Forecasting Chaos with Neural Networks

Timur Zhanabaev

MAST 680 - Assignment 3 - April 10, 2023

Abstract

In this assignment, a Recurrent Neural Network (RNN) is used to forecast the future steps of a Lorenz dynamical system. The model learns the dynamics for variable system parameters $\rho \in \{10, 28, 40\}$, and then tested with unseen system parameters $\rho \in \{17, 35\}$, where model performs better for stable chaotic parameters. The effects of time-scale is investigated, where a coarser grain data provides longer forecasting power. The final component is then to predict lobe transitions of the stable chaotic system ($\rho = 28$) with a simple neural network. The model is able to fit the states that determine the transitions states equally well for both training data and unseen data.

1 Introduction and Overview

For the final assignment, the theory of Neural Networks is briefly presented with the model of choice being a Recurrent Neural Network, taking the output of the model as input to learn the time progression of a dynamical system. As well, a brief introduction to chaotic system is given, with the system of interest being the Lorenz system. For the training data, the systems are simulated for various system parameters using the ODE solver Python package `scipy`, while the RNNs are built using the `TensorFlow` framework in Python. Further, the Lyapunov times are measured for each neural network as a measure of performance. For the final task, setting the system parameter to $\rho = 28$, a neural network is trained to predict the transitions from one lobe to another in the chaotic Lorenz system.

2 Theoretical Background

2.1 Neural Networks Introduction

The neural networks were popularized initially from the first attempts of modeling a biological neural circuit, where each neuron's activation influences the activations of neurons in it's pathway [1]. In essense, a neural network is nothing but a sequence of compositions of non-linear transformations with coefficients that fits our dataset, i.e. by finding a solution to coefficients that minimizes a metric between predicted and training data. Such models are flexible, as they extend linear regression to regression of non-linearly transformed data within each layer of the network, at the expense of losing interpretability of our coefficients.

2.2 Feed-forward Neural Networks

Given an input $x \in \mathbb{R}^M$ (M dimensions), the first layer's output is defined to be a non-linear or linear transformation of the input,

$$g_1(x) = V\sigma(W_1 x + b_1)$$

The **weights** of this mapping are then $W_1 \in \mathbb{R}^{n \times M}$, where n is the **width** of the layer 1 and the **bias** $b_1 \in \mathbb{R}^n$. Not to mention, the activation function $\sigma(\cdot)$ is performing the non-linear transformation. Some popular activations functions include; relu, tanh, sigmoid, etc.

With a neural network of **depth** $= d$, which determines the number of layers in the network, we use the output of previous layer as the input to next layer, effectively performing function compositions for each layer. The full neural network's output is then defined as,

$$g(x) = g_d(g_{d-1}(...g_1(x)) = g_d \circ g_{d-1} \circ ... \circ g_2 \circ g_1(x)$$

Now we specify flexibility to each layer, allowing differing activation functions and widths for each layer j to be n_j. The weights and biases are then $W_j \in \mathbb{R}^{n_{j+1} \times n_j}$ and $b_j \in \mathbb{R}^{n_j}$ (while keeping matching input output dimensions between layers). The output of the neural network is our response y that we wish to predict. The model is then defined as,

$$y = \sigma_d(W_d \sigma_{d-1}[W_{d-1}...(W_2 \sigma_1[W_1 x + b_1] + b_2)... + b_{d-1}] + b_d) \tag{1}$$

For dynamical systems, we wish to predict the next time step. Given the time series dataset, the predictors matrix is,

$$X = \begin{bmatrix} X_1 & X_2 & \dots & X_{K-1} \end{bmatrix} \in \mathbb{R}^{M \times (K-1)}$$

the response is then,

$$Y = \begin{bmatrix} X_2 & X_3 & \dots & X_K \end{bmatrix} \in \mathbb{R}^{M \times (K-1)}$$

The output from the neural network is then, $x_{k+1} = g(x_k)$.

To train the model, the difference between our network output and true response is to be minimized. The objective function is chosen to be the sum of squared error (SSE). For the next time step prediction, the objective function is,

$$\mathcal{L} = \frac{1}{2(K-1)} \sum_{k=1}^{K-1} ||X_{k+1} - g(X_k)||_2^2 \tag{2}$$

The optimization is performed with gradient descent, requiring the computation of chain-rule derivatives of the neural network.

2.3 Recurrent Neural Networks

As the next step to improving the artificial NNs for sequential data, a Recurrent NNs uses the idea of taking output of the network at time $t-1$ as the input to the same model predicting for time t. One hopes then to train a neural network that will map a sequence of data from an initial condition to future time. To incorporate the recurrent passage of information, the output of the RNN is then fed back into the RNN again as. This is done over s times, allowing the RNN to take initial input X_k and find weights β that best map the system s time steps away at X_{k+s},

$$g^s(X_k) = g^{(s)} \circ g^{(s-1)} \circ ... \circ g^{(1)}(X_k) \tag{3}$$

The loss or objective function for this system is then,

$$\mathcal{L} = \frac{1}{2S(K-1)} \sum_{s=1}^{S} \sum_{k=1}^{K-1} ||X_{k+s} - g^s(X_k)||_2^2 \tag{4}$$

2.4 Lorenz System

The chaotic dynamical system, the Lorenz system, a system of ordinary differential equations first developed to model *atmospheric convection*. Also known for the famous butterfly shape it produces for some specific system parameters [2]. The Lorenz equations governing the system are as follows,

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = x(\rho - z) - y \\ \frac{dz}{dt} = xy - \beta z \end{cases}$$

For simulations, the 2 parameters σ and β are set to 10 and 8/3, respectively. The training data is then simulated with variable ρ parameters, in order to train a network with ρ as input.

3 Algorithm Implementation and Development

3.1 Simulation

First off, the Lorenz system is simulated using the `scipy`'s ODE solver for system parameters $\rho = \{10, 28, 40\}$ which are later used as input for the neural network. For the time-scale of the systems, the interval for time is $t \in [0, 100]$ at $\triangle t = 0.01$, giving a total of 10001 data points for each ρ. The figures (1) showcase the Lorenz attractor simulations, where Lorenz system with $\rho = 28$ is known to be a chaotic system.

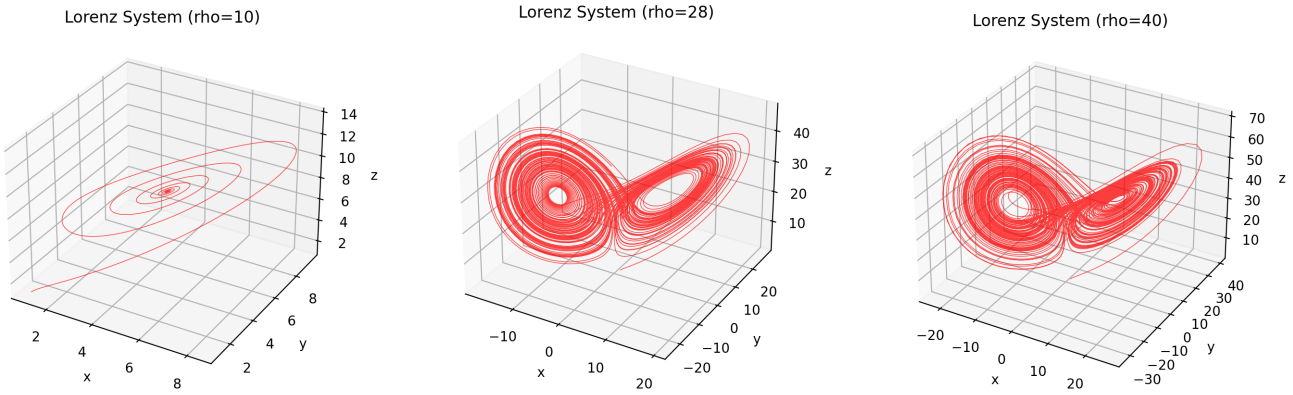

Figure 1: Lorenz attractor: $(\sigma = 10, \beta = 8/3)$; (left) $\rho = 10$, (middle) $\rho = 28$, (right) $\rho = 40$.

3.2 RNN Model Architecture

For the neural network implementation, a Recurrent Neural Network is implemented as introduced in Section 2.3 using the `TensorFlow` package. Most of the code is taken from Professor Bramburger's code repository forecasting the Henon map [1].

The model takes input (x, y, z, ρ) of 4 dimensions, the spatial dimensions of the system and the system parameter ρ. Hopefully this allows for variable ρ values, although this would not be directly related to the dynamical system's parameter, but as an indicator variable that trains (x, y, z) for such different ρ values.

Forecasting Model

The model built for forecasting contains 4 fully connected layers (`Dense` layers) with **width** of 50 for each layer and *SELU* activation functions. The loss functions include; MSE, MAE and Huber loss.

SELU activation:

$$f(x) = \lambda\alpha(e^x - 1)\mathbb{1}\{x \leq 0\} + \lambda x\mathbb{1}\{x > 0\}$$

Huber loss function:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2\mathbb{1}\{|y - \hat{y}| \leq \delta\} + \delta \cdot \left(|y - \hat{y}| - \frac{1}{2}\delta\right)\mathbb{1}\{|y - \hat{y}| > \delta\}$$

Lobe Prediction Model

To begin, a lobe transition is defined as $y_t = 0$ when $(x_{t-1}, y_{t-1}, z_{t-1})$ state is in the same lobe as (x_t, y_t, z_t) and $y_t = 1$ when (x_t, y_t, z_t) lobe has changed from $(x_{t-1}, y_{t-1}, z_{t-1})$ lobe. Since the response of the model is the lobe transition, several changes are made to the model. The change is made to the *output* layer of the model. The activation function is set to the *sigmoid* function, modelling probabilities of having a transition happening at the spatial state (x, y, z), that is $\hat{p_t} \in [0, 1]$, The model is then trained for data fixed only for $\rho = 28$, with the cutoff to detect it as a transition set to 0.5. The predicted transitions are then $\hat{y}_t = \mathbb{1}\{\hat{p}_t > 0.5\}$.

3.3 Experiments

1. For the first experiment, the Huber loss, MSE (Mean Squared Error) and MAE (Mean Absolute Error) are benchmarked with the Lyapunov time as the metric of performance for the trained models with ρ set to 28 (excluding $\rho = 10, 40$ to reduce training time)

2. For the second experiment, the training data is now 3 simulations of the Lorenz system for system parameters $\rho = 10, 28, 40$, and predictions are made for system parameters $\rho = 17$ and 35.

3. For the third experiment, the model is used to re-train the chaotic system for variable time increments. Two additional experiments are made over $t \in [0, 50]$ at $\triangle t = 0.005$ (10001 data points) and the second over $t \in [0, 250]$ at $\triangle t = 0.025$ (10001 data points)

4. The final fourth experiment involves training a simple neural network to predict transitions from one lobe to another. The states containing transitions are discriminated by the x-axis, where passing through $x = 0$ is a transition from one lobe to another.

4 Computational Results

4.1 Loss Function Experiments

The RNN models are trained for 20000 epochs each to evaluate loss function with 3 steps of network compositions. The Huber loss has shown the best forecasting performance ($1.449s$ Lyapunov time) of the Lorenz System set to $\rho = 28$, compared to MSE ($0.779s$) and MAE ($0.829s$). The loss of choice for next models is then taken to be the Huber loss.

The simulated data and the model forecasting from an initial condition $(x_0, y_0, z_0) = (5, 5, 10)$ are presented in Figures (2) for the Huber loss. Training the model on one system parameter works well, although changing the initial condition to $(x_0, y_0, z_0) = (1, 1, 1)$ lowers the model's performance. The Lyapunov time or the forecasting time is reduced to $0.07s$, shown in the Appendix Figure **??**. One main observation is that the neural network is able to learn the dynamics of the system, even if not following the exact path, it has the general shape of the butterfly as shown in Figure. 3.

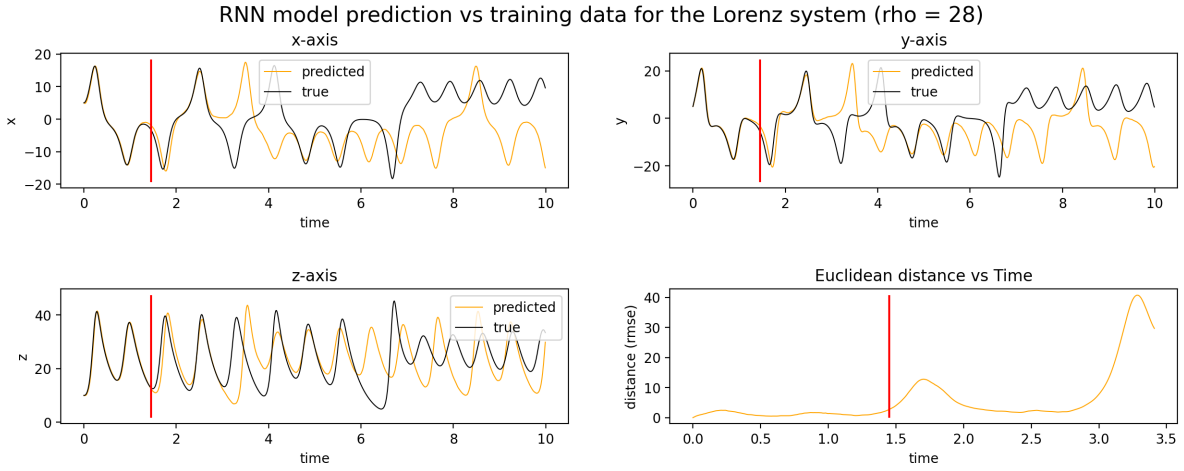

Figure 2: Lorenz System ($\rho = 28$, $(x_0, y_0, z_0) = (5, 5, 10)$): RNN forecast (orange) and training data (black) for 3 axes. As well as distance between predicted and training data (bottom right). The red vertical lines are markings for Lyapunov time.

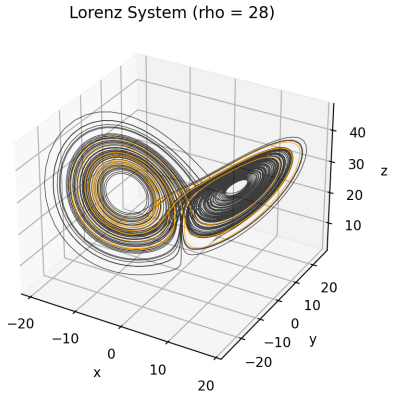

Figure 3: Simulated data (black) and RNN predictions (orange), showcasing the network's ability to learn the chaotic dynamics of the system.

4.2 Training RNN on a Larger Dataset

Having the models trained data with parameters $\rho = 10, 28$ and 40, the resulting Lyapunov times for seen data was calculated to be $0s$, $1.42s$ and $0.11s$, respectively. While for unseen data, for parameters $\rho = 17$ and 35, the Lyapunov times was calculated to be $0.03s$ and $0.69s$. It is evident then the model performs better for stable chaotic systems ($\rho = 28$), as those systems do not converge to some fixed point, essentially eliminating all the training data. For the stable systems,

our networks then have a bigger region of the space covered to learn the dynamics, improving the overall performance for stable system parameters.

4.3 Varying Training Data Time Scale

For the fine-grain time scale $t \in [0, 50]$ at $\triangle t = 0.005$ and for each system parameter $\rho = 10, 17, 28, 35, 40$ the calculated Lyapunov times are $0.83s, 0.005s, 1.585s, 0.625s$ and $1.21s$, respectively. For the second coarse-grain time scale $t \in [0, 250]$ at $\triangle t = 0.025$ and for each system parameter $\rho = 10, 17, 28, 35, 40$ the calculated Lyapunov times are $0.725s, 0.15s, 3.925s, 1.3s$ and $2.625s$, respectively. We observe then that the performance stays the same relative to each parameter, but increases for coarser grain time scale.

4.4 Lobe Transition Prediction

The training data for the lobe transitions are done for $\rho = 28$ with initial condition $(x_0, y_0, z_0) = (5, 5, 10)$ and the test generated from initial condition $(x_0, y_0, z_0) = (4, 4, 27)$. For training data, the lobe transitions are plotted in Figures (4), where $49/53$ transitions are labelled correctly. As well, for test data in Figures (4) $50/53$ transitions are labelled correctly.

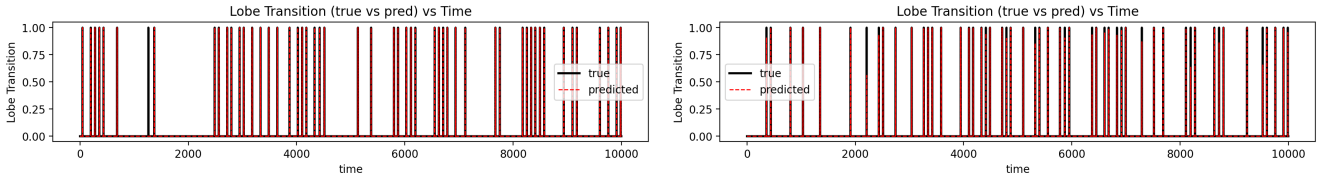

Figure 4: (Left) Training data transitions (black) and NN classification (red). (Right) Test data transitions (black) and NN classification (red).

5 Summary and Conclusions

To conclude the observations, the recurrent neural network is able to learn the dynamics of a stable chaotic system, although this performance requires training the model on more initial conditions to completely cover the training space. Next, allowing the model to train on system parameters may imbalance the training data, where if most data is trained on stable system parameters, the unstable systems may remain stable. Moreover, the coarser-grained time-series was found to perform better at forecasting over longer ranges. Finally, a network can successfully classify states labeled as transitory.

References

[1] J. J. Bramburger, Data-Driven Methods for Dynamic Systems, 2023.

[2] Wikipedia contributors. "Lorenz system." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 30 Mar. 2023. Web. 7 Apr. 2023.

Appendix A.1 Premade functions used and brief implementation explanation.

1. `odeint`: from scipy package, solve an ODE over a time range.

Appendix B. Computer codes