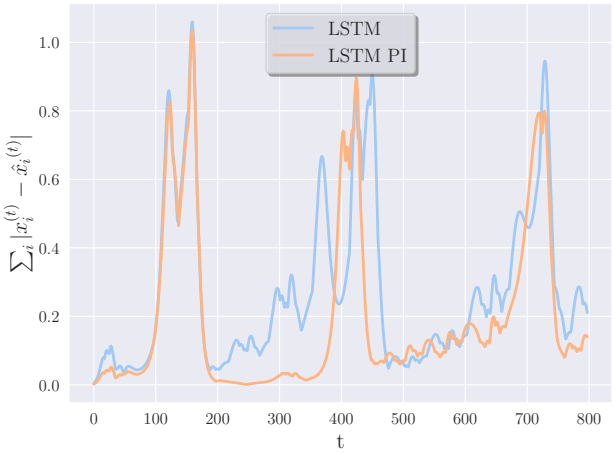

Figure 18: Sum of the absolute errors for all coordinates of the LSTM and physics-informed LSTM model for each time-step. These errors are from the trajectories displayed in Fig. (17).

To understand the effects of the addition of the physics-informed penalty to the training mean squared error values, we display Fig. (19). This figure shows the log of the evaluation metric while the model is being trained, and compares the LSTM networks with and without PI penalty.

Learning Rate	Epochs	Look-back	MSE	Model
10^{-3}	500	2	0.03	Vanilla
10^{-3}	500	2	0.02	LSTM
10^{-3}	500	2	0.01	LSTM PI

Table I: Overview of significant results in comparison of the models for the Lorenz attractor prediction. All of the models here used Adam as optimizer.

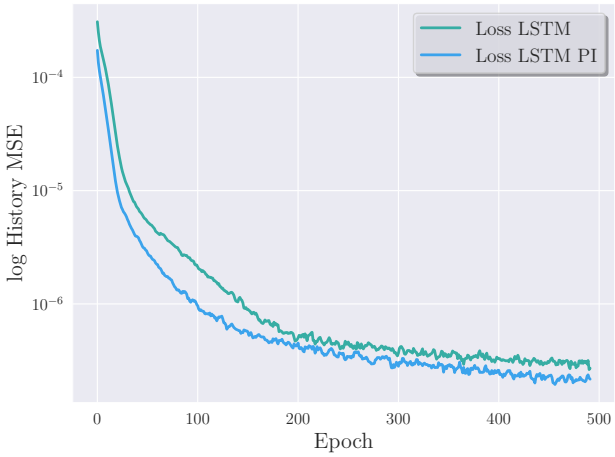

Figure 19: Log of the mean squared error of predictions throughout the training process.

Finally, a brief overview of the models and obtained mean squared error scores for the Lorenz trajectories can be seen in Tab. (I).

IV. DISCUSSION

A. Regular FFNN

We start by addressing the results for the regular FFN. Evidently, Fig. (7) shows a sub-optimal prediction for the test trajectory, and both Figs. (8) and (9) help explain why. While the loss of the testing set keeps arbitrarily improving, the validation loss worsens significantly after a very small number of epochs. This is the classical behavior of an overfitting network and goes to show that training on more epochs is not the way to improve performance in this case.

Reinforcing the point above, Fig. (9), seems to indicate the default prediction of the network is simply the mean of the coordinates. This will clearly not be

sufficient for good results, as the 3D trajectory seen in Fig. (7) has a wide distribution among negative and positive coordinates.

B. Regular RNN and LSTM

The far-from-desirable results of the previous section motivate the discussion of the afterward implemented methods. As expected, the introduction of the time dependency of the loop connections in the RNN was able to improve the predicted trajectories. This analysis needs not be intricate in terms of the assessed evaluation score. Indeed, Figs. (9) and (15) show unequivocally that the RNNs in general do a better job for this forecasting task.

To compare the vanilla RNN implementation to the LSTM, we can start with the grid search of parameters for the prediction of the stable spiral trajectory. While the colormap scale can be deceiving, a closer look at Figs. (10) and (11) show that the LSTM model was able to yield better predictions almost regardless of the number of epochs and length of look-back sequence. The superiority of the LSTM prediction is again confirmed in the trajectories of Fig. (14). Here we should mention that the MSE of the order of 10^{-5} on one of the grid search configurations was considered a fluctuation: further investigations were unable to generate such a low MSE.

When trying to optimize the model's training epochs and look-back length of the Lorenz attractor, we see that the comparison between the vanilla RNN and LSTM is not so simple. The grid searches of Figs. (12) and (13) show that both the neural network's implementations are capable of obtaining MSE scores of the order of 0.02, but the MSE values for the vanilla implementation across the hyperparameters have much higher average and variance across the parameters.

This last analysis indicates that especially when comparing a wider range of parameters, the addition of a more robust set of memory units of the LSTM can benefit the model prediction on complex and, in this case, chaotic trajectories.

Comparing the Lorenz test trajectories by eye against the predictions, as in Fig. (15), can be a non-trivial task. By breaking down the accumulated error of the coordinates as done in Fig. (16), we again confirm that, at least for those set of parameters, the LSTM gave overall better predictions of the test-set trajectory.

C. Discussing the hyperparameters

As we can see from the grid searches over the number of epochs, the performance of the network, in general, did not increase for reasonable-sized epochs, aligning with the aforementioned reasons. For most cases, the choice of 500 epochs was ideal, and values larger than that seemed to incur some detriment to the evaluation metric.

The choice of the look-back length plays an important role in learning the system dynamics and requires careful consideration. In the case of the stable system, a larger look-back is generally effective or does not harm the prediction. However, for chaotic and unstable systems, it is essential to keep this parameter reasonably small (even as small as 2 which our findings suggest to be sufficient). We suppose this is because the nature of chaotic and unstable dynamics inherently calls for a limited look-back.

We opted not to optimize the look-ahead parameter in our model. The rationale behind this decision was that increasing the prediction range would bear similar outcomes to executing multiple consecutive prediction steps.

D. Adding physics to the loss function

The comparison of the prediction of the LSTM network with the same network with an added physics-informed term can be done in Fig. (17), but it is not the most informative. From this graph, both models seem to be performing equally well. A closer analysis, from Fig. (18) shows, however, that the predictions of the model with the physics penalty are overall more accurate throughout the time evolution.

An interesting point is that the model is not better in every time step of the series. This can indicate that the physics loss penalty is successfully guiding the network to learn the differential equations but doing so in a fair way. If the physics-informed network displayed lower MSE for every time step, it could indicate that our implementation was leaking information from the testing set, among other potential problems.

An interesting observation can be made from Fig. (19) when the physics-informed loss is incorporated. Initially, it was anticipated that the PI loss would improve the LSTM's predictions on the test set compared to regular predictions. However, it was not necessarily expected to see an overall improvement in the training MSE throughout the training process.

This outcome is surprising because even though the PI term assists with predictions, particularly for unseen data, it still adds a positive value to the loss term as indicated by Eq. (10). We suspect the reason for this is that we have a noiseless data set, meaning the position and the velocities are completely correlated. Including the velocity is then equivalent to training the trajectories on more points, thus reinforcing the correct results and reducing the loss error. If the data set contains

experimentally measured data, the position would not be completely correlated to its velocity computed using the right hand side of Equation Eq. (2). Improvement in the loss function might not be as visible throughout training in that case. We also argue that this is not physics-informed machine learning in a traditional sense since the velocity of the system is not a physical law but rather just the derivative of the positions. This observation raises the need for further investigation into our implementations.

To summarize the comparison of the RNN models (vanilla, LSTM, and LSTM with PI loss), Tab. (I) shows that, for the correct set of parameters, our expectations on the models were satisfied. The MSE values for the vanilla implementation were outmatched by the LSTM which were in turn also worst than the same variation but with the PI term. The added complexity of each approach, in this case, was not in vain.

E. Discussing the data sets

Our intentions were to initially use the stable spiral trajectory as a benchmark for our network implementations. The task, however, proved to be harder than previously imagined. The obtained mean squared errors for the spiral were smaller than the ones obtained for the attractor, and that can be seen from the previously mentioned grid searches. Nonetheless, the predictions in Fig. (14), confirm that even predicting a spiral can be challenging when only the initial position is given for a network trained on different trajectories.

Regardless of the predictions of the spirals and attractors not being exactly in accordance with the test set, it is valid to note that the "shapes" of the trajectories are preserved in some loose sense. This serves as an indication that some information about the dynamics of the differential equations is indeed being learned. As will be discussed in the following subsection, it can be fruitful to train the networks in those types of time-independent properties of the system.

F. Additional considerations and comparison to literature

A point worth mentioning is the decision to not use any batching procedure in the optimization process. While adding stochasticity to network training is often beneficial, it should be done with caution in chaotic systems. In fact, the trajectory learned by the network is sensitive both to the initial condition and also the network's weights in an unpredictable way. Small updates to the weights will drastically influence the stability of the long-term predictions. Mini-batch training introduces noise into the gradient estimates due to the random selection of mini-batches so by using a gradient scheme based on batches, these effects would be detrimentally magnified during the training process.

According to a more rigorous mathematical analysis of [8], the challenge of training RNNs in chaotic dynamics cannot be overcome by the use of specific architecture designs, constraints, or regularizations. The theoretical results presented argue that the back-propagated gradients of the loss function will invariably explode, and it becomes necessary to limit the gradients in an optimal manner. One of the strategies is to use what is called *sparse teacher forcing*, which could be a topic for future investigation. The idea behind this method is to build up on concepts from dynamical control theory and combine the power of observed data with the RNN's internal states, promoting effective learning of complex sequential patterns.

To further explain the suboptimal results for the RNN predictions, we delve some more into the available literature. When learning and reconstructing dynamical systems, especially nonlinear ones, the works done in [9][10] show that evaluating and optimizing in "ahead-prediction" errors is not meaningful for chaotic time-series. A more modern and fruitful approach is to reproduce invariant or time-independent properties of the system such as the Kullback-Leibler divergence as done in [8]. To add to the complexity of the approach, they also used the so-called dimension-wise Hellinger distance which is a measure of temporal agreement of the observed and generated time series done in their power spectra of the Jacobian.

While the results here contained displayed inferior results when compared to those mentioned above, they stand out in their simplicity. Some representation and underlying dynamics of the attractor are effectively being reconstructed, even if the exact trajectories cannot be predicted. One potential area for enhancement could involve the use of techniques such as autoencoders, known for their capacity to extract useful features.

Finally, it is important to remember that our intent was not to devise a method to perfectly predict chaotic systems but rather to study the capacity of different neural networks in learning and reproducing dynamical systems. While the exact trajectories were not precisely predicted, the neural networks did in any rate learnt some underlying dynamics and recreated the attractor shapes, which is a significant finding in itself.

V. CONCLUSION

In this study, we focused on an exploration of how Feed Forward Neural Networks (FFNN), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks can be employed to understand

and predict dynamics of systems ranging from stable (e.g. the stable spiral) to chaotic(e.g. the Lorenz attractor).

Interestingly, LSTM RNNs were also proficient in studying less chaotic systems, like the stable spiral, which underlines their versatility and adaptability across diverse dynamic systems. However, in contrast to our initial hypothesis, the performance was not substantially better than that of the Lorenz attractor. Additionally, by incorporating physics-informed loss functions, we see an improvement of the results and obtained new insight to physics informed neural network.

It became clear that training RNNs for chaotic dynamics is a challenging task, and conventional approaches may not be sufficient. We recognise that there is still room for improvement and further exploration.

[1] K. Zhang, "The Lorenz system -An introduction to chaos," .
[2] M. L. Abell and J. P. Braselton, in *Mathematica by Example (Sixth Edition)*, edited by M. L. Abell and J. P. Braselton (Academic Press, 2022) sixth edition ed., pp. 409–523.
[3] B. Hasselblatt and A. Katok, *A first course in dynamics: with a panorama of recent developments* (Cambridge University Press, 2003).
[4] E. N. Lorenz, Journal of atmospheric sciences **20**, 130 (1963).
[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) pp. 380–381, http://www.deeplearningbook.org.
[6] S. Hochreiter and J. Schmidhuber, Neural Computation **9**, 1735 (1997).
[7] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," (2018).
[8] J. Mikhaeil, Z. Monfared, and D. Durstewitz, Advances in Neural Information Processing Systems **35**, 11297 (2022).
[9] G. Koppe, H. Toutounji, P. Kirsch, S. Lis, and D. Durstewitz, PLoS computational biology **15**, 1 (2019).
[10] S. N. Wood, Nature **466**, 1102 (2010).