Deep learning based Koopman operator estimation of Non-linear function

Rishant pal
Electronics and Communication Engineering, IIT Guwahati
p.rishant@iitg.ac.in

I. Introduction

This study explores complex systems that don't follow simple rules and are tough to understand. It focuses on a method called Koopman operator theory to make these systems easier to handle by turning them into simpler, linear models. To do this, the study suggests using neural networks, a type of artificial intelligence, to find key patterns in the data. This approach helps deal with two big challenges in studying dynamic systems: dealing with unknown equations, like in climate science or neuroscience, and simplifying the intricate details of these systems. The report outlines a step-by-step process, emphasizing the use of specific networks to create clear and simple models, and provides examples to show how well this method works for understanding complex systems.

II. Koopman Operator

In 1931, B. O. Koopman introduced the Koopman operator as a framework for describing dynamical systems through evolving functions in a measurement space. The recent surge in interest surrounding Koopman analysis stems from its capacity to linearly represent nonlinear dynamics, offering opportunities for advanced prediction and control through established linear system theories. However, obtaining finite approximations of this infinite-dimensional operator poses a formidable

$$\mathcal{K}g \triangleq g \circ \mathbf{F} \quad \Longrightarrow \quad \mathcal{K}g(\mathbf{x}_k) = g(\mathbf{x}_{k+1}).$$

challenge.

Dynamic Mode Decomposition (DMD) is a widely used method for creating finite approximations of the Koopman operator. It excels at identifying coherent structures in high-dimensional systems but faces limitations in capturing nonlinear behavior.

Deep learning methods have emerged as a promising solution for identifying Koopman eigenfunctions, which are essential for spanning invariant subspaces and providing a concise linear description. This pursuit is driven by the need to accurately and efficiently represent these eigenfunctions, prompting the exploration of powerful deep learning techniques.

III. Dataset Generation

For the data generation, we employ a nonlinear function to create a dynamic dataset. The nonlinear function, defined as

$$f(x) = x + \sin(x)$$

, serves as the governing rule for the evolution of each array. The dataset consists of a matrix with dimensions '5000 * 100'. The initial array, denoted as 'x1' of size 5000, is generated by evenly sampling values from -100 to 100, which are put into the first column of dataset. Subsequent column arrays are generated by applying the nonlinear function to the preceding array, creating a temporal sequence of states. This dataset is taken as input to the neural network model.

$$f(x) = x + \sin(x)$$

$$x_{k+1} = f(x_k)$$

The resulting dataset, named 'data array', is further processed to create an output array, 'data array op', where each column represents the next state of the corresponding column in the input array. This dataset generation process is designed to capture dynamic and nonlinear relationships among array elements, making it suitable for exploring complex system behaviors.

To evaluate the performance of models trained on this dataset, the data is split into training and testing sets. The training set, comprising 80% of the data, is denoted as 'train data', while the remaining 20% constitutes the testing set, denoted as 'test data'. Similarly, the corresponding output datasets are split into 'train data op' and 'test data op'. This division allows for the robust assessment of model generalization and predictive capabilities.

The code, written in Python, leverages the NumPy library for efficient array operations. This dataset generation methodology provides a versatile foundation for investigating and modeling dynamic systems, offering researchers an opportunity to explore the complexities inherent in nonlinear relationships.

IV. Neural network training

Neural networks are valuable for approximating the Koopman operator in non-linear functions. The architecture consists of:

1. Encoder (Non-linear): Transforms input data nonlinearly to capture intricate features.

2. BottleNeck (Linear Layers): Captures linear aspects, with weights corresponding to Koopman operator eigenvalues.

3. Decoder (Non-linear): Reconstructs transformed representation, incorporating non-linearity.

Training involves regression, optimizing parameters using gradient descent and the **Adam optimizer**. Post-training, bottleneck weights represent Koopman operator eigenvalues, crucial for understanding system dynamics.

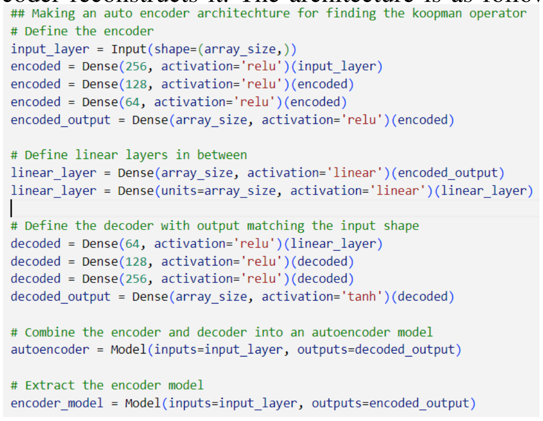
In summary, this architecture, with encoder-decoder and a linear bottleneck, is a powerful tool for approximating and extracting the Koopman operator from non-linear functions.

A. *Dataset*

We took the input and output data set from the data set generation part which we fed through the proposed model.

B. *Architecture of an Artificial Neural Network*

Neural network architecture, organized into input, hidden, and output layers with interconnected neurons characterized by learned weights and biases, adapts its depth and width to the task complexity, allowing deep networks for intricate patterns and shallow networks for simpler tasks. The autoencoder consists of an encoder and a decoder and the linear layer within. The encoder compresses the input data, and the decoder reconstructs it. The architecture is as follows:

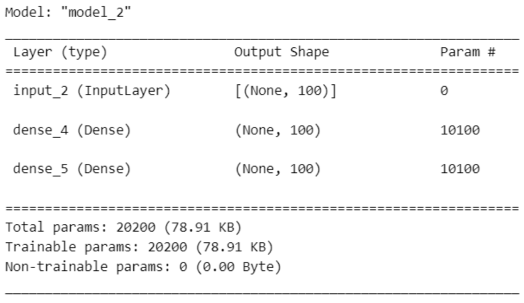


```
## Making an auto encoder architechture for finding the koopman operator
# Define the encoder
input_layer = Input(shape=(array_size,))
encoded = Dense(256, activation='relu')(input_layer)
encoded = Dense(128, activation='relu')(encoded)
encoded = Dense(64, activation='relu')(encoded)
encoded_output = Dense(array_size, activation='relu')(encoded)

# Define linear layers in between
linear_layer = Dense(array_size, activation='linear')(encoded_output)
linear_layer = Dense(units=array_size, activation='linear')(linear_layer)

# Define the decoder with output matching the input shape
decoded = Dense(64, activation='relu')(linear_layer)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(256, activation='relu')(decoded)
decoded_output = Dense(array_size, activation='tanh')(decoded)

# Combine the encoder and decoder into an autoencoder model
autoencoder = Model(inputs=input_layer, outputs=decoded_output)

# Extract the encoder model
encoder_model = Model(inputs=input_layer, outputs=encoded_output)
```

We then compiled the model and used mse or mean-squared error loss function for training the data through epochs(25) with batch size of 32. The mean squared error (MSE) is calculated using the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

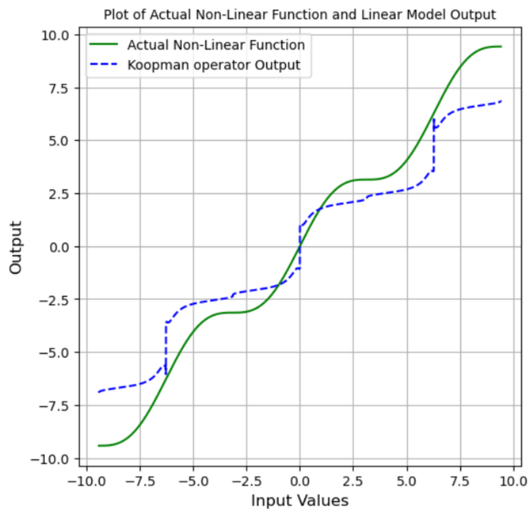
where n is the number of data points, y_i is the actual value, and $\hat{y}_i$ is the predicted value. We then extracted the linear model from the trained auto encoder model to generate the final prediction.(fig 2)



```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 100)]             0

 dense_4 (Dense)             (None, 100)               10100

 dense_5 (Dense)             (None, 100)               10100

=================================================================
Total params: 20200 (78.91 KB)
Trainable params: 20200 (78.91 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

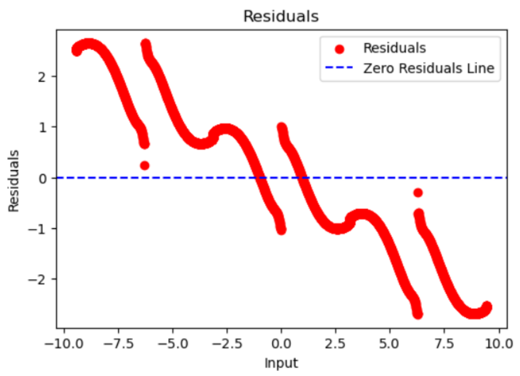
V. Results

After obtaining the weights of the linear model from the fully trained auto encoder, which represent the Koopman operator and the linear estimation of the nonlinear function, we visualized the chosen nonlinear function alongside the output of the linear model for a thorough analysis.



Subsequently, we assessed the performance using error metrics, specifically Mean Squared Error (MSE) and R-squared. The obtained values were as follows:
Mean Squared Error:2.558672877611617
R-squared:0.9247462013619882

To further scrutinize the predictive accuracy, we plotted the residuals to examine the deviations between the predicted output and the actual output.



VI. Conclusion

In summary, we designed a custom non-linear function tailored to our specific requirements and employed a neural network with an auto-encoder architecture. This network was trained to extract the Koopman operator from the linear model embedded within the encoder and decoder components. Subsequently, we conducted training for successive steps of the system and assessed the model's performance on validation data. Ultimately, we utilized the trained model to predict the output of the linear model using a sample array, comparing the results with the actual non-linear model output. This comprehensive analysis demonstrates that the Koopman operator-based model closely aligns with the non-linear function, as validated through both qualitative and quantitative measures.

References

[1] Lusch, B., Kutz, J.N. and Brunton, S.L., 2018. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature communications*, 9(1), p.4950.

[2] Haojie Shi, Graduate Student Member, IEEE, Max Q.-H. Meng† , Fellow, IEEE, Deep Koopman Operator with Control for Nonlinear Systems, *IEEE ROBOTICS AND AUTOMATION*.

Note: This report was submitted to the EE695 (Data Driven System theory) course assignment by Rishant Pal.