Assignment 3 Math 494

Alexander Vakkas (40098027)

April 10, 2023

Abstract

The goals that were set by this assignment were to establish the loss function that leads to the best predictions, to assess how many time steps the prediction remains correct, and to determine whether the recurrent neural networks (RNNs) that are properly trained to predict data. This is accomplished by employing ordinary differential equations, specifically the Lorenz equations, to train RNNs to take data from one time step to the next using the parameter ρ. To obtain the optimum error approximation, many loss functions were explored to forecast shifting data from one time step to the next.

1 Introduction and Overview

Over the years training neural networks has been very hard since there are only a limited number of differential equations that can be easily computed by hand for this specific training. It would be impossible to determine whether the neural network solutions were right without these solutions. As a result, the Lorenz equations, which are ordinary differential equations for which we have solutions, will be used to train recurrent neural networks. The training that will be undertaken will determine the loss function to the best of its ability by sending data from one time step to another, next would be to determine how future data is predicted, and finally using outside parameters we will test to see whether our data remains accurate.

2 Theoretical Background

2.1 Lorenz Equation

The Lorenz equation is named after its creator, Edward Norton Lorenz, who was an American mathematician and meteorologist. The Lorenz system is a system of ordinary differential equations that Lorenz devised. The reason for its importance is that Lorenz noticed when playing around with parameter values his system became chaotic. In popular media, the "butterfly effect" is derived from the real-world impact of the Lorenz attractor. This means that in a chaotic physical system, the ability to predict its future course always fails without perfect knowledge of its initial conditions [2]. The shape of the Lorenz attractor when plotted in phase space, see figure 1, also is seen to resemble a butterfly, hence the name.

The Lorenz equation is described as follows:

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

The state variables x, y, and z are described by equations as the rate of change over time. The state variables do not refer to spatial coordinates, it is very important to understand. In actuality, x denotes the overturning of convection on the plane, whereas y and z, respectively, denote the horizontal and vertical variations in temperature. The standard specifications for the Lorenz system uses initial conditions (x0, y0, z0) = (0, 1,

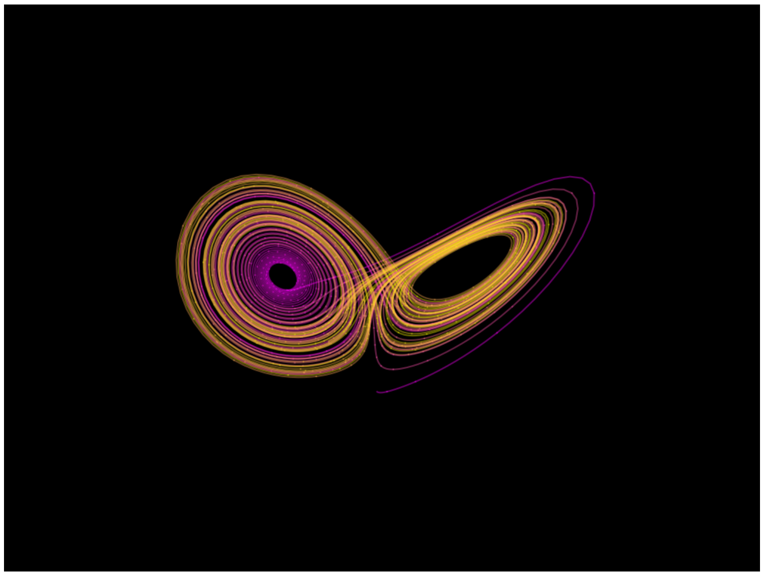

Figure 1: Figure 1 - 3-Dimensional Lorenz Model

0), $\sigma = 10$, $\beta = 8/3$ and ρ varies (ideally for ρ belonging to [24.74,30.1)). Overall, accurately predicting the system is nearly impossible very far into the future, so maybe we will have better luck in the near future [2].

2.2 Recurrent Neural Networks(RNNs)

Learning a little bit about function approximators will help you better grasp neural networks. A function approximator's primary objective is to, as the name suggests, approximate functions. It is done by employing simple functions listed bit by bit so that the behavior of the complete function can be approximated by smaller chunks of simpler functions [1].

$$x_{n+1} = x_n - \gamma \nabla f(x_n) \tag{2}$$

$$L = \frac{1}{2K} \sum_{k=1}^{K} ||Y_k - g(X_k)||_2^2 \tag{3}$$

By applying the all-purpose function approximator sigmoid σ, this nonlinear process is precisely what a neural network (NN) is. The NN minimizes the loss function while pushing data from one time step to the next using a single function. As a gauge of how well the prediction matches the real function, the aforementioned loss function is used [1]. NNs and machine learning both depend on the composition of functions. In fact, the loss function interpolates between what goes in and what comes out. The γ parameter is calibrated to take the appropriate step size towards the minimum, thereby preserving an accurate approximation. As a result, the loss function is based on reducing utilizing gradient descent. In contrast to the loss function, which utilizes an approximation of the true function, the gradient descent equation (2) uses the true function. The mean squared error (MSE) (3) loss function will be applied in this assignment [1].

The focus of this assignment revolves around recurrent neural networks (RNNs) which will complete our mathematical foundation. The RNNs augment the loss function using sequential data as opposed to applying it once. Next, using Lyaponov time, we can calculate how quickly the system will diverge and, consequently, how many steps our system is accurate in [1]. In simple terms, a RNNs loop through each element while keeping track of its current state, which is informed by what it has seen up to that point. An RNN is a particular kind of neural network with an internal loop.

3 Algorithm Implementation and Development

3.1 Forward Propagation

To achieve forward propagation, the first step in the study was to create the test data that would serve as the neural network's foundation. This was accomplished by using the standard specifications for Lorenz, as previously mentioned in the theory section, σ is equal to 10 β is equal to $\frac{8}{3}$, and the values for ρ were defined as 10, 28, and 40. The initial conditions were then set to $(x,\ y,\ z) = (0,\ 1,\ 1)$. Next, the data acquired was stacked into a (30000,4) matrix. Where each column represents a rho value as defined above (10, 28 and 40). The data is then rearranged into a Henkle matrix.

3.2 Backward Propagation

The second step involved training the neural network using the collected test data.Then, a piecewise decay of the learning rate was used to define the gradient descent. By doing this, the minimum is approached without being stepped over. The loss function MSE was then defined; it was chosen because it was the simplest to use. Next we tested the NN and ran data for ρ outside our recommended trained data. The data was then forecasted for ρ equal to 17 and 35, with the results shown in the next section.

4 Computational Results

The figures 2 are a representation of such apparently minimized loss function still incapable of predicting the chaotic nature of the Lorenz equations. As for 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 iterates. Note given more time the results would be reorganized into a proper manner.

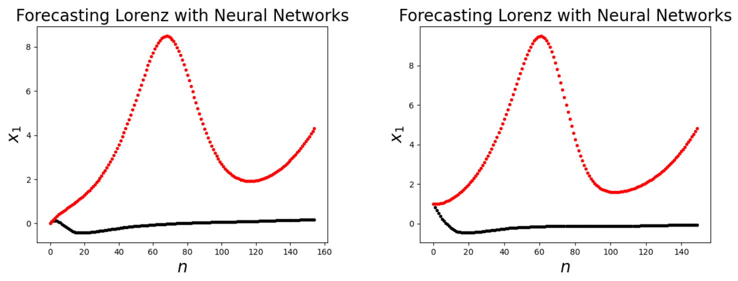

Figure 2: Rho data

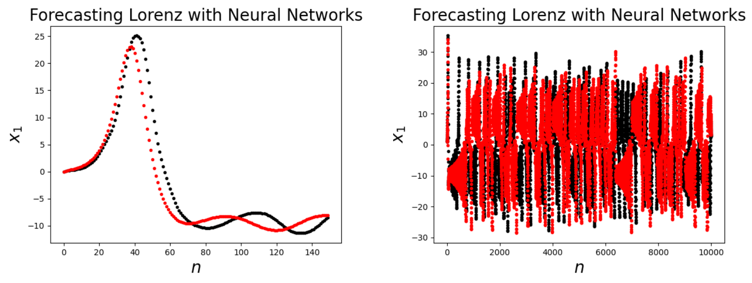

Figure 3: Non Recommended rho - rho 17 (left) and rho 35 (right)

5 Summary and Conclusions

To summarize, the objective of the study was to use recurrent neural networks to simulate the motion of the Lorenz equations. To find the loss function that produces the best predictions, we first have to train the NN using the data gathered from the Lorenz equations. In this case, the MSE was employed to acquire the results. The next objective was to ascertain how many time steps the prediction remained accurate, despite the shortage of time. The last step was determining if the recurrent neural networks had successfully learnt to predict data for the parameter ρ with values of 17 and 35 that were not included in the training set. Despite the fact that this study's results fell short of expectations, neural networks are still an effective function approximation tool.

References

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

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

Appendix A Python Functions

- `np.empty` Generates an empty array.
- `np.zeros` Generates an array of zeros.
- `np.linspace` Returns evenly spaced values over the specified interval.
- `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.
- `z = odeint(function, z0, t)`: calculates ordinary differential equations.

Appendix B Supplementary Graphs and Images

Appendix C 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

#Lorenz Parameters (rho = 10, 28, and 40)
sigma = 10
```

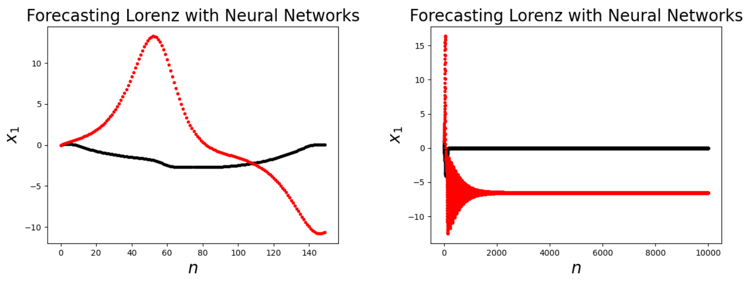

Figure 4: Extra forecast graphs

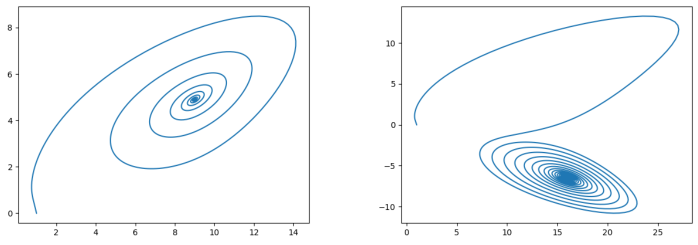

Figure 5: Extra lorenz graphs

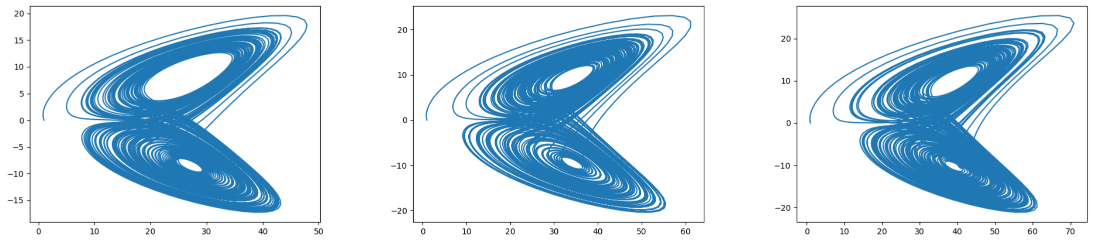

Figure 6: Extra lorenz graphs

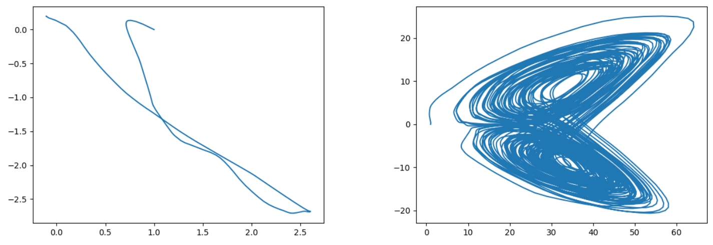

Figure 7: Extra lorenz graphs

```
rho = 28
beta = 8/3

#These are initial conditions (we can choose others)
x0 = 0
y0 = 1
z0 = 1
```

```python
#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.
sol1 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))

#Where the first column is composed of the values of x, the second of y and third of z
sol1.shape

#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

plt.plot(sol1[:,2], sol1[:, 0], label='x_coord')

#Solution of rho = 10
rho = 10
sol2 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol2[:,2], sol2[:, 0], label='x_coord')

#Solution of rho = 40
rho = 40
sol3 = odeint(Lorenz,(x0, y0, z0),t,args=(sigma, beta, rho))
plt.plot(sol3[:,2], sol3[:, 0], label='x_coord')

np.save('solutions_rho28', sol1)
np.save('solutions_rho10', sol2)
np.save('solutions_rho40', sol3)

#Data processing to feed into the mdoel
matrix_rho10 = np.load('/content/solutions_rho10.npy')
matrix_rho28 = np.load('/content/solutions_rho28.npy')
matrix_rho40 = np.load('/content/solutions_rho40.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.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"
#refactor this to loops because it's really ugly
data[0:10000,0:3] = matrix_rho10
data[10000:20000,0:3] = matrix_rho28
data[20000:30000,0:3] = matrix_rho40
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((9999*3,4))
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]
```

```python
xnp1 = np.empty((9999*3,4))
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]

#Just some testing to make sure that the slicing is done correctly
xnp1[0:1,0:4] == xn[1:2,0:4]
xnp1[9999:10000,0:4] == xn[10000:10001,0:4]
xnp1[19998:19999,0:4] == xn[19999:20000,0:4]
xnp1[20001:20002,0:4] ==xn[20002:20003,0:4]

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

def compute_loss(model, xn,xnp1, steps):
#Here you would have instead of xnforward, xn or xnp1

    loss = 0

    xpred = model(xn)

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

    return loss

def get_grad(model, xn,xnp1, steps):
#Here you would have instead of xnforward, xn or xnp1

    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 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 1000 steps use a learning rate of 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])
#lr = 0.00001

# Choose the optimizer
```

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

#Training
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))

#Forecasting the Lorenz
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(xpred[:155,0],'k.')
plt.plot(xn[:155,0],'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[:150,1],'k.')
plt.plot(xn[:150,1],'r.')
plt.title('Forecasting Lorenz with Neural Networks', fontsize = 20)
plt.xlabel('$n$', fontsize = 20)
plt.ylabel('$x_1$', fontsize = 20)

#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('/content/solutions_rho17.npy')
matrix_rho35=np.load('/content/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

#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

#Forecasting using new rho
M = 10000

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

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

#Plot Henon Trajectories for rho 35
fig = plt.figure()
plt.plot(xpred[:150,0],'k.')
plt.plot(dataRho35[: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 for rho 35
fig = plt.figure()
plt.plot(xpred[:10000,1],'k.')
plt.plot(dataRho35[: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 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 for rho 17
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')
```