Can an AI surpass the RK4 method in predicting the Lorenz 63 system?

Sebastian M.D.

February 2023

Abstract

This paper explores the application of neural networks in predicting the trajectory of the Lorenz 63 system, a set of differential equations that showcase chaotic behavior. The Lorenz System was originally stipulated by Edward N. Lorenz in 1963 as a mathematical model for atmospheric convection. It is commonly used as a toy problem to explore chaos theory. Traditional numerical methods such as the Runga Kutta 4th order method can be used to solve and predict the system's behavior. This study explores the use of neural networks as an alternative approach to predict chaos. The methodology involves training a neural network on a dataset generated from the Lorenz system via the RK4 method. By using a small step size and high computational resources the network can generalize patterns and possibly later on efficiently predict the system's future state with different initial conditions. This paper aims to test the RNN LSTM, Transformers and RC-ESN network architectures. RNN and Transforms architectures are known for their ability to handle sequential data, while RC-ESN is known for its ability to capture chaotic systems. The results of the study will be compared to the the RK4 method to determine if the neural networks could surpass it with greater prediction horizon given similiar computational resources.

1 Theory

1.1 The Lorenz System

The original Lorenz system is a set of three differential equations. It is one of the earliest and most studied examples of systems that exhibit chaotic behavior. It is defined by the following equations:

$$\frac{dx}{dt} = \sigma(y - x) \tag{1}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{2}$$

$$\frac{dz}{dt} = xy - \beta z \tag{3}$$

where x, y, and z make up the state, t is time, and σ, ρ, and β are parameters. Typically, the values $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$ are used.

The Lorenz system is known for its butterfly-shaped attractor, which is a set of two points the systems tends to evolve around, regardless of the starting conditions. The attractor is visualized in Figure 1.

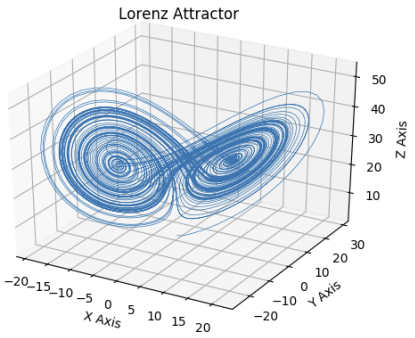

Figure 1: The Lorenz attractor for $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$.

1.2 Recurrent Neural Networks and LSTM

Neural networks are a type of machine learning AI model that are inspired by the structure of the human brain. They are composed of layers of interconnected nodes, which represent neurons, that process input data and produce an output. These nodes are usually connected to each other with linear transformations called weights and biases. The weights and biases are the parameters of the network that intially are randomly initialized and are optimized during the training process. Via the process of stochastic gradient descent the network can compute a gradient to slowly shift the parameters and minimize the error of the network's predictions. Over time the network can learn to make accurate predictions on the training data.

A Recurrent Neural Network (RNN) is a type of neural network architecture designed to recognize patterns in sequential data. What makes the RNN architecture special is that it's composed of a train of nodes, called cells, each connected to the next, where all the cells share the same parameters. When the input vector is fed to the first cell of the train, it creates an output and then the state of the node(the hidden state) is updated and passed along to the next cell. This update in hidden state makes it so the next cell can 'remember' the previous data inputed.

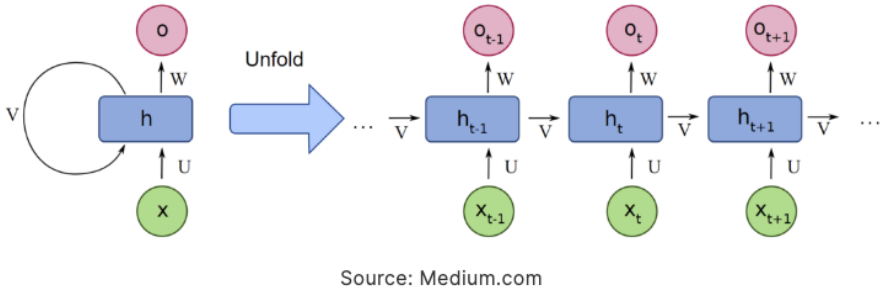

Figure 2: RNN architecture - https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/

However, RNNs have a significant limitation in that they struggle to learn long term dependencies due to the vanishing and exploding gradient problem. Long Short-Term Memory (LSTM) networks aim to solve this problem. The LSTM network is a modification of the RNN that introduces a second hidden state, called the cell state, which is updated differently from the traditional hidden state. This update process is controlled by some gates which detemine when to update the cell state. This modification of the hidden state is more effective when performing backpropgation

which allows the network to learn long-term dependencies.

1.3 Transformers

Transformers are a type of neural network architecture that was introduced in the paper "Attention is All You Need"(2017) by Google. Unlike RNNs, Transformers do not process the data in sequence, instead, they process the entire sequence at once. Transformers transform the data into an embedding layer where the positions are encoded into the data vectors themselves. This makes the network highly parallelizable. Transfomers has been quite revolutionary and is the basis of the state-of-the-art model GPT-4.

The key innovation however in Transformers is the self-attention mechanism. In self-attention, each token in the input sequence is transformed with trainable weights into three vectors, a query Q, key K and value V vector. The query vector states what a given token is looking for, the key vector states what the token offers, whilst the value vector is the information the token contains.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The self-attention mechanism takes the dot product between the keys and queries of the tokens, divides it by the square root of the key dimension(to prevent too small gradients) and then applies softmax to create a weights matrix. The weights matrix is used to determine how much each token in the sequence should contribute to the value vector of another token. For example, let the tokens be words in a sentence. In the sentence "The cat sat on the mat", the word "cat" would have a high affinity for the word "mat"(a query vector looking for the object of the sentence) and a low affinity for the word "the". This mechanism allows the network to learn the relationships between the tokens in the sequence.

The self attention mechanism is usually applied multiple times on the same tokens in parallell. This is called multi-head attention. Each 'head' has its own trainable transformations to look for different things in the tokens. The results are then concatenated and then aggregated through a trainable transformation to create the final ouput of the multi-head layer.

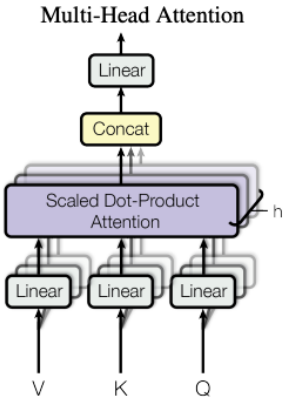

Figure 3: Taken from "All you need is attention" by Google.

The multi-head layer is paired with a position-wise feed-forward layer. The feed-forward layer applies two trainable linear transformations with a ReLU activation in between. The feed forward layer makes it so the network can learn more complex representations of the data. The

multi-head layer and feed-forward layer is usually repeated in blocks to form the transformer network. Between the layers goes a residual conection called layer normalization. Each time a multi-head or feed-forward layer has made a computation the result is added to the residual connection. This makes the gradient flow more stable.

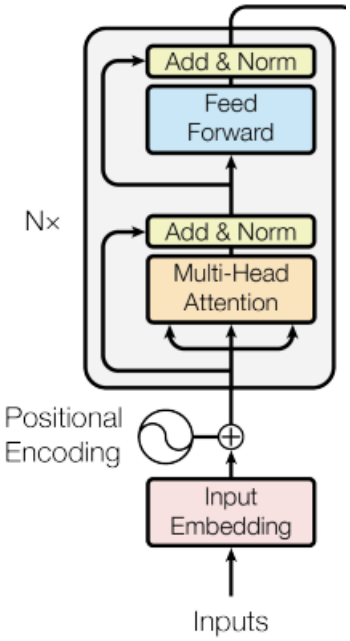

Figure 4: Basic Transformers architecture, taken from "All you need is attention" by Google.

2 Method

The code is written in Python and uses the PyTorch library for the neural network models. The project is publicly available on GitHub at: `https://github.com/SebCodesTheWeb/lorents-net`

2.1 Code Overview

The code is organized into several modules, each responsible for a specific part of the process. The main modules are:

- `lorenz.py`: This module implements the Lorenz system as well as the RK4 method.
- `generate_dataset.py`: This module generates the dataset used to train the networks. It uses the RK4 method to solve the Lorenz system and generates sequential data of the x, y, z values over time and saves them in a csv file
- `get_training_data.py`: Splits the dataset into training and testing data as well as into individual sequences of data.
- `train_network.py and train_transfomer.py`: These files train the RNN neural network and transformer network respectively.
- `lstm_rnn.py and transfomer.py`: These modules contain the class definitions for the neural networks.

- `backend.py`: Implements a simple backend to evaulate the networks in the 3d-visualizer frontend.

2.2 Key Code Examples

The data is initially processed in the `generate_dataset.py` module.

```
from lorenz import RK4
import pandas as pd
import numpy as np
from constants import seed_nbr, chunk_len, dt

np.random.seed(seed_nbr)
nbr_chunks = 100

dataset = []

for chunk_idx in range(nbr_chunks):
    init_pos = np.random.rand(3)
    pos = init_pos

    #offset each chunk by 2000 timsteps
    if chunk_idx > 0:
        for _ in range(2000):
            pos = RK4(pos, dt)

    for i in range(chunk_len):
        elapsedTime = i * dt
        pos = RK4(pos, dt)
        x, y, z = pos
        dataset.append({
            't': elapsedTime,
            'x': x,
            'y': y,
            'z': z
        })
```

In this code the constants are $dt = 0.05$, $seed\_nbr = 0$ and $chunk\_len = 2000$. The code generates 100 chunks of 2000 timesteps of the Lorenz system. The initial conditions are randomly generated and the system is solved using the RK4 method. The data is handled in separate chunks offsetted by 2000 timesteps to be trained on indivudually.

```
numerical_cols = ['x', 'y', 'z']
dataset[numerical_cols] = (
    dataset[numerical_cols] - dataset[numerical_cols].mean()
    ) / dataset[numerical_cols].std()

dataset.to_csv('lorentz-sequences.csv', index=False)
```

The dataset is then normalized to improve gradient flow and saved to a csv file.

The data is later on split trainable sequences in the `get_training_data.py` module.

```
import pandas as pd
import torch
import numpy as np
from constants import inp_seq_len, test_ratio, seed_nbr

dataset = pd.read_csv('lorentz-sequences.csv')
data_tensor = torch.tensor(dataset[['x', 'y', 'z']].values, dtype=torch.float32)
```

```

def create_seq(input):
    seq = []
    for i in range(len(input) - inp_seq_len):
        chunk = input[i:i + inp_seq_len]
        label = input[i + inp_seq_len:i + inp_seq_len+1]
        seq.append((chunk, label))
    return seq

data_seq = create_seq(data_tensor)
```

Here the *inp_seq_len* is set to 1. This makes is to the training data is split into a list of sequences, where each individual point has a label(the next value to predict) associated with it. These data sequences are laten split into a training and testing set with a 99/1 ratio.

The `LSTM_RNN` class defines a RNN network based on PyTorch's `nn.Module`:

```
import torch
import torch.nn as nn
from device import device

class LSTM_RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(LSTM_RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers=num_layers,
    batch_first=True)
        self.output_activation = nn.Linear(hidden_size, output_size)

    def forward(self, inputSeq):
        batch_size = inputSeq.size(0)
        state = self.init_state(batch_size)
        lstm_out, _ = self.lstm(inputSeq, state)

        prediction = self.output_activation(lstm_out[:, -1, :])

        return prediction

    def init_state(self, batch_size):
        state = (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(
    device),
                 torch.zeros(self.num_layers, batch_size, self.
    hidden_size).to(device))
        return state
```

`input_size` is the dimensionality of the input vector(in this case three for the x, y and z values). The constructor initializes the LSTM layer and output activation layer. The `forward` method is called during model training, sequentially passing the input through the LSTM cell and the activation layer to compute the output. The `init_state` method initializes the LSTM's internal state with zeros for each new batch.

The model is then instantiated like this:

```
#Hyperparams
hidden_size = 50
num_layers = 1
learning_rate = 0.0005
```

```
5
6  train_data = TensorDataset(x_train, y_train)
7  train_dataloader = DataLoader(train_data, batch_size=batch_size)
8
9  input_size = x_train.shape[2]
10 output_size = y_train.shape[2]
11 model = LSTM_RNN(input_size, hidden_size, output_size, num_layers).to(device)
12
13 loss_fn = nn.MSELoss()
14 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

In this case it uses a size for the hidden state of 50, which has proven effective, and uses a simple single layer RNN(as described in the theory). It uses mean squared error loss function and the adam optimizer with a learning rate of 0.0005.