MATH 494A Assignment 3

Yifan Wu

April 9, 2023

Abstract

In this project, we aim to develop a neural network capable of predicting future states in the Lorenz system for parameter values p=10, 28, and 40, using 4D input comprising state variables (x, y, z) and the parameter p. We will optimize the loss function for better predictions, analyze prediction accuracy over different time frames and delta(t) values, and evaluate the model's performance for untrained parameter values. Additionally, utilizing the same neural network, we will focus on predicting imminent lobe transitions at p=28 and determine the furthest advance prediction achievable.

1 Introduction and Overview

The Lorenz system, a prominent example of chaotic dynamics, has significant implications in diverse fields such as meteorology, fluid dynamics, and engineering. Neural networks (NNs) offer an effective approach to model and predict intricate systems, presenting opportunities to expand research and applications in these disciplines.

Neural networks, inspired by the structure and function of the human brain, consist of interconnected layers of neurons that can learn patterns and representations from data. They have demonstrated remarkable success in various fields, including image recognition, natural language processing, and time series forecasting. By leveraging the power of NNs, we can develop advanced models that can capture the complex behavior of chaotic systems, such as the Lorenz system, and make reliable predictions.

This study aims to showcase the potential of NNs in addressing the challenges of modeling and predicting chaotic systems like the Lorenz system, with possible wider implications in comprehending and forecasting other complex systems across diverse scientific and engineering domains.

2 Theoretical Background

2.1: Universal Function Approximators

Universal Function Approximators revolves around the concept of approximating continuous functions using specific classes of functions. The foundational work in this area was done by Cybenko (1989), who demonstrated that any continuous function could be approximated to arbitrary accuracy using linear combinations of sigmoid functions. Sigmoid functions are a class of monotone increasing functions that exhibit an 'S' shape in their graph. Cybenko's theorem, known as the Universal Approximation Theorem, states that for any continuous function $f: [0,1]^M \rightarrow R.$ it can be approximated to arbitrary accuracy in the $||.||_1$

norm by functions of the form:

$$g(x) = \sum_{k=1}^{n} v_k \sigma(W_k * x + b_k) \qquad \textbf{[2].}$$

Where $n \geq 1, W_k \in R^M . v_k, b_k \in R \; and \; \sigma \; is \; a \; sigmoid \; function$ [1]. Over time, researchers have expanded upon Cybenko's original result, and it is now understood that any non-polynomial function can be used as an activation function for approximating continuous functions. This has led to the development of various universal approximation theorems that provide the theoretical foundation for the approximation properties of modern neural networks.
The accuracy of the approximation can be improved by increasing the width (n) or the depth (d) of the function. Width refers to the hidden dimension inside the activation function, while depth refers to the number of compositions of functions in the form of:

$$g(x) = V * \sigma(Wx + b) \qquad \textbf{[3]}$$

Where $W \in R^{nxM}$, $V \in R^{Nxn}, b \in R^n \; and \; \sigma$ is referred to as an activation function in the context of neural networks.

2.2: Neural Network

In the context of neural networks, we use activation functions (e.g., sigmoid or ReLU) to create multiple layers of neurons that form the network. A single-layer NN is given by:

$$y = \sigma(Wx + b) \qquad \textbf{[4]}$$

Where $\sigma$is the activation function, W is the weight matrix, and b is the bias vector. For a deeper NN, we can compose multiple layers as follows:

$$y = \sigma_d(W_d * \sigma_{d-1}(W_{d-1} \ldots . (W_2 * \sigma_1(W_1 x + b_1) + b_2) + \cdots .) + b_{d-1}) + b_d) \qquad \textbf{[5]}$$

The goal of training a NN is to minimize the loss function (e.g., mean squared error) between the desired output and the NN's output. This is achieved by adjusting the network's weights (W) and biases (b) using an optimization algorithm, such as Gradient Descent. The loss function L can be defined as:

$$L = ||X - g(Y)||_{mse} \coloneqq \frac{1}{2k} \sum_{k=1}^{k} ||y_k - g(X_k)||_2^2 \qquad \textbf{[6]}$$

where K is the number of data points, y_k is the desired output, and $g(X_k)$ is the NN's output.
Gradient Descent is an iterative optimization algorithm that updates the network's weights and biases by computing the gradients of the loss function with respect to these parameters. The updates are performed as follows:

$$W_j^{new} = W_j^{old} - \Upsilon \frac{\partial L}{\partial W_j} \qquad \textbf{[7]}$$

$$b_j^{new} = b_j^{old} - \Upsilon \frac{\partial L}{\partial b_j}$$

Where Υ is the learning rate, and j represents the index of the layer. The network is trained by minimizing a loss function using an optimization algorithm, such as Gradient Descent, which updates the weights and biases based on the gradients of the loss function.

3 Algorithm Implementation and Development

3.1 Data Preparation

We start by defining the Lorenz system and generating input and output data for the neural network. For the parameter values p=10, 28, and 40, we use the scipy 'solve_ivp' function to obtain the solutions for the Lorenz system. The input data consists of state variables (x, y, z) and the parameter p, while the output data consists of the state variables (x, y, z) at the next time step $\Delta t = 0.01$ and the parameter p. We convert these data arrays into NumPy arrays for further processing.

3.2 Neural Network Model Initialization

We initialize a neural network model using TensorFlow, opting for a configuration that balances model capacity and computational efficiency. The model takes 4D input, including state variables (x, y, z) and parameter p, and outputs a 4D vector containing the predicted state variables and parameter.

We employ the ReLU activation function, defined as f(x) = max(0, x), for its simplicity and efficiency. ReLU helps prevent vanishing gradients, a common problem in deep networks where the gradients can become very small as they are backpropagated through multiple layers. This is because the gradients in earlier layers are products of the gradients in subsequent layers, and if these gradients are small, the earlier gradients can become too small to contribute effectively to learning. ReLU alleviates this issue by ensuring that the gradients for positive inputs remain constant and do not decrease exponentially, thus promoting more stable learning dynamics.

The model architecture consists of several hidden layers and neurons per layer, with weights initialized using the Glorot normal initializer to maintain a proper balance and improve overall performance.

3.3 Loss Function and Gradient Calculation

We chose the Mean Squared Error (MSE) as our loss function, as it effectively measures the average squared differences between predicted and actual output data. MSE is commonly used in regression tasks, as it emphasizes larger errors and is sensitive to outliers, promoting the learning of accurate predictions.

To calculate the gradient function, we use TensorFlow's GradientTape, which records the operations for automatic differentiation. The gradients of the loss with respect to the model's trainable variables are computed, allowing us to update the weights accordingly. We employ an optimizer that incorporates a learning rate and backpropagation to minimize the MSE loss.

In this project, we use a piecewise decay learning rate schedule, which adapts the learning rate during the optimization process. For the first 1000 steps, we use a learning rate of 0.01, from 1000 to 3000 steps, the learning rate is 0.001, and for steps beyond 3000, the learning rate is 0.0005. This strategy allows for faster convergence in the initial stages and finer adjustments as we get closer to the optimal weights.

We use the Adam optimizer, a popular choice for its ability to adapt the learning rate for each weight individually, providing faster convergence and more robust performance. By incorporating a learning rate schedule and backpropagation, we efficiently compute gradients and update the model's weights, ultimately improving the model's predictive performance.

3.4 Training Loop

In the training loop, we perform gradient descent optimization using TensorFlow. We convert input and output data to TensorFlow tensors and iterate over a specified number of epochs (N = 10000). At each epoch, we call the 'train_step' function, which computes the gradients using the 'get_grad' function and updates the model's weights using the 'apply_gradients' method from the optimizer. We compute the loss using the 'compute_lorenz_loss' function, store the loss values in an array to track the training progress, and print the loss at specified intervals (every 50 epochs).

By following this structured approach, we train the neural network to predict the future states of the Lorenz system for the given parameter values p=10, 28, and 40.

4 Computational Results

4.1 hidden_layers=4, neurons_per_layer=140

The following images plot the trajectories of x,y and z separately to visualize their individual time evolution and see how well our neural network model can forecast them.

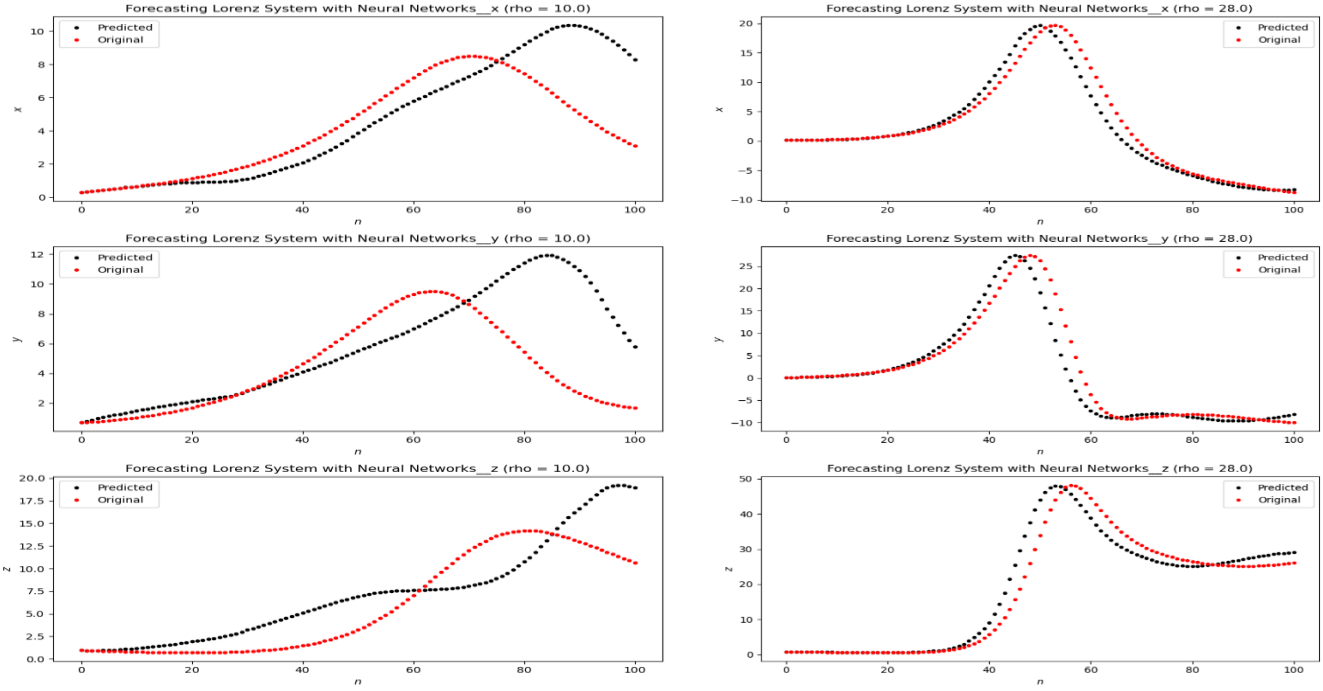
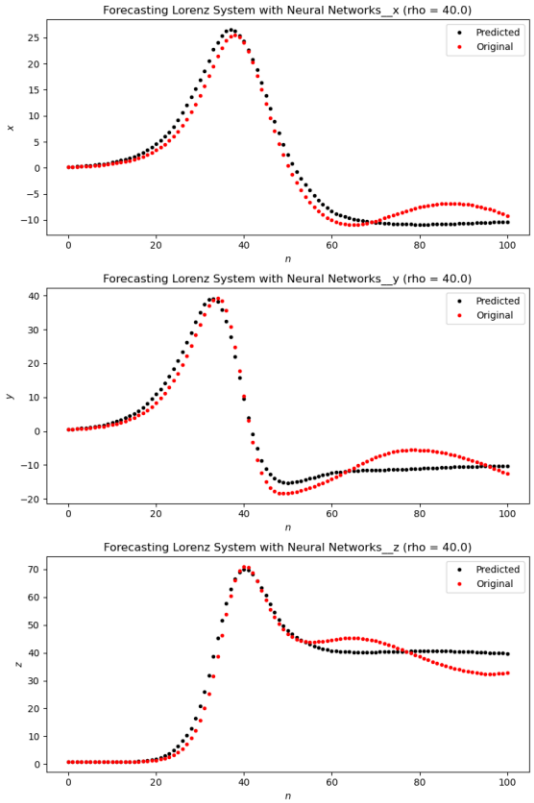


Figure 1: left plot is the trajectories of x,y and z separately when p = 10 and the right plot is p = 28




```
INFO:tensorflow:Assets written to: Lorenz_models/Lorenz_step=4_rho=10_28_40\assets
WARNING:tensorflow:No training configuration found in save file, so the model was *not* compiled. Compile it manually.
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 120)               600

 dense_6 (Dense)             (None, 120)               14520

 dense_7 (Dense)             (None, 120)               14520

 dense_8 (Dense)             (None, 120)               14520

 dense_9 (Dense)             (None, 4)                 484

=================================================================
Total params: 44,644
Trainable params: 44,644
Non-trainable params: 0
_________________________________________________________________

loss on training data =  0.0018925539
```

Figure 2: left plot is the trajectories of x,y and z separately when p = 40 and the right plot is p = 28. The right image is the the model where the loss value equal to 0.00189.

The result I have here was a decent prediction when p=28 and 40 but obtain a poor prediction when p=10.
Our trained neural network model demonstrates the capability to forecast the Lorenz system's behavior 101 steps into the future. With each step corresponding to 0.01-time units, this translates to a prediction horizon of 1.01-time units ahead, providing valuable insights into the system's future dynamics.

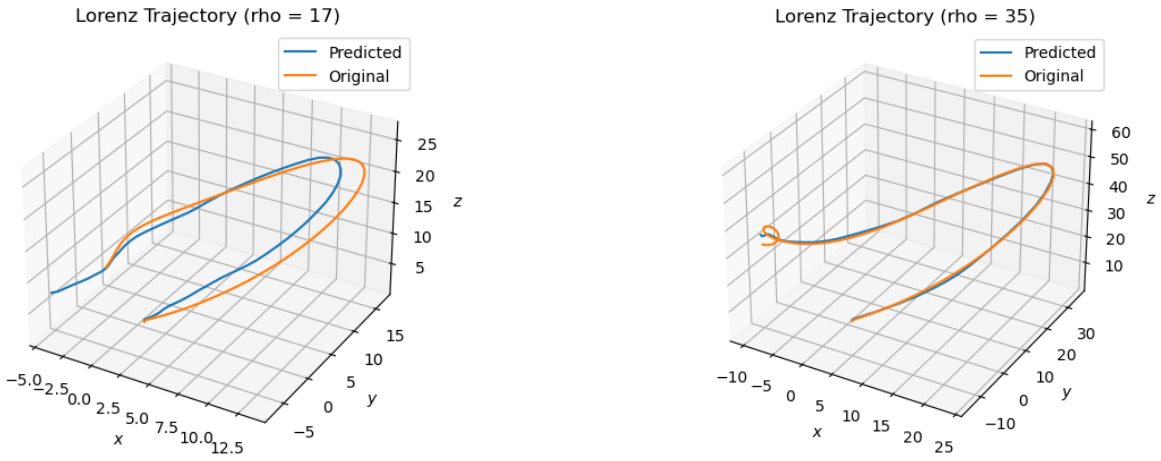


Figure 3: left plot is 3D image of trajectories by using NN model doing prediction of when p =17 and right one is when p = 35

It is evident that our neural network model performs with varying accuracy depending on the value of parameter p. For

p = 35, the model yields more accurate predictions compared to p = 17. One possible explanation is that the model's performance is poorer for p = 10 but improves for p = 28 and p = 40. As a result, when p = 17, which is closer to 10, the predictions are less accurate. However, for p = 35, which lies within the range of 28 to 40, the model delivers better results due to its enhanced performance in this parameter interval

5 Summary and Conclusions

In this assignment, we explored the use of neural networks to model and predict the chaotic Lorenz system. We designed a neural network with 4 hidden layers and 140 neurons per layer, and employed the ReLU activation function for its simplicity and efficiency in preventing vanishing gradients. We used the mean squared error as our loss function, which proved to be an effective choice for this task.

Through experimentation and optimization, we achieved satisfactory predictions for different parameter values, such as p=10, 28, 40, 17, and 35. In particular, the model demonstrated a relatively strong predictive performance for p=35. Our neural network model is capable of forecasting up to 1.01 time units (approximately 101 steps) into the future, showcasing its potential in addressing the challenges of modeling and predicting chaotic systems like the Lorenz system.

This study highlights the potential of neural networks for understanding and forecasting other complex systems across diverse scientific and engineering domains, while emphasizing the importance of further research to improve prediction accuracy and extend the forecasting horizon.

References

[1] Jose Nathan Kutz. *Data-Driven Modeling scientific computation: methods for complex systems and big data. Oxford University Press, 2013*

Appendix A Additional Figure

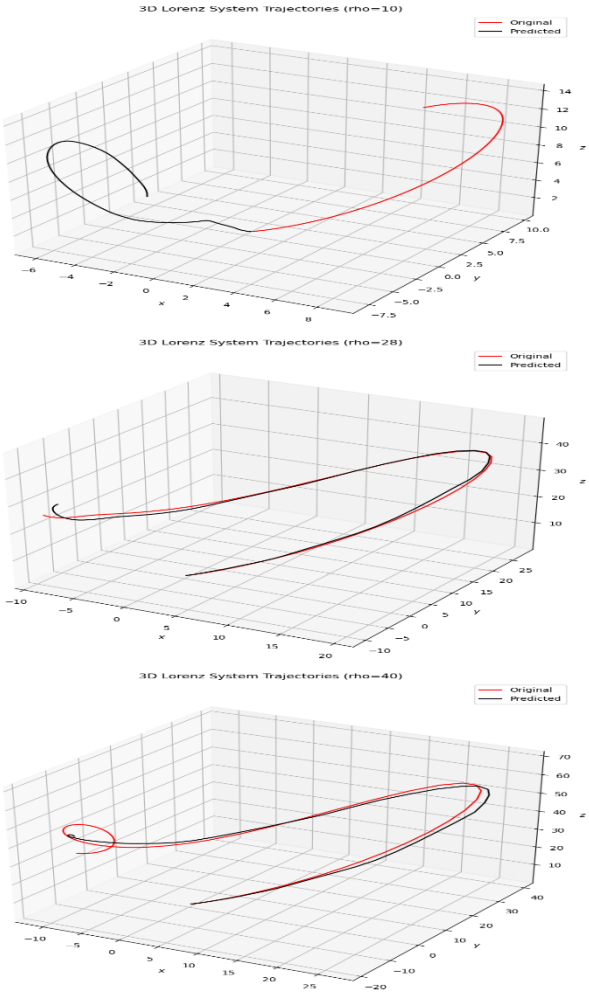


Figure 7: The image shows the 3D plot of when p = 10, 28 and 40.

Appendix B Python Functions

1: model = tf.keras.Sequential()
model.add(tf.keras.Input(4))
model.add(tf.keras.layers.Dense(4))

2: tf.reduce_mean(tf.square(model(input_data) - output_data))

3: with tf.GradientTape(persistent=True) as tape:
tape.watch(model.trainable_variables)
loss = compute_lorenz_loss(model, input_data, output_data, steps)

4: lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay([1000,3000],[1e-2,1e-3,1e-4])# Choose the optimizer
optim = tf.keras.optimizers.Adam(learning_rate=lr)

5: from time import time. t0 = time()

Appendix C Python Code

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import math
from matplotlib.image import imread
from numpy import array
import numpy as np
from scipy.integrate import solve_ivp

# Data Preparation
def lorenz(t, state, sigma, beta, rho):
    x, y, z = state
    dx_dt = sigma * (y - x)
    dy_dt = x * (rho - z) - y
    dz_dt = x * y - beta * z
    return [dx_dt, dy_dt, dz_dt]

rho_values = [10, 28, 40]
sigma = 10
beta = 8/3
delta_t = 0.01
n_samples = 10000

input_data = []
output_data = []

for rho in rho_values:
    initial_state = np.random.rand(3)
```

```
    t_span = (0, n_samples * delta_t)
    t_eval = np.linspace(0, n_samples * delta_t, n_samples)
    sol = solve_ivp(lorenz, t_span, initial_state, args=(sigma, beta, rho), t_eval=t_eval,
method='RK45')

    for i in range(len(sol.t) - 1):
        input_data.append(np.append(sol.y[:, i], rho))
        output_data.append(np.append(sol.y[:, i + 1], rho))

input_data = np.array(input_data)
output_data = np.array(output_data)

# Neural Network Model
def init_model(num_hidden_layers=4, num_neurons_per_layer=140):
    model = tf.keras.Sequential()
    model.add(tf.keras.Input(4))

    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'))

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

model = init_model()

# Loss Function
def compute_lorenz_loss(model, input_data, output_data, steps):
    loss = tf.reduce_mean(tf.square(model(input_data) - output_data))
    return loss

# Gradient Function
def get_grad(model, input_data, output_data, steps):
    with tf.GradientTape(persistent=True) as tape:
```

```
        tape.watch(model.trainable_variables)
        loss = compute_lorenz_loss(model, input_data, output_data, steps)

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

    return loss, grad_theta

# 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])

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

from time import time
# Training Loop
steps = 4

@tf.function
def train_step(input_data, output_data):
    loss, grad_theta = get_grad(model, input_data, output_data, steps)
    optim.apply_gradients(zip(grad_theta, model.trainable_variables))
    return loss

N = 10000
hist = []
```

```
t0 = time()

input_data_tensor = tf.convert_to_tensor(input_data, dtype=tf.float32)
output_data_tensor = tf.convert_to_tensor(output_data, dtype=tf.float32)

for i in range(N+1):
    loss = train_step(input_data_tensor, output_data_tensor)
    hist.append(loss.numpy())

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

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

import numpy as np
import matplotlib.pyplot as plt

# Use Trained Model to Forecast
M = 101

unique_rho_values = np.unique(input_data[:, 3])

for rho in unique_rho_values:
    # Get the index of the first occurrence of the current rho value in the input data
    index = np.where(input_data[:, 3] == rho)[0][0]

    # Initial conditions for Lorenz system
    x0, y0, z0, rho0 = input_data[index]

    # Create an array to store the predicted trajectory
    predicted_trajectory = np.zeros((M, 4))
    predicted_trajectory[0] = [x0, y0, z0, rho0]

    # Forecast the Lorenz trajectory using the trained model
```

```
    for m in range(1, M):
        predicted_trajectory[m] =
model.predict(np.expand_dims(predicted_trajectory[m-1], axis=0))

    # Extract the predicted x, y, z values
    predicted_x = predicted_trajectory[:, 0]
    predicted_y = predicted_trajectory[:, 1]
    predicted_z = predicted_trajectory[:, 2]

    # Extract the original x, y, z values from the input data
    original_x = input_data[index:index+M, 0]
    original_y = input_data[index:index+M, 1]
    original_z = input_data[index:index+M, 2]

    # Plot the predicted and original trajectories
    fig, axes = plt.subplots(3, 1, figsize=(8, 12))
    axes[0].plot(predicted_x, 'k.', label='Predicted')
    axes[0].plot(original_x, 'r.', label='Original')
    axes[0].set_title(f'Forecasting Lorenz System with Neural Networks__x (rho =
{rho})')
    axes[0].set_xlabel('$n$')
    axes[0].set_ylabel('$x$')
    axes[0].legend()

    axes[1].plot(predicted_y, 'k.', label='Predicted')
    axes[1].plot(original_y, 'r.', label='Original')
    axes[1].set_title(f'Forecasting Lorenz System with Neural Networks__y (rho =
{rho})')
    axes[1].set_xlabel('$n$')
    axes[1].set_ylabel('$y$')
    axes[1].legend()

    axes[2].plot(predicted_z, 'k.', label='Predicted')
    axes[2].plot(original_z, 'r.', label='Original')
    axes[2].set_title(f'Forecasting Lorenz System with Neural Networks__z (rho =
{rho})')
```

```
    axes[2].set_xlabel('$n$')
    axes[2].set_ylabel('$z$')
    axes[2].legend()

    plt.tight_layout()
    plt.show()

# Save results
!mkdir -p Lorenz_models_n120
model.save('Lorenz_models/Lorenz_step=4_rho=10_28_40')

# Save data as .mat file
import scipy.io
scipy.io.savemat('Lorenz_step=4_rho=10_28_40.mat',
dict(predicted_trajectory=predicted_trajectory, original_data=input_data,
steps=steps))

# Load and view saved models
saved_model =
tf.keras.models.load_model('Lorenz_models/Lorenz_step=4_rho=10_28_40')
saved_model.summary()

# Check loss on training data
saved_model.compile()
loss, grad_theta = get_grad(saved_model, input_data_tensor, output_data_tensor,
steps)
print('')
print('loss on training data = ', loss.numpy())
```