Physics-informed and recurrent neural networks applied to the Lorenz attractor

Marta Corioni, Luigi Ernesto Ghezzer, Francesco Pogliano

April 2024

Abstract

In this study, we compare the performance of fully-connected feed-forward, physics-informed, and recurrent neural networks (FFNNs, PINNs, and RNNs) when used to predict the solution of two systems described by differential equations: a damped harmonic oscillator (DHO) and the Lorenz attractor. In the DHO we show that even with the same architecture and computational resources PINN perform much better than plain FFNN. When the differential equations describing the system are not known and PINNs are not possible to deploy, RNNs serve as an alternative, being capable of achieving a high level of accuracy in capturing the DHO properties, despite their slower computation times. When trying to predict the future behavior of the Lorenz attractor, the PINN struggle to converge on the right solution. A learning schedule for the PINN algorithm was proposed mitigate the problem, but the convergence time is still much longer than for the DHO. The RNN can quite accurately predict the true solution for the Lorenz attractor during the initial few time steps, showing its effectiveness in capturing the initial dynamics. However, as the simulation progresses, the RNN starts to deviate from the true solution and begins following a consistent pattern, indicating some limitations in capturing the long-term behavior of the Lorenz attractor accurately.

1 Introduction

A damped harmonic oscillator (DHO) is an interesting system from a physics point of view. In its simplest representation, the DHO may describe the time evolution of the position of a body lying on a rough surface, when attached to a spring (see Fig. 1, left). The position of the body will oscillate around the rest position, but its amplitude will gradually decrease as energy is lost to friction. Similarly, in another representation, the body may be hanging vertically from a spring, and oscillating up and down, the friction being the internal one in the spring. A third, common representation is an RLC circuit, meaning an electric circuit consisting of a resistor, an inductor, and a capacitor connected in parallel (Fig. 1, right). The DHO will describe the current intensity (and direction) as a function of time as it bounces back and forth due to the interplay between

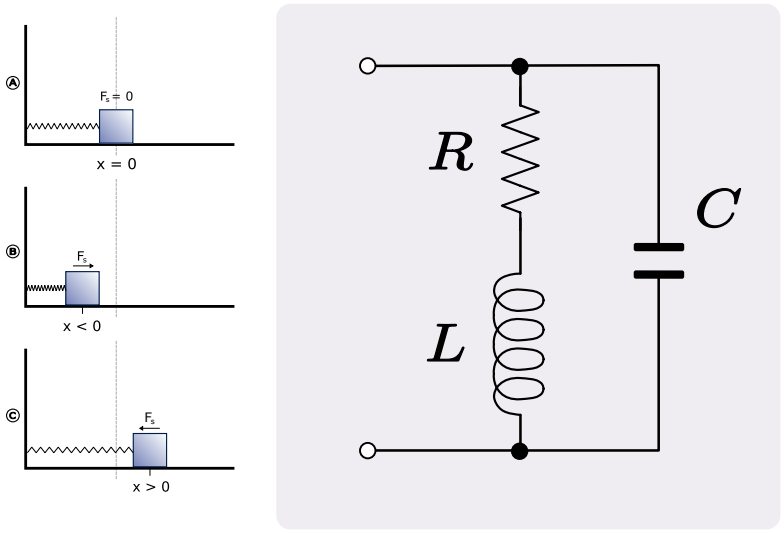

Figure 1: To the left, a sketch of the system of a body connected to a spring (Wikipedia, public domain). To the right, an RLC circuit (Wikimedia, user:Spinningspark, CC-BY 3.0). The displacement of the body from the rest position and the current in the circuit may be described using a DHO model.

capacitor and inductor, while the resistor dissipates the energy and damps the amplitude. Given its ubiquity in physics and the fact that its mathematical behavior is well studied and understood, it provides a good test case for studying the behavior of different machine learning (ML) predictive models.

Another interesting system in physics is the Lorenz attractor. This is one of the most famous examples of a chaotic system, where although its mathematics are well described and its evolution deterministic, a slight change in starting conditions may generate a big difference in predicted behavior. This makes the modeling of such systems increasingly harder the more into the future one tries to predict the results. In this report, we will compare how different ML algorithms will perform when describing these two systems, their strengths and weaknesses, and how they differ when trying to predict a non-chaotic such as the DHO, and a chaotic one such as the Lorenz system.

The algorithms used to study these two systems are *feed-forward, fully connected neural networks* (FFNNs), *physics-informed neural networks* (PINNs), and *recurrent neural networks* (RNNs). All these are neural networks (NNs), a powerful tool loosely modeled on the biology of the brain, first appeared in 1958 [1] and exploded in popularity once certain issues concerning the activation function were solved in 2010 [4]. The most "basic" NN is the fully connected FFNN, where every node in every layer is fully connected to the nodes of the following layers, and the values in each node are decided by the input and a set of previously trained weights and biases. From the FFNN were developed differ-

ent variants, each tailored to address specific types of problems more correctly and effectively.

Since the subject of this report is the time evolution of two physical systems, together with the simple FFNN we will consider two of its modifications: the PINN and the RNN. A more detailed and mathematical description of the physical systems at hand (the DHO and the Lorenz system) is given in Section 2, while a description of the three employed NN algorithms can be found in Section 3. The results of this analysis are then presented in Section 4 and a discussion follows in Section 5. The code used and referred to in this project report can be found in this GitHub repository: `https://github.com/luigiEG/FYS5429_Project1`.

2 The physical systems

2.1 The damped harmonic oscillator

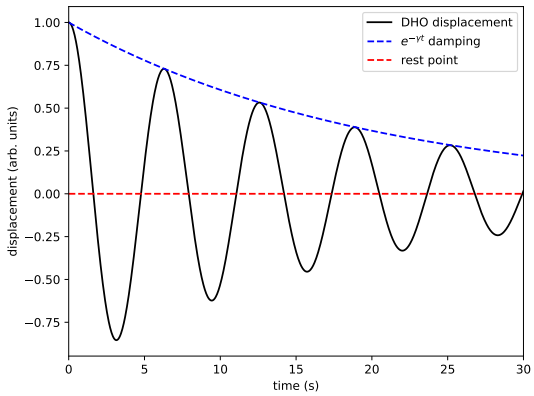

Figure 2: Plot of $x(t)$ as described in Eq. (2), for arbitrary units of displacement. This is a qualitative visualization of the damped harmonic oscillator, where a sinusoidal function is enveloped in a decaying exponential.

The damped harmonic oscillator (DHO) is a model that describes different physical systems, e.g. the oscillating motion of a body connected to a spring and experiencing friction. The body experiences two forces: F_s and F_f. The first one is the spring's drive and is given by Hooke's law $F_s = -kx$, where k is the spring's constant and x the displacement from the rest position, while the second is the friction, $F_f = cv$, where c is the friction coefficient and v the velocity of the body. By employing Newton's second law $\sum F = ma$ (where m is the body's mass and a its acceleration) and rewriting all in terms of derivatives of x, we obtain the following differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \frac{k}{m}x + \frac{c}{m}\frac{\mathrm{d}x}{\mathrm{d}t} = 0. \tag{1}$$

This is a second-order differential equation, and its solution can be easily found, and it depends on the value of k/m. In this report we will only consider the underdamped solution, meaning where $k/m < 2$. In this case, the analytical solution can be written as

$$x(t) = Ae^{-\gamma t}\cos(\omega t + \phi) \tag{2}$$

where $\gamma = c/2m$, $\omega = \sqrt{k/m - \gamma^2}$ are constants that reconnect to the body-spring configuration, and A and ϕ are the amplitude and phase, to be adjusted from the initial conditions. As we see in Fig. 2, this corresponds to an oscillatory motion where the total amplitude is modulated and decays exponentially.

2.2 The Lorenz attractor

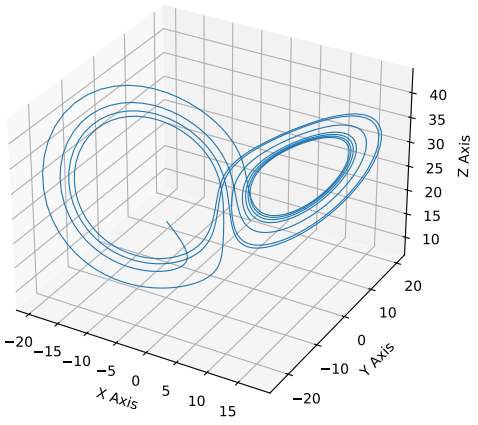

Figure 3: Plot of the Lorenz attractor as described in Eqs. (3) for $\sigma = 18$, $\rho = 28$ and $\beta = 8/3$.

The Lorenz attractor is a system of three, first-order differential equations first written in 1963 as a way to model atmospheric convection [2]. This is a 3D model, meaning that it describes the time evolution of three coordinates. The three differential equations are:

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \sigma(y - x), \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= x(\rho - z) - y, \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - \beta z, \end{aligned} \tag{3}$$

where σ, ρ and β are constants. This system is chaotic, meaning that a small variation in initial conditions will generate a very different time evolution. These

systems are notoriously difficult to handle since one is usually interested in making predictions when they occur in reality, and these predictions become rapidly non-accurate the further we go into the future. An interesting feature of the Lorenz system is the existence of an attractor: a set of the phase space where solutions with different initial condition tend to converge. In the Lorenz attractor, the trajectory circulates alternatively around two points, forming a sort of "butterfly" figure, as shown in Fig. 3.

3 The algorithms

3.1 Feed-forward neural networks

The fully-connected, feed-forward neural network (FFNN) may be seen as the precursor of all the more complex and problem-specific NN algorithms, such as *convolutional* NNs, *recurrent* NNs, *adversarial* NNs and so on. FFNNs can be understood schematically as nodes organized in layers. The first layer is called the *input layer*, the last is the *output layer*, and all layers in between *hidden layers*. Each node in the input layer contains a value, and these values are then propagated to the first hidden layer with the help of *weights*, *biases*, and *activation functions*. The value a_j^L in a node j of a hidden or output layer L is given by

$$a_j^L = f\left(\sum_i^N w_{ij} a_i^{L-1} + b_i\right), \tag{4}$$

where the sum $\sum_i^N$ runs through all the N nodes in the preceding layer $L-1$, multiplies each of their values a_i^{L-1} with the specific weight w_{ij} connecting node i in layer $L-1$ to node j in layer L, and adds their product to the bias b_i. The sum is then run through an activation function f. Examples of this function are the sigmoid function, the arctan function, the ReLU function and so on. Weights and biases are fitted to reproduce a dataset by *training* the network. The most popular training method is based on *back-propagation*: an optimization algorithm that allows us to minimize a custom *cost* or *loss function* $\mathcal{L}$. The most commonly used loss function would for example be the mean-square error of the predicted values in the output layer, and the expected or theoretical values. One may express the loss function in terms of the employed weights and biases, and thus imagine it as a multi-dimensional scalar field spanned by all the NN parameters. By calculating the negative gradient along each of these parameters allows us to update our set of weights and biases into one that yields a lower loss function, and thus a more correct result. The length of the step in the negative gradient direction is regulated by a *hyperparameter* called the *learning rate*, or η. Back-propagation is repeated until desired precision, and the network is trained. Ideally, this would correspond to a global minimum in the loss function. The choice of the number of layers, the number of nodes in each layer, their activation functions, and the value of the learning rate (in general, any parameter that is not optimized in the training) decide the hyperparameters

configuration of the FFNN. Different configurations may be faster to train or better at solving certain problems than others. Another issue when looking for a minimum in the loss function is the danger of converging in a local, instead of a global minimum. This is especially the case for multi-dimensional, rough loss function landscapes. A solution to this is to divide the training set into batches and calculate the negative gradient only on a subset of the whole training set at a time. In this way, the landscape is smoothed out and the algorithm has the chance of escaping local minima. The number of batches the whole training set is divided in is one more hyperparameter, and we talk about one *epoch* when we have run through all the batches once. The number of epochs is also a hyperparameter to be tuned. Although an arbitrarily complex FFNN architecture is theoretically able to approximate any function [3], this task can be complex and time-consuming. This is why there exist many variations of the concepts, which tailor the architecture of the NN in order to adapt to the type of problem at hand.

3.2 Physics-informed neural networks

FFNNs are useful when one has a lot of data on which the network can work on and find patterns. This approach is called *data-driven* machine learning. There are cases though where there are not a lot of data available, and we are asked to find the best pattern describing the few data points we have. This is usually the case in scientific research, where obtaining experimental results may be time-consuming, difficult and/or expensive. Motivated by these cases, scientific ML was developed, trying to replace the lack of data with information we already know about the system, namely the known physics laws that govern it [8]. In this scope, *physics-informed* neural networks (PINNs) are one of the many applications of scientific ML.

Let consider a general FFNN with one input, the independent parameter of the ordinary differential equation (ODE), t, and as output the state $\tilde{Y}(t)$ (this could also be a vector, as in the Lorenz case, see next section). In a PINN, the previously described FFNNs are penalized in the training when the results do not respect the ODE (other physics inspired constraints may be also used). This is done by providing the loss function with two additional terms: $\mathcal{L}_{\text{ODE}}$ and $\mathcal{L}_{\text{IC}}$. The first one includes the differential equations describing the system and encodes the physics. For example if the differential equation describing the system is $f' = cf$ (where f' is the time derivative of f, and c a constant), then

$$\mathcal{L}_{\text{ODE}} = \frac{1}{N}\sum_i (f'_{NN}(t_i) - cf_{NN}(t_i))^2, \tag{5}$$

where $f_{NN}(t)$ is the neural network and t_i are a selection of points in the time interval were the solution is searched. The number of these points, called *collocation points*[7], may be decided as a hyperparameter, and they may very well be positioned within the "test set", as we are only passing the independent variable t to the system, and not its solution $Y(t)$. While $f_{NN}(t_i)$ are

our model predictions, the derivatives $f'_{NN}(t_i)$ are calculated numerically with `pytorch.Autograd`. The second term, $\mathcal{L}_{\text{IC}}$, enforces the initial conditions of the system as $\mathcal{L}_{\text{IC}} = (f_{NN}(t_0) - Y(t_0))^2$, where $Y(t_0)$ is the initial condition. The full loss function will then take the form

$$\mathcal{L} = (\mathcal{L}_{\text{MSE}}) + \lambda \mathcal{L}_{\text{ODE}} + \mathcal{L}_{\text{IC}}, \tag{6}$$

where λ, weighting the two loss contributions, can be considered as another hyperparameter to be tuned. The $\mathcal{L}_{\text{MSE}}$ term, representing the mean square error loss, may be added if we have actual data to train, but may be also omitted, meaning that a PINN will be trained only by knowing the differential equation describing the system and the initial conditions. In this way, PINNs can also be used to obtain approximate solution of ODEs and partial differential equations, or PDEs[7].

3.3 Recurrent neural networks

As we saw, if we consider the FFNNs as general approximating machines, modifications and developments of their structure consist in shaping the neural network so that it is most prone to solve problems of the preferred type. One clear example of this are *convolutional* NNs, which we do not use here. CNNs are tailored to analyzing pictures by connecting nodes (representing color values in a pixel) only to the neighboring ones and thus making the network not fully connected. This constraint forces the architecture to only see patterns within a neighborhood of pixels or, in other words, visual and graphical patterns. PINNs may be seen in the same light, where the loss function is modified and a constraint is added so that the network is forced to take into account additional information about the physics of the system. Another sort of problem we may want to approach with ML is sequential data sets. In these, we may be dealt with a series of time coordinates with the corresponding time-varying variable and asked to predict the continuation of the series. In order to solve this, one may employ *recurrent* NNs. The concept of RNNs is to preserve some memory of the system, so that predictions will be forced to consider how data points follow each other, and not treat every data point as equal regardless of their position in time relative to the one we are trying to predict. If we define a simple FFN "network" consisting of one simple input node x_t, one hidden node h_t and one output node y_t, we may write

$$\begin{aligned} h_t &= f_{ih}(w_{ih}x_t + b_{ih}) \\ y_t &= f_{ho}(w_{ho}h_t + b_{ho}), \end{aligned}$$

where w_{ih}, b_{ih}, w_{ho} and b_{ho} are the weights and biases linking the input to the hidden, and the hidden to the output node respectively, and f_{ih} and f_{ho} their relative activation functions. A naive implementation of RNNs, often referred to as *vanilla* RNNs, consists in letting previously processed data to impact the output. This means that hidden nodes will be influenced by previous hidden

nodes, and a following h_{t+1} will be defined as

$$h_{t+1} = f_{ih}(w_{ih}x_{t+1} + w_{hh}h_t + b_{ih}), \tag{7}$$

where h_t carries the information from the previous input, and w_{hh} is the weight deciding how big an impact will have on previous layers. The vanilla RNN has many of the features one desires but suffers of one big problem: exploding/vanishing gradients. As we can see, w_{hh} operates between every time step. When back-propagating, this means that the gradient of the loss function with respect to w_{hh}, i.e. $\nabla_{w_{hh}}\mathcal{L}$, because of the chain rule will contain T terms like $\mathrm{d}h_{t+1}/\mathrm{d}h_t$, where the exponent T is the number of time steps provided in the training. This means in practice that the gradient will contain a term like w_{hh}^T, where T is usually a very large number. If w_{hh} is a small value below unity, this will eventually become vanishingly small, stopping the network from giving meaningful predictions. If the term is slightly above unity, it will explode. One way to solve this is by modifying the vanilla RNN approach by employing *long short term memory*, or LSTM. This modification consists in the introduction of a memory cell, and a cell state variable c. The memory cell is a series of operations that take in the values from the previous hidden layer h_{t-1} (called *short term memory*), the cell state variable from the previous layer c_{t-1} (called *long term memory*) and the input value of the current time step x_t. The memory cell is formed by three gates: the *forget* gate, the *input* gate and the *output* gate). In the forget gate, we calculate a factor f_t between 0 and 1, which will decide how much of the long-term memory from the previous cell we want to keep:

$$f_t = \sigma\left(w_f x_t + u_f h_{t-1} + b_f\right), \tag{8}$$

where w_f, u_f, and b_f are gate specific weights and biases (as will be the following ws, us and bs with different subscripts), and σ is the sigmoid activation function enforcing the result to be between 0 and 1. In the input gate, we try to find out two things: the features in the input, and how "remember-worthy" they are, before adding them to the long-term memory. These features are extracted as

$$\tilde{c}_t = \tanh\left(w_c x_t + u_c h_{t-1} + b_c\right) \tag{9}$$

for the feature extraction, and

$$I_t = \sigma\left(w_i x_t + u_i h_{t-1} + b_i\right) \tag{10}$$

for their "remember-worthiness". Here $\tilde{c}$ is calculated with tanh because it allows for an input between -1 and 1, and similarly to the forget gate, the sigmoid activation function is used to calculate how much of these input features we want to remember. The long term memory from the previous cell c_{t-1} is eventually updated to c_t as

$$c_t = f_t c_{t-1} + I_t \tilde{c}_t. \tag{11}$$

Finally, the output gate updates the short-term memory h as

$$o_t = \sigma\left(w_o x_t + u_o h_{t-1} + b_o\right), \tag{12}$$

$$h_t = \tanh\left(c_t\right) o_t, \tag{13}$$

which is both the output for the time step t, and one of the inputs for the following memory cell.

4 Results

In this section, we will present the results from the neural network calculations for both the DHO and the Lorenz attractor. The codes can be found in the GitHub repository referred to in the introduction. All codes are written in Python in the form of Jupyter notebooks, and we used the `PyTorch` package to model the neural networks.

4.1 The damped harmonic oscillator

The study of the damped harmonic oscillator (DHO) was carried out by first generating a solution of Eq. (1) with the parameters $m = 1.0$, $k = 1.0$ and $c = 0.1$, for initial conditions $x(t = 0) = 1.0$, $\mathrm{d}x(0)/\mathrm{d}t = 0$, from $t = 0$ to 30. A plot of this solution can be found in Fig. 2. The solution was generated in `src/DHOscillator_data_gen.py` and then saved to be later analyzed with a FFNN, a PINN and an RNN.

4.1.1 FFNN and PINN

Our study uses this solution to first compare the FFNN and the PINN performances using the same hyperparameters and architecture (called *baseline*). Since the FFNN baseline performance were much poorer than the PINN, we focused on the last one in a hyperparameter optimization searching for a better model (more on this in the Discussion section). A summed-up analysis comparing the performances of the FFNN and the PINN can be found in `src/DHOscillator_analysis.ipynb`. Details on the PINN tuning can be found in `DHOscillator_PINN_tuning.ipynb`, and for the baseline calculations in `DHOscillator_FFNN_baseline_model.ipynb` (similarly for the PINN). The hyperparameter tuning was carried out by using the `ray` package [6] and employing the asynchronous hyperband-type algorithm (`ASHAScheduler`, [5]), where several hyperparameter combinations were chosen at random and let compete against each other. After a certain number of epochs, the less-performing combinations are pruned to focus computational power on the most promising ones, until one combination survives as the winner. In particular, `ASHA` has three hyperparameter:

- `grace_period`: epochs between performance evaluation of the different models
- `reduction_factor`: fraction of models in a specific bucket to discard at the end of each `grace_period`
- `max_t`: max number of train epochs for a single model

We explored 128 hyperparameter configuration, a plot with the evaluation of the different combinations and the `ASHA` tuning is shown in Fig. 4. For details on the sampling of the hyperparameters `DHOscillator_PINN_tuning.ipynb`

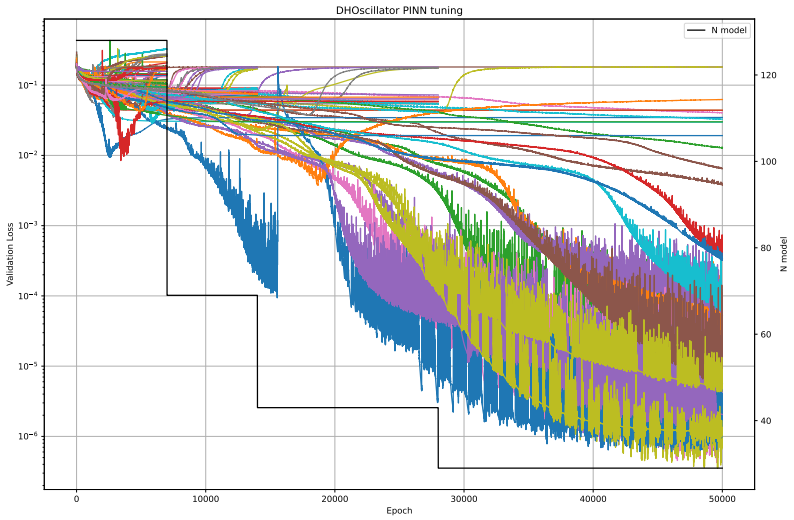

Figure 4: The results of the hyperband algorithm to search the best hyperparameters for the PINN.

The relevant hyperparameters for the FFNN and PINN are (baseline and tuned PINN hyperparameters in parenthesis):

- `n_layers`: number of layers (baseline: 3, tuned: 4)
- `n_neurons`: number of neurons/nodes per layer (baseline: 20, tuned: 28)
- `lr`: the learning rate (baseline: 0.1, tuned: 0.004)
- `factor`: factor by which the learning rate is reduced when patience is reached (baseline: 0.9, tuned: 0.74)
- `patience`: number of epochs on a loss plateau before lowering the learning rate (baseline: 200, tuned: 618)
- `batch_size`: data points in batch (baseline: 595, tuned: 946)
- `grid_num`: uniformly distributed collocation points (baseline (PINN): 595, tuned: 303[1])

[1] In this case `grid_num` is smaller than `batch_size`, meaning that the data set is not divided into minibatches. The number was anyway referred as both hyperparameter were tuned by the hyperband algorithm

- `lambda`: coefficient in front of $\mathcal{L}_{\text{ODE}}$, see Eq. (6) (baseline (PINN): 1.0, tuned: 23.9)
- `n_epochs`: number of epochs, fixed, and not tuned (baseline: 50000, tuned: 50000).

For the first part of the analysis, the FFNN and the PINN performances were compared. Both models were trained by dividing the dataset into 70% training, 15% validation and 15% test. While the FFNN was given train and validation data in the $0 \leq t < 25.5$ interval (25.5 being (70+15)% of 30), the PINN was given the same amount of training points, but now throughout the whole time interval $0 \leq t \leq 30$. The reason why this is not "cheating" by giving the network the solutions, is that the PINN loss function does not include a $\mathcal{L}_{\text{MSE}}$ term, so we are not providing solutions for the test set (or, for that matter, for the train set either). For a similar reason, no validation was performed on the PINN since only collocation points are passed for its training, and there was no danger for overfitting. By only providing the differential equation describing the system and its initial conditions, we are able to compare the model's predictions and its derivatives (which may be calculated with `pytorch.Autograd`) with the loss function, without necessarily providing the solution.

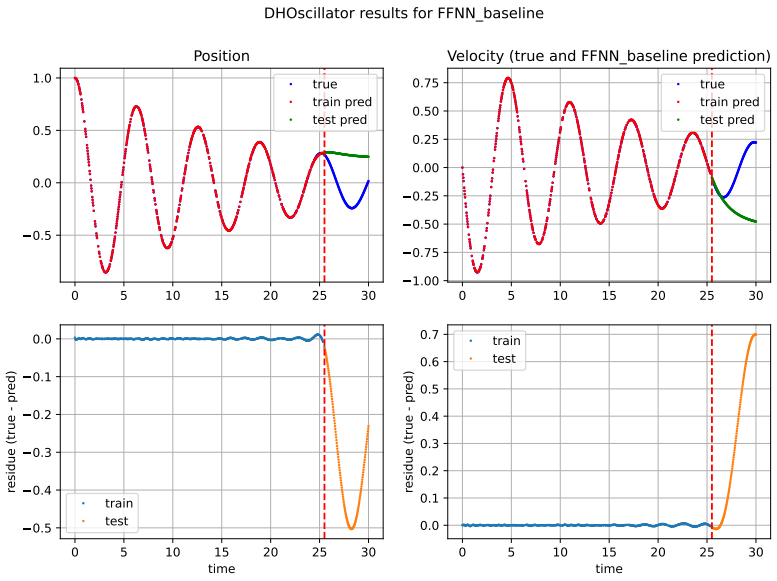

Figure 5: Predictions of the FFNN model on the DHO data. Here we see how precision is high for the training data, but quite low for the test (future) data, showing how time correlations are lost modeling differential equations with an FFNN.

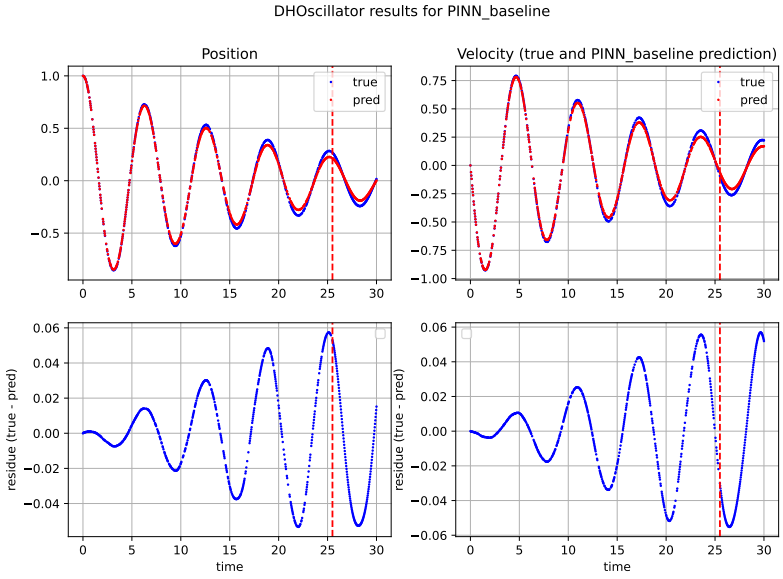

Figure 6: Predictions of the PINN model on the DHO data, using the same hyperparameters as the FFNN. This model does not divide data temporally into training and test, but it only provides the system's differential equation and boundary conditions as information. The model is not as precise as the FFNN for data below $t = 25.5$, but has an overall better prediction power.

A comparison between these two networks with the same hyperparameters and trained for the same amount of epochs can be seen in Figs. 5 and 6, where the position and the velocity train, validation, and test points are shown, as well as a plot of their residuals against the true, analytical value. From these plots, we see how the FFNN does good at fitting the train values, but does not catch the oscillatory motion. The PINN, instead, does well in guessing the general behavior of the oscillation, and although the fit for the train set is not as good as for the FFNN, the predictions in the test are much closer to the true values, two order of magnitude better. With the use of the hyperparameters found with the hyperband algorithm, the model becomes much better (see Fig. 7), showing in an even clearer way how providing information about the physical behavior of the system may improve the predictions dramatically.

4.1.2 RNN

We analyzed the same problem utilizing a simple RNN. The code for this can be found in the file named `src/DHOscillator_LSTM.ipynb`. We did not try to optimize the structure but aimed to build a network with similar complexity for comparison with the others. The RNN model in this case includes:

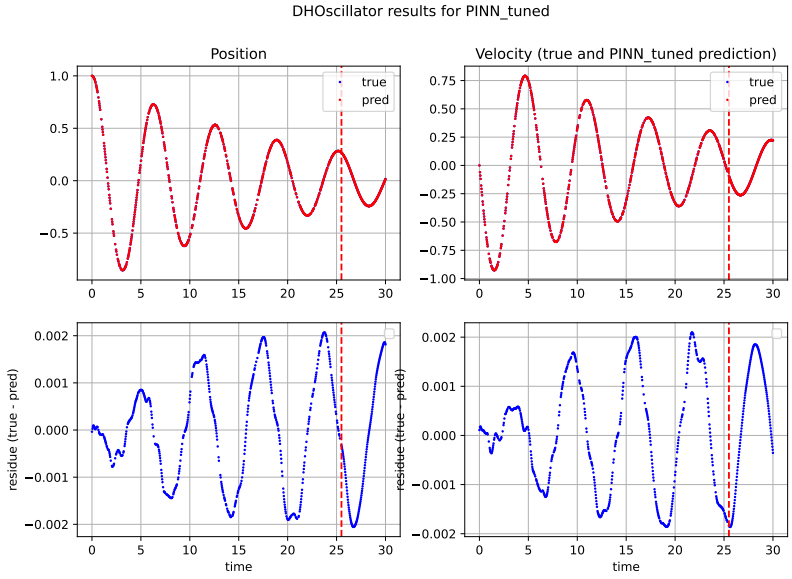

Figure 7: Predictions of a well-tuned PINN model on the DHO data. Here we attain both good precision and predictive power.

- One LSTM layer with an `input_size` of 2, which represents the position and velocity of the oscillator. The `hidden_size` of 50 refers to the number of hidden units within the LSTM layer. These hidden units serve as memory cells, allowing the model to capture and process information over a sequence of inputs.

- A linear layer that maps the hidden state, obtained from the LSTM layer, to an `output_size` of 2. This final layer provides predictions or estimates for the position and velocity of the oscillator.

The model is then instantiated and an `Adam` optimizer is used with the model parameters. The loss function is set to the mean squared error (`MSELoss`).

For training, the RNN is fed sequences – sets of consecutive data points – representing the position and velocity of a system at different time steps. The `lookback` parameter determines the number of previous time steps to consider when creating the feature and target pairs for training the RNN model. In this case, we choose a `lookback` of 4. For each sample in the input datasets, the feature is created by stacking `lookback` consecutive points of position and velocity, encoding the historical information leading up to the current time step. Similarly, the target is formed by stacking the subsequent `lookback` data points that immediately follow each of the feature's end points. This structure creates the output sequence that the RNN will be trained to predict. The pseudo-code illustrates how the dataset is constructed:

```
function create_dataset(dataset_p, dataset_v, lookback)
    Initialize empty lists X, y
    For i from 0 to (length of dataset_p - lookback)
        feature = stack_columns(
            dataset_p[i : i + lookback],
            dataset_v[i : i + lookback])
        target = stack_columns(
            dataset_p[i + 1 : i + 1 + lookback],
            dataset_v[i + 1 : i + 1 + lookback])
        Append feature to X
        Append target to y
    end for
    Return tensors(X), tensors(y)
end function
```

By adjusting the `lookback` parameter, you can control how much historical context and future information the model considers when making predictions. A larger `lookback` value allows the model to capture longer-term dependencies and patterns, but it also increases the dimensionality of the input and can lead to more complex training and slower convergence. On the other hand, a smaller `lookback` value focuses on more recent information and may result in faster training but potentially overlooks long-term patterns. For the training, we considered the following hyperparameters (introduced in the last section):

- `batch_size`: 8
- `n_epochs`: 1000
- `lr`: 0.001
- `factor`: 0.9
- `patience`: 250

The model was trained by dividing the dataset into 85% training and 15% test. The net was given train data in the $0 \leq t < 25.5$ interval as for the FFNN. We did not utilize a validation set to oversee the training of our LSTM-based RNN. This decision was made due to the nature of the inputs, which consist of subsequent sequences. If we were to select a portion of the input interval as a validation set, it would result in the loss of valuable data, leading to less accurate predictions during training. Furthermore, the possibility of overtraining was low because the subsequent nature of the input sequences helped to prevent the model from simply memorizing specific data points. This allowed the model to learn the underlying patterns and relationships in the data more effectively, reducing the risk of overfitting.

This model is designed to predict future values in a time series, doing so by examining a set of historical data points (`lookback`) and using this context to anticipate the next value in the sequence. In order to generate predictions for the test set, the model initially takes the last `lookback` number of data points from the training data as its input sequence. Utilizing this input sequence, the model forecasts the subsequent data point. After each prediction is made, this newly anticipated point is incorporated into the input sequence. This iterative process continues, progressively evolving the input sequence until the entire test set has been successfully found. This method for predicting data is much slower than the straightforward method used for FFNN and PINN, which feed future time to the trained net and obtain the predicted data without relying on a cycle.

Fig. 8 shows the position and the velocity train, validation and test points are shown, as well as a plot of their residuals against the true, analytical value. In Table 1 are shown all the performances for the models used on the DHO.

model	train MSE	validation MSE	test MSE	comp. time (s)
FFNN	5.001e-6	6.746e-6	0.147	(3.584 ± 3.820)e-6
PINN	6.239e-4	6.585e-4	1.568e-3	(3.318 ± 4.716)e-6
PINN tuned	1.268e-6	1.240e-6	1.749e-6	(4.956 ± 5.465)e-6
RNN	4.079e-7	-	3.451e-05	2.172e-2

Table 1: Performances for the different models on the DHO. Since the PINN is not trained on data points and does not need validation, their validation MSE points are calculated with respect to the same points used for the FFNN for comparison. The RNN is trained without validation as explained in the RNN section.

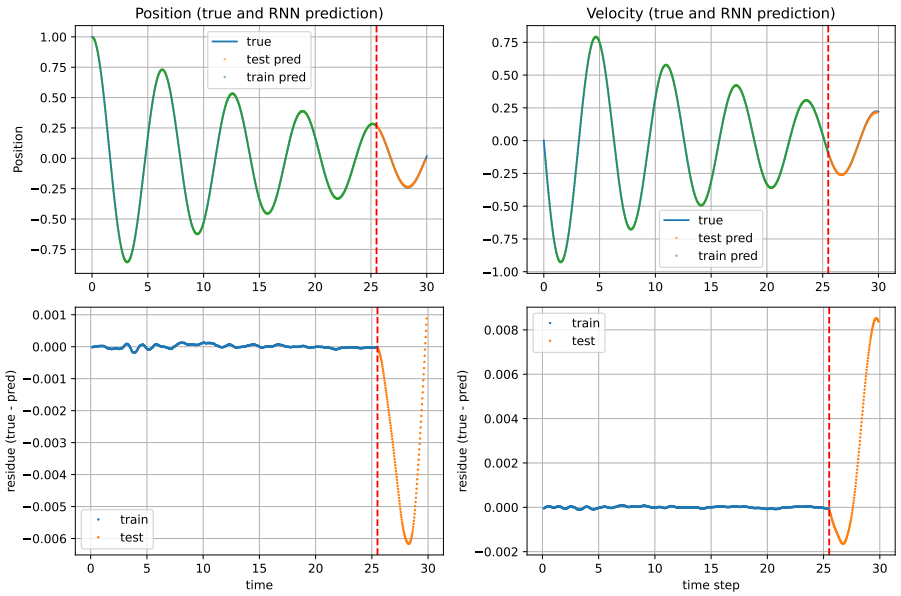

Figure 8: Predictions of the RNN model on the DHO data. The RNN model exhibits high precision in its predictions when applied to the DHO dataset, maintaining this accuracy across both training and future test data. Compared to the FFNN model, the RNN demonstrates superior performance when estimating future data points, with discrepancies that are smaller by several orders of magnitude.

4.2 The Lorenz attractor

As for the DHO, the study of the Lorenz attractor was carried out by generating a solution to the differential equation (numerically with Runge-Kutta with a short enough time step, in `src/Lorenz_data_gen.py`), and comparing network predictions to it. The study on this chaotic system was done only by using a PINN and an RNN, given the poor performance of the FFNN even on the relatively well-behaved DHO.

4.2.1 PINN

In order to study the Lorenz attractor, we made the initial guess of hyperparameters to be the same as for the fine-tuned DHO PINN model. After some tests, it was clear that PINN struggled to converge to the correct solution of the chaotic system. The same training method as for the DHO was used, although with five times more epochs, and a shorter time span of 8 units (compared to the 30 time units used for the DHO). The results were poor, with no apparent qualitative match. Only when trained on a much smaller time span $T = (0, 0.25)$, a solution that qualitatively resembled the correct one was observed. The next step was to multiply the time span by four to $T = (0, 1)$ and perform a training for four times the time needed to converge in the $T = (0, 0.25)$ case.

We observed that even if the PINN manages to converge to the right solution for the time span $T = (0, 0.25)$, it does not for $T = (0, 1)$, even with the increased number of epochs. Even with a number of epochs corresponding with two night of training on a personal computer, the PINN still converges to a solution non compatible with the one obtained with the initial conditions (see Fig. 9).

An interesting observation comes from analyzing the Lorenz $\mathcal{L}_{\text{ODE}}(t)$ as a function of time (Fig. 10). Here we can see that there are some spikes, especially in the first part of the time span.

In Fig. 11 we plot the RK solution using as initial condition the predicted state $\tilde{Y}(t_{\max})$, where $t_{\max}$ is the time location of the maximum in $\mathcal{L}_{\text{ODE}}(t_{\max})$ is located (excluding the initial point). Interestingly, the PINN actually converges to a solution of the Lorenz system, but one that corresponds to a different initial condition choice. We may say that the PINN "goes astray" for just a few time steps, but then finds another solution that still satisfies the ODE conditions, but not the desired one.

We can explain this fact by considering some interesting features of chaotic systems. A chaotic system is characterized by a Lyapunov exponent, which tell how fast the distance in phase space between two solutions on the differential equations evolves in time from close initial conditions. The Lyapunov exponent implies also a Lyapunov characteristic time given by $T_L = \frac{1}{\lambda}$. This is a characteristic time of the chaotic system that can be roughly interpreted as an upper limit of the time that one can propagate two close initial conditions for a chaotic system and still consider them to be from the same IC. Inspired by this rough interpretation and by our observations that the PINN struggles to converge for long time spans, we come up with the idea of a Learning Schedule for PINN.

Learning Schedule for PINN

It is difficult to train our Lorenz PINN for a period of time $T_L \sim \frac{1}{\lambda}$, λ being the Lyapunov exponent. This is because when the time span T where we calculate the $\mathcal{L}_{\text{ODE}}$ is bigger than the typical Lyapunov time T_L, there is the risk that the network

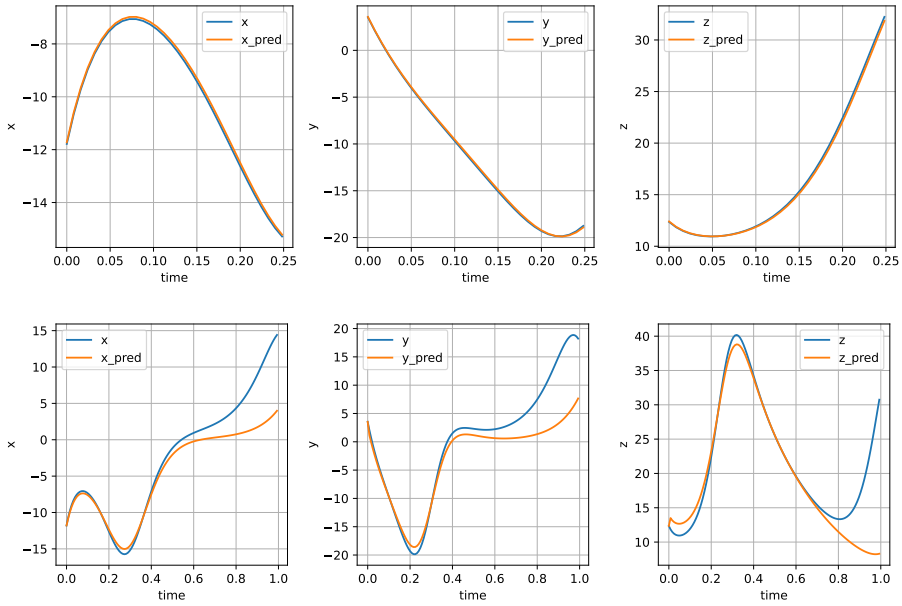

Figure 9: Three plots on top: PINN prediction on a shorter timespan of $T = (0.25)$. Three plots on bottom: PINN prediction on a longer timespan of $T = (0, 1)$.

"fails" for just a few points and falls on another "allowed" trajectory for the ODE, but not the intended one in terms of given IC.

This problem can be mitigated by training the network with a Learning Schedule (LS). Let Δt such that $\Delta t \ll \frac{1}{\lambda}$. Let also n be the number of mini time spans, equal to one at first.

- We provide the loss function points with time t such that $t \in (0, n\Delta t)$.
- We train the network until the loss is lower than a certain precision threshold p and we add one Δt to the time span, so $n \to n + 1$.

This method has only two hyperparameters, Δt, which have to respect the condition $\Delta t \ll \frac{1}{\lambda}$, and the threshold precision p. Since this method can be computationally expensive, some checkpoint methods and the possibility to restore and train the network in more sessions is implemented in `src/Lorenz_PINN_learning_schedule.ipynb`.

We tried the LS method to train again a PINN on the timespan $T = (0, 1)$. The MSE of the LS PINN is 2 orders of magnitude lower than the analogous model without LS. We can also qualitatively appreciate that solution obtained with the LS seems to be correct, suggesting that the method can be used to also obtain solutions for longer time spans. However, the clock time to obtain this solution was so long (approximately 3 days on a personal computer) that a deeper investigation of the hyperparameters $\Delta t, p$ was not performed.

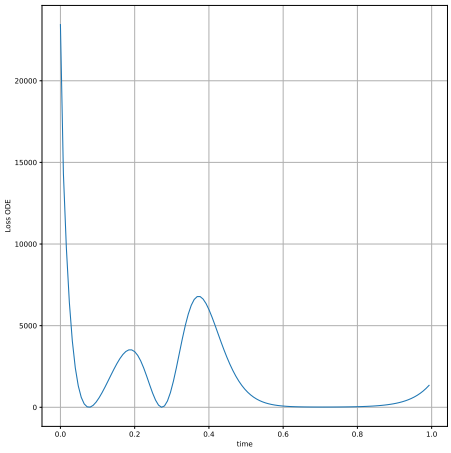

Figure 10: $\mathcal{L}_{\text{ODE}}(t)$, $0 \leq t \leq 1.0$ of the Lorenz PINN model.

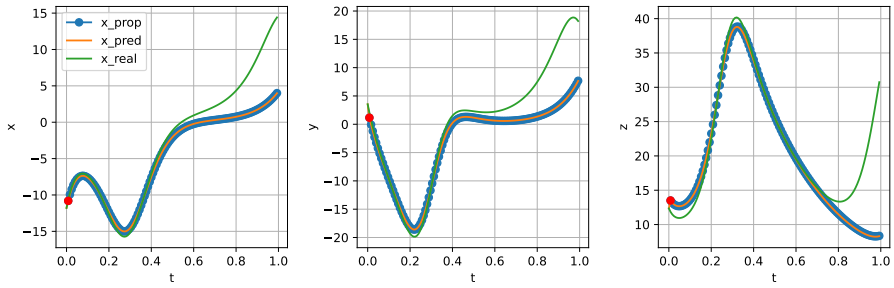

Figure 11: RK propagation of the predicted state with maximum $\mathcal{L}_{\text{ODE}}$ in Fig. 10 (excluding the first one).

4.2.2 RNN

We also carried out a study on the Lorenz attractor using a recurrent neural network (RNN). We chose an RNN with long short-term memory (LSTM) units as our model. Our RNN model consists of one LSTM layer with an `input_size` of 3, corresponding to the three-dimensional coordinates of the Lorenz system, and one linear layer for the outputs. We increased the number of units in the LSTM layer to 100 to enable the network to learn more complex structures in the data, details in `src/Lorenz_RNN.ipynb`. The model was fed the same sequence structure as for the DHO but with a `lookback` of 5. As for the DHO we did not employ a validation set.

We used the following parameters for the training:

- `lr`: 0.01
- `factor`: 0.995
- `patience`: 250
- `n_epochs`: 5000

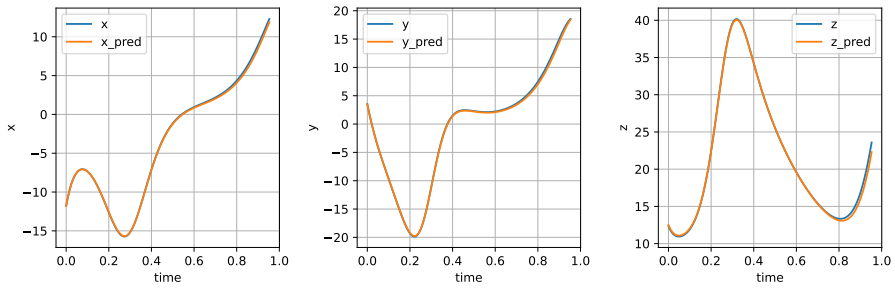

Figure 12: Learning Schedule PINN prediction, trained on $T = (0, 1)$.

	MSE (0,0.25)	MSE (0,1)
PINN short	0.00327	\
PINN long	\	21.5
LR PINN	\	0.067

After the model was trained, we compared the results with a RK numerical evaluation of the Lorenz system (see e.g. Fig. 14). The MSE for the test set can be found in table 2.

To compare the RNN's performance on Lorenz data to the PINN's, we trained the model on the same time interval $T = (0, 1)$. The RNN predictions begin at the `lookback` point, using a cycle of the prediction that originates from the starting points provided by the initial dataset. The RNN produces very accurate predictions, as shown in Table 2. One reason for this could be that the prediction is within the training interval. We can however still accept the results to some extent. In Fig. 13 one can see a direct comparison between the predicted and actual values within the range $T(0, 1)$. This visual representation provides a clear overview of the model's accurate predictions.

5 Discussion

As shown in the results section and in Fig. 5, the use of FFNNs to predict the time evolution of sequential data does not yield good results. This can be seen in the con-

model	train MSE	test MSE	comp. time (s)
RNN all	5.98e-05	6.453	1.1e-1
RNN 25%	”	0.446	”
RNN $(0, 1)$	”	9.675e-2	6.9e-2

Table 2: Performances for the RNN for the Lorentz data. The RNN 25% refers to the first 25% predictions, showing that at first the model predicts the test with a good accuracy and only after it deviates. RNN $[0, 1]$ refers to the prediction in the $[0, 1]$ interval, made for compare with the PINN results

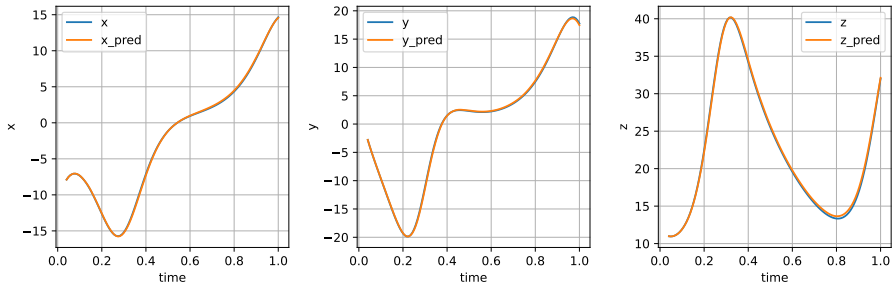

Figure 13: RNN prediction on a timespan (0,1)

text that FFNNs are fully-connected, and that their unbiased architecture does not make it easy to find sequential patterns. We see from the solutions that not only the FFNN fails to predict the damping, but also the oscillatory behavior, opting instead for a prediction that continues the curve from the training set and softly flattens. This problem can be solved by using PINNs, i.e. providing the FFNN with constraints that would make solutions obeying the system's differential equations preferable. We can see in Fig. 6 how the PINN greatly outperforms the FFNN when it comes to predictions, even if the two networks share the same hyperparameter and training epochs. We also showed how a hyperparameter optimization can increase the performance.

Our RNN model performed very well on the test data, with a smaller MSE compared to the FFNN and PINN baselines. However, it took more time for the RNN to make these predictions compared to the other models. This is because RNNs work by processing each step in the data one after the other, and it uses the outcome of the previous step to inform the next. This gives rise to increased computational times and prevents parallel computations. While the RNN performs well in terms of accuracy, the trade-off is large computation time. If we want to increase the speed of the RNN's predictions, we could consider using shorter sequences, although this might have an impact on the accuracy.

When it comes to modeling differential equations, the main question is if ML algorithms can outperform more traditional solving methods such as numerical evaluations with algorithms such as Runge-Kutta (RK). We are able to make the precision of the numerical methods arbitrarily small, given short enough time steps, but at the cost of computational time. For this reason, we wanted to compare the performance of our solutions to the RK algorithm by comparing their MSE errors and computation times. This comparison is plotted in Fig. 15, together with the errors and time of computation for the baseline FFNN and PINN (the two networks sharing hyperparameter values) and the properly-tuned PINN. We observe that if we do not require extreme precision and we are satisfied with an error of $\sim 10^{-6}$ (for a problem whose scale is around unity), the tuned PINN is a valid alternative to a RK evaluation, as its computation time is shorter by about 2-3 orders of magnitude. The PINN and the RK evaluations have an advantage, and it is knowing the physical laws governing the DHO system. This is not the case for either the FFNN or the RNN. As we see in Fig. 15, the RNN performs worse than the tuned PINN in terms of precision, and takes much longer time to evaluate because for every prediction a whole evaluation cycle has to take place. Nevertheless, this prediction is still quite precise, considering

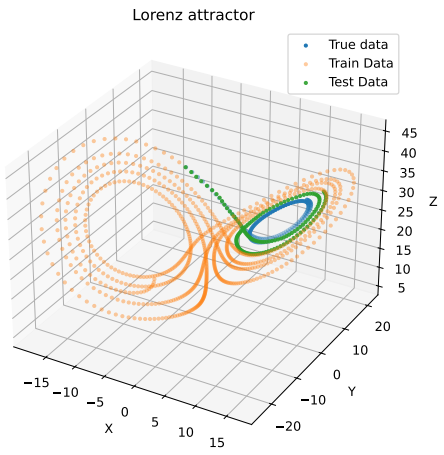

Figure 14: Prediction of the RNN model on the Lorenz data. The RNN is good at guessing the overall behavior of the function, but due to the chaotic nature of the Lorenz data even a small deviation or error in the initial prediction can magnify and result in a significant divergence from the true solution as time progresses.

that no differential equation was provided to the model. This would make RNNs powerful tools when the task is to predict the time evolution of (non-chaotic) sequential systems, when no information on its governing mathematical laws is available.

The main take-away from analyzing the models' performances when trying to predict the behavior of the Lorenz attractor, is that none is particularly good at solving the chaotic systems. Even a PINN, which knows the differential equations of the system, struggles to converge to the right solution for much smaller time periods compared to the DHO. We were able to mitigate the problem with the Learning Schedule method, finding a way to force the network to converge to the right solution, but the training is much slower than in the non chaotic example. The RNN can instead predict the solution over a considerably longer time period without increasing too much the training time compared to the DHO. However, while it produces predictions that are very close to the true solution for the initial steps, it deviates over longer times and falls into a pattern. To achieve a better match for longer time periods, one could opt for shorter time steps, even if it results in more computationally expensive calculations. This trade-off may be necessary to accurately capture the dynamics of the system over an extended period.

6 Conclusion

In this report, we describe our project comparing the performance of FFNNs, PINNs and RNNs on two time-dependent systems: one well-behaved and understood (the

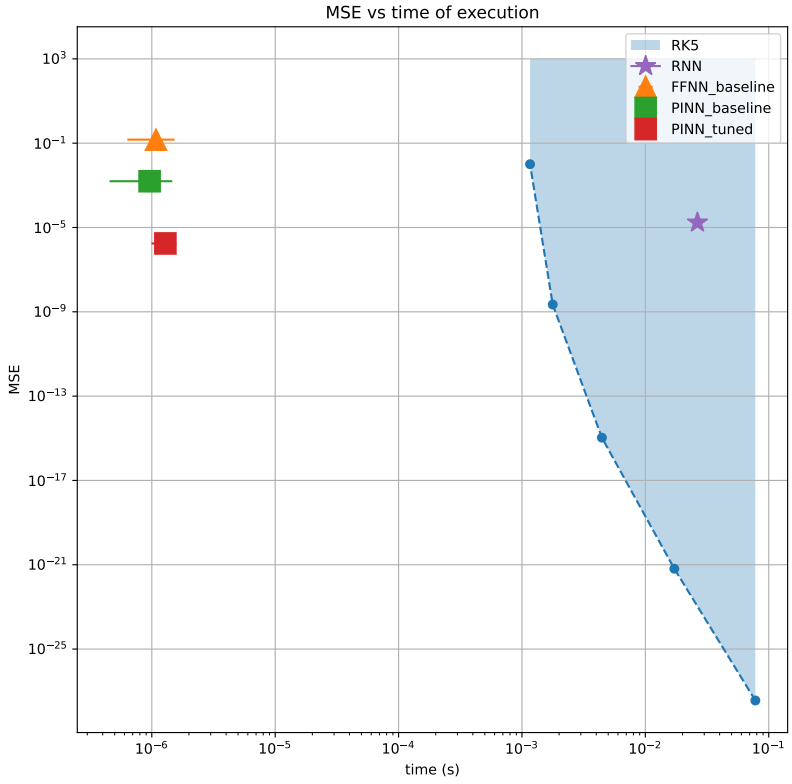

Figure 15: The FFNN, the PINN (tuned and with the same hyperparameters used for the FFNN) and the RNN errors and computation times are compared with the more traditional numerical evaluation of differential equations, here the Runge-Kutta algorithm. Although the Runge-Kutta may yield an arbitrarily good solution given enough computation time, the tuned PINN already performs much better and faster at evaluating differential equations, when extreme precision is not a concern. The RNN, although performing relatively slowly, yields a good guess given that information about the differential equation of the system was not provided. The MSE is obtained by comparing with the analytical solution.

damped harmonic oscillator), and one chaotic system (the Lorenz attractor). We first observe how the FFNN, although matching the DHO training data very well, performs poorly when trying to predict the future behavior of datasets with a strong sequential character. The test predictions do not mirror the actual behavior of the system. Besides of course the numeric methods for solving the differential equation describing

the system (as RK), we investigated two other NN methods. With the PINN, the precision is largely enhanced and the computation time, once the model is trained, is comparable to the one of the FFNN (and much less than a simple Runge-Kutta evaluation for similar MSE on the predicted data). When information on the physical behavior of the system is not available and the differential equations describing it are unknown, RNNs may provide a relatively computationally heavy alternative, but which is capable of producing predictions with slightly higher uncertainty than a well-tuned PINN. When it comes to modeling a chaotic system such as the Lorenz attractor, the situation becomes dire for all three networks. While the PINN is able to predict over a much shorter time frame compared to the DHO case (albeit at a high computational expense), the RNN can predict the solution over a considerably longer time period. However, while the RNN produces predictions that are very close to the true solution for the initial steps, over longer times it deviates and falls into a pattern. Shorter time steps may solve the problem, at the expense of more computation time.

References

[1] F Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X.

[2] Edward N. Lorenz. "Deterministic Nonperiodic Flow". In: *Journal of Atmospheric Sciences* 20.2 (1963), pp. 130–141. DOI: 10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/atsc/20/2/1520-0469_1963_020_0130_dnf_2_0_co_2.xml.

[3] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274.

[4] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[5] Lisha Li et al. "Massively Parallel Hyperparameter Tuning". In: (2018). URL: https://openreview.net/forum?id=S1Y7OOlRZ.

[6] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[7] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[8] B Moseley. "Physics-informed machine learning: from concepts to real-world applications". PhD thesis. University of Oxford, 2022. URL: https://ora.ox.ac.uk/objects/uuid:b790477c-771f-4926-99c6-d2f9d248cb23.