What Neural Network Architecture is Best for Predicting the Lorenz 63 System?

Sebastian M.D.

March 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.

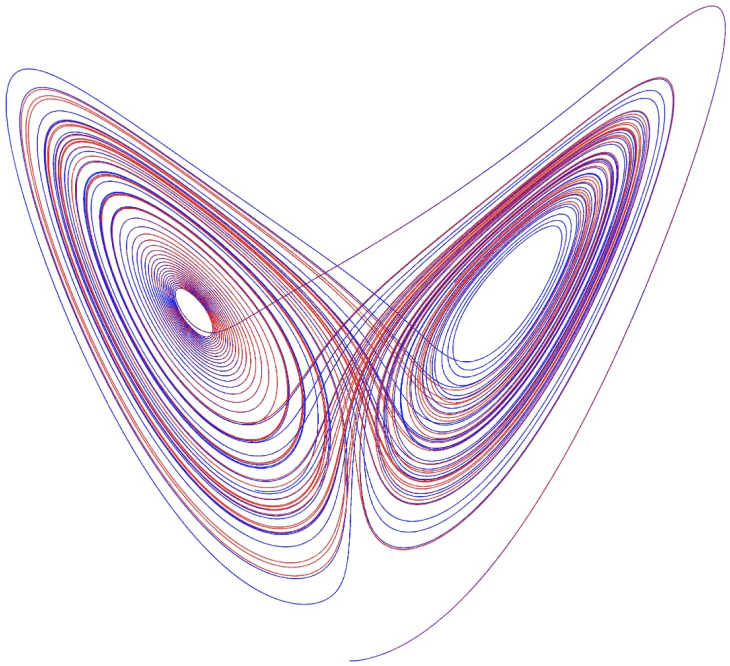

Figure 1: https://qparticle.wordpress.com/2014/07/11/lorenz-attractor/

Contents

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 2.

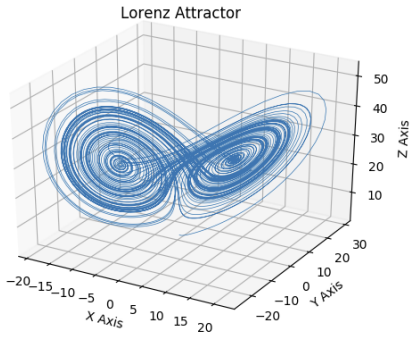

Figure 2: The Lorenz attractor for $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$.

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.

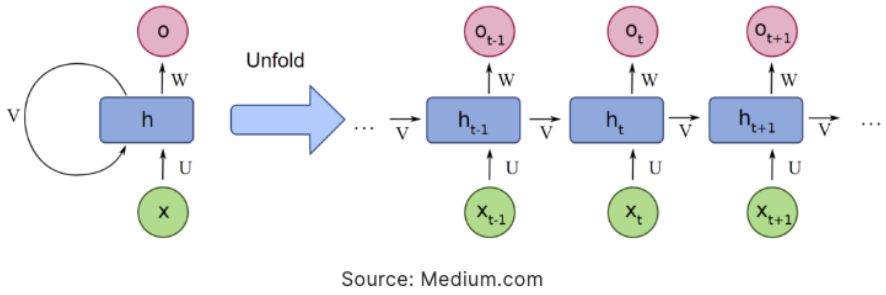

Figure 3: RNN architecture - https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/

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.

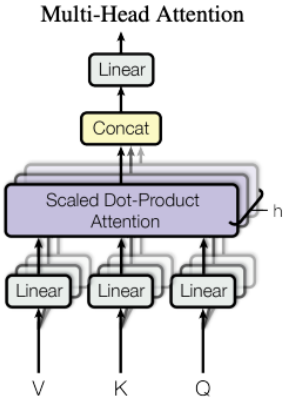

Figure 4: Taken from "All you need is attention" by Google.

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.

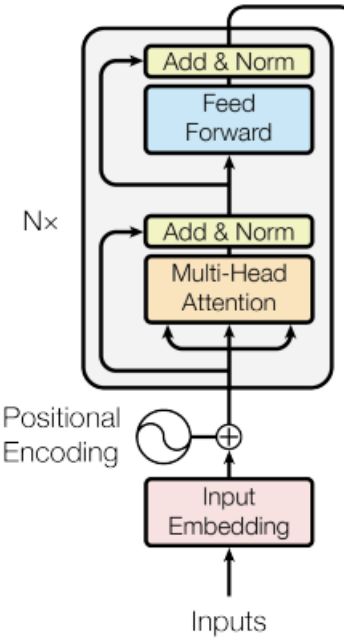

Figure 5: Basic Transformers architecture, taken from "All you need is attention" by Google.

1.4 Reservoir Computing and Echo State Networks

Reservoir Computing (RC) is a conceptual framework for understanding and designing RNNs, particularly in the context of dynamic systems and time series prediction. The fundamental principle behind reservoir computing is that only a part of the network is trained, while the rest of the network—or the "reservoir"—remains unchanged during training. This approach significantly reduces the computational cost of training RNNs and overcomes some of the issues related to gradient-based learning in traditional RNNs, such as vanishing and exploding gradients.

Echo State Networks (ESNs) are a particularly well-known implementation of reservoir computing. ESNs consist of a sparsely connected and randomly initialized recurrent layer—the reservoir—that is left untrained. The reservoir serves to project the input into a higher-dimensional space where the different parts of the input sequence become more linearly separable. Training only occurs in a readout layer, which is typically a linear model that is adjusted to map the reservoir states to the desired output.

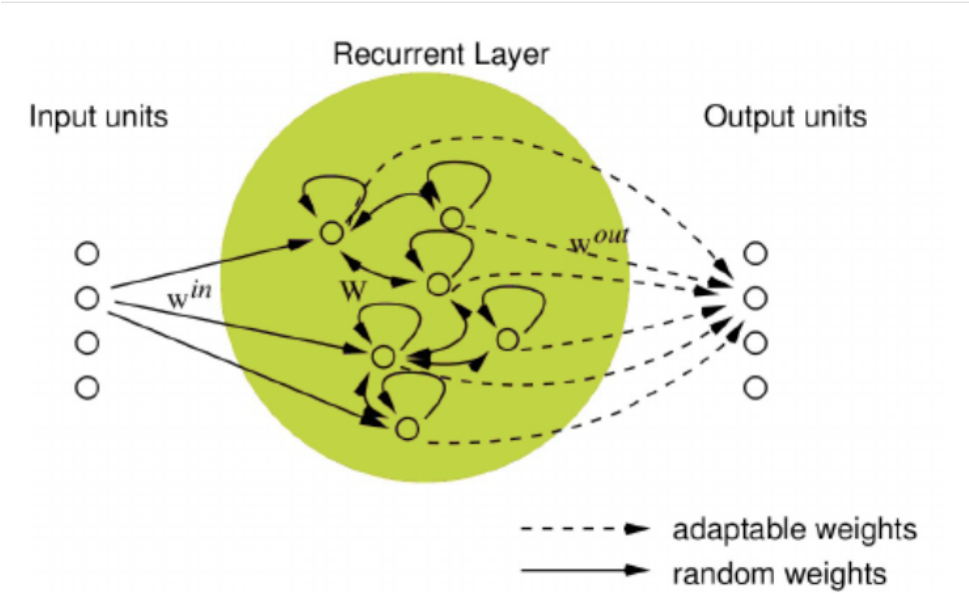

Figure 6: Research gate Joschka Boedecker

The key properties of an ESN are its dynamical richness and memory capacity, enabling it to process time-dependent information effectively. Reservoirs in ESNs possess a "fading memory," enabling them to handle input sequences with varying time scales. This property allows ESNs to maintain the context of earlier inputs while also adapting quickly to recent changes in the input stream.

The Echo State property is crucial for the functioning of ESNs. It dictates that the reservoir's state should eventually become independent of its initial state, given enough time and the right conditions for the reservoir's weights. Essentially, this means that the echoes of past inputs fade away, allowing the network to remain sensitive to the most recently received inputs while still retaining some historical context.

Training an ESN is straightforward and efficient. It involves collecting the states of the reservoir for a known input sequence (teaching signal), then using a supervised learning technique, such as linear regression, to train the readout weights such that the error between the predicted output and the actual target output is minimized.

One of the strengths of ESNs is that they can be applied to a variety of tasks without requiring the

laborious design of network architectures and weights that is typically necessary for conventional RNNs. This makes them versatile tools in situations where quick and efficient modeling of temporal dynamics is paramount.

In summary, Echo State Networks represent a streamlined, efficient approach to temporal pattern recognition and forecasting within the reservoir computing paradigm. Their simplicity and effectiveness have made them subjects of interest for various applications ranging from time series prediction to signal processing and control systems.

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 Generating Data

The training data is processed via the RK4 method:

```
import numpy as np

sigma =  10
rho = 28
beta = 8/3

def get_derivative(pos_vector):
    x, y, z = pos_vector

    dx_dt = sigma*(y-x)
    dy_dt = x*(rho-z)-y
    dz_dt = x*y - beta*z

    return np.array([dx_dt, dy_dt, dz_dt])

def RK4(pos_vector, dt):
```

```
    k1 = dt * get_derivative(pos_vector)
    k2 = dt * get_derivative(pos_vector + k1/2)
    k3 = dt * get_derivative(pos_vector+ k2/2)
    k4 = dt * get_derivative(pos_vector + k3)

    return pos_vector + (k1 + 2*k2 + 2*k3 + k4) / 6
```

Which is used 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 filel. Afterwards it is split into 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)

```

```python
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 the number of most recent data points the model will take into consideration when predicting the next point. By performing autocorrelation analysis a suitable *inp_seq_len* could be found.

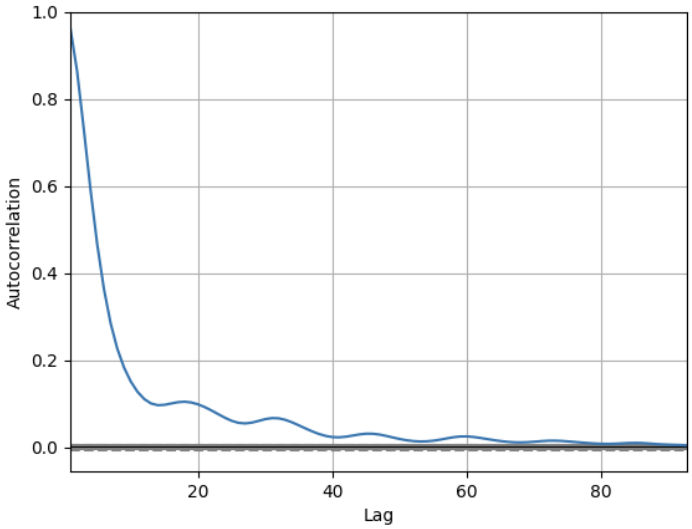

Figure 7: Autocorrelation shows how much the data is correlated with itself at different time lags

A significant drop of occurs at around lag 10-15, which makes *inp_seq_len* set to 10 a good value. Furthermore the function splits the training data into a list of separate, trainable sequences, where each individual sequence has a label(the next value to predict) associated with it. These data sequences are later split into a training and testing set with a 99/1 ratio.

2.3 Setting up the RNN

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

```python
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):
        """
        input_size: input feature size, in this case 3 for the Lorenz system
        output_size: output feature size, in this case 3 for the Lorenz system
        hidden_size: number of hidden units in the LSTM
        """
        super(LSTM_RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
15
16          self.lstm = nn.LSTM(input_size, hidden_size, num_layers=num_layers,
    batch_first=True)
17          self.output_activation = nn.Linear(hidden_size, output_size)
18
19      def forward(self, inputSeq):
20          batch_size = inputSeq.size(0) # inputSeq shape [batch_size, seq_len,
    feature_size]
21          state = self.init_state(batch_size)
22          lstm_out, _ = self.lstm(inputSeq, state)
23
24          # Select the last point in the sequence
25          prediction = self.output_activation(lstm_out[:, -1, :])
26
27          return prediction
28
29      def init_state(self, batch_size):
30          # Initializing the hidden and cell states for the LSTM based on the
    batch size
31          state = (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(
    device),
32                           torch.zeros(self.num_layers, batch_size, self.
    hidden_size).to(device))
33          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 passing the input through the torhc.nn.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:

```
1  #Hyperparams
2  hidden_size = 50
3  num_layers = 1
4  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.

The RNN network is later on trained with the following training loop:

```
1  def train(dataloader, model, loss_fn, optimizer):
2      size = len(dataloader.dataset)
3      num_batches = len(dataloader)
4      model.train()
5      running_loss = 0.0
6
```

```
    for batch_nbr, (seq, label) in enumerate(dataloader):
        seq, label = seq.to(device), label.to(device)
        label = label.squeeze(1) # Remove the extra middle dimension, in this case label shape is [batch_size, 1, feature_size]
        prediction = model(seq)
        loss = loss_fn(prediction, label)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_nbr % 100 == 0:
            current = batch_nbr * len(seq)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    running_loss /= num_batches
    print(f"Average loss for epoch: {running_loss:>7f}")

epochs = 5
for t in range(epochs):
    print(f"epoch {t + 1} \n--------------")
    train(train_dataloader, model, loss_fn, optimizer)
print("Done")
torch.save(model.state_dict(), 'lstm_rnn_lorenz.path')
```

The training loop iterates over the dataset, feeding batches of data to the model, calculating the loss, and updating the model parameters through backpropagation. Finally after five epochs, the model's parameters are saved.

2.4 Setting up the transformers architecture

The transfomers class is written as follows:

```
import math
import torch
from torch import nn, Tensor
from torch.nn import TransformerEncoder, TransformerEncoderLayer

class TransformerModel(nn.Module):
    def __init__(self, ntoken: int, d_model: int, nhead: int, d_hid: int, nlayers: int, dropout: float = 0.5):
        """
        ntoken: The size of the vocabulary (total number of unique tokens).
        d_model: The dimensionality of the token embeddings (the size of the vectors that represent each token).
        nhead: The number of attention heads in the multi-head attention mechanisms.
        d_hid: The dimensionality of the feedforward network model in the transformer encoder.
        nlayers: The number of sub-encoder-layers in the transformer encoder.
        dropout: The dropout rate, a regularization technique to prevent overfitting.


        """
        super().__init__()
        self.model_type = 'Transformer'
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        encoder_layers = TransformerEncoderLayer(d_model, nhead, d_hid, dropout, batch_first=True)
```

```
        self.transformer_encoder = TransformerEncoder(encoder_layers, nlayers)
        ##Use linear layer instead of traidiontal embedding layer due to
    continous data
        self.input_linear = nn.Linear(3, d_model)
        self.d_model = d_model
        self.output_linear = nn.Linear(d_model, 3)

        self.init_weights()

    def init_weights(self) -> None:
        initrange = 0.1
        self.input_linear.weight.data.uniform_(-initrange, initrange)
        self.input_linear.bias.data.zero_()
        self.output_linear.bias.data.zero_()
        self.output_linear.weight.data.uniform_(-initrange, initrange)

    def forward(self, src: Tensor) -> Tensor:
        """
        Arguments:
            src: Tensor, shape ``[seq_len, batch_size]``
            src_mask: Tensor, shape ``[seq_len, seq_len]``

        Returns:
            output Tensor of shape ``[seq_len, batch_size, ntoken]``
        """
        src = self.input_linear(src) * math.sqrt(self.d_model)
        src = self.pos_encoder(src)
        output = self.transformer_encoder(src)
        output = self.output_linear(output)
        return output
```

This transformers class utilizes the inbuild TransofrmerEncoder and TransformerEncoderLayer built into pytorch. The input is first passed through a linear layer to transform the input data into the d_model dimensionality. The input is then passed through a positional encoding layer to add information about the position of the tokens in the sequence. The output is then passed through the transformer encoder and then through a linear layer to transform the output back to the original dimensionality. The positional encoding is defined as:

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) /
     d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x) :
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The positional encoding is a sine and cosine function of different frequencies that are added to the input data. This is to give the network information about the position of the tokens in the

sequence. The frequencies are chosen to be logarithmically spaced.

The model is then trained in a similar fashion to the RNN model.

```
from get_transformer_training_data import x_train, y_train
from transformer import TransformerModel
from torch import nn
from device import device
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim
from torch.optim.lr_scheduler import ExponentialLR
import torch

# Hyperparams
hidden_dim = 500
nhead = 2
num_layers = 2
learning_rate = 0.0005
batch_size = 64
#vocab_size does not matter for this implementation
vocab_size = 3
# d_model has to be even due to how positional encoding is implemented,
    requiring pairs of sin and cosine positions
d_model = 128
dropout=0

train_data = TensorDataset(x_train, y_train)
train_dataloader = DataLoader(train_data, batch_size=batch_size)

model = TransformerModel(ntoken=vocab_size, d_model=d_model, nhead=nhead, d_hid=
    hidden_dim, nlayers=num_layers,dropout=dropout).to(device)

loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
scheduler = ExponentialLR(optimizer, gamma=0.9)
```

The main difference is that the model is trained with a learning rate scheduler to decay the learning rate over time, and there are some other hyperparameters that are different.

2.5 Setting up the RC-ESN

2.6 Training process