MAST 680 Assignment 3: Forecasting Chaos with Neural Networks

Will Sze

April 10, 2023

Abstract

The dynamics of non-linear systems are difficult to predict. Neural networks (NN) can be used to learn the dynamics of a complex system. This paper explores the use of NN to predict the trajectories of the Lorenz equations. TensorFlow 2.10 in Python was used to generate the models as it contains the necessary libraries to easily create these networks. The learned model makes accurate predictions with large time interval dt of the generated data and with sequential stepping S. However, the model showed inaccuracies when data outside of the model's training set was given, although the initial tendencies of the curve were captured. Finally, the algorithm was modified to learn the time for transitions between lobes and it performed decently well.

1 Introduction and Overview

Predicting the dynamics of a non-linear system is a difficult task. In assignment 2, the underlying dynamics behind a real system were uncovered using SINDy. Although this provided a general way of describing the system, a library of terms needed to be defined first which might not be obvious to choose. A method of learning non-linear dynamics without pre-defining a library is through neural networks (NN). Neural Networks date back to 1943 when McCulloch and Pitts attempted to model neurons in the brain [3]. Currently, NN has a wide range of applications, but they are all based on the same regression fitting principle.

In this paper, neural networks will be used to learn the dynamics of a chaotic system. The Lorenz differential equation will be used to train and evaluate the model. The general Lorenz equations with parameters σ,ρ and β as positive constants are given as

$$\begin{aligned} \dot{x} &= \sigma(y - x); \\ \dot{y} &= x(\rho - z) - y; \\ \dot{z} &= xy - \beta z. \end{aligned} \tag{1}$$

For this assignment, σ will be set to 10, β will be set to 8/3, and $\rho > 0$ will be an input parameter. The NN will be trained with data generated with values of ρ of 10, 28 and 40. The aim is to observe how far into the future the model can predict. Additionally, a model will be trained for its ability to detect the next instant the system changes lobes when the value of ρ is 28.

2 Theoretical Background

Neural networks function by linear regression between the input and output. It accomplishes that by using functions that can approximate other functions. Cybenko's original work on these functions were sigmoid functions [1], however, it turns out any non-polynomial functions can be used to perform the approximations. In neural networks, these functions are called activation functions and they are written as

$$g(x) = \tilde{\sigma}(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{2}$$

$\tilde{\sigma}$ is the activation function (the tilde is added to not be confused with the constant σ in the Lorenz system). For an M input and N output, $\mathbf{W} \in \mathbb{R}^{N \times M}$ contains the weights and $\mathbf{b} \in \mathbb{R}^N$ contains the biases. These weights and biases are parameters that can be modified to increase the accuracy of the approximation $g(x)$.

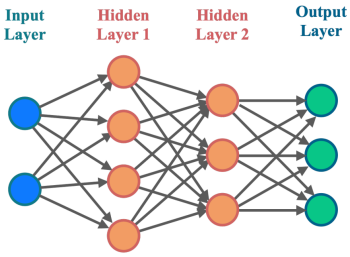

Figure 1: Representation of a neural network with an input, output and hidden layers in between. The network is 3 layers deep with 4 neurons in hidden layer 1, 3 neurons in hidden layer 2 and 3 neurons in the output. (image source:[2])

Equation 2 represents only one layer of the neural network. Multiple layers can be formed by composing Equation 2 by itself forming

$$g(x) = \sigma_d(W_d(\sigma_{d-1}(W_{d-1}(...(W_2(\sigma_1(W_1 x + b_1)) + b_2) + ...) + b_{d-1})) + b_d). \tag{3}$$

The subscript d represents the number of layers in the network and each $\mathbf{W}_j$ has n_j weights (also called neurons) where the subscript j is the layer number. A neural network with a large number of layers is called **deep** and a layer j that has a large n_j is called **wide**. Any layer that sits in between the input and the output is called a hidden layer. Having a deep and wide NN allows a more accurate fit to a given function. Figure 1 illustrates the neural network with its layers and widths. If the output in the network is continuously verified, it is called supervised learning. This method will be used for the learning of the dynamical system.

For a dataset $\mathbf{X} \in \mathbb{R}^{M \times K}$, where K is the number of data points, it is required to find a function $g(\mathbf{X})$ that maps the elements of column X_k into the elements of the k^{th} column of the output data $\mathbf{Y} \in \mathbb{R}^{N \times K}$. Since $g(\mathbf{X})$ only approximates $\mathbf{Y}$, there needs to be a way to verify the accuracy. Such a way is to calculate the error called a **Loss** using

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

Equation 4 is the mean square error of $\mathbf{Y} - g(\mathbf{X})$ where $\|.\|_2$ represents the Euclidean norm. For a system in which data points are sequential, equation 4 can be modified to calculate the loss after stepping s times into the future to increase the curve fitting. Stepping requires obtaining the output of the neural network and feeding it back into the network to obtain a new output such that $g^s(X_k)$ maps to X_{k+1+s}. g^s means s compositions. In this case, $\mathbf{Y}$ is formed from $\mathbf{X}$ where $Y_k = X_{k+1}$ such that the loss is calculated as

$$\mathcal{L} = \frac{1}{S} \sum_{s=1}^{S} \frac{1}{2(K-s)} \sum_{k=1}^{K-s} \|Y_{k+s} - g^s(X_k)\|_2^2. \tag{5}$$

S (capitalized s) is the total number of steps into the future. Generally, a larger number of steps could theoretically result in a better fit, however, at the expense of longer computations. The constant $\frac{1}{2}$ in equations 4 and 5 allows a term to be cancelled when taking the gradient of $\mathcal{L}$. A reason for taking the gradient comes from the minimization problem of the loss function so that $g(\mathbf{X})$ becomes as close as possible to Y. The gradient of $\mathcal{L}$ shows the direction in which the weights and biases need to move for the loss to decrease the most. The weights and biases are updated in iterative steps at each layer j and it is termed the gradient descent.

$$\begin{aligned} \mathbf{W}_j^{new} &= \mathbf{W}_j^{old} + \gamma \frac{\partial \mathcal{L}}{\partial W_j}; \\ \mathbf{b}_j^{new} &= \mathbf{b}_j^{old} + \gamma \frac{\partial \mathcal{L}}{\partial b_j} \end{aligned} \tag{6}$$

γ is the learning rate. It represents the magnitude in which $\mathcal{L}$ moves in the direction of the steepest descent. It is generally a user-defined parameter that needs to be set carefully as too large of a step can result in overshooting the minimum of $\mathcal{L}$, and too small of a step can result in a lengthy time to reach the minimum.

3 Algorithm Implementation and Development

The algorithms for the neural network were programmed using Tensorflow 2.10 with Python which allows the use of built-in libraries to easily create neural networks. The code from Jason's Forecasting Dynamics of the Hénon map [2] was adapted to generate data for a continuous system rather than for a discrete system. The algorithm starts with generating the data, and then training the network using pre-made functions from appendix A.

Generating the data involves the use of the function `odeint` from the Scipy package to numerically solve the Lorenz system. ρ is set as an input argument in the definition of the function. A time interval dt of 0.05 with N = 10000 data points was used with initial conditions (x,y,z) as (1,2,3). The solution was solved for three different values of ρ: 10, 28 and 40. To include all three solutions in the training set, the solutions were stacked in one large matrix X. The first three columns contain the spatial coordinates x, y and z, and the last column contain the value of ρ. Before training the network, input and output matrices were formed from X to allow mapping of $X_k \rightarrow X_{k+1} \rightarrow X_{k+2}$.... More matrices formed will allow a larger step S to be performed.

The network model is initialized with four inputs and four outputs with 10 hidden layers at 100 neurons. This amount of hidden layers and neurons was chosen as it gave decent results without being too computationally intensive. In each layer, the activation function was set to ReLU which is given as

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0. \end{cases} \tag{7}$$

In this way, the computations will be kept to a minimum since the derivative of such function is only 0 for $x \leq 0$ and 1 for $x > 0$. The network was trained at 30 000 epochs to reduce the loss as much as possible. During each epoch, the gradient descent was done through pre-made functions which required calculating the loss and setting the learning rate. The loss was computed as in equation 5 using `reduce_mean` of the squared difference between X_k and $g(X_k)$ and summing the losses from each step s. The total number of stepping S was experimented with values from 1 to 3. The learning rate for the gradient descent was set to 1e-3, 5e-4, 1e-4, 5e-5, 2.5e-5 and 1e-5 between intervals [0,1000], [1000,4000], [4000,8000], [8000,15000], [15000,20000] and [20000,30000] epochs respectively using `PiecewiseConstantDecay`. These values and intervals were chosen experimentally. In addition, TensorFlow provides multiple optimizers for gradient descent. The `Adam` optimizer was used as it provides a faster convergence to the local minima. Once the network is trained, the initial conditions used for the data generation along with the desired value of ρ were provided to the model and the output was consecutively fed back into the input to advance forward in time. The data points predicted by the network were then compared to the original Data matrix.

Algorithm 1: Pseudo-algorithm for neural network implementation in Python (full code can be found in appendix B)

```
Generate data points using odeint for all values of rho = 10, 28 and 40
Group all data into large matrix X
Create input and output matrices
Initialize model with 4 inputs, 10 hidden layers at 100 neurons per layer and 4 outputs
for i in range(Numberofepoch) do
  for s in range(Numberofsteps) do
    output ← model(input)
    input ← output
    loss + = reduce_mean(square(output − true))
  end for
  Perform gradient descent
end for
for m in range(number of advancements) do
  xp_m ← model(xp_{m−1})
end for
```

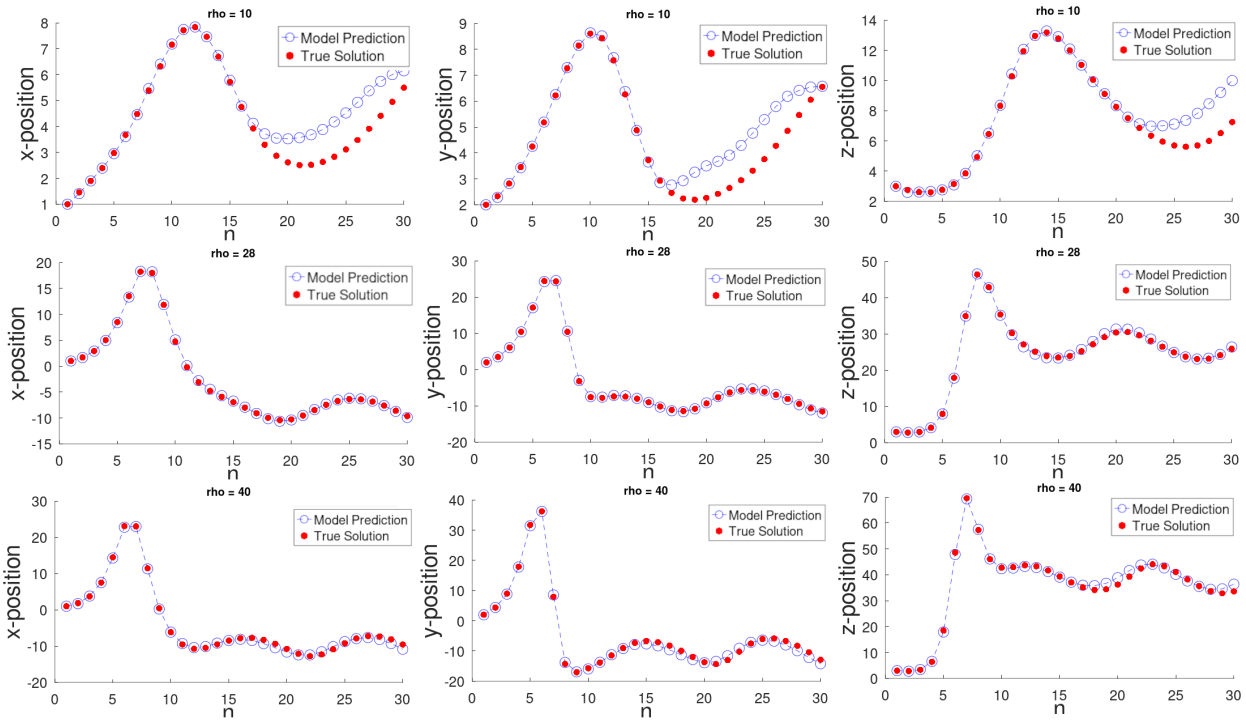

Figure 2: Model predictions for the first 30 iterations of the Lorenz equations with (σ, ρ, β). All σ and β values remain the same (10 and 8/3). The value of ρ from the top row to the bottom row is 10, 28 and 40. Training was performed using N = 10000, dt = 0.05, S = 3

4 Computational Results

After experimenting with multiple tunable parameters and running the algorithm multiple times, the model which outputs the best predictions was kept. Figure 2 shows the model prediction in comparison with the actual data for the first 30 advancements. When $\rho = 10$ is selected, accurate predictions can be obtained up to the 16^{th} datapoint after which the prediction deviates significantly from the true solution. Using the Euclidean norm to calculate the absolute error, the error calculated at the 30^{th} point is 2.85. When $\rho = 28$, the model predicts very accurately the system over the 30 data points displayed. The error at the 30^{th} point is 0.911. At $\rho = 40$, again, the model seems to perform well overall. However, a slight shift in the curve can be seen after 16 predictions. After 30 predictions, the error is 3.397. The model seems to provide accurate predictions at $\rho = 28$ and is less accurate with the other two values even though the network was trained with all three values. A possible reason for the model predicting less accurately the system when $\rho = 10$, even though the dynamics seem the easiest, is that the shapes produced by the other two values of ρ are similar to each other. This results in the training set containing more of the "Butterfly" shape and less of the "Spiral" shape. Needless to say, the model performs very well overall compared to the Lyapunov time, which describes the time for trajectories to diverge when given small differences in the initial conditions, calculated as the inverse of the largest Lyapunov exponent [2]. The Lyapunov time for the Lorenz system with parameters $(\sigma, \rho, \beta) = (10, 28, 8/3)$ is around 1.1s (1/0.90566) [4]. This equates to around 22 forward iterations at a dt of 0.05, which the model successfully meets. For the other values of ρ, the Lyapunov time was not found. This could be calculated in future works.

The number of steps S used was 3. When no step is used during training (S = 1), the model prediction is less accurate further into the future which is to be expected. The plots for $\rho = 28$ can be viewed in appendix C. Other values of ρ follow the same conclusion.

As a reminder, the time interval dt used in the data generation was 0.05. Smaller and larger dt were explored. Figure 3 shows the results with two different values of dt separated with a green line. When the

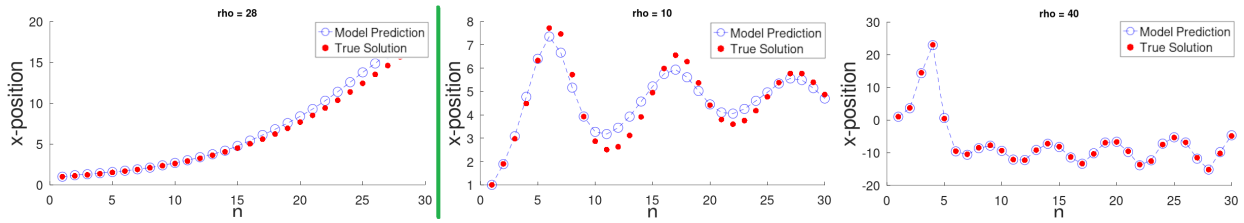

Figure 3: Model predictions for the first 30 iterations using training datasets generated at 2 different time intervals dt (0.01 at the left of the vertical green line and 0.1 at the right of the vertical green line)

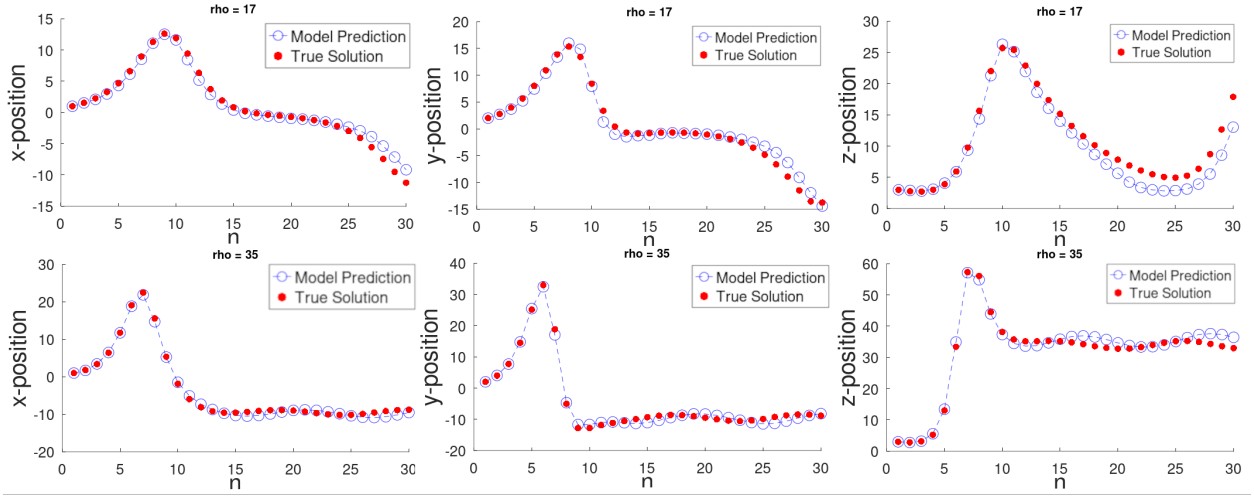

Figure 4: Trained neural network model tested for values of ρ outside of the training set.

time interval is decreased to 0.01, the model has a harder time predicting far into the future even with the same step size S of 3. In fact, the prediction deviates from the true solution once it reaches a significant curve. This may be because, for a step size of 3, less curvature is captured with the steps. More steps are needed to obtain more accurate predictions although, this leads to an increase in computation. When dt is increased to 0.1, mixed results can be seen. For $\rho = 10$, the model only predicts accurately up to 3 steps into the future before losing accuracy. However, for $\rho = 40$, the predictions almost perfectly reproduce the true solution with an error of only 0.49881 at the 30^{th} data point.

The results discussed up to now uses values of ρ with which the model has been trained on. Thus relatively good predictions can be expected. It is interesting to look at how the model can predict with other values of ρ. Figure 4 shows the predictions for ρ of 17 and 35. The network is able to predict the overall shape and tendencies inside 30 iterates, however, the accuracy falls after six or seven iterates. The error at the 30^{th} datapoint is around 5.6 and 3.9 for ρ of 17 and 35 respectively.

To see if neural networks can predict when a transition from one lobe to the other is imminent in the Lorenz system with $(\sigma, \rho, \beta) = (10, 28, 8/3)$, the algorithm is modified such that the model takes in the spatial coordinates and outputs the number of additional iterates needed for the switch. The training set was generated by observing the x-position of the system. The signs of two consecutive points were used to determine the transition. The network was trained and verified with the first 500 points of this data. Figure 5 and figure 6 show the comparison between the model prediction and the true solution as well as the comparison between the model prediction and the x position of the Lorenz system. The model accurately reproduces the "saw tooth" shape within the trained region. As soon as it reaches beyond 500 points, the model has more difficulties. For regions where it has similar counterparts to the trained data (e.g. between 540 and 710 advancements), the prediction matches roughly with the actual solution. However, for areas where no similar counterpart can be found (e.g. between 710 and 870), the model struggles to predict the next transition and constantly adjusts its values, although, near the end, it was able to correct itself and

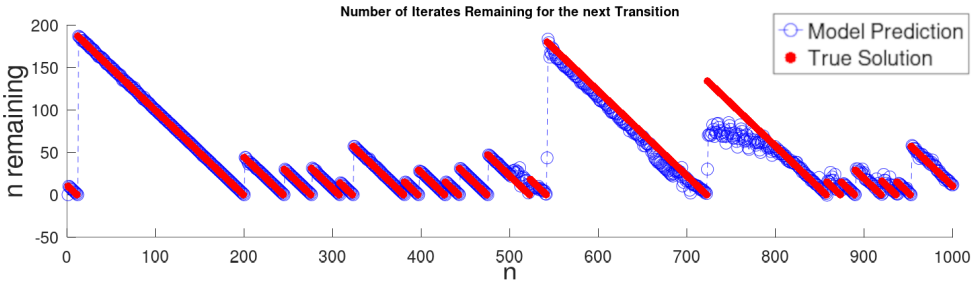

Figure 5: Comparision between model prediction and true solution of the remaining steps to transition from lobe to lobe of the Lorenz system.

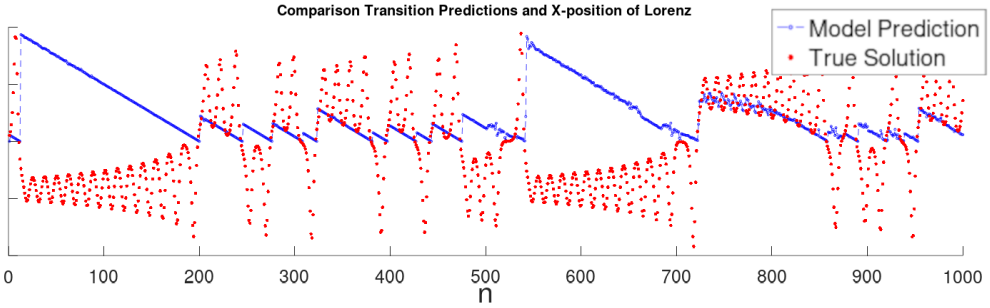

Figure 6: Comparison of the model prediction of the remaining steps to transition from lobe to lobe and the solution to the Lorenz equation in the x direction.

locate the transition.

5 Summary and Conclusions

In conclusion, neural networks are capable of learning the dynamics of non-linear systems. It does so by using activation functions to approximate the underlying function. A network was constructed to learn the equations of the Lorenz system. The model was able to predict fairly accurately when the inputs fall within the training set, however, the accuracy drops when values are outset of the training set. A drop in accuracy can also be observed with a decreasing dt and a smaller sequential step S. When training the network to predict the moment when it jumps from one lobe to the other, the model does well within the range of the training set but has more difficulties with prediction outside of the training set. Even though NN predictions have some inaccuracies after certain iterations, the models seem to capture the general tendencies of the system. If more time and computational resources are available, it would be interesting to increase the layers or increase the step size to past 10 and observe the effects.

References

[1] G Cybenko. "Approximation by superpositions of a sigmoidal function". In: (1989). URL: https://doi-org.lib-ezproxy.concordia.ca/10.1007/BF02551274.

[2] Jason J. Bramburger. *Data-Driven Methods for Dynamic Systems*. Concordia University, 2023.

[3] Bohdan Macukow. "Neural Networks – State of Art, Brief History, Basic Models and Architecture". In: *Computer Information Systems and Industrial Management*. Ed. by Khalid Saeed and Władysław Homenda. Cham: Springer International Publishing, 2016, pp. 3–14. ISBN: "978-3-319-45378-1".

[4] Divakar Viswanath. "Lyapunov Exponents from Random Fibonacci Sequences to the Lorenz Equations". PhD thesis. 1998. URL: https://hdl.handle.net/1813/7351.

Appendix A Premade Functions Used

- `odeint(function, $X_o$, t, arguments)` is a function from the Scipy package and it numerically integrates a set of differential equations in function, using initial conditions X_o, the time span t and function arguments.
- `tf.keras.Sequential()` set up the model with layers ordered sequentially.
- `model.add(arguments)` allows layers to be added to the model.
- `tf.keras.Input(n)` allows the model to accept n inputs.
- `tf.keras.layers.Dense()` sets the number of neurons in a layer. The activation function can also be set.
- `tf.reduce_mean(t)` calculates the mean of elements across dimensions of tensor t.
- `tf.square(t)` calculates the square of elements tensor.
- `tf.keras.optimizers.schedules.PiecewiseConstantDecay()` allows different learning rates to be set at different intervals of training.
- `tf.keras.optimizers.Adam(lr)` Provides an adaptive learning rate for the gradient descent using set values of the learning rate lr.s

Appendix B Python Code

B.1 Forecasting the Lorenz system with Neural Network

```
#Forecasting the Lorenz system with Neural Network
#Code adapted from https://github.com/jbramburger/DataDrivenDynSyst
    Learning Dynamics with Neural Networks\Forecast.ipynb

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as scipint

# Lorenz System function definition
def Lorenz(y, t, rho, b):

    dydt = [10.0*(y[1]-y[0]), y[0]*(rho - y[2]) - y[1], y[0]*y[1]-8.0/3*y
        [2], 0]

    return dydt

#------ Begin Generating Lorenz Data

# Initializations

ti = 0
tfin = 500
```

```
N = 5001
sol_rhoall = np.zeros((3*N,4))
t = np.linspace(ti,tfin,N)
dt = t[2]-t[1]
print("dt = ", dt)

# Lorenz parameter #1
rho = 10.0
x0 = [1,2,3, rho]
sol_rho10 = scipint.odeint(Lorenz,x0, t, args=(rho,0))
sol_rhoall[0:N,:] = sol_rho10

fig = plt.figure()
plt.plot(sol_rho10[:,0],sol_rho10[:,2],'k')
plt.title('The Lorenz Attractor with parameter rho = 10.0', fontsize = 20)
plt.xlabel('$x$', fontsize = 20)
plt.ylabel('$z$', fontsize = 20)

# Lorenz parameter #2
rho = 28.0;
x0 = [1,2,3, rho]
sol_rho28 = scipint.odeint(Lorenz,x0, t, args=(rho,0))
sol_rhoall[N:2*N,:] = sol_rho28

fig = plt.figure()
plt.plot(sol_rho28[:,0],sol_rho28[:,2],'k')
plt.title('The Lorenz Attractor rho = 28.0', fontsize = 20)
plt.xlabel('$x$', fontsize = 20)
plt.ylabel('$z$', fontsize = 20)

# Lorenz parameter #3
rho = 40.0;
x0 = [1,2,3, rho]
sol_rho40 = scipint.odeint(Lorenz,x0, t, args=(rho,0))
sol_rhoall[2*N:3*N,:] = sol_rho40

fig = plt.figure()
plt.plot(sol_rho40[:,0],sol_rho40[:,2],'k')
plt.title('The Lorenz Attractor rho = 40.0', fontsize = 20)
plt.xlabel('$x$', fontsize = 20)
plt.ylabel('$z$', fontsize = 20)

#------ End Generating Lorenz Data


#------ Begin Forecastiong using neural networks with the Lorenz Data

forward_iters = 10 # Number of forward iterations
xnforward = [] #initialize matrix for training data


ENDrow = np.size(sol_rhoall,0)
for j in range(forward_iters):
    xnforward.append(sol_rhoall[j:ENDrow-forward_iters+j])
```

```
80
81  input("Press Enter to continue...")
82
83  # Initializes the neural network model
84  def init_model(num_hidden_layers = 10, num_neurons_per_layer = 100):
85
86      model = tf.keras.Sequential()
87
88      # Input is (x,y,z,rho)
89      model.add(tf.keras.Input(4))
90
91      for _ in range(num_hidden_layers):
92          #adds the number of layer at each _ hidden layers
93          model.add(tf.keras.layers.Dense(num_neurons_per_layer,
94              activation=tf.keras.activations.get('relu'),
95              kernel_initializer='glorot_normal'))
96
97          #model.add(tf.keras.layers.Dropout(0.3))
98
99      # Output is (x,y,z,rho)
100     model.add(tf.keras.layers.Dense(4))
101
102     return model
103
104 def compute_loss(model, xnforward, steps):
105
106     loss = 0
107
108     for s in range(steps):
109
110         if s == 0:
111             xpred = model(xnforward[0])
112         else:
113             xpred = model(xpred) # x_(n+1) = model(x_n)
114
115         xnp1 = xnforward[s+1] # Gets the next true point to compare with
                the prediction
116
117         loss += tf.reduce_mean(tf.square(xpred-xnp1))/steps
118
119     return loss
120
121 def get_grad(model, xnforward, steps):
122
123     with tf.GradientTape(persistent=True) as tape:
124         # This tape is for derivatives with respect to trainable vriables.
125         tape.watch(model.trainable_variables)
126         loss = compute_loss(model, xnforward, steps)
127
128     g = tape.gradient(loss, model.trainable_variables)
129     del tape
130
131     return loss, g
132
```

```
# get neural network model
num_hidden_layers = 10
num_neurons_per_layer = 100;
model = init_model(num_hidden_layers,num_neurons_per_layer)

# Learning rate chosen as increasing steps
lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay
    ([1000,4000,8000,15000,20000], [1e-3,5e-4,1e-4,5e-5,2.5e-5,1e-5])

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


# add time function from the time package
from time import time

steps = 3


@tf.function
def train_step():
    # Compute current loss and gradient w.r.t. parameters.
    loss, grad_theta = get_grad(model, xnforward, steps)

    # Perform gradient descent step
    optim.apply_gradients(zip(grad_theta, model.trainable_variables))

    return loss

# Number of training epochs
N_training = 30000
Loss_hist = [] # Matrix to collect losses

# Start timer
t0 = time()


# Train the data
for i in range(N_training+1):
    loss = train_step()

    Loss_hist.append(loss.numpy())

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

# Print overal computation time
CompTime = time()-t0
print('\nComputation time:{} seconds'.format(CompTime))



# Use Trained Model to Forecast
M = 201

```

```
xpred = np.zeros((M,4))


rho = 10; # <--- Change parameter
xpred[0] = [1, 2, 3, rho] #inital conditions
for m in range(1,M):
    xpred[m] = model(xpred[m-1:m,:])


if rho == 28:
    DataStart = N
elif rho == 40:
    DataStart = 2*N
elif rho == 10:
    DataStart = 0
else:
    x0 = [1,2,3, rho]
    sol_rho_cust = scipint.odeint(Lorenz,x0, t, args=(rho,0))

    fig = plt.figure()
    plt.plot(sol_rho_cust[:,0],sol_rho_cust[:,2],'k')
    plt.title('The Lorenz Attractor with parameter rho = other', fontsize
        = 20)
    plt.xlabel('$x$', fontsize = 20)
    plt.ylabel('$z$', fontsize = 20)

DataEndPred = 150
DataStart = 0
DataEnd = DataStart+DataEndPred



if rho == 10 or rho == 28 or  rho == 40:

    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,0],'b--o')
    plt.plot(sol_rhoall[DataStart:DataEnd,0], 'r.')
    plt.title('Forecasting Lorenz xt with Neural Networks', fontsize = 20)
    plt.xlabel('$t$', fontsize = 20)
    plt.ylabel('$x$', fontsize = 20)

    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,2],'b--o')
    plt.plot(sol_rhoall[DataStart:DataEnd,2], 'r.')
    plt.title('Forecasting Lorenz zt with Neural Networks', fontsize = 20)
    plt.xlabel('$t$', fontsize = 20)
    plt.ylabel('$z$', fontsize = 20)

    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,0], xpred[:DataEndPred,2],'b--o')
    plt.plot(sol_rhoall[DataStart:DataEnd,0],sol_rhoall[DataStart:DataEnd
        ,2], 'r.')
    plt.title('Forecasting Lorenz xz with Neural Networks', fontsize = 20)
    plt.xlabel('$x$', fontsize = 20)
```

```
    plt.ylabel('$z$', fontsize = 20)


else:
    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,0],'b--o')
    plt.plot(sol_rho_cust[:DataEndPred,0], 'r.')
    plt.title('Forecasting Lorenz xt with Neural Networks', fontsize = 20)
    plt.xlabel('$t$', fontsize = 20)
    plt.ylabel('$x$', fontsize = 20)

    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,2],'b--o')
    plt.plot(sol_rho_cust[:DataEndPred,2], 'r.')
    plt.title('Forecasting Lorenz zt with Neural Networks', fontsize = 20)
    plt.xlabel('$t$', fontsize = 20)
    plt.ylabel('$z$', fontsize = 20)

    fig = plt.figure()
    plt.plot(xpred[:DataEndPred,0], xpred[:DataEndPred,2],'b--o')
    plt.plot(sol_rho_cust[:DataEndPred,0],sol_rho_cust[:DataEndPred,2], 'r
        .')
    plt.title('Forecasting Lorenz xz with Neural Networks', fontsize = 20)
    plt.xlabel('$x$', fontsize = 20)
    plt.ylabel('$z$', fontsize = 20)


#Show all the plots
plt.show(block=True)

#Save model
model.save('Lorenz_models/LorenzTRNDALL_step=3_rho=10largedt')

# Save data as .mat file
import scipy.io

Param = [dt, N, num_hidden_layers, num_neurons_per_layer, steps, CompTime]
if rho == 10 or rho == 28 or  rho == 40:
    scipy.io.savemat('LorenzTRNDALL_step=3_rho=10largedt.mat', dict(xpred
        = xpred, xtrue = sol_rhoall[DataStart:DataStart+M,:], Param = Param
        , rho = rho, loss = Loss_hist))
else:
    scipy.io.savemat('LorenzTRNDALL_step=3_rho=17LossN10000.mat', dict(
        xpred = xpred, xtrue = sol_rho_cust[:M,:], Param = Param, rho = rho
        , loss = Loss_hist))
```

B.2 Predicting lobe transitions in the Lorenz system with Neural Network

```
#Predicting lobe transitions in the Lorenz system with Neural Network
#Code adapted from https://github.com/jbramburger/DataDrivenDynSyst
   Learning Dynamics with Neural Networks\Forecast.ipynb

import os
```

```
from string import printable
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as scipint

# Lorenz System function definition
def Lorenz(y, t, rho, b):

    dydt = [10.0*(y[1]-y[0]), y[0]*(rho - y[2]) - y[1], y[0]*y[1]-8.0/3*y
        [2]]

    return dydt

#------ Begin Generating Lorenz Data

# Initializations

ti = 0
tfin = 500
N = 10001
t = np.linspace(ti,tfin,N)
dt = t[2]-t[1]
print("dt = ", dt)


# Lorenz parameter #2
rho = 28.0;
x0 = [1,2,3]
sol_rho28 = scipint.odeint(Lorenz,x0, t, args=(rho,0))

M_end = 500;
fig = plt.figure()
plt.plot(t[:M_end],sol_rho28[:M_end,0],'k')
plt.title('The Lorenz Attractor rho = 28.0', fontsize = 20)
plt.xlabel('$x$', fontsize = 20)
plt.ylabel('$z$', fontsize = 20)

# Finding the transition points and set countdowns
tJump = np.zeros((N,1));
n = 0
for i in range(N-1):
    curr = sol_rho28[i+1,0]
    prev = sol_rho28[i,0]

    if (np.sign(curr) !=  np.sign(prev)):
        tJump[i+1,0] = 0.0 #1
        for j in range(i+1-n):
            if n==0:
                tJump[j+n,0] = i+1-n-j
            else:
```

```
                if j != 0:
                    tJump[j+n,0] = i+1-n-j

        n = i+1
    else:
        tJump[i+1,0] = 1.0 #0


fig = plt.figure()
plt.plot(tJump[:M_end],'k')
plt.title('T jump', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$t$', fontsize = 20)


#------ End Generating Lorenz and time till jump Data

#------ Begin Forecastiong using neural networks with the Lorenz Data
xnforward = [] #initialize matrix for training data

xnforward.append(sol_rho28[0:500,:])
xnforward.append(tJump[0:500])

input("Press Enter to continue...")

# Initializes the neural network model
def init_model(num_hidden_layers = 10, num_neurons_per_layer = 100):

    model = tf.keras.Sequential()

    # Input is (x,y,z,rho)
    model.add(tf.keras.Input(3))

    for _ in range(num_hidden_layers):
        #adds the number of layer at each _ hidden layers
        model.add(tf.keras.layers.Dense(num_neurons_per_layer,
            activation=tf.keras.activations.get('relu'),
            kernel_initializer='glorot_normal'))


    # Output is (t)
    model.add(tf.keras.layers.Dense(1))

    return model

def compute_loss(model, xnforward):

    loss = 0
    tpred = model(xnforward[0])
    xnp1 = xnforward[1]

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

    return loss
```

```
112
113 def get_grad(model, xnforward):
114
115     with tf.GradientTape(persistent=True) as tape:
116         # This tape is for derivatives with respect to trainable vriables.
117         tape.watch(model.trainable_variables)
118         loss = compute_loss(model, xnforward)#, tJump, steps)
119
120     g = tape.gradient(loss, model.trainable_variables)
121     del tape
122
123     return loss, g
124
125 # get neural network model
126 num_hidden_layers = 10
127 num_neurons_per_layer = 100
128 model = init_model(num_hidden_layers,num_neurons_per_layer)
129
130 # Learning rate chosen as decremental steps
131 lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay
    ([1000,3000,8000], [1e-2,1e-3,1e-4,1e-5])
132
133 optim = tf.keras.optimizers.Adam(learning_rate=lr)
134
135
136 # add time function from the time package
137 from time import time
138
139 steps = 1
140
141
142 @tf.function
143 def train_step():
144     # Compute current loss and gradient w.r.t. parameters.
145     loss, grad_theta = get_grad(model, xnforward)
146
147     # Perform gradient descent step
148     optim.apply_gradients(zip(grad_theta, model.trainable_variables))
149
150     return loss
151
152 # Number of training epochs
153 N_training = 20000
154 Loss_hist = [] # Matrix to collect losses
155
156 # Start timer
157 t0 = time()
158
159
160 # Train the data
161 for i in range(N_training+1):
162     loss = train_step()
163
164     Loss_hist.append(loss.numpy())
```

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

# Print overal computation time
CompTime = time()-t0
print('\nComputation time:{} seconds'.format(CompTime))



# Use Trained Model to Forecast
M = 1100

tpred = np.zeros((M,1))

for m in range(1,M):
    tpred[m] = model(sol_rho28[m-1:m,:])


print(tJump[0:50])
print(tpred[0:50])

input("Press Enter to continue...")
DataStartPred = 0
DataEndPred = 1000

fig = plt.figure()
plt.plot(1/10*tpred[DataStartPred:DataEndPred,0],'b--o')
plt.plot(sol_rho28[DataStartPred:DataEndPred,0], 'r.')
plt.title('Forecasting tJump with Neural Networks', fontsize = 20)
plt.xlabel('$t$', fontsize = 20)
plt.ylabel('$x$', fontsize = 20)


plt.show(block=True)

input("Press Enter to continue...")
# Save model
model.save('Lorenz_models/LorenztJumpPred_rho=28Saw')

# Save data as .mat file
import scipy.io

Param = [dt, N, num_hidden_layers, num_neurons_per_layer, CompTime]
scipy.io.savemat('LorenztJumpPred_rho=28Saw.mat', dict(tpred = tpred,
    ttrue = tJump, FullSOl = sol_rho28, Param = Param, rho = rho, loss =
    Loss_hist))
```

Appendix C Complementary Figures

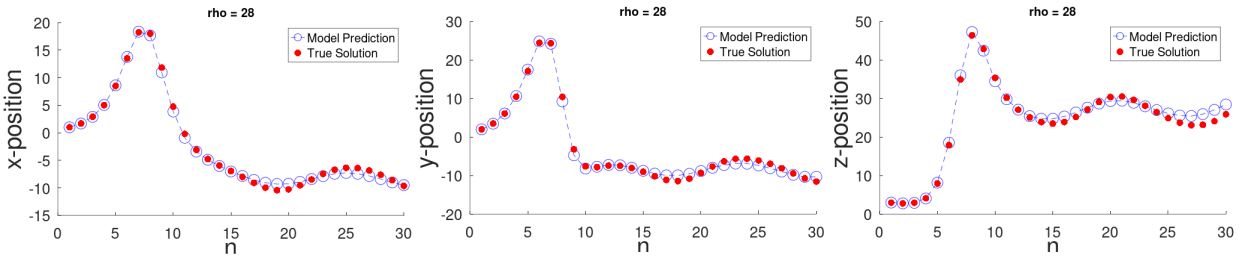

Figure 7: Model predictions for the first 30 iterations of the Lorenz equations with $(\sigma, \rho, \beta) = (10, 28, 8/3)$. Training was performed using N = 10000, dt = 0.05, S = 1.