EE695 - Koopman Assignment

Manish Kumar
200102108
ECE
manish200104057@iitg.ac.in

July 11, 2024

Abstract

This report explores the application of neural networks (NN) in the identification of Koopman operators for non-linear dynamic systems. The Koopman operator, a mathematical construct in the field of dynamical systems, provides a powerful framework for analyzing and understanding complex behaviors in nonlinear systems. Traditionally, obtaining the Koopman operator involves solving high-dimensional linear equations, which can be computationally challenging. Leveraging the capabilities of neural networks, this report investigate an alternative approach to learn the Koopman operator directly from data.

1 Introduction

The basic idea is to fisrt define a non-linear dynamic system that evolves through time. Now our main aim is to learn the koopman operator that basically acts as a linear system for this but in another space. So our first step is to project this non-linear system into some other space where it is described by Linear Dynamics. Then we learn the koopman operator in that space that will increment the system one time step in that space. Finally we use a decoder model to project it back to the original space.

Here is the explanation.

2 Methodology

This section describes the detailed procedure I followed to learn the Koopman operator.

2.1 Defining the non-linear function

I have taken a 2D function so that its easier to see/plot its trajectory evolving over time. The non-linear system I chose is:

$$\dot{x} = \frac{dx}{dt} = 2.0 * sin(x) + y - 3.5 \tag{1}$$

$$\dot{y} = \frac{dy}{dt} = x - 3.5 * cos(y) + 2.9 \tag{2}$$

where,

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \tag{3}$$

$$\frac{DX}{Dt} = f(X) = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{4}$$

$$\tag{5}$$

X is a vector with x and y coordinates of system at any time-instant. I took the dynamic system (differential eqn.) to be only function of x and y and not time instant (t) for sake of simplicity. so basically for every initial starting position there will always be a unique trajectory of the system through the time-steps.

2.2 Generating training Data

Next I generate the dataset that will be used for training the neural network. The dataset is basically the simulation of the system over 150 time steps for 1500 such random paths. By random paths I means i have randomly taken the staring point coordinates (x_0, y_0) to be $\sim U(-2, 2)$ and let the system evolve over time to get the data points.

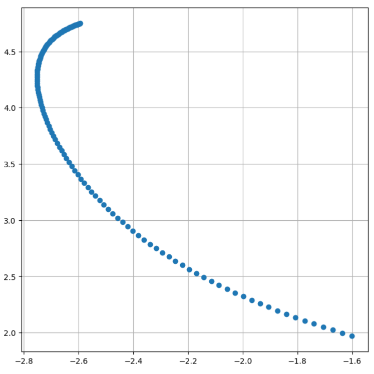


Figure 1: This shows one such trajectory evolving over time taking random initial x and y coordinates

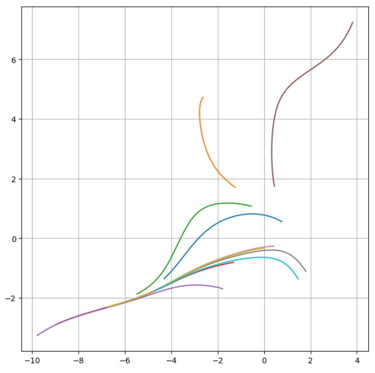


Figure 2: This shows 10 such paths

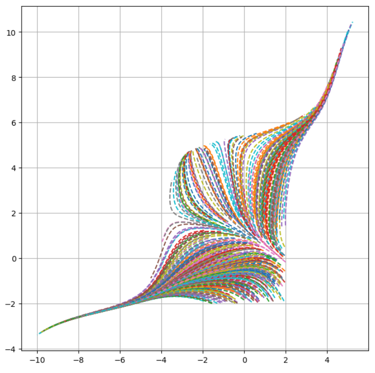
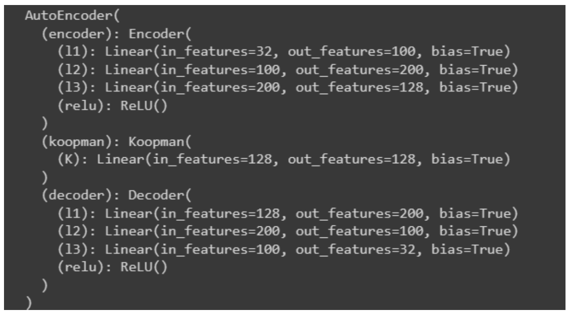


Figure 3: This shows the flow-field

2.3 Defining NN architecture

The NN architecture used here is a simple Encoder-Decoder Architecture with the following parameters.



```
AutoEncoder(
  (encoder): Encoder(
    (l1): Linear(in_features=32, out_features=100, bias=True)
    (l2): Linear(in_features=100, out_features=200, bias=True)
    (l3): Linear(in_features=200, out_features=128, bias=True)
    (relu): ReLU()
  )
  (koopman): Koopman(
    (K): Linear(in_features=128, out_features=128, bias=True)
  )
  (decoder): Decoder(
    (l1): Linear(in_features=128, out_features=200, bias=True)
    (l2): Linear(in_features=200, out_features=100, bias=True)
    (l3): Linear(in_features=100, out_features=32, bias=True)
    (relu): ReLU()
  )
)
```

Figure 4: Encoder Decoder based NN architecture

It has 3 layers with ReLU activation function applied on 1st 2 layers. The encoder takes the input features which is 32 dimensional vector. I have taken prev 16 time-steps. 32 comes from the fact that I have concatenated the x coordinates of the positions with y coordinates to get the final input feature i.e if points are $(x_t, y_t), (x_{t-1}, y_{t-1}), ..., (x_{t-15}, y_{t-15})$ then the input feature vector for time-step (t) looks like this:

$$X_f(t) = \begin{bmatrix} x_{t-15} \\ x_{t-14} \\ \vdots \\ x_{t-1} \\ x_t \\ y_{t-15} \\ y_{t-14} \\ \vdots \\ y_{t-1} \\ y_t \end{bmatrix} \tag{6}$$

Similar structure I have followed for the output vector from decoder layer. In latent representation this vector is just a 128 dimensional vector

2.4 Defining the Loss functions

I have taken 3 loss functions for training. Note that all these three losses are the standard MSE loss provided by

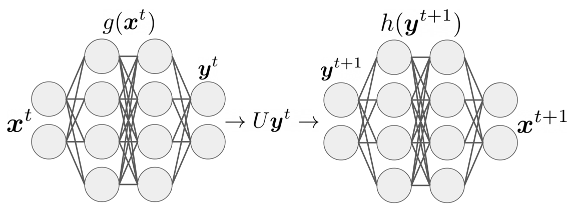


Figure 5: Encoder Decoder based NN architecture (U is the koopman operator)

py-Torch but the basic difference is between what feature vectors are we taking the loss. Those are:

1. The Reconstruction Loss (Loss 1)
2. The State prediction Loss (Loss 2)
3. The Koopman observable Loss (Loss 3)

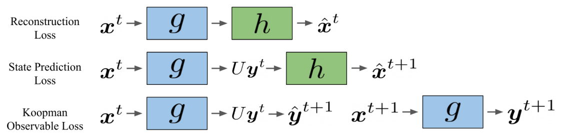

Reconstruction Loss
$\boldsymbol{x}^t \rightarrow g \rightarrow h \rightarrow \hat{\boldsymbol{x}}^t$
State Prediction Loss
$\boldsymbol{x}^t \rightarrow g \rightarrow U\boldsymbol{y}^t \rightarrow h \rightarrow \hat{\boldsymbol{x}}^{t+1}$
Koopman Observable Loss
$\boldsymbol{x}^t \rightarrow g \rightarrow U\boldsymbol{y}^t \rightarrow \hat{\boldsymbol{y}}^{t+1}$ $\boldsymbol{x}^{t+1} \rightarrow g \rightarrow \boldsymbol{y}^{t+1}$

Figure 6: Overview of 3 losses

2.4.1 Loss 1

The reconstruction loss is defined as the loss of the auto-encoder itself. It is just the loss between the input-features and output-features.

$$loss_1 = MSE(x^t, \hat{x}^t) \tag{7}$$

Here x^t is the input feature at time-step (t) as described above and $\hat{x}^t$ is the decoded version of it after passing through encoder layer (g) and decoder layer (h).

2.4.2 Loss 2

In state prediction loss we first get the latent representation on input-features and then increment it (think of it as the system moving forward time step in the latent dimension) by multiplying it with the Koopman operator. Then we decode it by passing through the decoder layer and compare the output-features with the actual data-point in our dataset.

$$loss_2 = MSE(x^{t+1}, \hat{x}^{t+1}) \tag{8}$$

Here x^{t+1} is the feature vector at time-step (t+1) and $\hat{x}^{t+1}$ is the incremented and decoded version of x^t after passing through encoder layer (g) , koopman operator and decoder layer (h).

2.4.3 Loss 3

The Koopman observable loss is basically the loss between next time steps of input features, but in the latent dimension. Similar to before here also we get the latent representaion of input-features and increment it. Then we take the encoded version of input-features which are incremented in our original space. Then we compare these two to get our 3rd loss.

$$loss_3 = MSE(y^{t+1}, \hat{y}^{t+1}) \tag{9}$$

This is basically same as previous one but here the featuress are taken from latent representaions. so y^t is basically the encoded form of x^t.

The final loss is taken as the sum of all these three and this is used in training.

$$loss = loss_1 + loss_2 + loss_3$$

2.5 Training Details

The model was trained in google colab notebook for 150 epochs. I used the ADAM optimizer with learning-rate parameter set to 0.001. The batch size taken was of 1024. Following are the hyper-parametrs used :

2.5.1 Hyperprams taken

- number of epochs: 150
- batch size: 1024
- input dimension: 32
- latent dimension: 128
- learning rate: 0.001

- number of previous samples taken: 16
- optimizer: ADAM

2.5.2 Loss curve

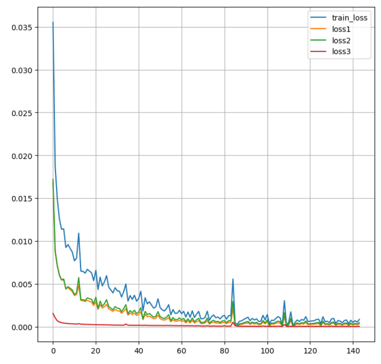


Figure 7: Loss progression over 150 epochs

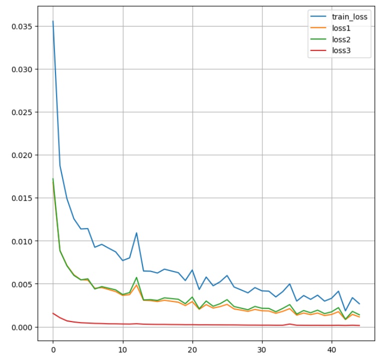


Figure 8: Zoomed-in version

3 Results

The model is abble to reconstruct the behaviour of dynamic system, but not perfectly. For the inference, I first initialized few random starting points and then pass its corresponding input-features through the model to get the latent representation. Having learnt the koopman operator, now I iteratively multiplied the input features with K matrix to increment it in latent space and use the decoder to decode it to original space. This way I generated for around 100 time-steps and the compared its trajectory with the original paths. Here are some results:

4 Conclusion

So, we saw how we can use auto encoder based NN architecture to learn the system dynamics of a non-linear system in a data-driven manner. various other methods like DMD can be used for this too but, The main draw back of traditional DMD based methods is that one needs to pre-specify a library of potential observables to learn the Koopman operator from and here use are using data.

However we can improve the system more to give us more accurate predictions if we try to enforce the skew-symmetricness property of the Koopman operator (which is just a matrix of size (latent-dim X latent-dim) and add sparsity to its weights. This concludes the report.

References

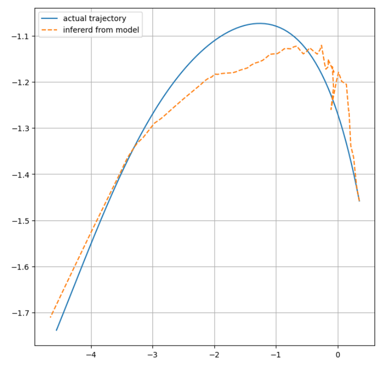


Figure 9: Inference for one such path taking random initial point. dotted one is predicted and solid is actual

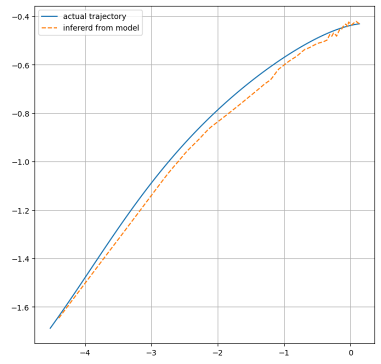


Figure 10: another such path

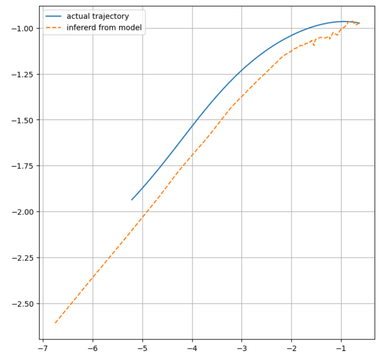


Figure 11: another path

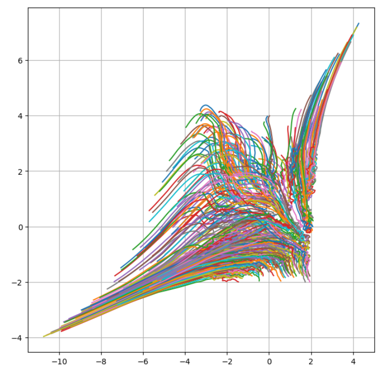


Figure 12: This shows the flow field learnt by the model