MATH 494-A: Assignment 3
Forecasting Chaos with Recurrent Neural Networks

Myriam Boucher-Pinard (40109509)

April 10, 2023

Abstract

This study has three specific aims: to determine the loss function leading to the best predictions; to determine with how many time steps the prediction remains accurate; and to see whether the RNNs properly learned to predict data for values of parameter ρ other than the training data. This is achieved through the use of ordinary differential equations, particularly the Lorenz equations, to train recurrent neural networks (RNNs) to take data from one time step to the next using the parameter ρ. As the only tested loss function, the predictions for taking data from one time step to another were that of the mean square error (MSE).

1 Introduction and Overview

There are not many differential equations for which the solution may be found easily; indeed, only a few may be computed by hand, and these are the ones that must be used to train neural networks (NNs). Without these solutions, it would not be possible to know whether the NN solutions were correct. Differential equations often have several solutions and the initial conditions shall lead to drastically diverging solutions. This makes forecasting data quite difficult, as the nature of these equations means the motion is often chaotic. As such, this study is composed of using the Lorenz equations, which are ordinary differential equations for which we do have the solutions, to train recurrent neural networks (RNNs). This training has three aims, more particularly: to determine the loss function which produces the best predictions in sending data from one time step to the next; to determine how many steps further data may be predicted, with enough accuracy that the approximated function is not completely diverging; and to use parameters that were not a part of the training data for the RNNs and see whether our predicted data remains accurate.

2 Theoretical Background

2.1 The Lorenz Equations

The mathematical understanding of the Lorenz equations draws from its origins in weather systems.

$$\begin{aligned}\frac{dx}{dt} &= 10(y-x)\\ \frac{dy}{dt} &= x(\rho - z) - y\\ \frac{dz}{dt} &= xy - \frac{8}{3}z\end{aligned} \tag{1}$$

When this model was conceived, meteorological data was at the forefront of this application. Most importantly, x, y, and z in (1) are not coordinates in space, but rather convective motion and temperature change. Thus, even for this study, although the focal point is not a weather system, it is important to consider x, y, and z as variables of change. Furthermore, the values 10 and $\frac{8}{3}$ (what would have been σ and β, respectively) (1) were found to be suitable values representing the chaotic dynamics of the weather

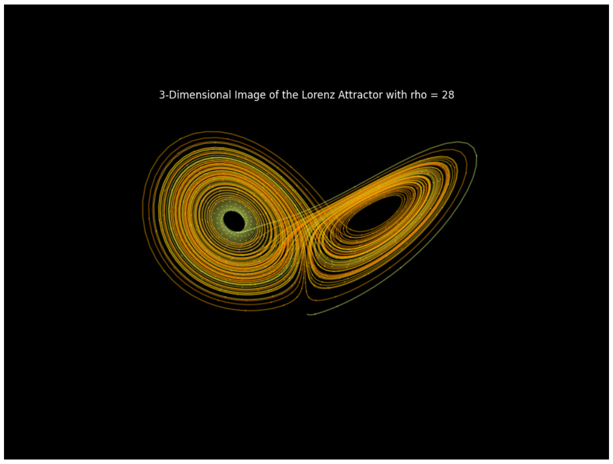


Figure 1: 3-Dimensional Image of the Lorenz Attractor with rho = 28.

system. Thus remains ρ (1), the difference between the top and bottom of the plane (for temperature with respect to the weather system) [2].

The observable effect of the Lorenz model is in its Lorenz attractor, which is also known as the Butterfly Effect, pictured in figure 1. This chaotic behaviour is non-periodic, for ρ belonging to [24.74, 30.1), in that the trajectory borrowed by the model never repeats itself. Thus, accurately predicting the system is nearly impossible very far into the future, and thus may only be done somewhat accurately in the near future (as observed in weather systems) [2].

2.2 (Recurrent) Neural Networks

The process of understanding neural networks begins with some background on function approximators. Indeed, the goal of a function approximator is precisely that: to approximate functions. It is accomplished using simple functions listed in a piecewise manner, such that the behaviour of the entire function is approximated with simpler pieces of simple functions [1].

$$y = \sigma(Wx + b) \tag{2}$$

$$x_{n+1} = x_n - \gamma \nabla f(x_n) \tag{3}$$

$$L = \frac{1}{2K} \sum_{k=1}^{K} ||Y_k - g(X_k)||_2^2 \tag{4}$$

This non-linear process is precisely that which a neural network (NN) is; function approximation using the universal function approximator Rectified Linear Unit (ReLU). Using but one piecewise function, the NN pushes data from one time step to the next while minimizing the loss function. This aforementioned loss function serves as a measure of how close prediction is to the actual function [1]. Machine learning and NNs operate on function composition (2). This means that the composition of this equation leads to a more adjusted function approximation, by creating layers of weights (W) and biases (b), and determining the optimal values for these two parameters in order to make the best predictions. These layers then form the neural network [1].

Thus, the loss function interpolates between that which goes in and that which comes out, both of which we know. The loss function is therefore based on minimizing using gradient descent (3), rather than the ascent, such that the γ parameter is calibrated to take the correct step size towards the minimum, effectively maintaining an accurate approximation. The gradient descent equation (3) uses the true function, while the loss function uses the approximated function. For this study, the mean square error (MSE) loss function (4) shall be used below [1].

The final component to generate a sound mathematical background for this study is the more specific recurrent neural network (RNN). Rather than applying the loss function once, the RNN uses sequential data to augment the loss function. Lyaponov time, named after the mathematician to define it, shall then determine how quickly the system will diverge, effectively giving us the number of steps in which our system is accurate [1].

3 Algorithm Implementation and Development

3.1 Forward Propagation

The first component of the study was to build the test data that would be at the core of the neural network, to achieve forward propagation. This was accomplished by defining the Lorenz equations using the same values as the weather system, with σ and β equal to 10 and $\frac{8}{3}$, respectively [2]. The values for ρ were defined as 10, 28, and 40, in order to build data for below, within, and above the chaotic threshold of these equations. Meanwhile, the initial conditions were set arbitrarily to $(x, y, z) = (0, 1, 1)$. Following this, the computed data was stacked into a single (30000, 4) matrix, the first 10000 rows (for the number of time steps) belonging to $\rho = 10$, and so on, and the columns representing x, y, z, and ρ.

Then, the data was restacked into a Henkel matrix, with rows representing the data length and the columns each representing a step forward in time. The feedforward NN was fitted with four hidden layers and 80 neurons.

3.2 Backward Propagation

The second component of the study was to then train the neural network on the compiled test data. First, the gradient descent was defined, with a piecewise decay of the learning rate. This is done in order to move towards the minimum without stepping over it. Then, the loss function MSE was defined, the use of this loss function due to its familiarity and lack of time to experiment with more. The loss function garnered a loss of 2 x 10^{-3}, which is comparable to the loss discussed with our peers.

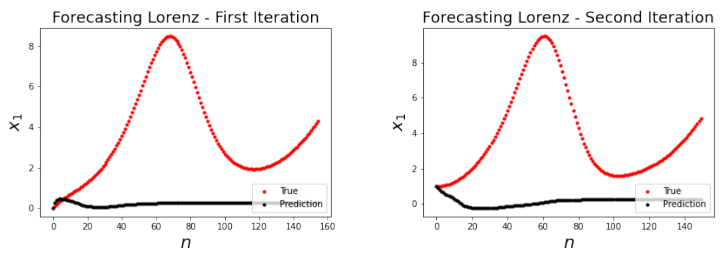
The final portion of this study required an examination of the capacity of the NN to run the data for ρ different than the training data. The data was then forecasted for ρ equal to 17 and 35, with the results below.

4 Computational Results

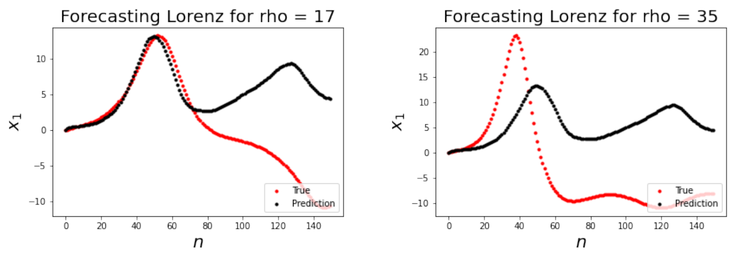
Overall, while this study was not a definitive success, some results may be drawn despite the lack of time for experiment. Although perhaps not the best predictor, the error computed, as mentioned above, was comparable to our peers' error, such that the prediction may still be quite accurate due to the minimized gradient descent. Unfortunately, however, the predictions severely lacked accuracy.

The above figures 2 and 3 are a representation of such apparently minimized loss function still incapable of predicting the chaotic nature of the Lorenz equations. Indeed, the prediction is barely accurate for a single time step. As such, perhaps a future direction for this study would be to indeed experiment with other loss functions to obtain a more accurate prediction of the Lorenz equations. Thus, taking a further prediction of the data for a different Δt would have given very little accuracy. Unfortunately, we remain uncertain as to where the issue lies, as the model was run numerous times, and the code has been viewed by our peers with their feedback being positive.

Conversely, the data was tested on the untrained values of ρ, being 17 and 35. Surprisingly, the predictions for both ρ were much closer to the true system, for approximately 10 time steps, at least for the first



Figures 2 and 3: Forecasting Lorenz with Neural Networks, in the first iterate (left) and the second iterate (right).



Figures 4 and 5: Forecasting Lorenz with Neural Networks, in the first iterates of $\rho = 17$ (left) and $\rho = 35$ (right).

iterate, pictured in figures 4 and 5.

Time permitting, it would have been possible to observe when a transition from one lobe to the next is imminent within the NN, and thus create a prediction on this transition. We recognize that this should be a future direction for this study.

5 Summary and Conclusions

To conclude, the goal of the study was to model the motion of the Lorenz equations using recurrent neural networks. Initially, we had to train the NN using the data compiled from the Lorenz equations, to then determine the loss function leading to the best predictions. Here the MSE was used in order to obtain the results. Then, although time lacking, the following goal was to determine within how many time steps the prediction remained accurate. The final component was to see whether the recurrent neural networks properly learned to predict data for values of 17 and 35 for parameter ρ which were not part of the training data. Although this study did not obtain the desired results, neural networks remain a powerful tool in function approximation.

References

[1] Jason Bramburger. *Data-Driven Methods for Dynamic Systems: A new perspective on old problems.* 2023.

[2] University of Waterloo - Department of Mathematics. *Chaos Theory and the Lorenz Equation: History, Analysis, and Application.* 2012.

Appendix A Python Premade Functions

- `np.empty` Generates an empty array.

- `np.zeros` Generates an array of zeros.
- `np.linspace` Returns evenly spaced values over the specified interval.
- `odeint` Solves an ordinary differential equation through integration.
- `plt.plot` Generates a plot (plt has many applications, for instance, generating labels and legends).
- `np.save` Saves a matrix as a file.
- `np.load` Loads a saved matrix back into the program.
- `.add` Adds the specified element to the set.
- `.gradient` Computes the gradient of the specified element.
- `tf.GradientTape` Provides the derivatives for the trainable variables (there are many TensorFlow functions that have been used to define this neural network, such as optimizers (for the minimized loss function)).
- `np.append` Stacks an array into an existing array.

Appendix B Python Code

```
from scipy.integrate import odeint
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

tf.version.VERSION # This verifies the version of Tensorflow.

## LORENZ PARAMETERS AND INITIAL CONDITIONS:
# The parameter rho shall have the values of 10, 28, and 40 (and later 17 and 35).

sigma = 10
rho = 28
beta = 8/3

x0 = 0
y0 = 1
z0 = 1

# The initial conditions were selected based on Christian Hill (2016) -
↪  https://scipython.com/blog/the-lorenz-attractor/.

## THE LORENZ EQUATIONS:

def Lorenz(X, t, simga, beta, rho):
    x,y,z = X
    dxdt = -sigma * (x - y)
    dydt = x * (rho - z) - y
    dzdt = (x * y) - (beta * z)
    return dxdt, dydt, dzdt

# Maximum time point and total number of time points:

tmax = 100
n = 10000

# Interpolate solution onto the time grid, t:

t = np.linspace(0, tmax, n)

# Integrate the Lorenz equations:
```

```python
sol = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))

print(sol.shape)

# There are three columns: x, y, z.

## CREATING THE IMAGE OF THE LORENZ ATTRACTOR:
# Based on Christian Hill (2016) - https://scipython.com/blog/the-lorenz-attractor/.

WIDTH, HEIGHT, DPI = 1000, 750, 100

fig = plt.figure(facecolor = 'k', figsize = (WIDTH/DPI, HEIGHT/DPI))
ax = plt.axes(projection='3d')
ax.set_facecolor('k')
fig.subplots_adjust(left=0, right=1, bottom=0, top=1)

x = sol[:, 0]
y = sol[:, 1]
z = sol[:, 2]

s = 10

cmap = plt.cm.Wistia

for i in range(0,n-s,s):
    ax.plot(x[i:i+s+1], y[i:i+s+1], z[i:i+s+1], color=cmap(i/n), alpha=0.4)

ax.set_axis_off()

plt.title("3-Dimensional Image of the Lorenz Attractor with rho = 28", color = "white", x = 0.5, y = 0.85)
plt.savefig('lorenz_28.png', dpi=DPI)
plt.show()

# Note: images of the attractor have been made for rho = 10, 28, 40.

## PARTICULAR SOLUTIONS: RHO = 10

rho = 10
sol10 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
np.save("sol10", sol10)

## PARTICULAR SOLUTIONS: RHO = 28

rho = 28
sol28 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
np.save("sol28", sol28)

## PARTICULAR SOLUTIONS: RHO = 40

rho = 40
sol40= odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
np.save("sol40", sol40)

## DATA PROCESSING TO FEED THE MODEL: Merci, Nada !

mat10 = np.load("sol10.npy")
mat28 = np.load("sol28.npy")
mat40 = np.load("sol40.npy")

data = np.empty((30000, 4))
print(data.shape)

# Four columns, for x, y, z, and rho.

# Stacking all of our data from each rho into one single matrix:

data[0:10000, 0:3] = mat10
```

```
data[10000:20000, 0:3] = mat28
data[20000:30000, 0:3] = mat40
data[0:10000, 3:4] = 10
data[10000:20000, 3:4] = 28
data[20000:30000, 3:4] = 40

# Here we create matrices xn and xnp1 where xnp1 represents the forecast of xn one step into the future:
xn = np.empty((29997, 4))
xn[0:9999,0:4] = data[0:9999,0:4]
xn[9999:19998,0:4] = data[10000:19999,0:4]
xn[19998:29997,0:4] = data[20000:29999,0:4]

xnp1 = np.empty((29997,4))
xnp1[0:9999,0:4] = data[1:10000,0:4]
xnp1[9999:19998,0:4] = data[10001:20000,0:4]
xnp1[19998:29997,0:4] = data[20001:30000,0:4]

## DEFINING THE NEURAL NETWORK:

def init_model(num_hidden_layers = 4, num_neurons_per_layer = 80):

    # Initialize a feedforward neural network:

    model = tf.keras.Sequential()

    # Input is 2D - for each component of the Henon output:

    model.add(tf.keras.Input(4))

    # Append hidden layers:

    for _ in range(num_hidden_layers):
        model.add(tf.keras.layers.Dense(num_neurons_per_layer,
            activation=tf.keras.activations.get('relu'),
            kernel_initializer='glorot_normal'))

    # Output is 2D - for the next step of Henon map:

    model.add(tf.keras.layers.Dense(4))

    return model

## DEFINING THE LOSS FUNCTION:

def compute_loss(model, xn, xnp1, steps):

    loss = 0

    xpred = model(xn)

    loss += tf.reduce_mean(tf.square(xpred - xnp1))

    return loss

## COMPUTING THE GRADIENT:

def get_grad(model, xn,xnp1, steps):

    with tf.GradientTape(persistent=True) as tape:
        # This tape is for derivatives with respect to trainable variables.
        tape.watch(model.trainable_variables)
        loss = compute_loss(model, xn,xnp1, steps)

    g = tape.gradient(loss, model.trainable_variables)
    del tape

    return loss, g
```

```
## INITIALIZE MODEL (TILDE U):

model = init_model()

# We choose a piecewise decay of the learning rate, i.e., the step size in the gradient descent type algorithm.
# From 0 - 999, learning rate = 0.01; from 1000 - 3000, learning rate = 0.001; from 3000 onwards, learning rate
↪  = 0.0005.

lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay([1000,3000],[1e-2,1e-3,1e-4])

# The learning rate is decreased along the way to decrease the size of the set, in order to ensure the minimum
↪  is not missed.

# Choose the optimizer:

optim = tf.keras.optimizers.Adam(learning_rate=lr)

from time import time

steps = 4

# Define one training step as a TensorFlow function to increase speed of training:

@tf.function
def train_step():

    # Compute current loss and gradient with respect to parameters:

    loss, grad_theta = get_grad(model, xn, xnp1, steps)

    #Perform gradient descent step:

    optim.apply_gradients(zip(grad_theta, model.trainable_variables))

    return loss

# Number of training epochs:

N = 10000
hist = []

# Start timer:

t0 = time()

for i in range(N+1):

    loss = train_step()

    # Append current loss to hist:

    hist.append(loss.numpy())

    #Output current loss after 50 iterates:

    if i%50 == 0:
        print('It {:05d}: loss = {:10.8e}'.format(i,loss))

#Print computation time:

print('Computation time: {} seconds'.format(time()-t0))

M = 1000

xpred = np.zeros((M,4))
xpred[0] = xn[0,:]

for m in range(1,M):
    xpred[m] = model(xpred[m-1:m,:])

#Plot Henon Trajectories:

fig = plt.figure()
plt.plot(xn[:155,0],'r.', label = "True")
plt.plot(xpred[:155,0],'k.', label = "Prediction")
plt.title('Forecasting Lorenz - First Iteration', fontsize = 18)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)
plt.legend(loc = 'lower right', prop = {'size':10})
plt.savefig("Forecast 0")

#Plot Henon Trajectories:

fig = plt.figure()
plt.plot(xn[:150,1],'r.', label = "True")
plt.plot(xpred[:150,1],'k.', label = "Prediction")
plt.title('Forecasting Lorenz - Second Iteration', fontsize = 18)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)
plt.legend(loc = 'lower right', prop = {'size':10})
plt.savefig("Forecast 1")

#Testing the other rho values:

rho = 17
sol4 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol4[:,2], sol4[:, 0], label='x_coord')

rho = 35
sol5 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol5[:,2], sol5[:, 0], label='x_coord')

np.save('solutions_rho17', sol4)
np.save('solutions_rho35', sol5)

matrix_rho17=np.load('solutions_rho17.npy')
matrix_rho35=np.load('solutions_rho35.npy')

#rho 17 data
size_input = matrix_rho17.shape[0]*3
dataRho17 = np.empty((size_input,4))
dataRho17[0:10000,0:3] = matrix_rho17
dataRho17[0:10000,3:4] = 17

M = 10000

xpred = np.zeros((M,4))
xpred[0] = dataRho17[0,:]

for m in range(1,M):
    xpred[m] = model(xpred[m-1:m,:])

#Plot Henon Trajectories for rho 17
fig = plt.figure()
plt.plot(dataRho17[:150,0],'r.', label = "True")
plt.plot(xpred[:150,0],'k.', label = "Prediction")
plt.title('Forecasting Lorenz for rho = 17', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)
plt.legend(loc = 'lower right', prop = {'size':10})
plt.savefig("Forecast 17")

#rho 35 data
size_input = matrix_rho35.shape[0]*3
dataRho35 = np.empty((size_input,4))
dataRho35[0:10000,0:3] = matrix_rho35
```

```
dataRho35[0:10000,3:4] = 35

#Plot Henon Trajectories for rho 35
fig = plt.figure()
plt.plot(dataRho35[:150,0],'r.', label = "True")
plt.plot(xpred[:150,0],'k.', label = "Prediction")
plt.title('Forecasting Lorenz for rho = 35', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)
plt.legend(loc = 'lower right', prop = {'size':10})
plt.savefig("Forecast 35")

#Plot of rho 35 data
plt.plot(xpred[:,2], xpred[:, 0], label='x_coord')

M = 10000

xpred = np.zeros((M,4))
xpred[0] = dataRho17[0,:]

for m in range(1,M):
    xpred[m] = model(xpred[m-1:m,:])

#Plot Henon Trajectories for rho 17
fig = plt.figure()
plt.plot(xpred[:150,0],'k.')
plt.plot(dataRho17[:150,0],'r.')
plt.title('Forecasting Lorenz with Neural Networks', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)

#Plot Henon Trajectories
fig = plt.figure()
plt.plot(xpred[:10000,1],'k.')
plt.plot(dataRho17[:10000,1],'r.')
plt.title('Forecasting Lorenz with Neural Networks', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)

#Plot of rho 17 data
plt.plot(xpred[:,2], xpred[:, 0], label='x_coord')
```