Dynamical Systems Theory in Machine Learning & Data Science Final Project

Maiqi Zhou, 4732289, Data and Computer Science

github:https://github.com/maiqizhou/DSML

Task 1: Summary of the paper

Paper: Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case

Introduction: The authors developed a transformer-based model for time series forecasting. It processes an entire data sequence and uses self-attention mechanisms to learn dependencies in the sequence. It applied influenza like illness(ILI) forecasting as a case study.

Model:

Data: The data is normalized using min-max scaling, which is based on the dataset's maximum and minimum values. Then it uses a fixed-lenth window to construct labeled datasets.

The input time series data is first passed through a fully connected layer to map it to a higher-dimensional space. Since the Transformer does not inherently process data in a sequential manner, positional encoding is added to the input to provide information about the order of the time series. It uses sine and cosine functions to encode the position of each time step in the sequence.

The transformer-based model consists of an encoder and a decoder. The encoder processes a time series input, experiences self-attention and feeds forward layers, then the decoder also has a time series input, which makes it different and predicts more effectively.

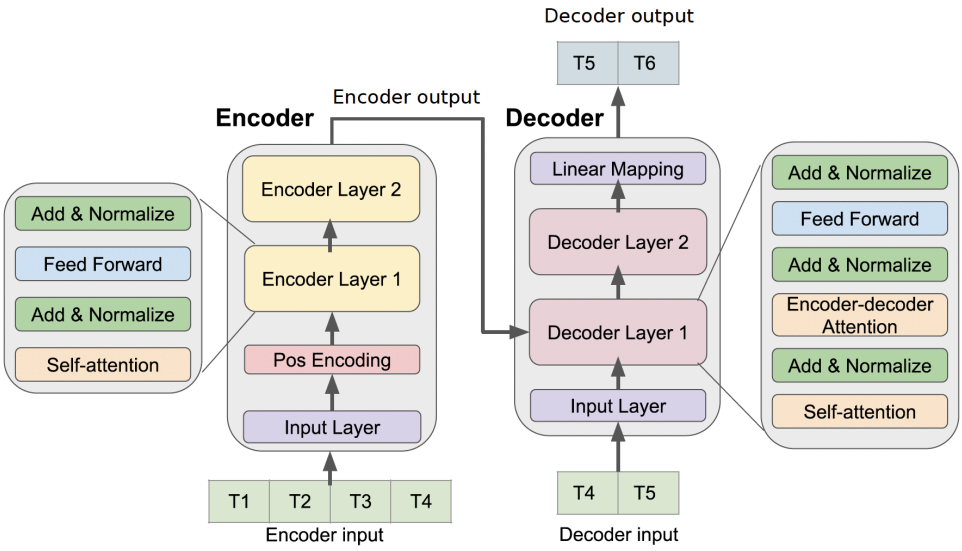


Figure 1: Architecture of the transformer model

For the time delay embedding, this work leverages the state space modeling concept to capture the complex dynamics of influenza prevalence, which enables the model to learn both short and long-term dependencies in the time series data. The formula is as follows:

$$\mathrm{TDE}_{d,\tau}(x_t) = \left(x_t, x_{t-\tau}, \ldots, x_{t-(d-1)\tau}\right)$$

This method is based on (Takens' theorem) (Takens, 1981), which states variables ("phase space")can be recovered in the delay coordinates of the observed variables. Each scalar value x_t into a d-dimensional time-delay space.

The authors also mentioned that sequence models like RNNS, LSTM, and Seq2Seq are used to handle sequential data. LSTMs address the gradient vanishing and exploding problems in RNNs by using three gates (input, forget, and output) to control information flow. LSTMs address the limitations of RNNs in capturing long-term dependencies.

Experiments:

1. One-Step-Ahead Forecasting Using ILI Data Alone. Among Transformer, ARIMA, LSTM, Seq2Seq with attention models, the transformer shows the best performance.
2. One-Step-Ahead Forecasting Using Feature Vectors. This experiment aims to test if adding features improves the accuracy of forecasting. The results show improvement is slight.
3. Experiment 4: Forecasting Using Time Delay Embedding. Through modeling the phase space of the dynamical system, the authors show time delay embeddings improve the model's ability to capture underlying system dynamics.

Conclusions:

1. The transformer model is a powerful model to forecasting time series data. Compared to other deep learning models, it can learn complex dependencies through self-attention mechanisms.
2. The model is more flexible for more complex cases as a generic framework.

Task 2,3 and bonus: Compare the power spectra between the ground truth time series and the generated time series

In [9]:
```
from psd import power_spectrum_error
import torch
import numpy as np
```

Compute the average power spectrum of Lorenz-63(epoch=10)

In [ ]:
```
# Load the generated time series
generated_series = np.load("generated_lorenz63(epoch=10, lr=0.0001).npy")
true_series = np.load("lorenz63_test.npy")


# Convert to tensors
generated_series = torch.tensor(generated_series, dtype=torch.float32)
true_series = torch.tensor(true_series, dtype=torch.float32)
# Ensure both have the same shape for comparison
generated_series = generated_series.unsqueeze(0)
true_series = true_series.unsqueeze(0)
initial_steps = true_series[:, :30, :]
generated_series = torch.cat((initial_steps, generated_series), dim=1)

ps_distance = power_spectrum_error(generated_series, true_series)
print(f"\nPower Spectrum Distance(Lorenz-63, 20epochs): {ps_distance:.6f}
```

```
Power Spectrum Distance(Lorenz-63, 20epochs): 0.660247
```

Compute the average power spectrum of Lorenz-63(epoch=20)

In [ ]:
```
# Load the generated time series
generated_series = np.load("generated_lorenz63(epoch=20, lr=0.00007).npy"
true_series = np.load("lorenz63_test.npy")


# Convert to tensors
generated_series = torch.tensor(generated_series, dtype=torch.float32)
true_series = torch.tensor(true_series, dtype=torch.float32)
# Ensure both have the same shape for comparison
generated_series = generated_series.unsqueeze(0)
true_series = true_series.unsqueeze(0)
initial_steps = true_series[:, :30, :]
generated_series = torch.cat((initial_steps, generated_series), dim=1)

ps_distance = power_spectrum_error(generated_series, true_series)
print(f"\nPower Spectrum Distance(Lorenz-63, 20epochs): {ps_distance:.6f}
```

```
Power Spectrum Distance(Lorenz-63, 20epochs): 0.568479
```

Compute the average power spectrum of Lorenz-96(epoch=20)

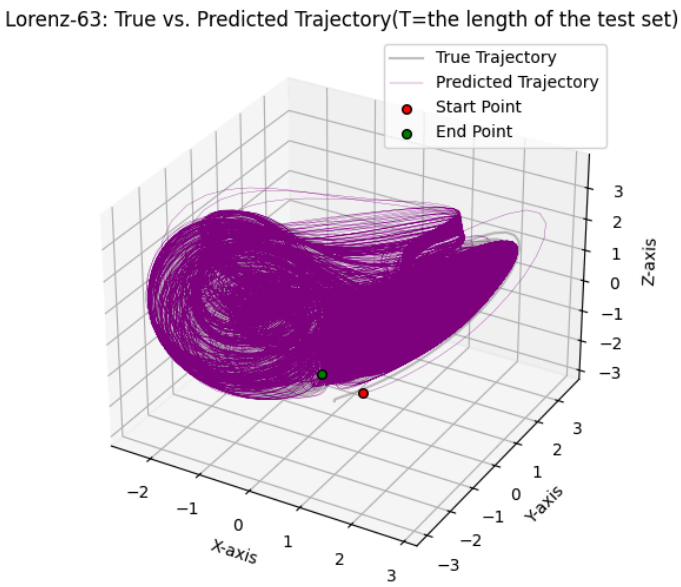
In [ ]:
```
# Load the generated time series
generated_series = np.load("generated_lorenz96(epoch=20, lr=0.00007).npy"
true_series = np.load("lorenz96_test.npy")


# Convert to tensors
generated_series = torch.tensor(generated_series, dtype=torch.float32)
true_series = torch.tensor(true_series, dtype=torch.float32)
# Ensure both have the same shape for comparison
generated_series = generated_series.unsqueeze(0)
true_series = true_series.unsqueeze(0)
initial_steps = true_series[:, :30, :]
generated_series = torch.cat((initial_steps, generated_series), dim=1)

ps_distance = power_spectrum_error(generated_series, true_series)
print(f"\nPower Spectrum Distance(Lorenz-96, 20epochs): {ps_distance:.6f}
```

```
Power Spectrum Distance(Lorenz-96, 20epochs): 0.467019
```

Visualization

1. The image below shows the lorenz-63 trained for 10 epochs with transformer model generated the whole test dataset steps, which the initial sequence is the first encoder sequence lenth steps.

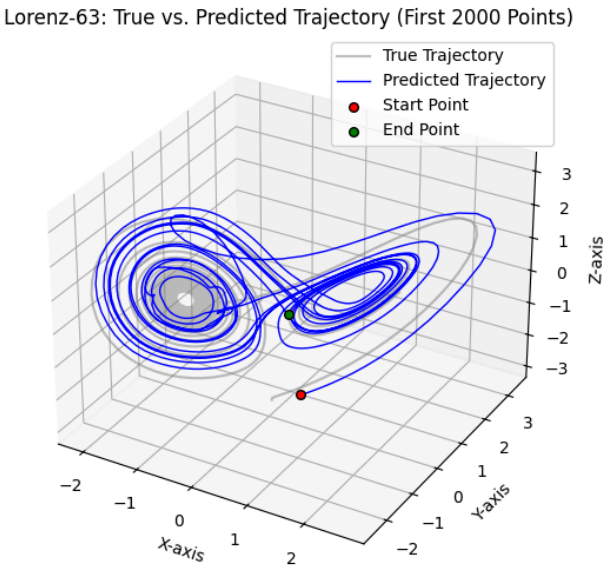
In [35]:
```
image_path = "Lorenz63(10epochs)_100000steps.png"
display(Image(filename=image_path, width=600))
```



2. This image shows the lorenz-63 trained for 20 epochs with transformer model generated 2000 total steps, which the initial sequence is the first encoder sequence lenth steps.

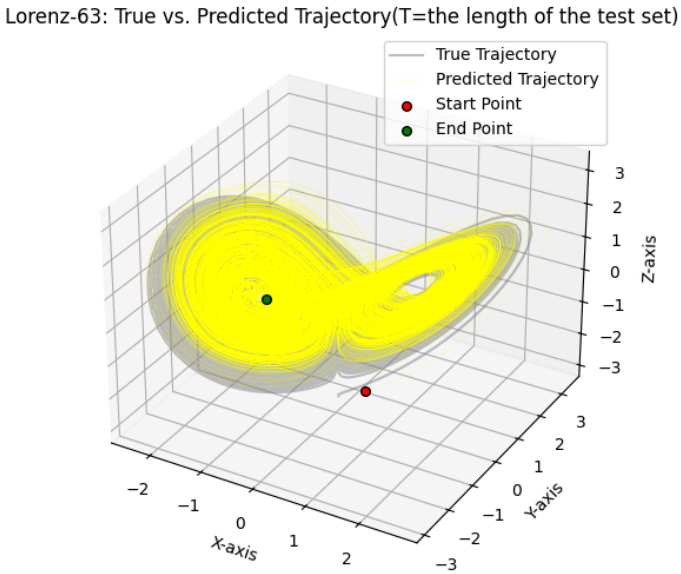
In [34]:
```
from IPython.display import Image, display
```

```
image_path = "Lorenz63_2000steps.png"
display(Image(filename=image_path, width=600))
```



3. The image below shows the lorenz-63 trained for 20 epochs with transformer model generated the whole test dataset steps, which the initial sequence is the first encoder sequence lenth steps.

In [17]:
```
image_path = "Lorenz63_100000steps.png"
display(Image(filename=image_path, width=600))
```



We can see that the model performs better after training for 20 epochs.

4. The image below shows the Lorenz-96 system trained for 20 epochs. Since it has 20 features, I visualize only three of them for clarity.

In [37]:
```
image_path = "Lorenz96(20epochs)_100000steps.png"
display(Image(filename=image_path, width=600))
```