MATH 494 Homework 3

Nada El Arabi

April 10th 2023

Abstract

In this report, we discuss the training of a forecasting model for the Lorenz system via the use of Recurrent Neural Networks. We begin by generating the solutions of our dynamical system for various values of the variable ρ, some of which induce chaotic behaviour while others do not. We feed our generated data to a Recurrent Neural Network and optimize for performance by examining the effects of changes to the loss function and the size of the time step of the variables on the model's ability to forecast the solutions of the Lorenz system for both known and novel values of ρ.

1 Introduction and Overview

Neural Networks have seen a meteoric rise in popularity in the past decade thanks to their ability to efficiently model extremely non-linear data [1]. A pointed example of such data is found in dynamical systems which often feature nonlinear relationships between the the values of variables in time [3]. Recurrent Neural Networks in particular are an augmentation on traditional feed forward models where the weight parameter is shared across each layer of the neural network.

In this paper, we attempt to forecast in time the values of variables $\mathbf{x}, \mathbf{y}, \mathbf{z}$ of the Lorenz system for values of $\rho = 17, 35$. We do this by first calculating the solution of the Lorenz system with initial conditions $x_0, y_0, z_0 = 0, 1, 1$ and $\beta, \sigma = 8/3$ and 10, respectively. We do this for values of $\rho = 10, 28$ and 40. We stack the solutions in an array and add a fourth column to contain each of the values of ρ. We then feed the data to a Recurrent Neural Network constructed with Python's Deep Learning library: Tensorflow. Once the model finishes training, we examine and comment on its performance in forecasting solutions to the Lorenz model for values 17 and 35 of the variable ρ.

2 Theoretical Background

The Lorenz system is a nonlinear dynamical system which has applications in meteorology, in particular it is a simplified weather model [2]. It is characterized by the following three equations:

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

Where x represents convection, y is the horizontal temperature variable and z is the vertical temperature variable.

Three parameters are present in this model: σ a ratio of fluid viscosity to thermal conductivity, ρ the difference of temperature between the top and the bottom of the plane and finally, β which is the ratio of the width to the height of the plane. The Lorenz equations have been proven to behave chaotically for $\rho \in [24.74, 30.1]$, though chaotic behaviour is certainly possible outside of this range [2].

Neural Networks, though are not an entirely recent invention, have revolutionized our ability to handle nonlinear data.

3 Algorithm Implementation and Development

The first step in accomplishing our goal is to generate the solutions of the Lorenz system to train and test the model. To that effect we settle on generating solutions for initial conditions $x_0, y_0, z_0 = 0, 1, 1$. For σ and β, we follow in Edward Lorenz's footsteps and choose 10 and 8/3 respectively [2], while noting that other initial conditions could be chosen just as successfully. We select $\rho = 10, 28, 40$ for the training data and $\rho = 17, 35$ for our test data. The particular Δt we select to generate our data can have significant impact on the results and will be discussed at length in the Computational Results section below. For now, we will simply generate three sets of data for all values of ρ with $\Delta t = 0.1, 0.01, 0.005$ which correspond to 1,000, 10,000 and 20,000 data points, respectively. To do so we define a function "Lorenz" which takes inputs consisting of variables x, y, z in a single placeholder variable X, t for Δt, σ, β and finally ρ. The function returns derivatives in time for variables x, y, z as defined in the above discussion of the Lorenz system. We calculate the solutions to our function via SciPy's legacy odeint method and save the results in NumPy matrices (Please refer to lines 26-86 of Appendix C and note that only the code for $\Delta t = 0.005$ is provided as the procedure is much the same for other values of Δt.)

An intermediary step necessary to proceed to the training of the model is of course to transform it into a convenience format for the Neural Network to process. To that purpose, we create a large matrix "data" which is large enough to contain all solutions of x, y, z for $\rho = 10, 28, 40$, stacked horizontally where each row is a snapshot in time, as well as a fourth column for each ρ. This implementation is chosen as we wish to train on particular values of ρ while keeping these static in the training of the Neural Network. We then proceed to create two new matrices, "xn" and "xnp1" where the first is simply all of the snapshots in time of the "data" matrix, and the latter is the same, only one step forward in time (see lines 92-128).

The second objective consists in constructing a Recurrent Neural Network. To do so, we make use of Python's Tensorflow 2.0. We first define our model to be a sequential model of four hidden layers of eighty neurons each which takes four inputs and returns 4 outputs (the next time step of the Lorenz system, or indeed, the next row of data). For our activation function we choose the ReLu function as it is currently one of the most performant available [1]. On the particular choice of number of hidden layers and neurons per layer, it was determined by trial and error that the most accurate results were given with these features, where increasing either results in significant computation overhead for almost no additional precision and on the contrary, decreasing either one results in significant loss of accuracy (see lines 134-150).

We then define our loss function which we select to be the MSE or Mean Square Error, and note that we focused our fine tuning efforts on the learning rate of the gradient descent algorithm, though the search for a better loss function would be a worthwhile pursuit. To our loss function we give as inputs our model and our two solution matrices, xn and xnp1 so that the model can learn to forecast solutions in time. We proceed to define the gradient descent function and to initialize the model with its learning rate (see lines 156-242). The learning rate is characterized by two optimizers: a Stochastic Gradient Descent optimizer: Adam, and a piecewise constant decay schedule. Significant time was invested in fine-tuning the latter, largely by trial and error and observation of possible patterns in the value of the loss function as the Gradient Descent algorithm is performed. The particular schedule outlined on lines 192-197 of Appendix C has been found to consistently produce final loss values in the magnitude of $[1e-03, 8e-03]$ for data where $\Delta t = 0.1, 0.01$ and even as little as $9.299e-04$ for data where $\Delta t = 0.005$.

Finally the last objective in achieving our desired goal is to validate the model to determine how successful our Recurrent Neural Network model is in its ability to forecast solutions in the future and how well it can model the behaviour of novel Lorenz systems. In both goals, the procedure remains the same the model is validated by giving it the initial conditions of the data we wish to validate against and repeatedly calculate the forward iteration by way of the model we trained (see lines 248-308). Python's library Matplotlib can be utilized to generate graphs of the results which we will discuss in the next section.

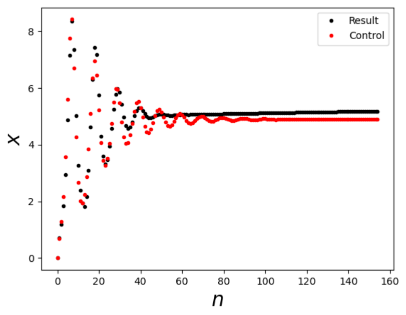
4 Computational Results

The implementation discussed above has provided interesting results which we will explore in this section. We begin by noting that all the graphs presented examine only the behaviour of the variable x as detailed discussion of every variable would be quite lengthy. A second important consideration is the fact that all results analyzed here are born out the same three models, namely one for each of $\Delta t = 0.1, 0.01, 0.005$. Respectively, these models had final loss function values of $7.53079494e-03, 1.40194956e-03$ and $9.29846137e-04$. At first, these figures seem impressive, but upon closer inspection of Figure 1, we can easily observe that the models seem to perform quite poorly in terms of the number of steps in the future they can forecast. The ability seems to only decrease as we decrease Δt which is contrary to expectation. With $\Delta t = 0.1$ we are able to forecast merely two steps in the future and none accurately with the others.

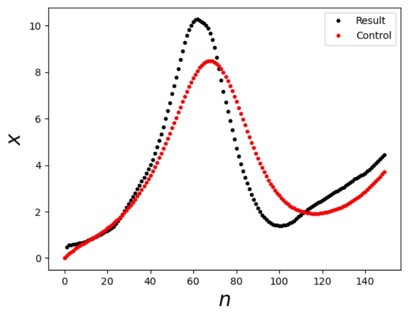
Unexpected results can be found in the models' ability to forecast solutions of Lorenz systems with novel values of ρ. These results can be observed in Figures 4 and 5 in Appendix C. Indeed the model with $\Delta t = 0.1$ seems to fare better for both $\rho = 17$ and $\rho = 35$, being able to forecast with much precision respectively three and two steps in the future. Even more striking are the results seen in subfigures 5b and 4b where the first seems to be an excellent forecast over 20 steps in the future while the second experiences a complete inversion of trajectory. Figure 5 seems to indicate that chaotic behaviour is well taken into account by the models while Figure 4 suggests the opposite for convergent trajectories. A possible explanation for this the fact that chaotic behaviour is represented twice-to-one in our training data, making the models more apt to detect it.

The analysis above has provided significant insight into the ability of the models to forecast solutions very early in time, but it would be a glaring omission to not question how well the models fare in loosely predicting long term behaviour, especially since Figures 1, 4 and 5 generally indicate that the models are doing well in that endeavor. Of course, so far we have only looked at a single variable and only approximately 150 steps into the future, we can instead look to plot the $xz-plane$ for both $\rho = 17$ and $\rho = 35$ for all models. For reference, exact values are offered for both values of ρ in Appendix C, Figures 6 and 7.

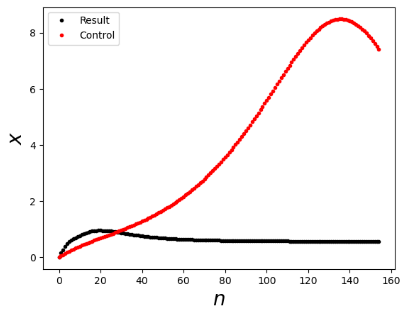
We now examine Figures 2 and 3 and notice surprising results!
All models have correctly identified the system as chaotic or convergent. Though we initially expected poor performance from the models on modelling non-chaotic behaviour, Figure 2 demonstrates that the behaviour is clearly captured. Interestingly for both figures, the marginally weaker model appears to be the one where $\Delta t = 0.01$.



(a) $\Delta t = 0.1$

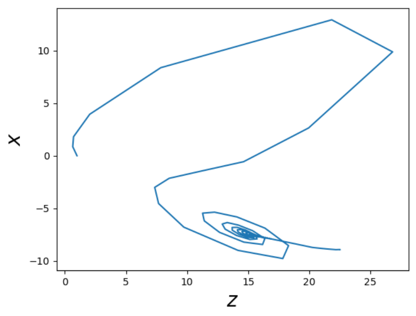


(b) $\Delta t = 0.01$

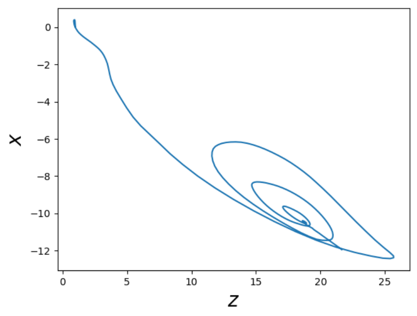


(c) $\Delta t = 0.005$

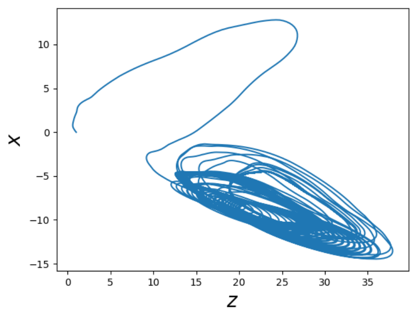
Figure 1: Forecast of the x variable with respect to discrete time step n



(a) $\Delta t = 0.1$

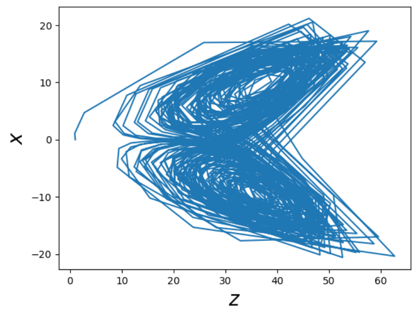


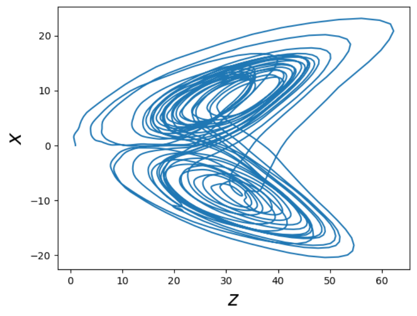
(b) $\Delta t = 0.01$



(c) $\Delta t = 0.005$

Figure 2: Prediction of the $xz - plane$ for $\rho = 17$


(a) $\Delta t = 0.1$


(b) $\Delta t = 0.01$

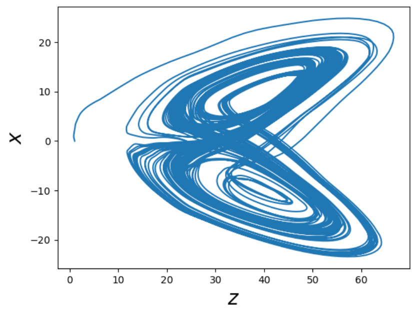

(c) $\Delta t = 0.005$

Figure 3: Prediction of the $xz - plane$ for $\rho = 35$

5 Summary and Conclusions

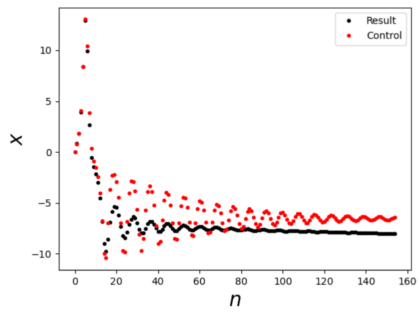
By generating solutions for a number of Lorenz systems, chaotic and otherwise, we were able to construct a data set to give to a Recurrent Neural Network whose parameters we tweaked to attempt to produce accurate forecasts. Though the loss function values seemed promising at first, forecasting the systems' solutions proved challenging. Increasing the quantity of data points did not improve the predictions definitively and at times seemed to worsen it. The Neural Network was however quite successful in capturing the the nature of the system's behaviour in the long run, correctly modelling chaotic and non-chaotic behaviours alike, highlighting the potential to provide valuable insight into similar systems.

References

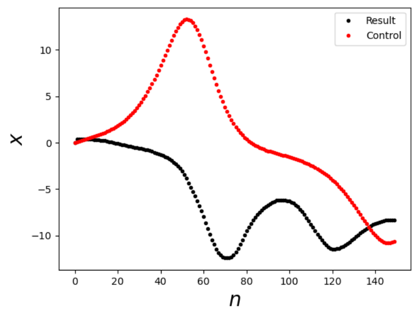
[1] Jason Bramburger. *Data-Driven Methods for Dynamic Systems: A new perspective on old problems.* 2023.

[2] University of Waterloo Faculty of Mathematics. *Chaos Theory and the Lorenz Equation: History, Analysis, and Application.* Apr. 2012. URL: `https://links.uwaterloo.ca/pmath370w14/PMATH370/lorenz%5C%20Latex.pdf`.

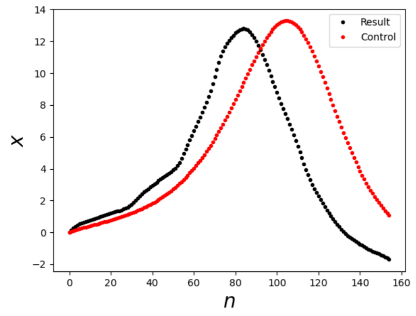
[3] Yonggi Park et al. *Recurrent Neural Networks for Dynamical Systems: Applications to Ordinary Differential Equations, Collective Motion, and Hydrological Modeling.* Feb. 2022. URL: `https://arxiv.org/abs/2202.07022`.



(a) $\Delta t = 0.1$

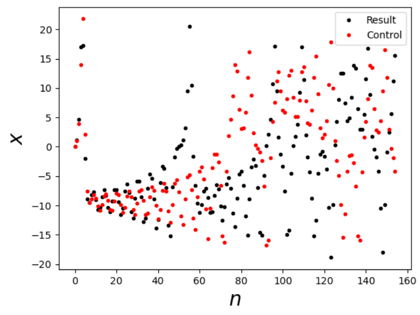


(b) $\Delta t = 0.01$

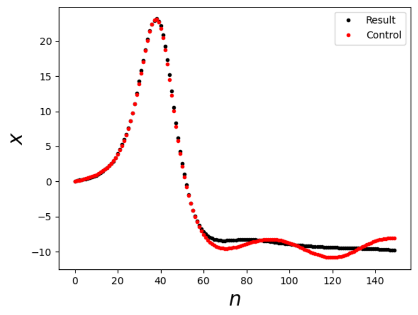


(c) $\Delta t = 0.005$

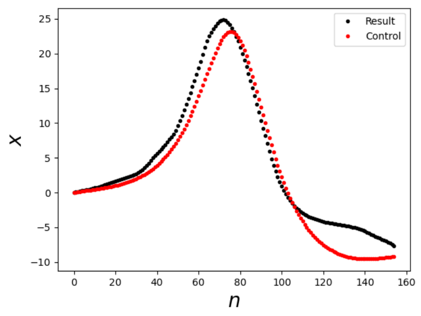
Figure 4: Forecast of the x variable with respect to discrete time step n for $\rho = 17$



(a) $\Delta t = 0.1$

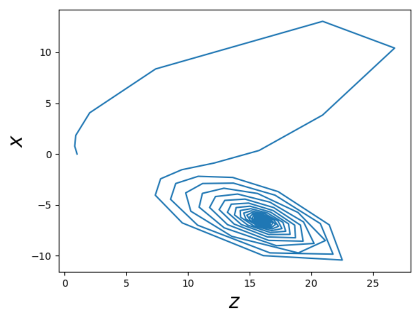


(b) $\Delta t = 0.01$

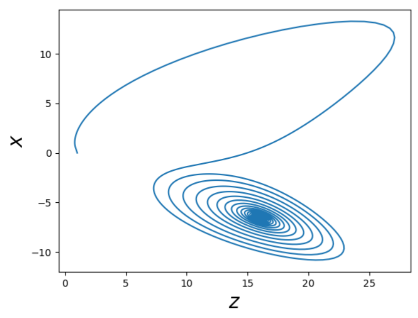


(c) $\Delta t = 0.005$

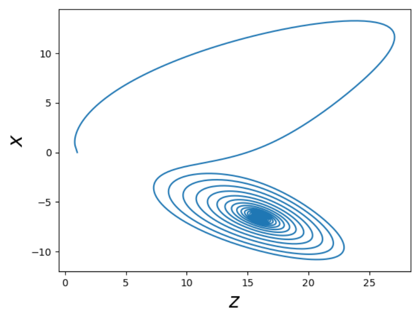
Figure 5: Forecast of the x variable with respect to discrete time step n for $\rho = 35$



(a) $\Delta t = 0.1$

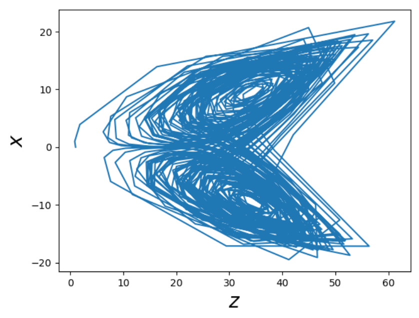


(b) $\Delta t = 0.01$

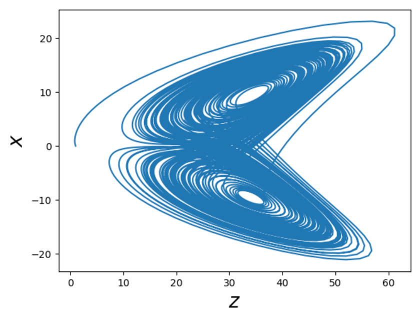


(c) $\Delta t = 0.005$

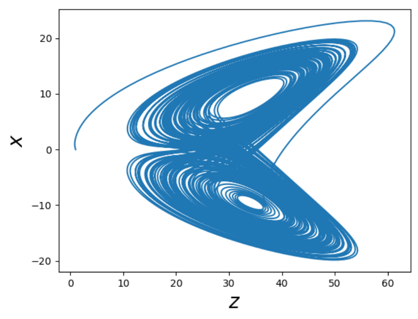
Figure 6: Solutions to the Lorenz system with $\rho = 17$



(a) $\Delta t = 0.1$



(b) $\Delta t = 0.01$



(c) $\Delta t = 0.005$

Figure 7: Solutions to the Lorenz system with $\rho = 35$

Appendix A Additional graphs

Appendix B Python Functions

This is how to make an **unordered** list:

- `y = linspace(x1,x2,n)` returns a row vector of `n` evenly spaced points between `x1` and `x2`.
- `[X,Y] = meshgrid(x,y)` returns 2-D grid coordinates based on the coordinates contained in the vectors `x` and `y`. X is a matrix where each row is a copy of `x`, and `Y` is a matrix where each column is a copy of `y`. The grid represented by the coordinates `X` and `Y` has `length(y)` rows and `length(x)` columns.

Appendix C Python Code

```python

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


# In[2]:


tf.version.VERSION


# In[3]:


from platform import python_version

print(python_version())


# In[67]:


#We note that the Lorenz system displays chaos for values of rho between ~24.7 and ~30.1,
    since rho is 10 we shouldn't observe chaotic behaviour
beta = 8/3
sigma = 10
rho = 10

#These are some initial conditions, it's possible to choose others
x0 = 0
y0 = 1
z0 = 1


# In[68]:


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


# In[69]:

```

```

#We set up the time points
tmax = 100 #This is the maximum time point
n = 20000 #Going from 10k to 20k means we halve delta t
t = np.linspace(0, tmax,n)


# In[70]:


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


# In[42]:


#Where the first column is composed of the values of x, the second of y and third of z
sol.shape


# In[16]:


rho = 10
sol = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol[:,2], sol[:, 0], label='x_coord')
np.save('solutions_rho10_20k', sol)

rho = 40
sol = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol[:,2], sol[:, 0], label='x_coord')
np.save('solutions_rho40_20k', sol)

rho = 28
sol = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol[:,2], sol[:, 0], label='x_coord')
np.save('solutions_rho28_20k', sol)


# In[ ]:


#Data processing to feed into the mdoel
#You have to stack all of the data with each rho on top of each other into a matrix xn
#You then also have to stack the data for one time stamp in the future into a matrix xnp1
#Then you feed both xn and xnp1 into the model
matrix_rho10_20k = np.load('/Users/nadaelarabi/Data-driven-dynamical-systems/M494H3/
    solutions_rho10_20k.npy')
matrix_rho28_20k = np.load('/Users/nadaelarabi/Data-driven-dynamical-systems/M494H3/
    solutions_rho28_20k.npy')
matrix_rho40_20k = np.load('/Users/nadaelarabi/Data-driven-dynamical-systems/M494H3/
    solutions_rho40_20k.npy')


#That was too complicated! Just get the entire data, all 10,000 x 3 into a matrix and then
    slice that
size_input = matrix_rho10_20k.shape[0]*3
data = np.empty((size_input,4))
data.shape

#Here we stack our all of our data from each rho into one single matrix "data"
data[0:20000,0:3] = matrix_rho10_20k
data[20000:40000,0:3] = matrix_rho28_20k
data[40000:60000,0:3] = matrix_rho40_20k
data[0:20000,3:4] = 10
data[20000:40000,3:4] = 28
data[40000:60000,3:4] = 40

```

```python
xn = np.empty((19999*3,4))
xn[0:19999,0:4] = data[0:19999,0:4]
xn[19999:39998,0:4] = data[20000:39999,0:4]
xn[39998:59997,0:4] = data[40000:59999,0:4]

xnp1 = np.empty((19999*3,4))
xnp1[0:19999,0:4] = data[1:20000,0:4]
xnp1[19999:39998,0:4] = data[20001:40000,0:4]
xnp1[39998:59997,0:4] = data[40001:60000,0:4]

#Just some testing to make sure that the slicing is done correctly
xnp1[0:1,0:4] == xn[1:2,0:4]
xnp1[19999:20000,0:4] == xn[20000:20001,0:4]
xnp1[39998:39999,0:4] == xn[39999:40000,0:4]
xnp1[40001:40002,0:4] ==xn[40002:40003,0:4]


# In[2]:


def init_model(num_hidden_layers = 4, num_neurons_per_layer = 80):
    # Initialize a feedforward neural network
    model = tf.keras.Sequential()

    # Input is 4D - each variable of the Lorenz system along with their corresponding value
    of rho
    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 4D - next step of Lorenz system
    model.add(tf.keras.layers.Dense(4))

    return model


# In[3]:


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

    loss = 0

    xpred = model(xn)

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

    return loss



# In[4]:


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



# In[5]:


# Initialize model aka 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
# the first 800 steps use a learning rate of 0.01
# from 800 - 3000: learning rate = 0.003
# from 3000 -7500: learning rate = 0.0006
# from 7500 onwards: 0.0001

lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay([800,3000,7500],[1e-2,3e-3,6e-4,1e
    -4])

# Choose the optimizer
optim = tf.keras.optimizers.Adam(learning_rate=lr)


# In[7]:


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 w.r.t. 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('\nComputation time: {} seconds'.format(time()-t0))


# In[9]:


```

```
matrix_rho17 = np.load('/Users/nadaelarabi/Data-driven-dynamical-systems/M494H3/
    solutions_rho17_20k.npy')
size_input = matrix_rho17.shape[0]*3
dataRho17 = np.empty((size_input,4))
dataRho17[0:20000,0:3] = matrix_rho17
dataRho17[0:20000,3:4] = 17


# In[8]:


matrix_rho35 = np.load('/Users/nadaelarabi/Data-driven-dynamical-systems/M494H3/
    solutions_rho35_20k.npy')
size_input = matrix_rho35.shape[0]*3
dataRho35 = np.empty((size_input,4))
dataRho35[0:20000,0:3] = matrix_rho35
dataRho35[0:20000,3:4] = 35


# In[27]:


#Utilizing the model we trained:
M = 20000

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

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


# Plot trajectory of the x-variable of the Lorenz system
fig = plt.figure()
plt.plot(xpred[:20000,0],'k.',label='Result')
plt.plot(xn[:20000,0],'r.',label='Control')
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x$', fontsize = 20)
plt.legend()
plt.savefig('xnPred_20k_allvalues.png')


# In[39]:


#Utilizing the model we trained:
M = 20000

xpred = np.zeros((M,4))
xpred[0] = dataRho35[0,:]

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


# Plot trajectory of the x-variable of the Lorenz system
fig = plt.figure()
plt.plot(xpred[:155,0],'k.',label='Result')
plt.plot(dataRho35[:155,0],'r.',label='Control')
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x$', fontsize = 20)
plt.legend()
plt.savefig('xnPred_20k_rho35.png')


# In[ ]:


```

```
#Saving the graph of our control data
plt.plot(matrix_rho17[:,2], matrix_rho17[:, 0], label='x_coord')
plt.xlabel('$z$', fontsize = 20)
plt.ylabel('$x$', fontsize = 20)
plt.savefig('Control_17_20k.png')


# In[43]:


#Saving the graph of a result
plt.plot(xpred[:,2], xpred[:, 0], label='x_coord')
plt.xlabel('$z$', fontsize = 20)
plt.ylabel('$x$', fontsize = 20)
plt.savefig('Result_35_20k.png')
```