**ECE 397: Individual Study**

**Neural Network Approximation of Koopman Operators**

Atindraya Ganti (ganti3)

Fall 2019

Professor Subhonmesh Bose

1. Introduction

During this semester, I was tasked with researching the use of neural network models in the analysis of nonlinear dynamical systems. Specifically, this project focuses on using deep neural networks to learn dictionary functions that capture dominant Koopman eigenfunctions of nonlinear power system dynamics.

Koopman operators are infinite-dimensional linear operators that can describe the evolution of a non-linear dynamical system (1). The goal of this project is to utilize a neural network framework to approximate the Koopman operator matrix that can be used to accurately forecast the next state of a time series function. The application we focus on is with dynamical power systems where Koopman operators can be utilized to observe if a transient system will stabilize to equilibrium given some dynamic change in its state- for example a line failure in a power grid (1).

It is known that the classical approximation of Koopman operators occurs through dynamic mode decomposition (DMD), which is the process of identifying linear operators through temporally or spatially linked data (2). This project aims to replicate that approximation using a deep neural network by feeding in a time series input along with its next-state output with the goal of outputting a linear operator that can map between state and state . Specifically, our data is the 2-dimensional input of a single machine infinite bus (SMIB) with a 2-dimensional next-state output. The Koopman operator we approximate comes out as a 2-dimensional matrix that maps between the input and output of the SMIB system. The design considerations and implementations of the neural network as well as its evaluation will be explored in this paper.

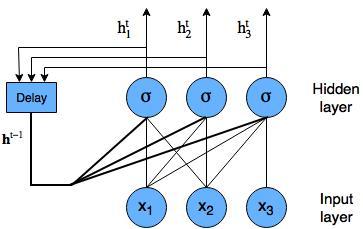
1. Design and Implementation

Before we can approach the design and implementation of the neural network system, we

we must understand the problem that we are trying to solve. This problem is defined as follows.

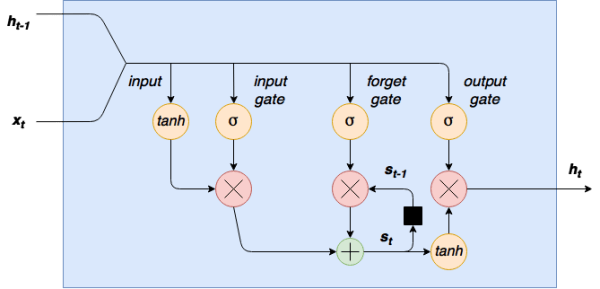
Given a function or and its output or , we define the Koopman operator , such that . Our task is to utilize known and to approximate . This approximation occurs through the minimization of the least squares problem, , which when you solve for is simplified to (1). We hope to use the neural network to approximate by training it on this simplified equation. We train it on the following input taken from SMIB data- , 2-dimensional vector of current-state values; , 2-dimensional vector of next-state values- combined to form a 4-dimensional input of both the current and next-state. The training output is a 2-dimensional vector of which is formed by taking the multiplication of and the inverse of (1).

When designing the neural network, the biggest consideration I took was the sequential nature of the data. Since is a time-series input at time and is nothing more than its value at time . I decided to treat the approximation of in a similar manner as a time-series value. To conduct the approximation, I decided to utilize a recurrent neural network (RNN), which uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about timesteps that has been seen so far (3). The nature of the recurrent neural network is shown in this image:



Recurrent Neural Network with nodes shown (4)

Normally, most recurrent neural networks run into the issue of the vanishing gradient problem where small gradients and weights are multiplied many times over through multiple time steps and shrink asymptotically to zero (4). The concept of Long Short-Term Memory (LSTM) aims to solve this. LSTM is a recurrent system with cells that have three main components, the input gate, forget gate and output gate as shown here:



LSTM Cell (4)

How the LSTM model works is it takes the output of the previous cell and the input of the current cell, activates it through a function and sigmoid function (separately) and multiplies those two outputs together. The input sigmoid removes any unnecessary features from the input and is stabilized with the input activation function. The next component is the forget gate, which takes in the input and activates it through a sigmoid function which is then multiplied with a recurrent cell that contains the added output of the previous timestep’s input gate and forget gate- the final forget gate output is added to the output of the input gate and fed into the recurrent cell for the next time step. Finally, the output gate takes the original input and runs it through another sigmoid activation function which is the multiplied with the output of the forget layer activated through the function. This output is the final output of the LSTM cell (4). What LSTM allows us to do is fix the vanishing gradient problem of the classical RNN by switching-off certain inputs by using sigmoid functions to output zero for them instead of continuously adding them to the overall gradient.

Outside of the recurrent layer, the goal was to keep the deep neural network relatively shallow with no more than three hidden layers. The hidden layers were standard neural network layers that utilized a RELU activation function. Through my research, I found that the RELU function, , was the fastest way to train non-linear systems (5). The output layer was another standard neural network layer that condensed the output into a 2-dimensional space that matched the training data. The summation of the neural network is shown here:

Layer (type) Output Shape Param #

=================================================================

lstm\_4 (LSTM) (None, 1, 32) 4736

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_16 (Dense) (None, 1, 64) 2112

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_17 (Dense) (None, 1, 128) 8320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_18 (Dense) (None, 1, 256) 33024

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_19 (Dense) (None, 1, 2) 514

=================================================================

Total params: 48,706

Trainable params: 48,706

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Neural Network Model Summary

Finally, I used mean absolute error (MAE) for the loss function with an ADAM optimizer. Initially, the plan was the use mean squared error (MSE) but the performance dipped greatly on this small of a dataset with MSE when compared to MAE. To optimize the loss function, we used the Adaptive Moment Estimation (ADAM) algorithm, which computes an adaptive learning rate for each parameter through the storage of exponentially decaying averages of past gradients and exponentially decaying averages of the momentum of past gradients (6).

The dataset of SMIB values had 100000 length vectors for both and . However, as I was working primarily on my personal computer, I did not have the computational power to train on the entire dataset, so I condensed the training on the first 10000 samples. I created the training and testing output for by simply using the equation . I utilized the standard train/test ratio of 7/3, where 70% of the samples were used to train and 30% were used to train. Due to the sequential nature of the data, I was unable to use the standard random train-test split algorithm provided by the Keras library, so I had to use the first 7000 samples of the 10K sample being utilized for training and the next 3000 samples for testing.

1. Evaluation

The Koopman dataset is a 2-dimensional matrix of n-timestep values (in our case 10000)

calculated by using the equation . The output of the neural network is a similar 2-dimensional matrix trained to fit over the Koopman dataset. The values in the dataset average at about 0.95 for the first dimension and 0.51 for the second dimension.

As mentioned in implementation, I utilized the mean absolute error function (MAE)

calculate loss for training the model. This takes the average distance of the predicted Koopman operator set to the provided training values. There is a significant reduction in distance in the early training epochs, but the MAE values start to asymptotically flatten between 1.20 and 1.30 on most runs. When compared to the average values in the dataset, this is a relatively poorly performing model that is unable to predict outlying values. The training of the model using 7000 samples of being fit on 7000 samples of is shown here:

Epoch 1/5

10/10 [==============================] - 49s 5s/step - loss: 1.9508

Epoch 2/5

10/10 [==============================] - 59s 6s/step - loss: 1.7185

Epoch 3/5

10/10 [==============================] - 67s 7s/step - loss: 1.4433

Epoch 4/5

10/10 [==============================] - 49s 5s/step - loss: 1.3246

Epoch 5/5

10/10 [==============================] - 45s 4s/step - loss: 1.2915

Koopman Model Training Summary

I have only shown for five epochs because the model begins to converge here, but I believe that the main problem with this model is the use of only the first 10000 data samples of which only 7000 are used for training. This leaves out many of the outlying values that could adjust the training performance of the neural network as well as reduces the performance of the deep neural network that works best on larger sets of data. The model was then used to output a predicted Koopman matrix of dimensions .

The next component of the Koopman operator evaluation occurs through the plotting of the eigenfunctions of the Koopman matrix to confirm the Koopman operators. The challenge here is the Koopman matrix is of dimension where n is the size of the dataset, which means it is not a square matrix and the eigenfunctions cannot be calculated through traditional eigen-decomposition. Through my research I found the formula for calculating Koopman eigenfunctions to be where is the input vector at the current state , is the Koopman mode and is the associated eigenfunction. Here is where I relied on the dynamic mode decomposition algorithm (DMD). I utilized an opensource library called PyDMD (7), which outputted the eigenvalues, modes and dynamics of the inputted matrix. The results of the function are shown here:

eigenvalues [0.99948096 0.9349783 ]

shape: (2,)

modes: [[-0.69939506 -0.71943116]

[-0.7147354 -0.6945638 ]]

shape: (2, 2)

dynamics: [[-1.19623482e+00 -1.19561393e+00 -1.19499336e+00 ... -2.52390112e-01

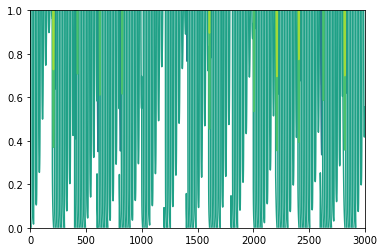
-2.52259112e-01 -2.52128180e-01]

[-1.17487989e-01 -1.09848721e-01 -1.02706171e-01 ... -3.64913639e-89

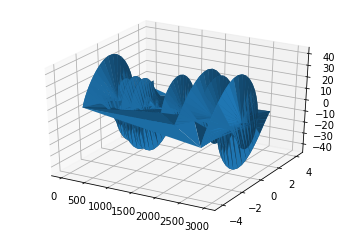
-3.41186336e-89 -3.19001822e-89]]

shape: (2, 3000)

Using the strongest outputted Koopman mode I was able to calculate the eigenfunction for the predicted Koopman matrix and plot it as shown below both on a contour plot and 3-D surface plot using the 3000 timesteps as the x-scale.

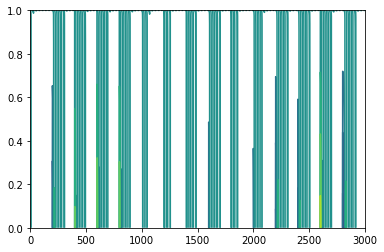


Predicted Koopman Eigenfunction Contour

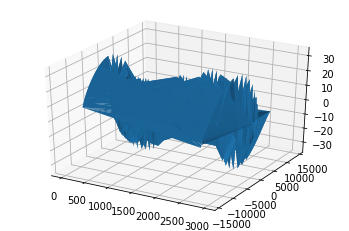


Predicted Koopman Eigenfunction Surface Plot

Using the same algorithm, I calculated the eigenfunctions for the test dataset, which outputted the following plots:



Actual Koopman Eigenfunction Contour Plot



Actual Koopman Eigenfunction Surface Plot

The difference in the two plots is most likely a result of the outlying Koopman values not being accurately captured in the condensed dataset.

1. Conclusion and Further Considerations

Overall, I considered this to be a challenging and insightful semester of research. I was able

to gain exposure to advanced topics such as non-linear dynamic systems and learn about the various methods in modeling and analyzing them. I have always been interested in time-series forecasting and learning about Koopman Operator theory has opened a lot of insight for me to create future models of a similar nature. Finally, I was able to hone my skills in TensorFlow modeling by working with different types of neural networks and prediction algorithms.

Future considerations for this project would be to utilize cloud computing to be able to run the entire dataset through the model- I believe that if the model was able to run on the entire dataset that encompassed a larger portion of outlying Koopman operators, it would’ve performed at a much better rate. Additionally, I would like to refine my method of finding eigenfunctions and visualizing them.

1. References

[1] Hyungjin Choi, Subhonmesh Bose *Transient Stability Analysis of Power Systems using Koopman Operators* (

[2] Eunoch Yeung, Soumya Kundu, Nathan O. Hodas *Learning Deep Neural Network Representations for Koopman Operators of Nonlinear Dynamical Systems*

[3] *Recurrent Neural Networks (RNN) with Keras API Documentation,* <https://www.tensorflow.org/guide/keras/rnn>

[4] *Keras LSTM tutorial – How to easily build a powerful deep learning language model*, <https://adventuresinmachinelearning.com/keras-lstm-tutorial/>

[5] Bethany Lusch, J. Nathan Kutz, Steven L. Brunton *Deep learning for universal linear embeddings of nonlinear dynamics*

[6] Anish Singh Walia *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*, <https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f>