2020

DEEP LEARNING OF NONLINEAR DYNAMICAL SYSTEM

Aditya Wagh

DEEP LEARNING OF NONLINEAR DYNAMICAL SYSTEM

By

Aditya V. Wagh

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Mechanical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2020

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Mechanical Engineering.

Department of Mechanical Engineering-Engineering Mechanics

Thesis Advisor:	*Dr. Yongchao Yang*
Committee Member:	*Dr. Zequn Wang*
Committee Member:	*Dr. Pengfei Xue*
Department Chair:	*Dr. William W. Predebon*

Dedication

To my father, mother, teachers and friends

who didn't hesitate to criticize my work at every stage - without which I would neither be who I am nor would this work be what it is today. Great appreciation to all of you for the constant support during the tough times throughout the year.

Contents

List of Figures

List of Tables

Abstract

Data-driven approach, such as neural networks, is an alternative to traditional parametric-model methods for nonlinear system identification. Recently, long Short-Term Memory (LSTM) neural networks have been studied to model nonlinear dynamical systems. However, many of these contributions are made considering that the input to the system is known or measurable, which often may not be the case. This thesis presents a method based on LSTM for output-only modeling, identification, and prediction of nonlinear systems. Numerical study is performed and discussed on Duffing systems with various cubic nonlinearity.

Chapter 1

Introduction

Non-linear systems in mechanical engineering, civil engineering, and other disciplines are the systems whose behaviour is not proportional to their inputs, whereas linear systems have proportional input-output relationships. Overall to say, most real-life applications involve nonlinear systems. These type of systems do not have the properties of linear systems like additivity/superposition principle, which makes them difficult to model and analyze. Nonlinear system identification thus remains a challenging task.

Identifying a system basically is mathematical modeling of the physical system, which allows us to predict or simulate the behaviour of the system. While identifying the system is a complex process, it usually follows a few steps [1]. The first step includes choosing the inputs or excitation signals, when available, to the system. Next, the selection of the model form or architecture is a very challenging part of the process. The final step usually involves the selection of the order of the model, which requires expertise in domain knowledge. This is difficult often times and one of the goals of this thesis is to provide an alternative to this step. Furthermore, selection of the model structure and parameters may be automated but require interference from a user or prior knowledge about the system, especially in the case of nonlinear systems. Lastly, the identified model needs to be validated and tested before use. This process of validation indicates quality of the model and its limitations.

Broadly categorizing the models for nonlinear systems into White Box, Black Box and Grey Box modeling provides an initial step in system identification [2]. Based on first principles, White Box modeling is the most effective way of modeling but rather difficult as it requires accurate physics knowledge and is often time-consuming. Moreover, it is difficult to generalize the model to a variety of systems. An alternative to such pure theoretical, physics-based modeling is Black Box modeling which characterizes the model based only on experimental data. There is very little to no physics knowledge required which makes it easy to use but at the risk of lesser interpretability. This type of modeling provides path towards data-driven approach towards system identification. In this thesis, a *hybrid* methodology, physics-informed data-driven modeling through integrating Black Box models (deep neural networks) with generic or incomplete prior physics knowledge, is explored for nonlinear system modeling and identification.

Recently research have emerged in data-driven nonlinear system identification using machine learning [3] or neural networks. Neural networks is a type of data-driven approach which is subset of black box modeling. Pioneering work includes Koopman approximation of nonlinear system [4],[5] and linear embeddings of the nonlinear systems [6]. Specific applications of have also been shown using Long Short-Term Memory (LSTM) for identifying nonlinear systems examples [7] and [8] and predicting structural seismic response using Deep LSTM [9],[10]. Although these applications are successful they follow classic input-output modeling process for the systems where inputs are explicitly known or can be measured [11] and [12]. Many of the times, however, only outputs or the system's response can be measured. Output-only parametric modeling of linear systems with known input and output response has been widely studied, especially in output-only modal parameter estimation and analysis [13] [14]. Thus data-driven modeling of output-only nonlinear dynamic systems remains an important problem to study.

This thesis aims to develop a LSTM neural network based method for modeling output-only nonlinear dynamic systems. Specific deep Neural Network architecture using LSTM and learning algorithm incorporating the physics of nonlinear dynamics are developed and described in detail. A connection is made between LSTMs and traditional modeling methods of nonlinear system identification.

1.1 Motivation

A broader perspective of this study is to establish a new physics-informed data-driven modeling method for identifying nonlinear systems using output-only data and predicting their behaviour with data-driven modeling. The new approach may be scaled to high-dimensional system response data, such as those captured by using a digital camera, where the pixel of the camera represents a single sensor [15], as compared to traditional point-based sensors. A continuous video encodes a time-varying response of the structure and every pixel would be then a sensor capturing the time-varying dynamic response of a (nonlinear) structure. Using a camera provides advantages like high resolution, agility, remote working, and a wide coverage of structures/systems simultaneously along with cheaper alternative to hardware sensors.

With the systems response as time sequences, the developed LSTM neural network ”sequence to sequence” model can be used to identify the nonlinear dynamic system and then predict the future response of that structure. This is advantageous in monitoring the health of the structure remotely and continuously [16]. If the health of the physical system deteriorates, its dynamical response can be captured and compared with the prediction of the physical systems equivalent, LSTM neural network model.

Taking a small step towards the bigger goal, this thesis presents the development of an LSTM based data-driven approach for nonlinear system identification and demonstrates its capability on Duffing systems. It provides an alternative to parametric methods for identifying nonlinear dynamical systems. Physics-informed schemes are incorporated in the loss function for optimizing the training and learning process for the time-varying dynamics of nonlinear systems.

Chapter 2

Theory and Practice

2.1 Nonlinear System Identification

Any real world system is a nonlinear and varies with time. Unlike linear systems, the output and input are non-linearly related implying the high complexity of the a nonlinear system. To deal with real world applications, modeling and identification of nonlinear systems becomes crucial for design, manufacturing and testing of complex systems.

The traditional way of nonlinear system identification is by transforming it into its linear equivalent. There have also been contributions in characterizing the system using Non-linear Normal Modes, see for instance [17] [18]. Similarly, Jacob Roll et al. add on an advanced estimator using Direct Weight Optimization to estimate nonlinear system[19].

There are many models defined [2]. A black box models in particular tries to describe the system behavior with mapping the input and output relationship. This method is an alternative to traditional methods where physics of the system has to be known. They make system identification simple and quicker. The general way of representing a system by a black box model is by hypothesizing a functional relationship between input and output as show in Figure 2.1.

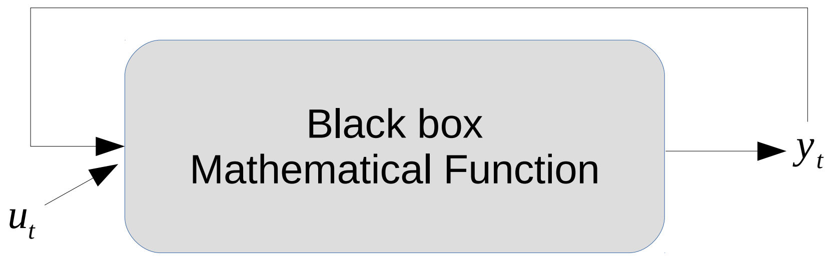


Figure 2.1: A Black Box model layout used for system identification, where u_t is the system input at time t and y_t is the corresponding system output. The system output is fed back into the input in some cases.

Black Box modeling deals with a number of parameters to estimate a function. This makes it easier, as it involves less knowledge of first principles but also difficult as these parameters are hard to estimate and interpret. An overview of such models have been covered in [20]. To solve the black box model, there are various classical methods [21] and modern methods including neural networks [22].

A special case of parametric black box modeling is NARMAX model which is characterized by the equation (2.1). In this model the system output is considered as function of past values of system input and prediction errors [23].

$$\begin{aligned} y(k) = F[y(k-1), y(k-2), ... \quad & y(k-n_y), \\ & u(k-d-1), u(k-d-2), ... \quad u(k-d-n_u), \\ & e(k-1), e(k-2), ... \quad e(k-n_e)] + e(k) \end{aligned} \tag{2.1}$$

where $y(k)$ is the system output, $u(k)$ is the system input, $e(k)$ is the prediction error, d is the time delay,n_y, n_u, n_e are the maximum lag of system output, input and error. F is a nonlinear function. However, it is not always possible to know the system input and the only information one has is of the system outputs. So modeling has to be done using only system output. Thus the new equation looks like equation (2.2)

$$\begin{aligned} y(k) = F[y(k-1), y(k-2), ... \quad & y(k-n_y), \\ & e(k-1), e(k-2), ... \quad e(k-n_e)] + e(k) \end{aligned} \tag{2.2}$$

This type of model is now an output only model and can be called as NARMA. Having experimental data, we can now learn the function F. We can fit the past system outputs with current responses and then fitted model will represent the function F. This data driven approach is very time efficient. Since we are fitting the model, a regression tool is the best fit for such problem, especially Multi Layer Perceptron (MLP). According to Universal Approximation Theory, MLP architecture can approximate any nonlinear function as long as the function is continuous.

2.2 Neural Networks

In traditional programming, a model is created which is given Data and certain defined rules and it produces answers. For example, Fibonacci sequence, where next number is sum of past two numbers. The computer model follows the rule of adding past two values to produce next value. However, in recent advances, the computer can be taught to learn the rules, given data and answers, refer Figure 2.2. Such computer models can approximately learn the rule of Fibonacci sequence provided enough data-answer pairs. There is quite similarity between the Figure 2.1 and Figure 2.3. Thus Neural Networks can represent the nonlinear function F in the equation (2.2).

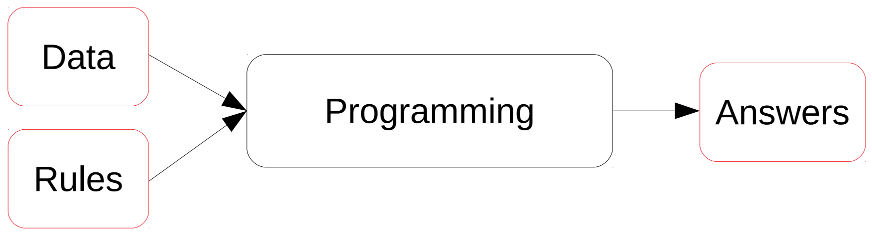


Figure 2.2: Flow of Traditional Programming. Rule based modeling.

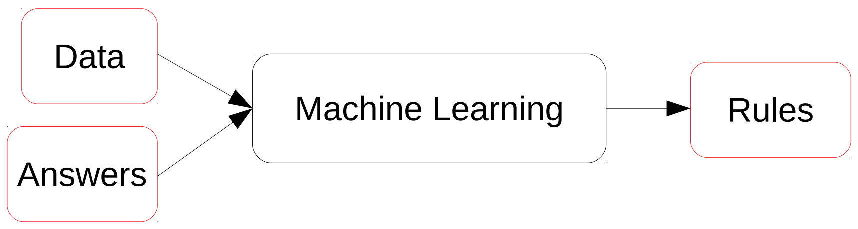


Figure 2.3: Basic methodology of Machine Learning.

According to Universal Approximation Theorem [24], neural networks can be used to learn the nonlinearity from the system response. several advances and optimizations have been proposed in using neural networks and gradient methods to learn/identify nonlinear system [25],[26], [27],[28],[29]. Neural Networks can be used in any field because of the flexibility to add domain knowledge while training. With optimized training and loss function, the physics of the system can be learnt, thus making a novel neural network model, see for instance [30] and [31]

Although MLP can be use to approximate nonlinear functions, when the data is in the form of sequence, Recurrent Neural Networks have proven to be more efficient and reliable [32]. RNNs has several types viz. the simple RNN, Gated-Recurrent Unit(GRU) and Long short-term Memory(LSTM) with increasing complexity. The simple RNN suffers from the problem of vanishing gradients when dealing with long term sequences. So the advanced GRU and LSTM are used especially for tasks involving speech signals, or music which are essentially a form of vibration [33].

Our attempt has shown that LSTM, a fundamental and complex form of basic RNN can be used to approximate the nonlinear function similar to F of the NARMA model.

2.3 Duffing System

Duffing system is a second order differential equation characterizing certain damped and driven oscillations. We use this system to test our ideology. Being the fundamental of all nonlinear systems, if LSTM can represent output only Duffing system, the method can be scaled to any other nonlinear dynamical system.

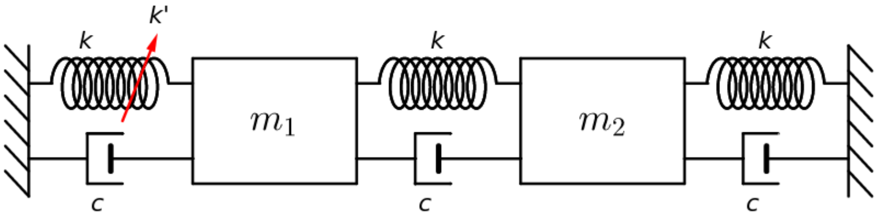


Figure 2.4: Duffing system with 2 DOF.

Figure 2.4 shows the spring-mass-damper diagram of the Duffing system. The same can be mathematically explained with the equations (2.3) for 2 DOF, where x_1 and x_2 are displacements of respective masses, $\dot{x}$ and $\ddot{x}$ are the first and second order derivatives of the displacements. The force component is zero as we are dealing with free vibrations. The x^3 term is the cubic non-linearity and is weighted 0.5 and 1 for testing our ideology.

$$\ddot{x}_1 + (0.02\dot{x}_1 - 0.01\dot{x}_2) + (2x_1 - x_2) + 0.5x_1^3 = 0 \tag{2.3}$$

$$\ddot{x}_2 + (0.02\dot{x}_2 - 0.01\dot{x}_1) + (2x_2 - x_1) = 0 \tag{2.4}$$

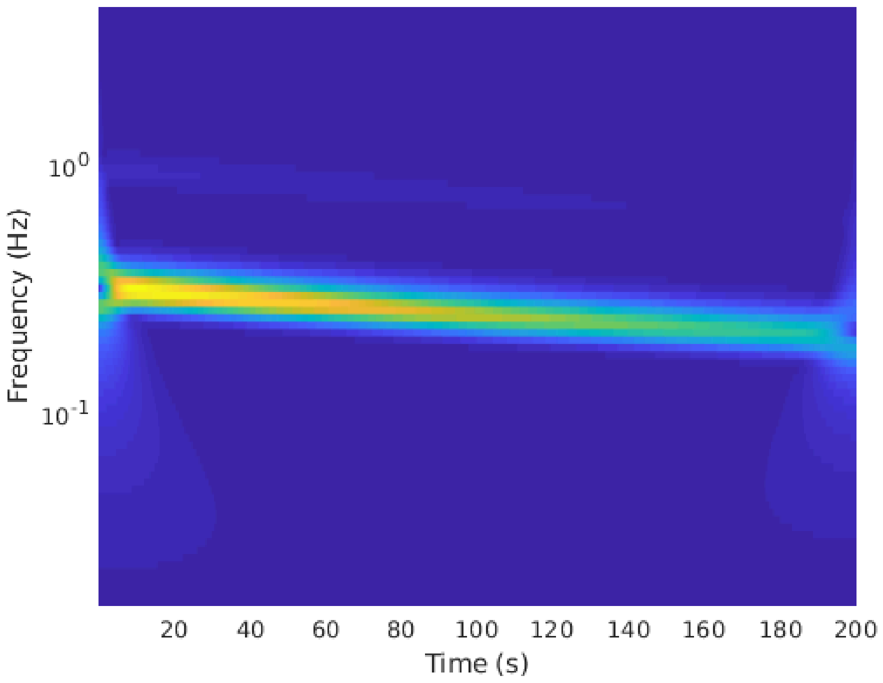


Figure 2.5: Change in Frequency with Time in structural response of Duffing System.

Time variant nonlinear dynamics: The response from the Duffing system can be analysed with the Figure 2.5. The Frequency of the response is decreasing with time. Thus the non-linearity of the system is observed maximum in the initial time after excitation. The nonlinear behaviour decreases exponentially eventually becoming linear. Thus training the neural network in the initial steps is more challenging and we have trained the LSTM on the systems initial response. If LSTM model can sufficiently represent higher nonlinear behaviour then it's expected to perform good on the behaviour in the later time with weaker nonlinearity as well.

Chapter 3

Neural Networks

3.1 Fundamentals of Neural Networks

Neural Networks are combinations of neurons, where each neuron is a regression equation like $y = mx + c$. There are stacks of neurons which are called layers. Every layer can have any number of neurons and thus this number is a hyper-parameter. Each neuron in a layer is connected to every other neuron in its adjacent layer and thus form a network. This can be visualized with the Figure 3.1. The layers apart from input and output are called hidden layers. The number of hidden also being a hyper-parameter, it is known to govern the ability of neural network to learn complex features within an input. The regression equation with each neuron is characterized as,

$$a^l_{n_l} = g(\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l-1}}[w^l_{ij}X_i + b_{ij}]) \tag{3.1}$$

$$\hat{y} = g(\sum_{j=1}^{n_l} w^{l+1}_{ij} a^l_j + b_j) \tag{3.2}$$

, where the w's are equivalent to the m(slope) in the regression equation of line and are known as weights. Thus every neuron is a weighted sum of input. These weights are the characteristics of every neural network and are known to represent the learnt neural networks. When a neural network learns, it means that these weights are optimised. The b's are called biases, equivalent to c of the equation of line. The are embedded in the weights matrix while training the neural network.

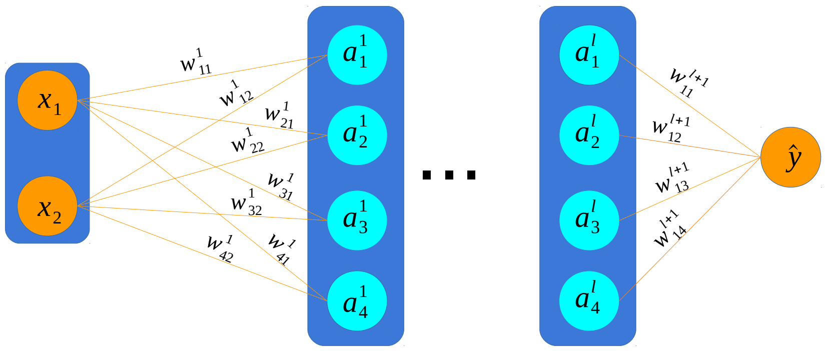


Figure 3.1: Connections in a Deep Neural Network

The function g is called the activation function. As the name suggests, this function controls whether to activate the neuron , meaning a non-zero value or to deactivate the neuron, meaning assign zero value to the neuron. When a neuron is activated, the weighted sum of its inputs are carried forward for further computation. Usually the activation functions used are *tanh*, *sigmoid*, or *relu*.

$$\begin{aligned} \tanh(x) &= \frac{e^{2x}-1}{e^{2x}+1} \\ \sigma(x) &= \frac{1}{e^{x}+1} \\ relu(x) &= \max(0,x) \end{aligned} \tag{3.3}$$

The predicted output $\hat{y}$ is essentially a complicated function of input X. This function can be shown represent any nonlinear function. The combinations of such regression functions or neurons can be used to learn features of certain type of data. The simple Deep Neural Network is not effective in every situation and thus we explore different types or architectures of neural networks [34].

3.2 Types of Neural Networks

3.2.1 Deep Neural Network

As represented in the Figure 3.1, this is the most fundamental architecture of neural networks. Every advanced form of neural network is built on this architecture. The

'DEEP' is a term referring to multiple hidden layers between input and output layers. With the use in traditional data fitting, flexibility of this architecture has made it useful in nonlinear system representation.

3.2.2 Convolutional Neural Networks

These networks are advanced form of Deep Neural Networks where the emphasis is on learning features in an image. A special operation of convolution is performed between a designed filter and the image to learn the features. Convolutional Neural Networks along with Deep Neural Networks are also effective in nonlinear system identification [35].

3.2.3 Recurrent Neural Networks

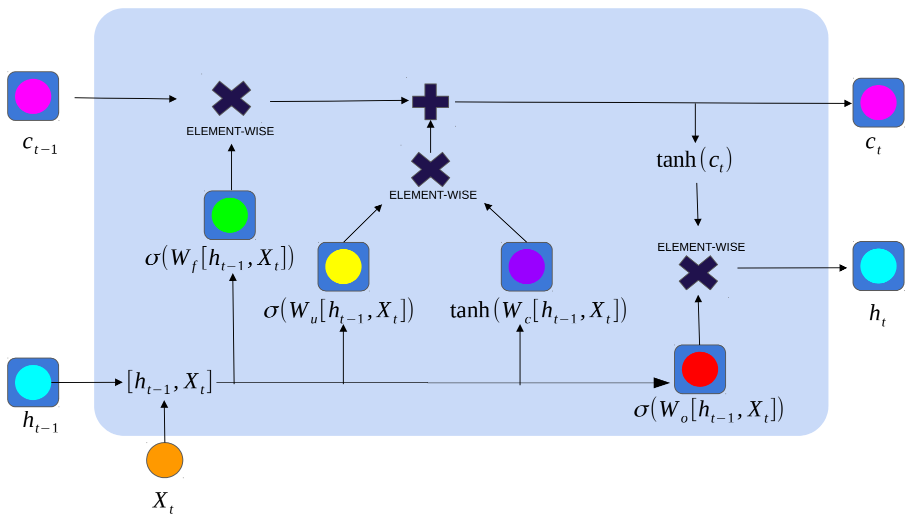
A special form of neural network architecture which emphasises the learning of features from a time sequence data. Recurrent Neural Networks(RNN), as the name stands for, computes regression on every time step of the data. This helps its learning through each time step. This architecture is shown in Figure . As the length of time sequence increases, RNN are found to be unreliable. The updating of weights in any neural networks aims to minimize a loss function. More about this is explained the Section 3.5.

For larger time sequences, the loss function doesn't optimize significantly and this leads to the problem of vanishing gradient. The gradient meaning the change in loss function with respect to the weights. Thus the weights do not update to better values and the learning of this neural network reaches a stagnant point. To address this problem Long Short-Term Memory neural networks are used, which have additional computations to resolve the issue of vanishing gradients.

3.3 Long Short-Term Memory

Long Short-Term Memory(LSTM) [36] was rather invented before RNN however, their use is profound in the recent years. The special elements in LSTM are the memory states which allows retaining information from a given time step till the end of the time sequence. This speciality of LSTM makes it much more efficient and reliable. The individual cell can be visualised with Figure 3.2.

The Figure 3.3 shows that the LSTM is built on basic neural network architecture. The LSTM gates viz. Forget gate, Update gate are responsible to pass on or forget the vital information carried in the time series. The output gates influences the hidden states that carry the information of dynamical system, with the vital information



c_{t-1}
ELEMENT-WISE
ELEMENT-WISE
$\tanh(c_t)$
c_t
ELEMENT-WISE
$\sigma(W_f[h_{t-1}, X_t])$
$\sigma(W_u[h_{t-1}, X_t])$
$\tanh(W_c[h_{t-1}, X_t])$
h_t
h_{t-1}
$[h_{t-1}, X_t]$
$\sigma(W_o[h_{t-1}, X_t])$
X_t

Figure 3.2: Individual cell of LSTM depicting number of operations, cell memory and cell activation states.

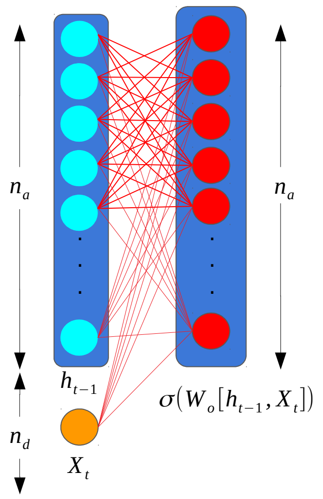


Figure 3.3: Expansion of one of the operations in the cell showing the computations at root level.

which is being carried throughout the series. Mathematically, the equations (3.4) describe the operation in the Figure 3.2,

$$
\begin{aligned}
\tilde{c}_t &= \tanh(W_c[h_{t-1}, X_t] + b_c) \\
\Gamma_f &= \sigma(W_f[h_{t-1}, X_t] + b_f) \\
\Gamma_u &= \sigma(W_u[h_{t-1}, X_t] + b_u) \\
\Gamma_o &= \sigma(W_o[h_{t-1}, X_t] + b_o) \\
c_t &= \Gamma_u * \tilde{c}_t + \Gamma_f * c_{t-1} \\
h_t &= \Gamma_o * \tanh(c_t)
\end{aligned}
\tag{3.4}
$$

Looking at our problem of representing nonlinear dynamical system with only output known, it is a very good choice to use LSTM for two reasons viz. a) We can learn dynamics with every time step of sequence essentially, without knowing the order of the model. b) The equation (3.4) shows that the output is indirectly a nonlinear function of the input.

Thus by fitting the data in an LSTM model, we can represent a nonlinear system without knowing its actual order.

3.3.1 Encoder Decoder Architecture

Since we only have output of the system, we have to use a specific architecture so that LSTM can learn the dynamics. The idea is to use the given output response, break into smaller sub sequences and teach the LSTM model to predict itself in future by providing present time timesteps. This allows us to not only learn the dynamics but also augment our data into larger dataset which helps in training the model.

To apply such self predicting methodology the best architecture is sequence to sequence model or also called as encoder-decoder model.

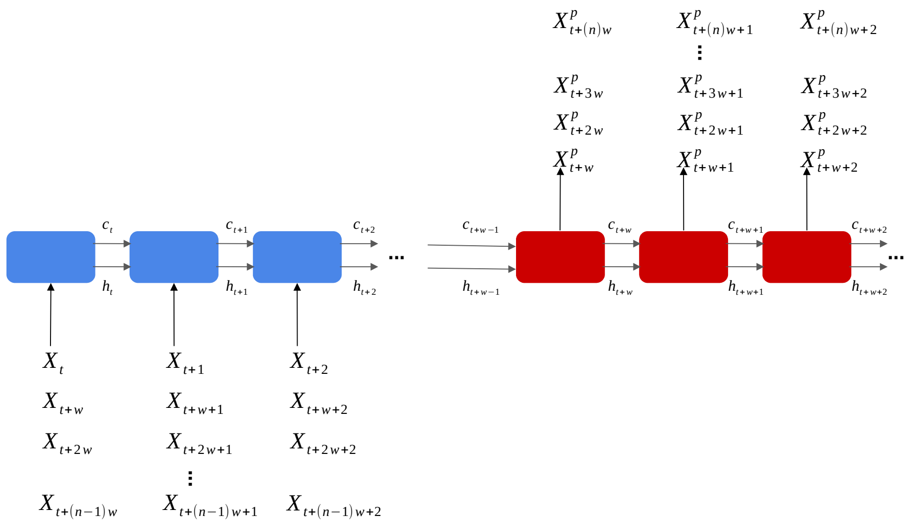


Figure 3.4: LSTM model architecture for Training.

This architecture uses the encoder model to learn the sub-sequence we created and then predict the next sub-sequence forward in time. The encoder is an LSTM cell as shown in Figure 3.2, which computes the regression function for every time step. Thus the LSTM cell repeats itself as many number of times as the number of time step in the sub sequence. After the last time step of the sub-sequence, the cell memory and activation states are carried forward to the decoder. The decoder is another LSTM cell which takes in the information of cell memory and activation states from last time step of the previous sub-sequence, computes regression for every time step. As the cell computes the regression for every time step, the information for every time step is learnt uniquely. As shown in Figure 3.3 the output of LSTM cell is a layer of neurons. This layer is converged into single neuron representing single time step. For that we use the Dense layers which are nothing but few more layers of connected neurons but the number of neurons decreases in numbers eventually reaching one.

3.4 Loss Function and Optimizer

3.4.1 Loss Function

A loss function calculates error of predicted data with the original data. This loss function is the key to train a neural network for the specific purpose and application. Loss function for classifications are different from loss functions for regression as the expected output is binary or real numbers. The error calculated by the loss function is to be minimized. So the loss function needs to be continuous and differentiable. Since we are predicting time sequence, the values are going to be real numbers, so its best to use Mean Squared Error(MSE) loss function. This would teach the LSTM to predict as close to the given values as possible.

Modification to the loss functions are necessary when dealing with special cases. Most researchers modify the loss function to best learn the features of the given data. This makes the neural network model more robust and reliable. We are using the Duffing System to test our results. The responses have a changing frequency with time as shown by the Figure 2.5. Moreover we break sequences into smaller sub-sequences which creates a larger non uniformity in the scale of each sub-sequence. Some sub-sequences with larger amplitude outweigh the sub-sequences with smaller amplitude. Since the neurons calculate the weighted average of the inputs during regression, the sub-sequences with smaller amplitude contribute very less towards updating the weights of neural net.

$$\mathcal{L} = \alpha_1(\mathcal{L}_{pred}) + \alpha_2(\mathcal{L}_{normpred}) \tag{3.5}$$

$$\mathcal{L}_{pred} = \|X_t - X_t^p\|_{MSE} \tag{3.6}$$

$$\mathcal{L}_{normpred} = \|(X_{norm})_t - (X_{norm}^p)_t\|_{MSE} \tag{3.7}$$

$$X_{norm} = \frac{X - \max(X)}{\max(X) - \min(X)} \tag{3.8}$$

So we add a term in addition to the MSE loss function which makes sure the every sub-sequence contributes equally to the loss function. We normalize every predicted and original sub-sequence to a constant scale of $[0, 1]$, and then find the MSE again. This term is added to the traditional MSE term with a factor controlling it's influence on the loss function. This factor is a hyperparameter and is decided after analyzing the performance of the LSTM model. The new loss function looks like equation (3.5), where $\alpha_1 and \alpha_2$ are hyperparameters.

3.4.2 Optimizer

Optimizing is process where the weights are updated to minimize the loss function. Imagine the a loss function with a graph shown in the Figure 3.5. Since the loss function computes the error of prediction, the error needs to be minimum. Thus taking the derivative of loss function with respect to parameter w updates the w to reduce the loss. So the parameters w have to reach the minimum of the loss function. This is achieved by iteratively updating the weights as shown in equation (3.9).

$$w_{t+1} = w_t - \eta \frac{d\mathcal{L}}{dw} \tag{3.9}$$

This equation is also known as equation of gradient descent. The learning rate (η) is the hyperparameter which governs the rate of convergence of loss to the minimum.

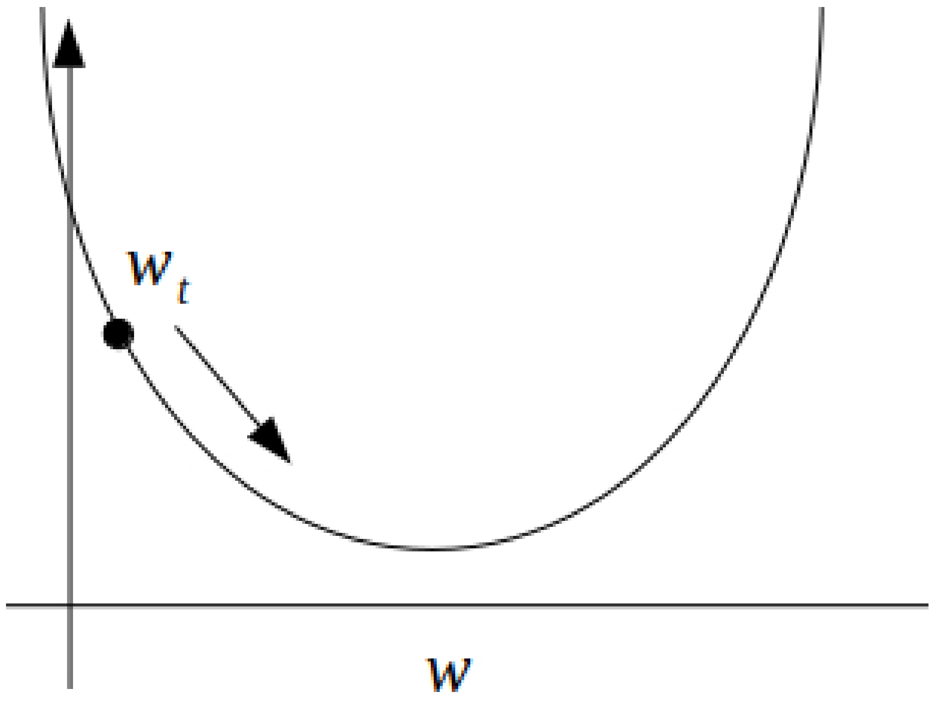


Figure 3.5: Picturing gradient descent on a loss function

With every update of weights, the value of the weight closer to the minimum by a factor of the learning rate. So, if this parameter is too large, the model will overshoot the minimum, whereas keeping it too low, will lead to longer training time. However, the loss function is not always this simple as shown in the Figure 3.5. As the function gets complex, there is not always one minima. But, optimizing the weights to reach the local minima has shown to suffice most of the times. Moreover, more advanced optimizer have been developed to reach the minima. The most efficient is Adam optimizer [37]. The name is derived from 'adaptive moment estimation'.

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\frac{d\mathcal{L}}{dw} \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)(\frac{d\mathcal{L}}{dw})^2 \\
w_{t+1} &= w_t - \eta \times \frac{m_{t+1}}{\sqrt{v_{t+1}+\epsilon}}
\end{aligned}
\tag{3.10}
$$

,where β_1 and β_2 are hyper-parameters. This optimizer is a combination of 'gradient descent with momentum' and 'RMSprop' optimizer and has proven to produce the best results. Judging from the complexity of our data, we use Adam optimizer with varying parameters mentioned in the results section.

3.5 Training and Testing

We use Tensorflow framework [38] and Keras [39] in Python environment to train and test the model. Training is the crucial step in modeling the neural network. This is the process when weights update iteratively fitting the data. The steps of taking input, calculating regression functions, calculating the loss, and updating the weights for the whole set of data is known as a epoch. With every epoch the weights are optimized layer by layer in reverse. That means the weights of the neurons that computes the final output are updated first and the weights of neurons that compute regression of the first inputs are updated last. This is known as Back Propagation.

In the case of LSTM, the weights of the Dense Layer of the Decoder with last time step are updated first and then the ones with the previous time step and so on till the weights of the neurons for the first time step in the encoder layer. Since we are back propagating through time, it is known as back propagation through time[BPTT]. This is an important characteristic of RNN. It is in this BPTT, RNN experiences vanishing gradient problem. And since LSTM carries a separate cell memory state, the information through forward time is retained for a longer sequence of time separately and gradients don't vanish.

Training for our architecture needs setting up of data. We have generated sequences of Duffing system response. They are each broken down in sub-sequences of length determined by the parameter window. After trading off with training time, and test performance and considering length of at least one period of time sequence, we have kept the length of window as 50. For time length of 50s and sampling frequency of 10 Hz, each sequence is of 500 time steps. We have 1000 such sequences. Out of the 1000 sequences, 950 are used for training and 50 are used for validation. Breaking every sequence into sub-sequences creates 10 such sub-sequences for every sequence. Now, since the idea is to self predict, the first sub-sequence predicts the next sub-sequence, the set of the sub sequences is arranged as shown in the Figure 3.6. Meaning that the first sequence is the input and the second sub-sequence is the output, and then second sub-sequence is the input and the third is the output and so on.

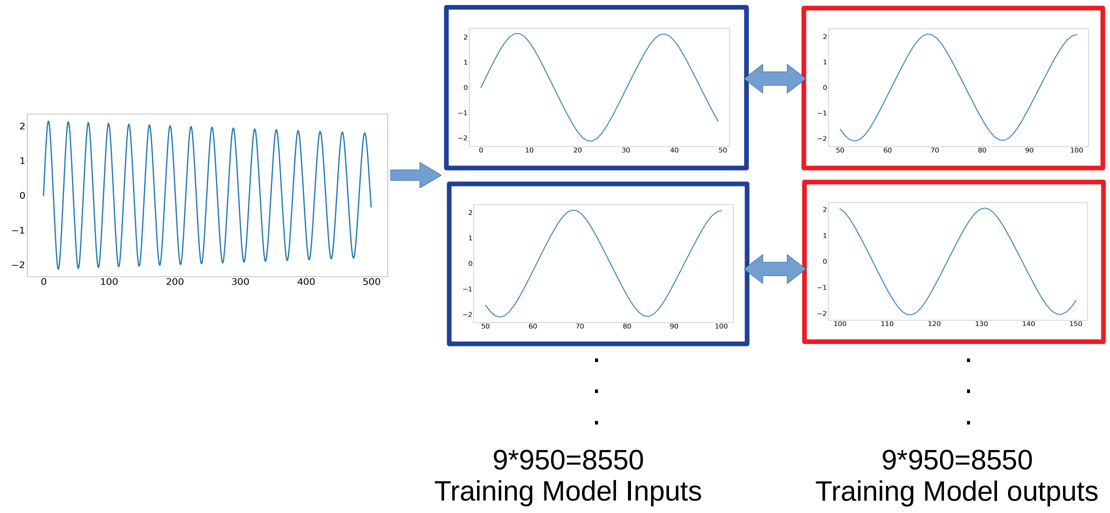


Figure 3.6: Restacking sequences, creating training dataset for LSTM model

So, for every sequence, we have 9 sets of training input and 9 set of training outputs. Extending to every sequence, we have a total of 8550 sub-sequences which are now treated as examples by the neural network. These are fed to the LSTM model shown in Figure 3.7. The LSTM model is a representation of the Figure 3.4.

$$y(t+(n-1)w), y(t+(n-1)w+1) \;\dots y(t+nw-1) \rightarrow \boxed{\begin{array}{c}\text{LSTM}\\\text{MODEL}\end{array}} \rightarrow y(t+nw), y(t+nw+1) \;\dots y(t+(n+1)w-1)$$

Figure 3.7: Mathematically representing input and output sub-sequences to LSTM model

Same is done for validation sequences as well, creating 450 sub-sequences. The model is then trained for 10000 epochs or until it starts to overfit. The phenomenon of overfitting is observed when the loss on the validation dataset starts increasing than loss on training dataset. If no overfitting is experienced the the model can be stopped

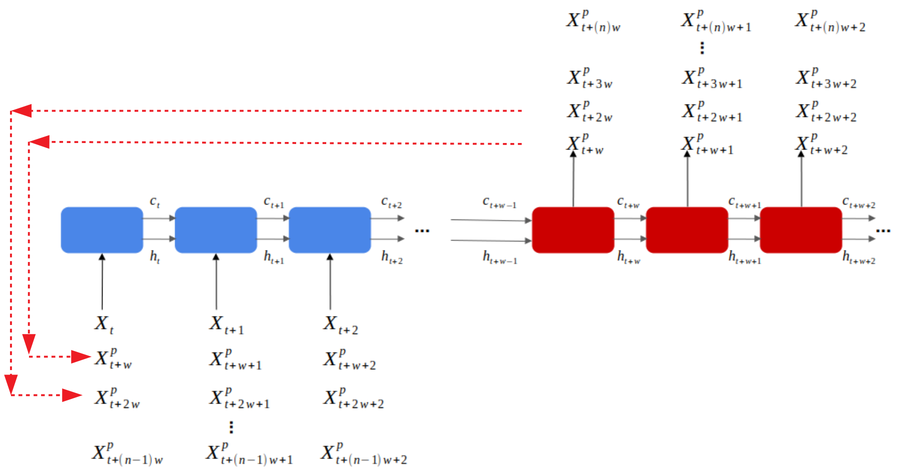


Figure 3.8: Testing architecture for LSTM model.

training when the loss doesn't seem to converge much with every epoch.

Testing of the model is done in a slightly different way. The model has learnt to predict a time sequence of length of a window, which is 50. To create a longer sequence, the predicted sub-sequence has to be fed back to encoder which will then predict the next sub-sequence and so on. This can be visualized with the Figure 3.8.

The sub-sequences predicted are then concatenated to form the longer desired sequence. This whole sequence is then compared with the original sequence to test the quality prediction results.

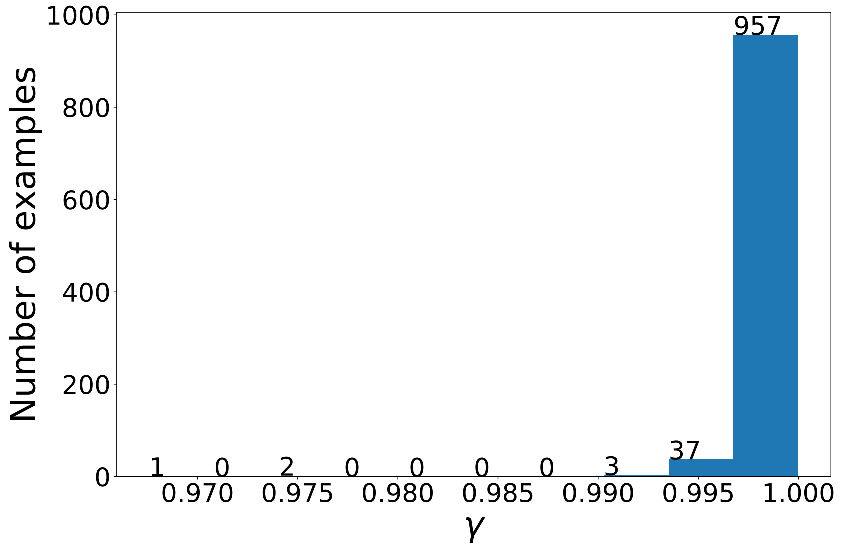
Chapter 4

Numerical Study on Duffing Systems

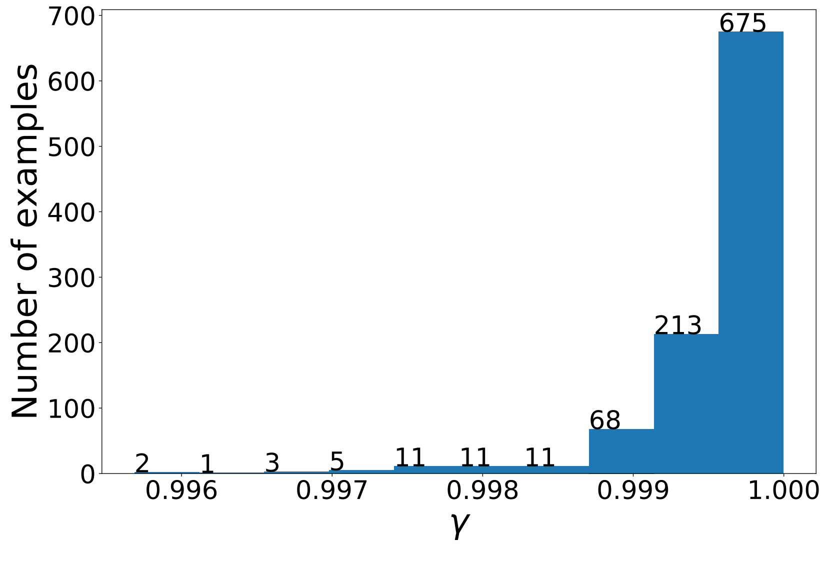
4.1 1 DOF

4.1.1 Non-linearity=0.5

For testing the hypothesis, we start with 1 DOF and 0.5 cubic non-linearity. The following results show the effect of adding the optimizing normalization term in the loss function. The concatenated sequences are used to check the quality of prediction. It can be seen in the Figure 4.2 that even though the correlation γ is high enough, the actual prediction is not stable for sequence with lower amplitude. However, adding the optimizing term solves that issue.



(a) with MSE



(b) with MSE and normalization

Figure 4.1: Histogram of performance of Non-linearity=0.5 examples.

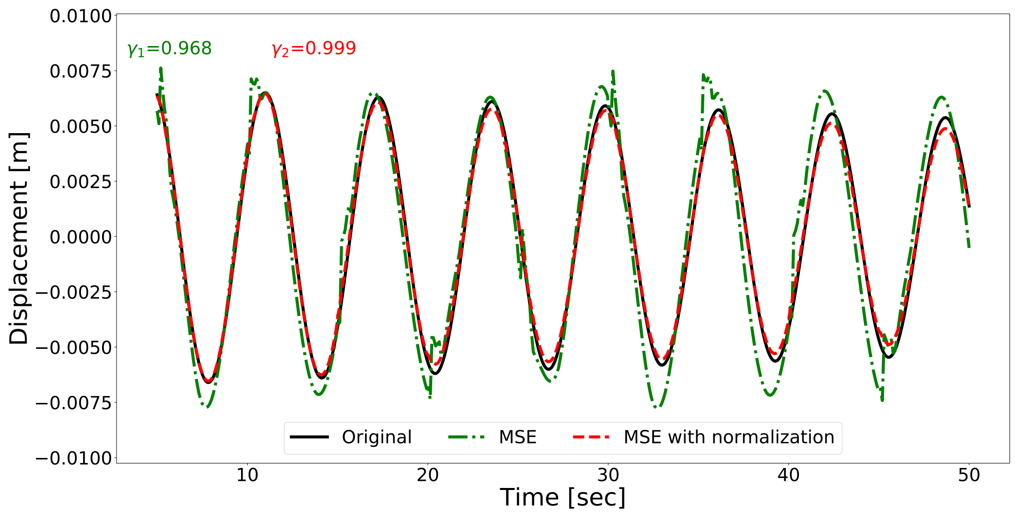
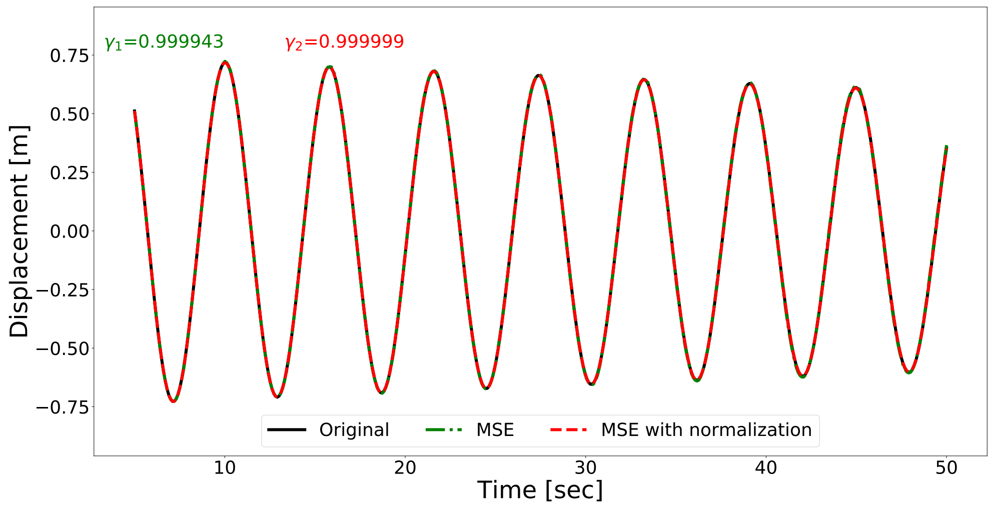


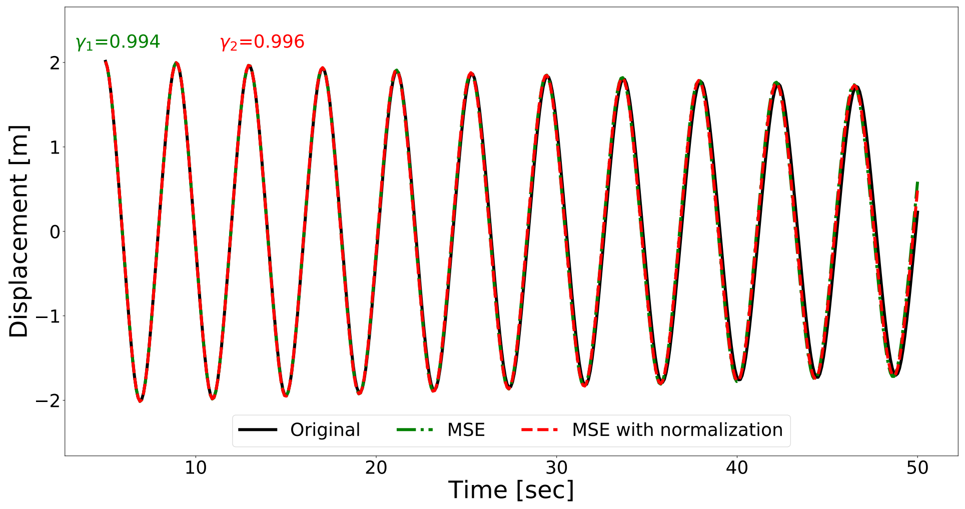
Figure 4.2: Effect of optimized loss function on Non-linearity=0.5 examples. γ is the correlation of predicted vs original sequence.

4.1.2 Non-linearity=1

For non-linearity of 1, similar steps of training and testing are repeated using higher nonlinear data. The results align with the discussions from results of predictions of examples with non-linearity of 0.5. Thus the LSTM model can also be used to represent higher nonlinearity with excellent results. The Figures 4.4, 4.5 and 4.6 show the perform of LSTM model on high nonlinear Duffing system.



(a) Best performance on prediction



(b) Worst performance on prediction

Figure 4.3: Non-linearity=0.5. γ is the correlation between original and predicted sequence.

4.2 2 DOF

Testing for 2 DOF, we gave responses of both the DOF for 5 seconds as input to the LSTM model and then the model predicting the next 5 seconds, which were fed back as input to the LSTM network and so on till we predicted 50 seconds. The results can be seen in the Figures 4.7 for first degree of freedom and in Figures 4.8 for second degree of freedom. The hyperparameters in the loss function have to adjusted which are mentioned in the code in Appendix A.1.

The Figures 4.9 depict the best predictions on test data set after concatenation of all the sub-sequences while the Figures 4.10 show the effect of the loss function on the worst performed sequence in the dataset.

4.3 Effect of window size

The hyperparameter window size has a physical significance when it comes to dynamical systems. It is the upper limit of the order of the system. While one needs to determine the order of the system before using the parametric methods of system identification, LSTM sequence to sequence model allows one to define the upper limit of the order. The actual order of the system is learnt during the training of the model implicitly. To choose a value for this parameter, one must consider the dataset available and sampling frequency used. The window size must not be lower than the order of the system. Although choosing a higher size results in better prediction, it is computationally inefficient to train a sequence to sequence model for longer sequences.

4.4 Effect of Loss Function

The response of the duffing system, i.e the displacement vs time graph, decays to zero exponentially. Since the system is nonlinear, the frequency of vibration also decays to a constant value, that is when the nonlinear system transforms into a linear system. This changing amplitude and frequency is to be learnt by the model in order to learn the dynamics of the system. Dividing the response sequence into smaller sub-sequences helps learning the changing frequency but it is still incomplete. The changing amplitude must also be considered while learning the dynamics. In short, the loss function must be such that the local dynamics of the sequence is learnt. The breaking of sequences into smaller sequences, creates an uneven scale of data. To weigh in every sub-sequence equally, we add another parameter to the loss function which normalizes every sub-sequence. This helps learning the local dynamics which is necessary for predicting long term sequences.

4.5 Effect of weights in loss function

The loss function consists of two terms, one that measures closeness with actual and predicted data while the other measures the closeness between normalized actual and normalized predicted data. Although, both the terms are important, they must be weighted, considering the uncommonness in the data. The data is a sub-sequence, which can be of varied amplitude. The model prediction would largely get skewed towards normalized data if the weights are equal. To keep the originality of the data while also learning the local dynamics, we try various values for α_2 in the loss function as shown in the table 4.1.

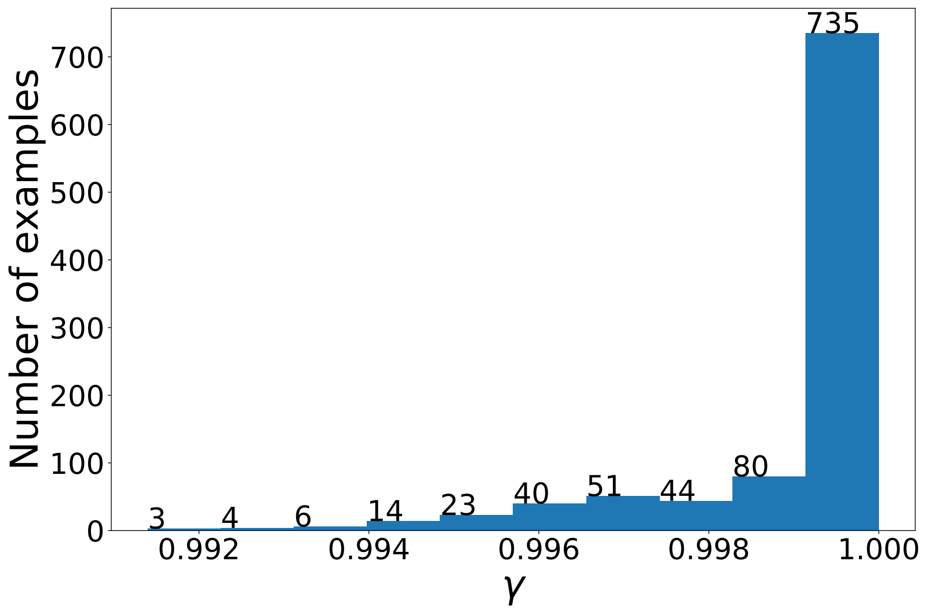
2 DOF (first DOF)	α_2=0	$\alpha_2 = 1e^{-4}$	$\alpha_2 = 1e^{-5}$
Non-linearity=0.5	0.74-0.99	0.97-0.99	0.94-0.99
Non-linearity=1	0.986-0.999	0.996-0.999	0.995-0.999

Table 4.1
Effect of hyperparameter on range of correlation values obtained on testing dataset.

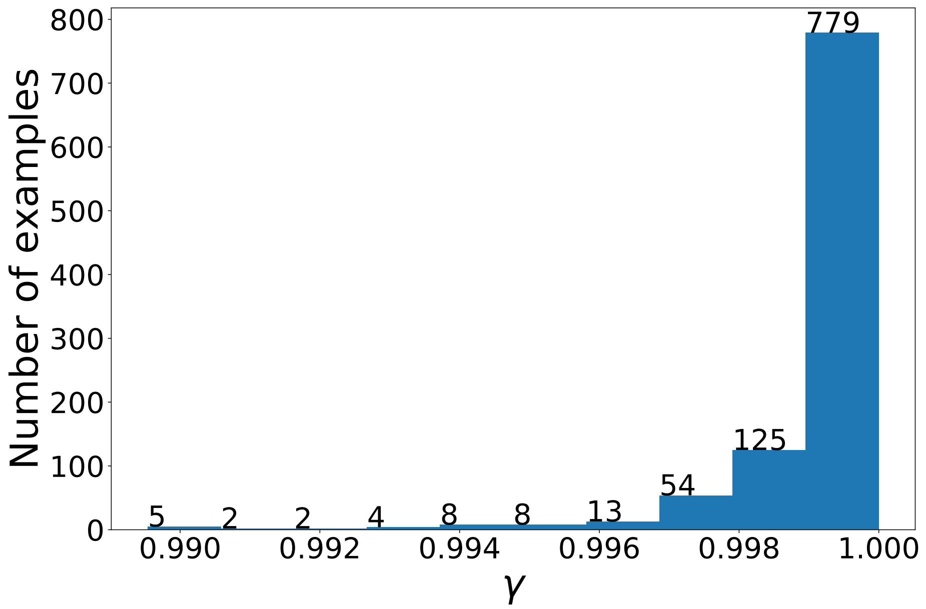
4.6 Testing for longer sequences

One of the effect of using the proposed loss function is on long term predictions as the model learns the local dynamics from the training sequence. The model is trained on dataset containing dynamical responses of 50s and tested on long term prediction upto

150s. Although the model is trained on dynamic response of 50s, while predicting, only the initial 5s of response is known. Although, the model is assumed to have learnt the dynamics to predict upto 50s, given proper training, predicting further into future tests the real reliability of the model. The Figures 4.15, 4.16 and 4.17 show the results of 1 and 2 DOF duffing systems with the new loss function.



(a) with MSE



(b) with MSE and normalization

Figure 4.4: Histogram of performance of Non-linearity=1 examples.

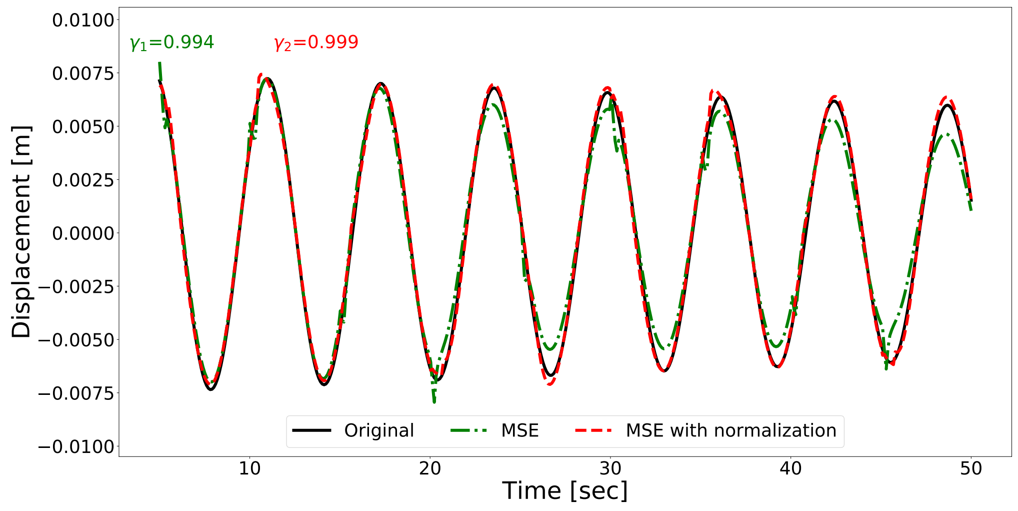
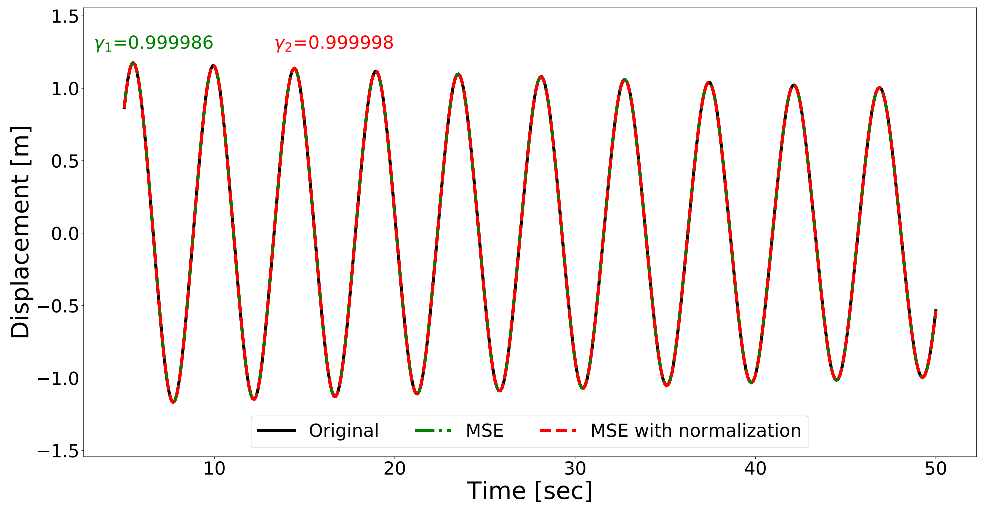
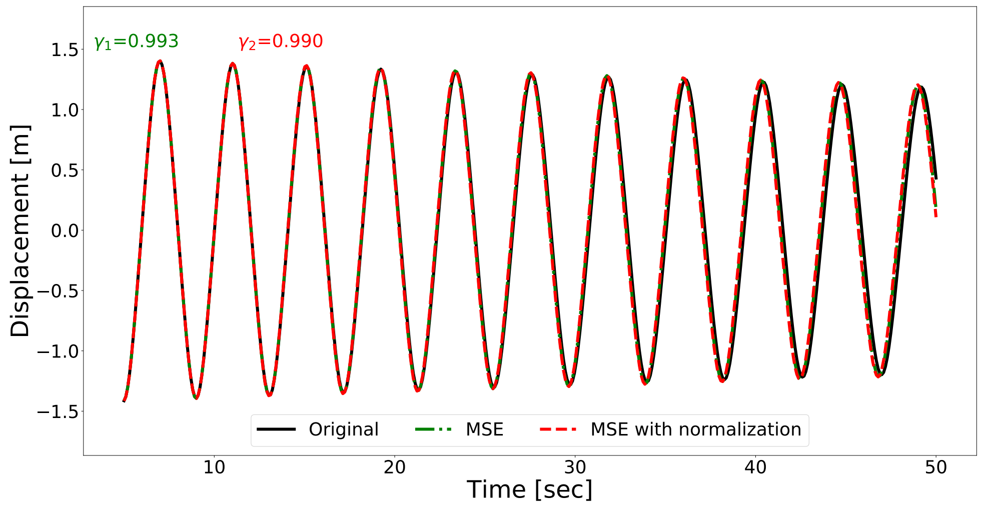


Figure 4.5: Effect of optimized loss function on Non-linearity=1 examples. γ is the correlation of predicted vs original sequence.

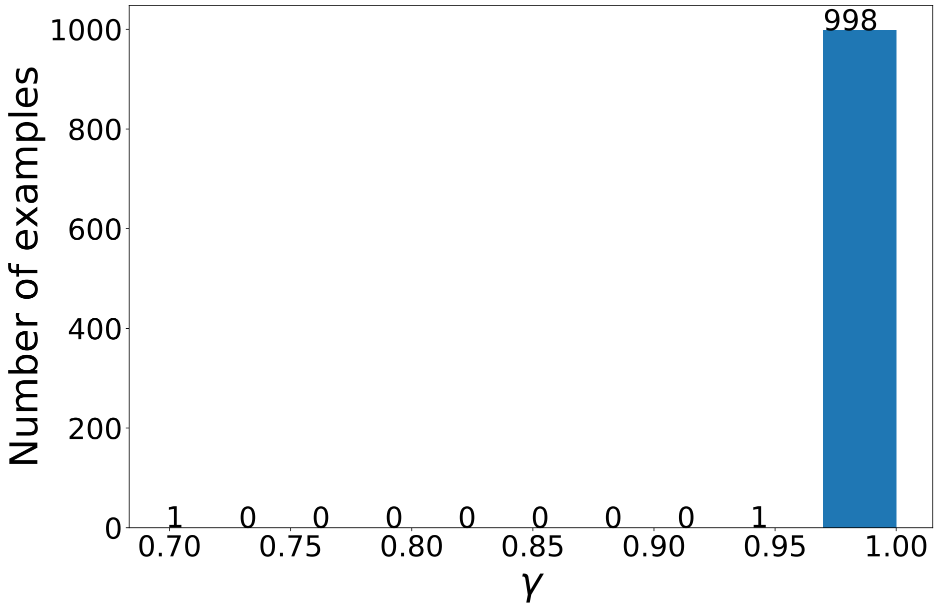


(a) Best performance on prediction

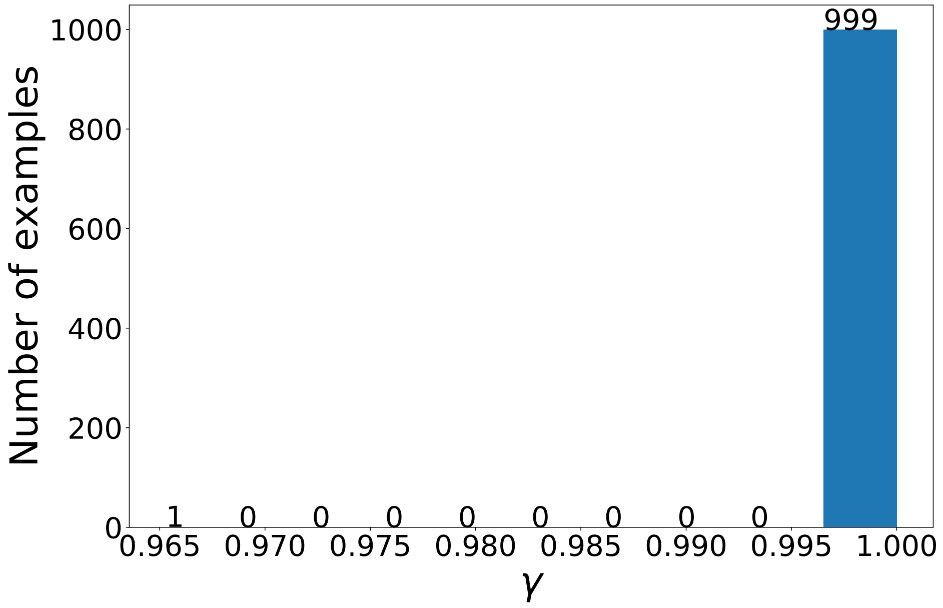


(b) Worst performance on prediction

Figure 4.6: Non-linearity=1. γ is the correlation between original and predicted sequence.

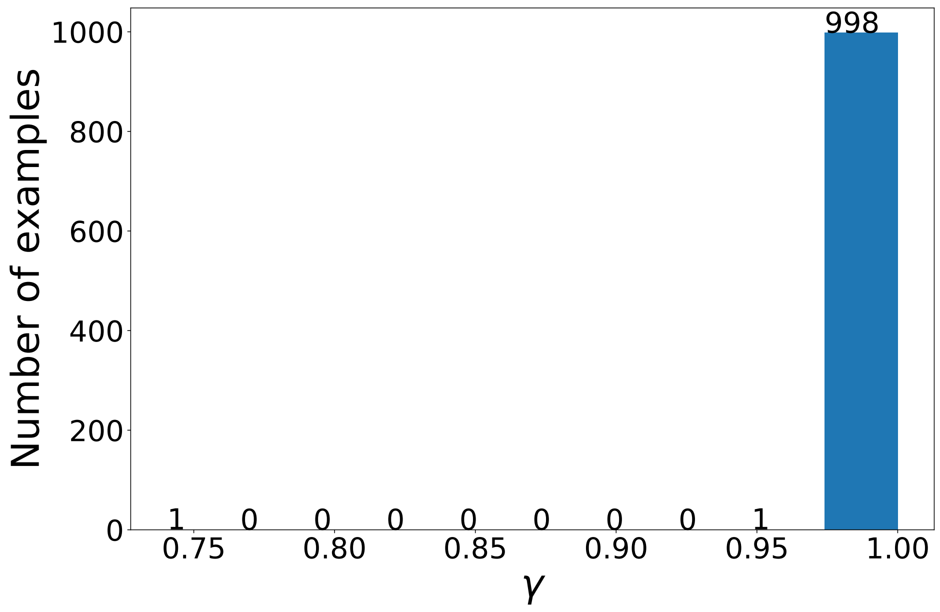


(a) with MSE

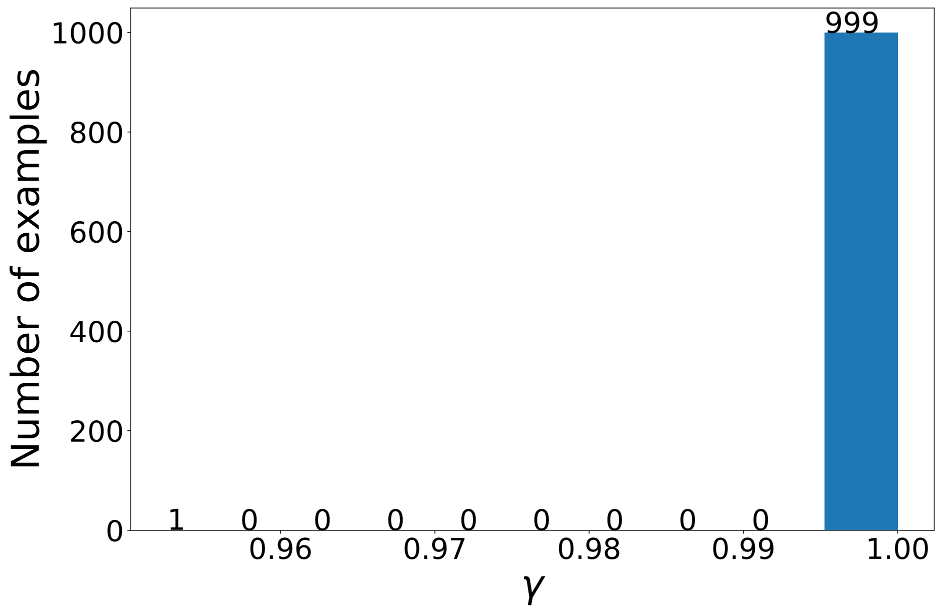


(b) with MSE and normalization

Figure 4.7: Histogram of performance of Non-linearity=0.5 on first degree of freedom.

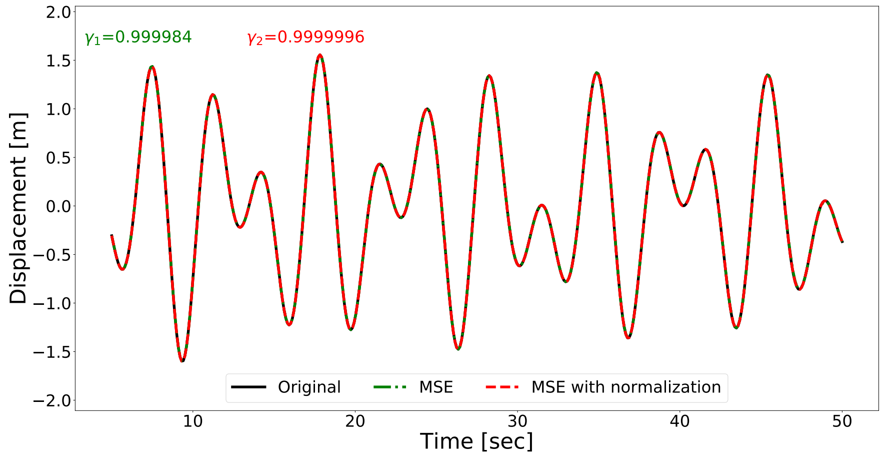


(a) with MSE

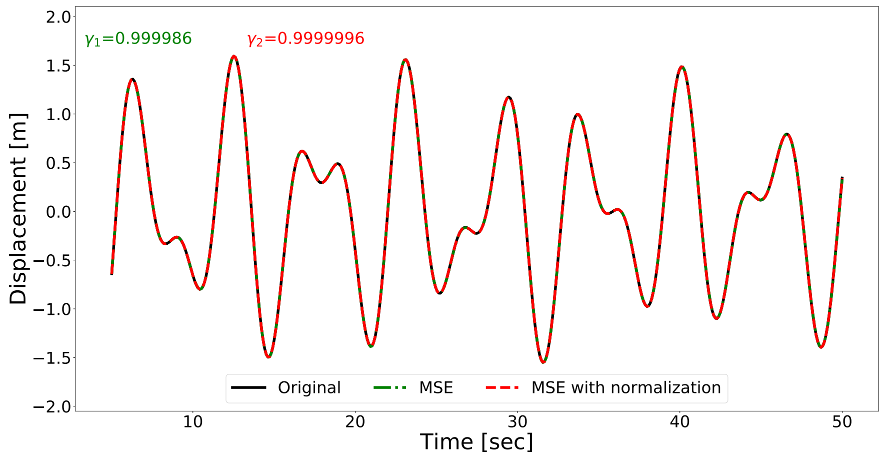


(b) with MSE and normalization

Figure 4.8: Histogram of performance of Non-linearity=0.5 on second degree of freedom.

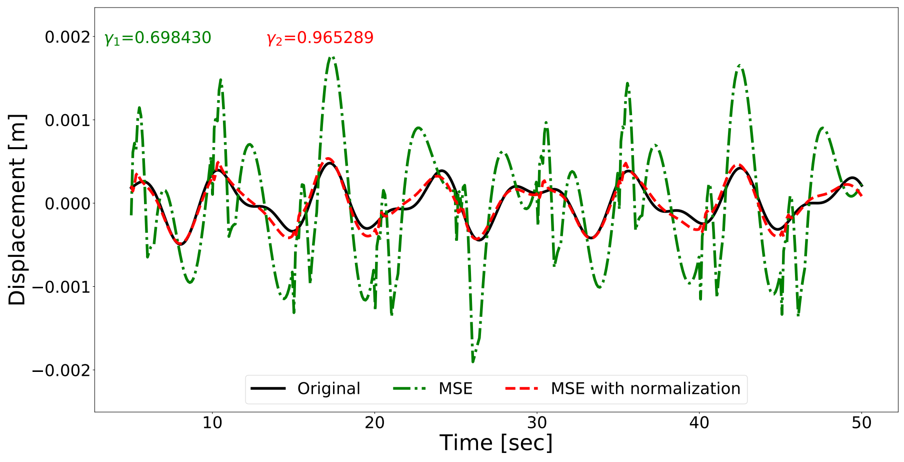


(a) first degree of freedom

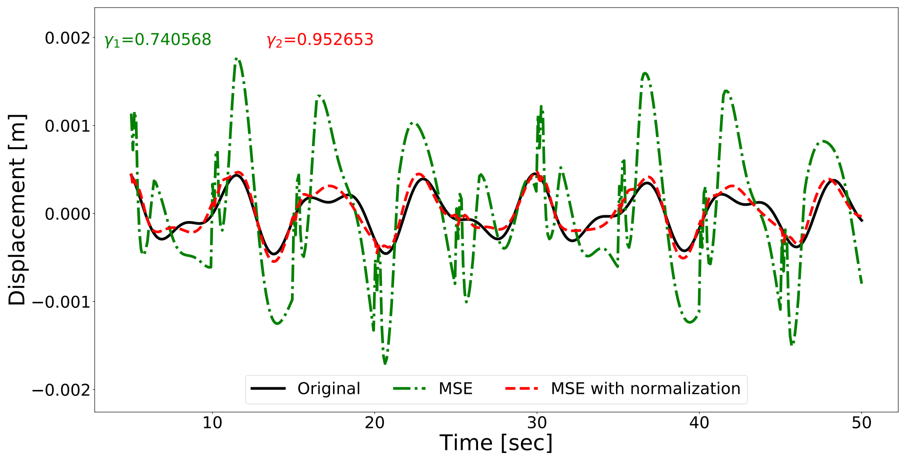


(b) second degree of freedom

Figure 4.9: Best performance of Non-linearity=0.5

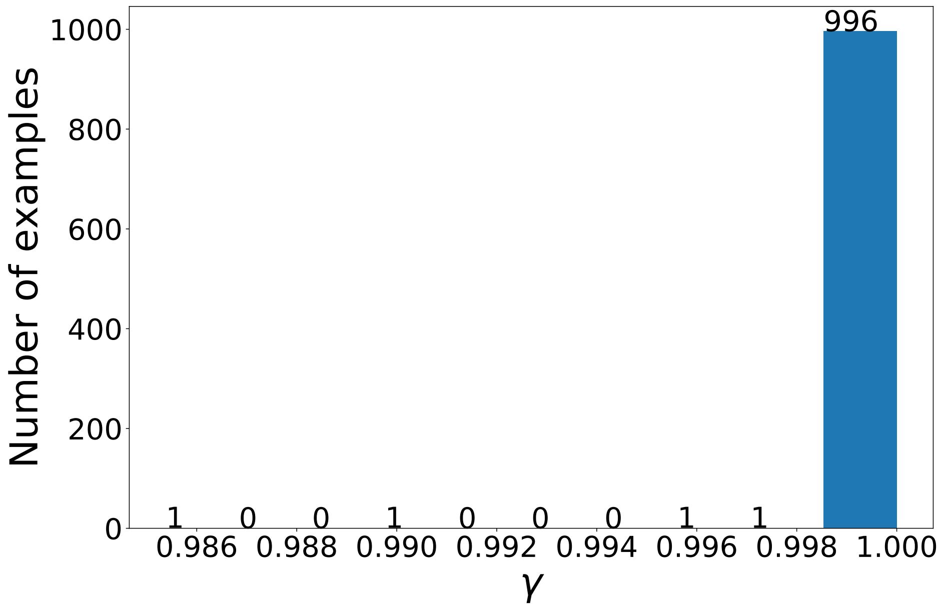


(a) first degree of freedom

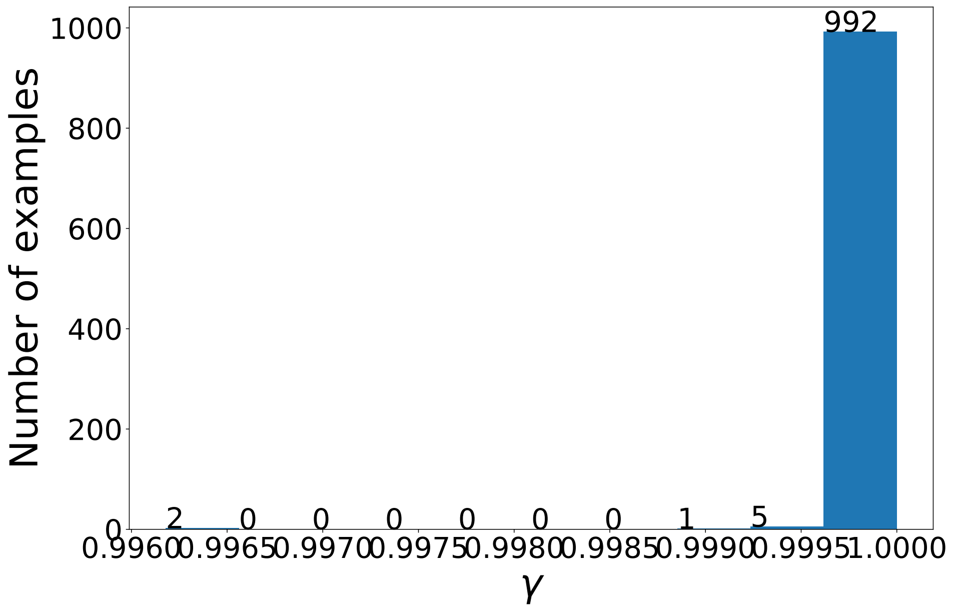


(b) second degree of freedom

Figure 4.10: Worst performance and comparison of loss function of Nonlinearity=0.5

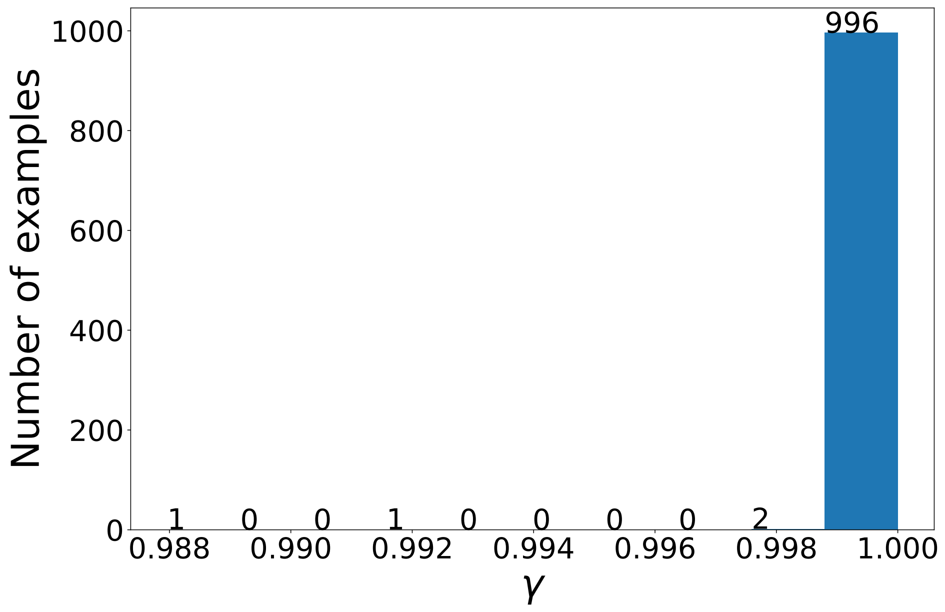


(a) with MSE

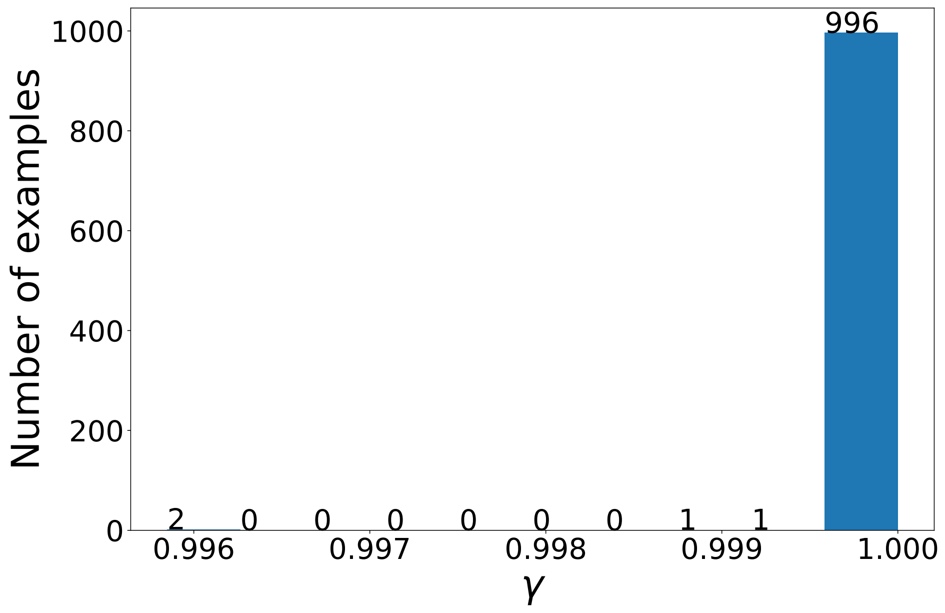


(b) with MSE and normalization

Figure 4.11: Histogram of performance of Non-linearity=1 on first degree of freedom.

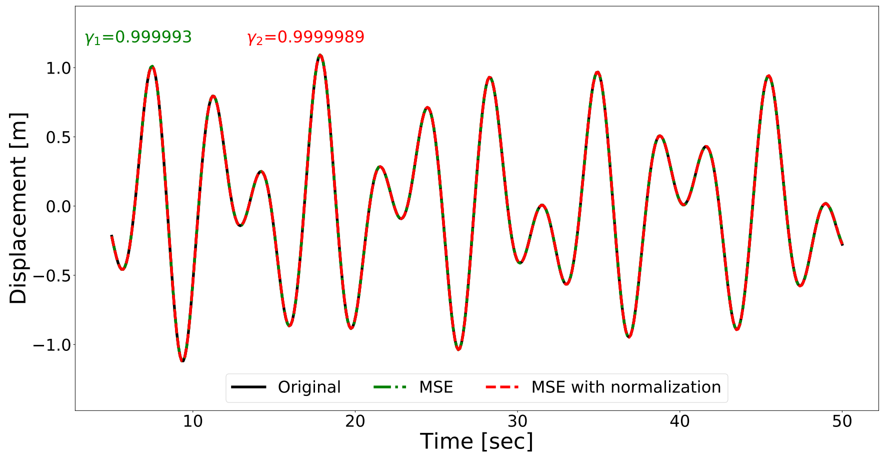


(a) with MSE

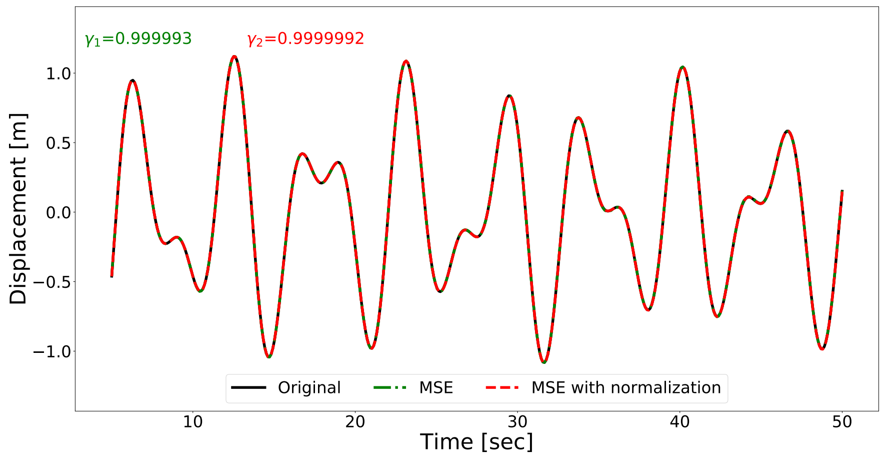


(b) with MSE and normalization

Figure 4.12: Histogram of performance of Non-linearity=1 on second degree of freedom.

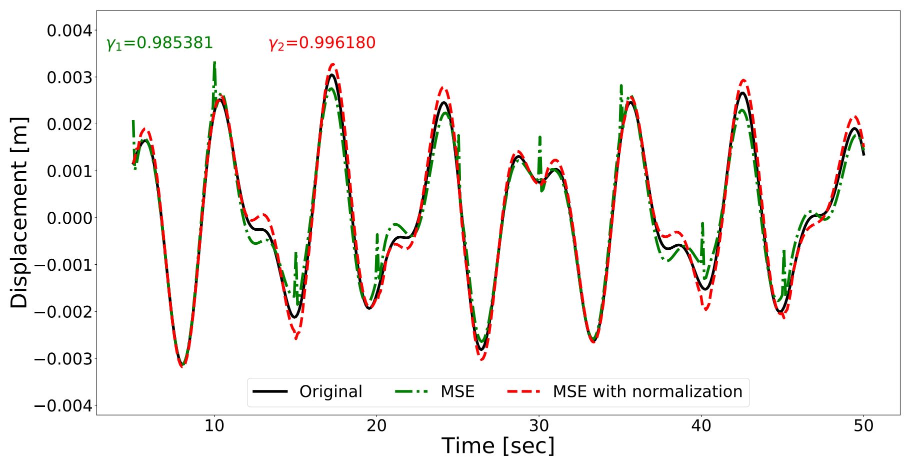


(a) first degree of freedom

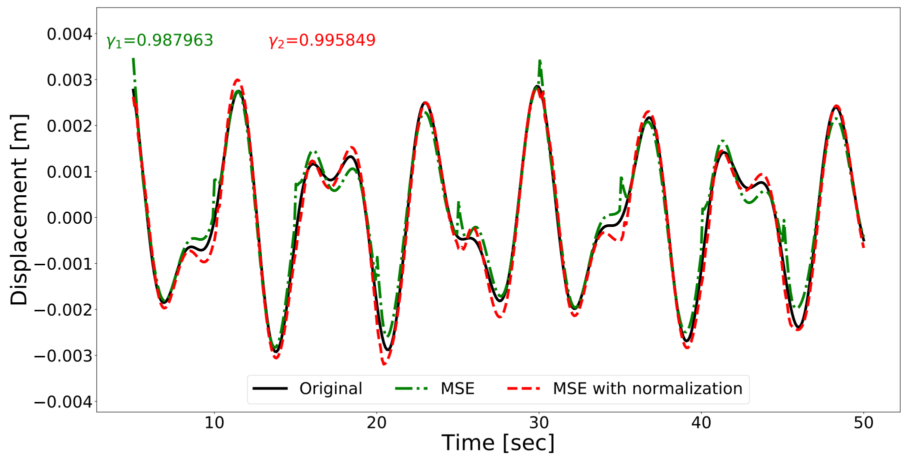


(b) second degree of freedom

Figure 4.13: Best performance of Non-linearity=0.5

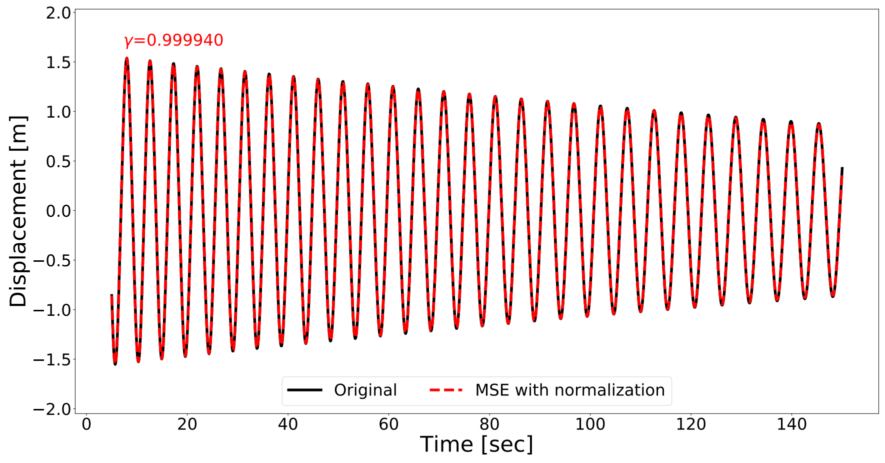


(a) first degree of freedom

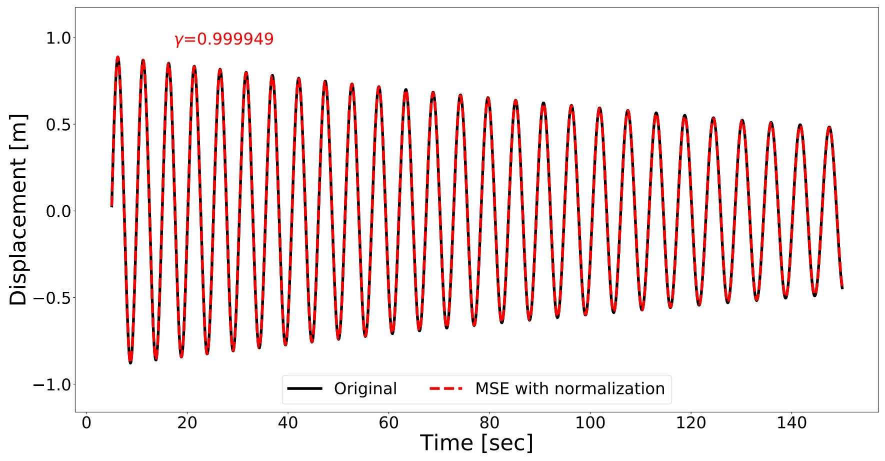


(b) second degree of freedom

Figure 4.14: Worst performance and comparison of loss function of Non-linearity=1

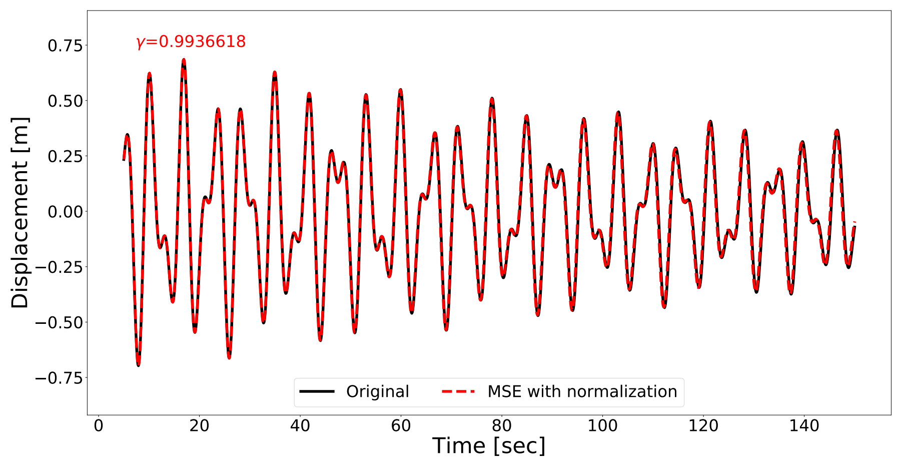


(a) non-linearity of 0.5.

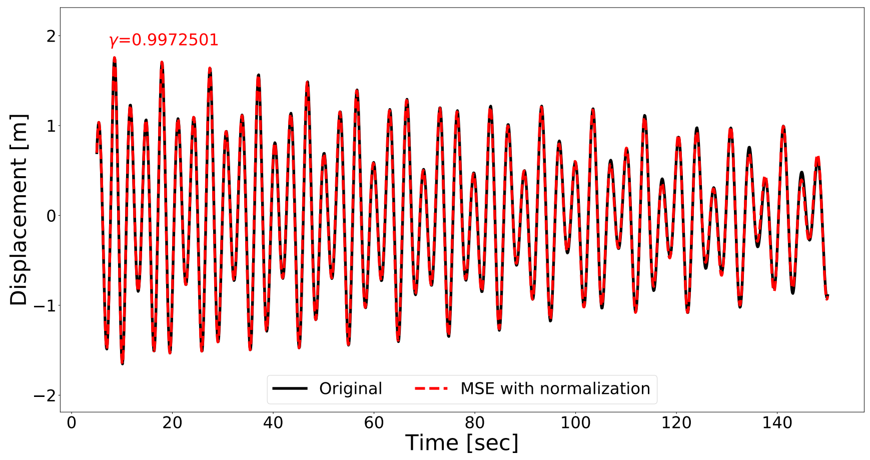


(b) non-linearity of 1.

Figure 4.15: 1 DOF predicted response for extended time (150s).

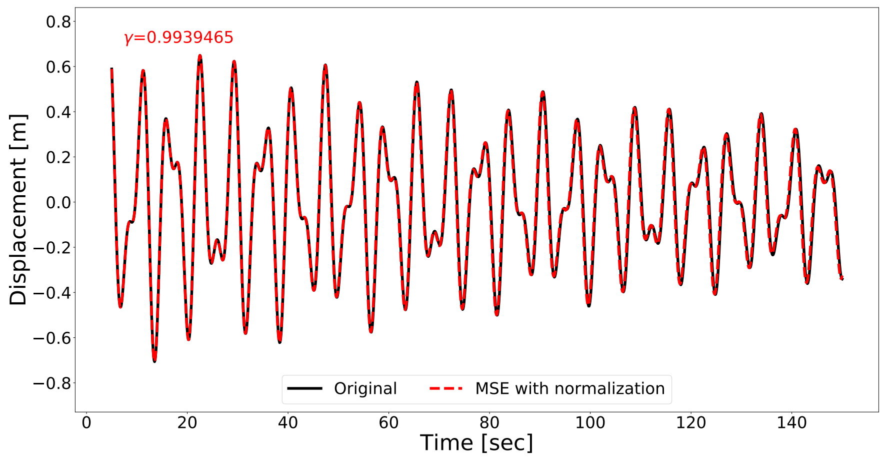


(a) non-linearity of 0.5.

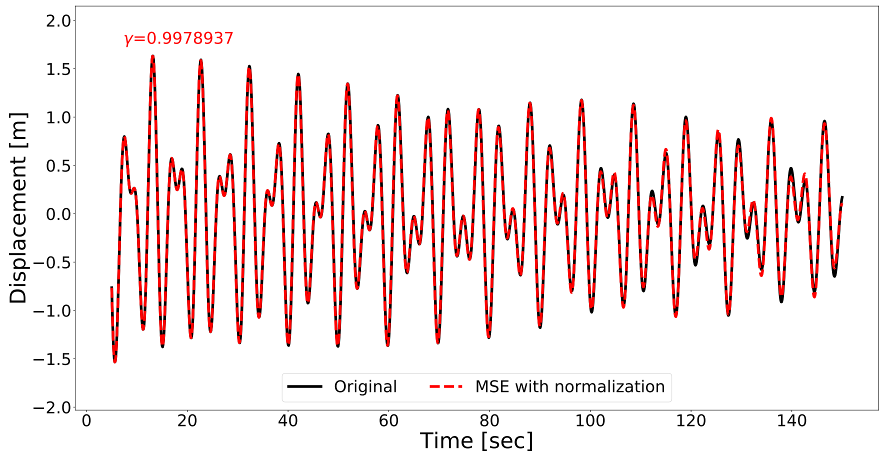


(b) non-linearity of 1.

Figure 4.16: First DOF predicted response of 2DOF system for extended time(150s).



(a) non-linearity of 0.5.



(b) non-linearity of 1.

Figure 4.17: Second DOF predicted response of 2DOF system for extended time(150s).

Chapter 5

Conclusion

The study in this thesis shows that LSTM informed by certain physics knowledge is a feasible data-driven model for representing and identifying nonlinear dynamic system with only system outputs. LSTM with model architecture of encoder-decoder can model the dynamics of Duffing Systems with lower and higher non-linearity for both single DOF and multiple DOFs. The scheme of using sub-sequences instead of the complete time sequence, taking into account of the time-varying dynamics of nonlinear system, is shown efficient with lesser data requirement. Moreover the additional term in the loss function for weighting every sub-sequence with non-uniform amplitude is introduced by considering the amplitude-dependent property of nonlinear dynamic systems.

Future work is considered to adapt the LSTM based approach for real-world applications such as learning the nonlinear dynamics from the structural response captured by the digital camera. Also, there is potential to test the model reliability for system with random excitation force. Moreover, the testing of this approach for variety of nonlinear dynamical system is also one of the future goals.

References

[1] O. Nelles, *Nonlinear System Identification:from Classical Approaches to Neural Network and Fuzzy Models.* Berlin, Germany: Springer, 2001, pp. 1–19.

[2] J. Schoukens and L. Ljung, “Nonlinear system identification: A user-oriented roadmap,” *CoRR*, vol. abs/1902.00683, 2019. [Online]. Available: http://arxiv.org/abs/1902.00683

[3] K. Worden and P. Green, “A machine learning approach to nonlinear modal analysis,” *Mechanical Systems and Signal Processing*, vol. 84, pp. 34–53, 2017.

[4] M. Korda and I. Mezić, “Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control,” *Automatica*, vol. 93, pp. 149–160, 2018.

[5] I. Mezić, “Spectrum of the koopman operator, spectral expansions in functional spaces, and state-space geometry,” *Journal of Nonlinear Science*, pp. 1–55, 2019.

[6] B. Lusch, J. N. Kutz, and S. L. Brunton, "Deep learning for universal linear embeddings of nonlinear dynamics," *Nature Communications*, vol. 9, no. 1, Nov 2018. [Online]. Available: http://dx.doi.org/10.1038/s41467-018-07210-0

[7] Yu Wang, "A new concept using lstm neural networks for dynamic system identification," in *2017 American Control Conference (ACC)*, 2017, pp. 5324–5329.

[8] J. Gonzalez and W. Yu, "Non-linear system modeling using lstm neural networks," *IFAC-PapersOnLine*, vol. 51, pp. 485–489, 2018.

[9] R. Zhang, Z. Chen, S. Chen, J. Zheng, O. *Büyüköztürk*, and H. Sun, "Deep long short-term memory neural networks for nonlinear structural seismic response prediction," *Computers and Structures*, vol. 220, pp. 55–68, August 2019.

[10] R. Zhang, Y. Liu, and H. Sun, "Physics-informed multi-lstm networks for metamodeling of nonlinear structures," *ArXiv*, vol. abs/2002.10253, 2020.

[11] R. J. Allemang, *Vibrations: Experimental Modal Analysis*, Structural Dynamics Research Laboratory, University of Cincinnati, OH., 1999.

[12] W. Heylen, S. Lammens, and P. Sas, *Modal Analysis Theory and testing.* Department of Mechanical Engineering, Katholieke Universiteit leuven, Leuven, Belgium: Katholieke Universiteit Leuven, Belgium, 1995.

[13] B. Peeters and G. De Roeck, "Stochastic System Identification for Operational Modal Analysis: A Review ," *Journal of Dynamic Systems, Measurement,*

and Control, vol. 123, no. 4, pp. 659–667, 02 2001. [Online]. Available: https://doi.org/10.1115/1.1410370

[14] P. Bart and G. De Roeck, "Reference-based stochastic subspace identification for output-only modal analysis," *Mechanical systems and signal processing*, vol. 13, no. 6, pp. 855–878, 1999.

[15] Y. Yang, C. Dorn, T. Mancini, Z. Talken, G. Kenyon, C. Farrar, and D. Mascareñas, "Blind identification of full-field vibration modes from video measurements with phase-based video motion magnification," *Mechanical Systems and Signal Processing*, vol. 85, pp. 567–590, 2017.

[16] C. R. Farrar and K. Worden, "An introduction to structural health monitoring," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 365, no. 1851, pp. 303–315, 2007.

[17] G. Kerschen, K. Worden, A. Vakakis, and J.-C. Golinval, "Past, present and future of nonlinear system identification in structural dynamics," *Mechanical Systems and Signal Processing - MECH SYST SIGNAL PROCESS*, vol. 20, pp. 505–592, 04 2006.

[18] M. Peeters, "Theoretical and experimental modal analysis of nonlinear vibrating structures using nonlinear normal modes," Ph.D. dissertation, University of Liege, 2010.

[19] J. Roll, A. Nazin, and L. Ljung, "Nonlinear system identification via direct weight optimization," *Automatica*, vol. 41, pp. 475–490, 03 2005.

[20] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Deylon, P. yves Glorennec, H. Hjalmarsson, and A. Juditsky, "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, vol. 31, pp. 1691–1724, 1995.

[21] K. Petsounis and S. Fassois, "Parametric time-domain methods for the identification of vibrating structures - a critical comparison and assessment," *Mechanical Systems and Signal Processing*, vol. 15, pp. 1031–1060, 11 2001.

[22] A. Juditsky, H. Hjalmarsson, A. Benveniste, B. Delyon, L. Ljung, J. Sjöberg, and Q. Zhang, "Nonlinear black-box models in system identification: Mathematical foundations," *Automatica*, vol. 31, no. 12, p. 1725–1750, 1995.

[23] S. A. Billings, *Nonlinear System Identification: NARMAX Methods in the Time, Frequency, and Spatio–Temporal Domains.* Chichester, UK: John Wiley & Sons, Ltd, 2013, pp. 1–11.

[24] B. C. Csáji *et al.*, "Approximation with artificial neural networks," *Msc thesis, Eötvös Loránd University*, 2001.

[25] C. S. Huang, S. Hung, C. Wen, and T. Tu, "A neural network approach for structural identification and diagnosis of a building from seismic response data," *Earthquake Engineering '—&' Structural Dynamics*, vol. 32, pp. 187 – 206, 02 2003.

[26] K. S. Narendra and K. Parthasarathy, “Gradient methods for the optimization of dynamical systems containing neural networks,” *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 252–262, 1991.

[27] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, “Convergence analysis of distributed stochastic gradient descent with shuffling,” *Neurocomputing*, vol. 337, pp. 46–57, 2019.

[28] K. S. Narendra and K. Parthasarathy, “Identification and control of dynamical systems using neural networks,” *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

[29] G. Lightbody and G. Irwin, “Multi-layer perceptron based modelling of nonlinear systems,” *Fuzzy Sets and Systems - FSS*, vol. 79, pp. 93–112, 04 1996.

[30] M. Raissi, “Deep hidden physics models: Deep learning of nonlinear partial differential equations,” *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 932–955, 2018.

[31] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations,” *arXiv preprint arXiv:1711.10561*, 2017.

[32] L. Medsker and L. C. Jain, *Recurrent Neural Networks: design and applications.* CRC Press, 1999.

[33] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” 2014.

[34] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” *Nature*, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[35] Q. Teng and L. Zhang, “Data driven nonlinear dynamical systems identification using multi-step cldnn,” *AIP Advances*, vol. 9, p. 085311, 08 2019.

[36] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[37] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2015.

[38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[39] F. Chollet, “keras,” https://github.com/fchollet/keras, 2015.

Appendix A

Sample Code

The following code is written in Python using Jupyter Notebook. This code uses Keras library for creating LSTM model and matplotlib to plot the results.

A.1 LSTM main.py

```
%matplotlib inline

from keras.models import Model
from keras.layers import CuDNNLSTM, Dense, Input, ↩
   Dropout
from keras.optimizers import Adam
from HelpingFunctions import *
```

```
import matplotlib.pyplot as plt
import numpy as np
import time

t0 = time.time()

Input_data='Y'
number_of_points=500
number_of_example=950
val_m = 50


training_data = np.loadtxt('./data/↩
   SingleDOFDuffingInitial_LHSDESIGN_N=1_[-4,4]_train_{}_↩
   {}*{}\
.csv'.format(Input_data,number_of_points,↩
   number_of_example),delimiter=',', dtype=np.float64)[:,↩
   np.newaxis]

validation_data = np.loadtxt('./data/↩
   SingleDOFDuffingInitial_LHSDESIGN_N=1_[-4,4]↩
   _validation_{}_{}*{}\
.csv'.format(Input_data,number_of_points,val_m),↩
   delimiter=',', dtype=np.float64)[:,np.newaxis]

print('Training data shape: ',training_data.shape)

number_of_features=training_data.shape[-1]
m=number_of_example

Tx=int(len(training_data)/number_of_example)
print('Length of each signal: ',Tx)
```

```
window_size=50
window_shift=window_size

print('WINDOW SIZE:  ',window_size)
print('WINDOW SHIFT: ',window_shift)
n_steps_in, n_steps_out = window_size, window_size

n_a=64


X_encoder,X_decoder,training_Y,val_data = ready_data(↩
   training_data,validation_data,number_of_features,m,↩
   val_m,Tx,window_size,window_shift,n_steps_in,↩
   n_steps_out,normalization=False)


print('Encoder X shape: ',X_encoder.shape)
print('Encoder val_x shape: ',val_data[0][0].shape)

print('Training Y shape: ',training_Y.shape)
print('Validation Y shape: ',val_data[1].shape)


LSTM_cell_encoder = CuDNNLSTM(n_a, return_state = True,↩
   name ='Encoder')
LSTM_cell_decoder = CuDNNLSTM(n_a, return_sequences = ↩
   True, return_state = True,name='Decoder')
dense_1 = Dense(32,activation='relu',name='Dense_1')
dense_2 = Dense(number_of_features,name='Dense_2')

def duffingModel(n_steps_in, n_steps_out, ↩
   number_of_features):
     """
     Arguments
```

```
    n_steps_in -- number of timesteps input to encoder.
    n_steps_out -- number of timesteps input to decoder.
    number_of_features -- number of variables that make ↩
       up a time-step.

    Returns
    model -- a keras model
    """

    encoder_X = Input(shape=(n_steps_in,↩
       number_of_features))
    decoder_X = Input(shape=(n_steps_out,↩
       number_of_features))
    X_e=encoder_X
    X_d=decoder_X

    a_e,h_e,c_e = LSTM_cell_encoder(X_e)
    a_d,h_d,c_d = LSTM_cell_decoder(X_d, initial_state=[↩
       h_e,c_e])
    out = dense_1(a_d)
    output = dense_2(out)


    model = Model(inputs=[encoder_X, decoder_X], outputs↩
       =output)

    return model


####################################
# TensorFlow wizardry
config = tf.ConfigProto()
```

```
# Don't pre-allocate memory; allocate as-needed
config.gpu_options.allow_growth = True

# Only allow a total of half the GPU memory to be ↩
   allocated
config.gpu_options.per_process_gpu_memory_fraction = 0.5

# Create a session with the above options specified.
K.tensorflow_backend.set_session(tf.Session(config=↩
   config))
###################################


Train_model = duffingModel(n_steps_in, n_steps_out, ↩
   number_of_features)
learning_rate = 0.01
opt = Adam(lr=learning_rate, beta_1=0.9, beta_2=0.999, ↩
   decay=0.01)

Train_model.compile(loss=custom_loss,optimizer=opt)

train_error=[]
validation_error=[]

######################################
epochs=10000

for epoch in range(epochs):
    t1=time.time()
    print('EPOCH: {}/{}'.format(epoch+1,epochs))

    fitted_model=Train_model.fit([X_encoder, X_decoder],↩
       training_Y,
                                 validation_data=↩
                                   val_data, batch_size↩
                                   =30, epochs=1, ↩
                                   verbose=0)

    train_error.extend(fitted_model.history['loss'])
    validation_error.extend(fitted_model.history['↩
       val_loss'])

    if (epoch+1)%100 ==0:
        plt.plot(np.log(train_error))
        plt.plot(np.log(validation_error))
        plt.show()

    t2=time.time()
    print('Time for this epoch: {:0.2f} minutes'.format↩
       ((t2-t1)/60))

t3 = time.time()
total = (t3-t0)/60
print("\nTotal time: {:0.2f} minutes".format(total))
```

A.2 Helping Functions.py

```
import numpy as np
import tensorflow as tf
import keras.backend as K
import matplotlib.pyplot as plt

def shuffle(x,y,d,M):
    '''
    Arguments

    x -- encoder input
    y -- target data
    d -- decoder input
    M -- number of examples

    Returns

    shuffled_x, shuffled_y, shuffled_d -- Shuffled ↩
       examples within the data
    '''

    # create list of indices [0,1,2...]
    indices=np.arange(M)
    np.random.shuffle(indices)


    shuffled_x=np.zeros(x.shape)
    shuffled_y=np.zeros(y.shape)
    shuffled_d=np.zeros(d.shape)

    for i in range(M):
        shuffled_x[i,:,:]=x[indices[i],:,:]
        shuffled_y[i,:,:]=y[indices[i],:,:]
        shuffled_d[i,:,:]=d[indices[i],:,:]

    return shuffled_x,shuffled_y,shuffled_d



def restack_data(sequence, n_steps_in, n_steps_out,step)↩
   :
    X, y = list(), list()
    x_in = 0
    y_out = 0
    count = 0
    while y_out < len(sequence):
        x_out = x_in + n_steps_in
        y_in = x_out
        y_out = y_in + n_steps_out

        seq_x, seq_y = sequence[x_in:x_out], sequence[↩
           y_in:y_out]

        x_in = x_in + step

        X.append(seq_x)
        y.append(seq_y)
        count+=1

    return np.array(X), np.array(y),count

def normalize(data,m,Tx,number_of_features):
    data = data.reshape(m,Tx,number_of_features)

    Max = np.max(data,axis=(1,2),keepdims=True)
    Min = np.min(data,axis=(1,2),keepdims=True)
    scaled_data = -1+2*(data-Min)/(Max-Min)
    scaled_data = scaled_data.reshape(-1,↩
       number_of_features)

    return scaled_data
```

```
def ready_data(training_data,validation_data,↩
   number_of_features,m,val_m,Tx,window_size,window_shift↩
   ,n_steps_in,n_steps_out, normalization):

    _,_,n_s = restack_data(training_data[0:Tx,0], ↩
       n_steps_in, n_steps_out,window_shift)

    for i in range(m):
        plt.plot(training_data[i*Tx:i*Tx+Tx,0])
    plt.show()

    #normalizing the data
    if normalization:
        training_data = normalize(training_data,m,Tx,↩
           number_of_features)
        for i in range(m):
            plt.plot(training_data[i*Tx:i*Tx+Tx,0])
        plt.show()
        validation_data = normalize(validation_data,↩
           val_m,Tx,number_of_features)
        print('\n\nEACH EXAMPLE IS NORMALIZED\n\n')

    # Restacking data in the shape (number of examples, ↩
       number of samples, n_steps_in)
    X0 = np.zeros((m, n_s, n_steps_in, ↩
       number_of_features))
    Y0 = np.zeros((m, n_s, n_steps_out, ↩
       number_of_features))
    D0 = np.zeros((m, n_s, n_steps_out, ↩
       number_of_features))

    val_X0 = np.zeros((val_m, n_s, n_steps_in, ↩
       number_of_features))
```

```
val_Y0 = np.zeros((val_m, n_s, n_steps_out, ↩
   number_of_features))
val_D0 = np.zeros((val_m, n_s, n_steps_out, ↩
   number_of_features))
for i in range(m):
    for j in range(number_of_features):
        X0[i,:,:,j], Y0[i,:,:,j], n_s = restack_data↩
           (training_data[Tx*i:Tx*i+Tx,j], n_steps_in↩
           , n_steps_out,window_shift)

for i in range(val_m):
    for j in range(number_of_features):
        val_X0[i,:,:,j], val_Y0[i,:,:,j], n_s = ↩
           restack_data(validation_data[Tx*i:Tx*i+Tx,↩
           j], n_steps_in, n_steps_out,window_shift)


print('X shape: ',X0.shape)
print('validation X shape: ',val_X0.shape)

# Training,validation sets

x = X0.reshape(-1,X0.shape[2],X0.shape[3])
y = Y0.reshape(-1,Y0.shape[2],Y0.shape[3])
d = D0.reshape(-1,D0.shape[2],D0.shape[3])

# shuffle all examples
x,y,d=shuffle(x,y,d,x.shape[0])

val_x = val_X0.reshape(-1,val_X0.shape[2],val_X0.↩
   shape[3])
val_y = val_Y0.reshape(-1,val_Y0.shape[2],val_Y0.↩
   shape[3])
```

```
    val_d = val_D0.reshape(-1,val_D0.shape[2],val_D0.↩
       shape[3])

    val_data = [[val_x,val_d],val_y]

    return x,d,y,val_data

def max_freq(y):
    w=50
    y=tf.cast(y,tf.complex64)
    ft = tf.signal.fft(y)
    freq = tf.constant(np.linspace(0,10/2,int(w/2)),↩
       dtype=tf.float32)   # given sampling freq = 10
    fft = abs(ft[:,0:int(w/2)])* (2/w)
    indices = tf.argmax(fft,axis=-1)
    Freq = tf.map_fn(lambda x: freq[x],indices,dtype=tf.↩
       float32)

    return Freq

def normalize_tensor(y):
    ymin=tf.reduce_min(y,axis=(-1,-2),keepdims=True)
    ymax=tf.reduce_max(y,axis=(-1,-2),keepdims=True)

    y_n = (y-ymin)/(ymax-ymin)
    return y_n

def custom_loss(y_true,y_pred):

    alpha_1 =  1
    alpha_2 =  0.000005
    print('\n\nALPHA 2 : ',alpha_2,'\n\n')

    #F_actual_1 = max_freq(y_true[:,:,0])
    #F_pred_1 = max_freq(y_pred[:,:,0])

    y_true_N = normalize_tensor(y_true)
    y_pred_N = normalize_tensor(y_pred)

    loss_value = K.mean(K.square(y_true-y_pred),axis=-1)↩
        + alpha_2*K.mean(K.square(y_true_N-y_pred_N),axis↩
       =-1) #+ 0.0001*K.mean(K.square(F_actual_1-F_pred_1↩
       ),axis=-1)

    return loss_value
```

A.3 Prediction.py

```
######################################
%matplotlib inline
from numpy import array
from keras.models import Model, load_model
from keras.layers import CuDNNLSTM, Dense, Input, Lambda↩
   , Reshape
from keras.optimizers import Adam
import keras.backend as K
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import time
t0 = time.time()
```

```
####################################
# TensorFlow wizardry
config = tf.ConfigProto()

# Don't pre-allocate memory; allocate as-needed
config.gpu_options.allow_growth = True

# Only allow a total of half the GPU memory to be ↩
   allocated
config.gpu_options.per_process_gpu_memory_fraction = 0.5

# Create a session with the above options specified.
K.tensorflow_backend.set_session(tf.Session(config=↩
   config))
####################################




Input_data='Y'
number_of_points=500
number_of_example=1000


data = np.loadtxt('./data/↩
   SingleDOFDuffingInitial_LHSDESIGN_N=1_[-4,4]_test_{}_↩
   {}*{}\
.csv'.format(Input_data,number_of_points,↩
   number_of_example),delimiter=',', dtype=np.float64)[:,↩
   np.newaxis]

print(data.shape)
number_of_features=data.shape[-1]
```

```
m=number_of_example

Tx=int(len(data)/number_of_example)

window_size=50
n_steps_in, n_steps_out = window_size, window_size

Ty=Tx # length of predicted signal
n_s = int((Ty-window_size)/window_size)


def normalize(data,m,Tx,number_of_features):
    data = data.reshape(m,Tx,number_of_features)

    Max = np.max(data[:,:50,:],axis=(1,2),keepdims=True)
    Min = np.min(data[:,:50,:],axis=(1,2),keepdims=True)
    scaled_data = -1+2*(data[:,:50,:]-Min)/(Max-Min)
    scaled_data = scaled_data.reshape(-1,↩
       number_of_features)
    return scaled_data

testing_data = data #normalize(data,m,Tx,↩
   number_of_features)

print('Testing data shape: ',testing_data.shape)



# number of activation units in LSTM
n_a=64
# Creating instances of LSTM and other layers. Default ↩
   activation in CuDNNLSTM is tanh
LSTM_cell_encoder = CuDNNLSTM(n_a, return_state = True,↩
   name ='Encoder')
```

```
LSTM_cell_decoder = CuDNNLSTM(n_a, return_sequences = ↩
   True, return_state = True,name='Decoder')
dense_1 = Dense(32,activation='relu',name='Dense_1')
dense_2 = Dense(number_of_features,name='Dense_2')

def CreateDuffing(n_s, n_steps_in, n_steps_out, ↩
   number_of_features):
    """
    Arguments

    n_s -- number of samples after using window.
    n_steps_in -- number of timesteps input to encoder.
    n_steps_out -- number of timesteps input to decoder.
    number_of_features -- number of variables that make ↩
       up a time-step.

    Returns
    model -- a keras model
    """

    encoder_X = Input(shape=(n_steps_in,↩
       number_of_features))
    X_e = encoder_X
    decoder_X = Input(shape=(n_steps_out,↩
       number_of_features))
    X_d = decoder_X

    a_e,h_e,c_e = LSTM_cell_encoder(X_e)

    a_d,h_d,c_d = LSTM_cell_decoder(X_d, initial_state=[↩
       h_e,c_e])

    out = dense_1(a_d)
    output = dense_2(out)
    duffing_model = Model(inputs=[encoder_X,decoder_X], ↩
       outputs=output)

    return duffing_model

Duffing_model = CreateDuffing(n_s, n_steps_in, ↩
   n_steps_out, number_of_features)

Duffing_model.load_weights('Duffing_Model_1DOF_N=1_↩
   [-4,4]_Y_custom loss_v3.1.5.5_Tx=500,\
TrainEx=950,ValEx=50,MOVINGwindow=50_STEP=50_epochs↩
   =10000_lr=0.01.h5')

#Tx=50
predicted_2=np.zeros((m,500,number_of_features))
for example in range(m):
    X_test = testing_data[example*Tx:example*Tx+↩
       n_steps_in].reshape(1,n_steps_in,↩
       number_of_features)
    predictions=[]
    for n in range(n_s):
        decoder_input=np.zeros((1,n_steps_out,↩
           number_of_features))
        prediction = Duffing_model.predict([X_test,↩
           decoder_input])
        prediction = array([np.squeeze(i) for i in ↩
           prediction])
        X_test=prediction.reshape(1,n_steps_in,↩
           number_of_features)
        predictions.extend(prediction.reshape(n_steps_in↩
           ,number_of_features))
    predicted_2[example,n_steps_in:,:]=predictions
    predicted_2[example,:n_steps_in,:]=testing_data[↩
        example*Tx:example*Tx+n_steps_in].reshape(1,↩
        n_steps_in,number_of_features)


t3 = time.time()
total = (t3-t0)/60
print("Total time: {:0.2f} minutes".format(total))
########################################

#CALCULATING ERROR

%matplotlib inline

MSE_1=[]
MSE_2=[]

for example in range(m):
    MSE_1.append(np.mean((testing_data[example*Tx+↩
        n_steps_in:example*Tx+Tx,0]-predicted_1[example,↩
        n_steps_in:,0])**2))
    MSE_2.append(np.mean((testing_data[example*Tx+↩
        n_steps_in:example*Tx+Tx,0]-predicted_2[example,↩
        n_steps_in:,0])**2))

NMSE_1=[]
NMSE_2=[]

for example in range(m):
    x = testing_data[example*Tx+n_steps_in:example*Tx+Tx↩
        ,0]
    y_1 = predicted_1[example,n_steps_in:,0]
    Sx_1=(np.sum((x-np.mean(x))**2)/(450-1))
    NMSE_1.append(np.mean((x-y_1)**2)/Sx_1)
    y_2 = predicted_2[example,n_steps_in:,0]
    NMSE_2.append(np.mean((x-y_2)**2)/Sx_1)


crcf_1=[]
crcf_2=[]
for example in range(m):
    coeff = np.corrcoef(testing_data[example*Tx+↩
       n_steps_in:example*Tx+Tx,0],predicted_1[example,↩
       n_steps_in:,0])
    crcf_1.append(coeff[0,1])
    coeff = np.corrcoef(testing_data[example*Tx+↩
       n_steps_in:example*Tx+Tx,0],predicted_2[example,↩
       n_steps_in:,0])
    crcf_2.append(coeff[0,1])

#######################################
# PLOTTING RESULTS

plt.plot(MSE_1)
print('Max error: ',max(MSE_1))
print('Example number with  max mse: ',np.argmax(MSE_1))
print('Min error: ',min(MSE_1))
print('Example number with  min mse: ',np.argmin(MSE_1))
plt.title('MSE Y1')
plt.show()

example=np.argmax(MSE_1)
coeff = np.corrcoef(testing_data[example*Tx+n_steps_in:↩
   example*Tx+Tx,0],predicted_1[example,n_steps_in:,0])
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
```

```
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Worst mse example Y1\nExample: {}  CRCF: ↩
   {:0.7f}\nMSE: {:0.7f} NMSE: {:0.7f}'.format(example,↩
   coeff[0,1],MSE_1[example],NMSE_1[example]))
plt.show()

example=np.argmin(MSE_1)
coeff = np.corrcoef(testing_data[example*Tx+n_steps_in:↩
   example*Tx+Tx,0],predicted_1[example,n_steps_in:,0])
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Best mse example Y1\nExample: {}  CRCF: {:0.7↩
   f}\nMSE: {:0.7f} NMSE: {:0.7f}'.format(example,coeff↩
   [0,1],MSE_1[example],NMSE_1[example]))
plt.show()

#####################################



plt.plot(NMSE_1)
print('Max error: ',max(NMSE_1))
print('Example number with  max NMSE: ',np.argmax(NMSE_1↩
   ))
print('Min error: ',min(NMSE_1))
print('Example number with  min NMSE: ',np.argmin(NMSE_1↩
   ))
print('Mean of NMSE: ',np.mean(NMSE_1))
plt.title('NMSE Y1')
plt.show()

example=np.argmax(NMSE_1)
```

```
coeff = np.corrcoef(testing_data[example*Tx+n_steps_in:↩
   example*Tx+Tx,0],predicted_1[example,n_steps_in:,0])
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Worst NMSE example Y1\nExample: {}  CRCF: ↩
   {:0.7f}\nMSE: {:0.7f}  NMSE: {:0.7f}'.format(example,↩
   coeff[0,1],MSE_1[example],NMSE_1[example]))
plt.show()

example=np.argmin(NMSE_1)
coeff = np.corrcoef(testing_data[example*Tx+n_steps_in:↩
   example*Tx+Tx,0],predicted_1[example,n_steps_in:,0])
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Best NMSE example Y1\nExample: {}  CRCF: ↩
   {:0.7f}\nMSE: {:0.7f}  NMSE: {:0.7f}'.format(example,↩
   coeff[0,1],MSE_1[example],NMSE_1[example]))
plt.show()

#######################################

n,bins,_ = plt.hist(crcf_1)
no_of_bins = len(n)
for i in range(no_of_bins):
    plt.text(bins[i],n[i],str(n[i]))

plt.grid(axis='y')
plt.xlabel('Correlation Coefficient')
plt.ylabel('Number of Examples')
plt.title('crcf Y1')
plt.show()

example=np.argmin(crcf_1)
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Worst crcf example Y1\nExample: {}  CRCF: ↩
   {:0.7f}\nMSE: {:0.7f}  NMSE: {:0.7f}'.format(example,↩
   crcf_1[example],MSE_1[example],NMSE_1[example]))
plt.show()

example=np.argmax(crcf_1)
plt.plot(testing_data[example*Tx+n_steps_in:example*Tx+↩
   Tx,0], label='Actual')
plt.plot(predicted_1[example,n_steps_in:,0], label='↩
   Predicted')
plt.title('Best crcf example Y1\nExample: {}  CRCF: ↩
   {:0.7f}\nMSE: {:0.7f}  NMSE: {:0.7f}'.format(example,↩
   crcf_1[example],MSE_1[example],NMSE_1[example]))
plt.show()
```