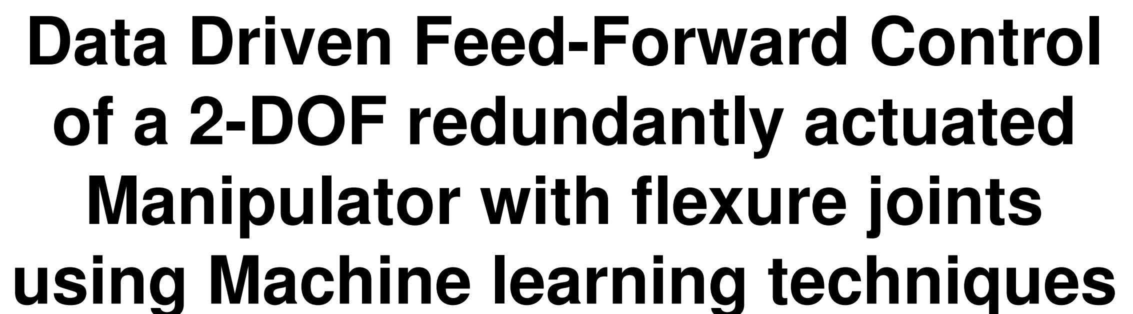
Faculty of Engineering Technology



Prasanna Sathiyanarayanan

M.Sc. Thesis Systems and Control
November 2021

Supervisors:
Prof. Dr. Ing. B. Rosic.
Dr. Ir. R.G.K.M. Aarts.

Chair of Applied Mechanics and Data Analysis
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Acknowledgement

I would like to express my heartfelt gratitude to my daily supervisors Dr. Ronald Aarts and Dr. Bojana Rosic. Both of them were instrumental in fulfilling every stage of my learning through out this thesis.

Dr. Rosic's strong knowledge in machine learning were the driving force behind my accomplishments, her inputs gave me new perspectives on the project which I hadn't considered yet.

The time Dr. Aarts spent with me in the laboratory guiding me through the technicalities of the manipulator and his valuable inputs were vital in shaping up the outcome of this thesis.

A special thanks to Leo Tiemersma for his mechanical guidance on the manipulator.

I would finally like to thank my family and friends for their incredible moral support during this academic endeavour.

Abstract

Robotic manipulators are capable of performing various tasks at speeds and accuracies that far exceeds those of human operators. They find applications in various industries ranging from manufacturing to precision engineering. To perform their tasks reliably at such high speeds with precision the end effectors of the manipulator are controlled digitally. Control of robot manipulators can be greatly improved with the use of feedforward control. However, the effectiveness of the feedforward control heavily relies on the accuracy of the inverse dynamics model of the manipulator. Classical physical modelling approaches suffers from modelling inaccuracies due to uncertain factors of the manipulator such as flexibility, friction and hysteresis, etc. Alternatively, artificial intelligence techniques are becoming increasingly popular in robotics applications in the recent years. It is hypothesised that Machine Learning techniques offer an alternative for the (complex) modelling approach and still enable an efficient and accurate implementation of the feedforward control.

In this thesis, a data-driven approach using machine learning techniques for modelling the inverse dynamics of the manipulator is studied. It is researched which type of algorithms are suited for this purpose. Based on the research three neural network-based machine learning techniques are proposed and developed, namely, LSTM network, CNN-LSTM network and TCN network. A comparison study on the performance of these networks are drawn. Investigations are made on the feasability of applying these techniques for a feedforward control.

In this study, an existing flexure based 2 DOF redundantly actuated manipulator is considered as the real time test setup upon which feedforward control using the proposed machine learning techniques are applied. The machine learning algorithms are first developed on a simulation model of the test setup, later it is developed for the real time setup. A feedforward control is designed based on the developed algorithms and they are put to test on both the simulation environment and on the real time system. Several use cases were investigated and experiments were performed. The results showcased that machine learning algorithms are powerful tool for modelling the inverse dynamics, and the designed feedforward control based on these techniques improved the manipulator's precision and tracking performance.

List of Figures

1.1 Redundantly actuated 2-DOF manipulator with flexure joints [1] 2
1.2 Comparison between RNN and FNN [2] 4

2.1 Manipulator rigid body diagram 8
2.2 Control Schematic 10
2.3 Recurrent Neural Network Cell 13
2.4 Recurrent Neural Network Structure 14

3.1 LSTM Cell Architecture 16
3.2 1D Convolution operation 18
3.3 CNN-LSTM Network 18
3.4 Standard Convolution (on the left) and Causal convolution (on the right) 19
3.5 Temporal convolutional network Architecture [3] 20

4.1 Simulink Block Diagram 22
4.2 Redundant Manipulator test setup 23
4.3 Circular Trajectories 26
4.4 Random Trajectories 27
4.5 Sample trajectory with quantization and white noises 28
4.6 Training and Testing Data-points (Simulation Environment) 29
4.7 Circular Trajectories 31
4.8 Random Trajectories in Real time environment 32
4.9 Training and Testing Data-points (Real time environment) 33

5.1 Lag Plots of End effector forces along x and y directions 36
5.2 LSTM Network Design 37
5.3 Loss Function plot - Comparison between different learning rate . . . 38
5.4 Loss function against Learning rate 39
5.5 Loss Function plot - Comparison between different hidden neurons . . 40
5.6 Loss function against number of hidden neurons 40
5.7 LSTM Training and testing loss over 1000 epochs 41
5.8 Loss function against maximum number of epochs 41
5.9 LSTM Predictions 42

5.10 TCN Network Design 43
5.11 Loss Function plot - Comparison between different learning rate (TCN) 44
5.12 Loss function against Learning rate (TCN) 45
5.13 Loss Function plot - Comparison between different hidden neurons (TCN) 46
5.14 Loss function against number of hidden neurons (TCN) 46
5.15 TCN Training and testing loss over 1500 epochs 47
5.16 Loss function against maximum number of epochs (TCN) 47
5.17 TCN Predictions 48
5.18 1D CNN-LSTM Network Design 48
5.19 CNN-LSTM Training and testing loss 50
5.20 CNN-LSTM Predictions 50
5.21 Time taken by networks 51
5.22 parameters to train by the networks 51
5.23 Simulink Block Diagram with feedforward control 52
5.24 Reference Path 53
5.25 Feedforward Evaluation 54
5.26 Error Plot 55

6.1 Lag Plots of End effector forces along x and y directions 58
6.2 LSTM Network Design 59
6.3 LSTM Training and testing loss 60
6.4 LSTMPredictions 60
6.5 TCN Network Design 61
6.6 TCN Training and testing loss 62
6.7 TCN Predictions 63
6.8 Time taken by networks 64
6.9 parameters to train by the networks 64
6.10 Feedforward evaluation on real time system 66
6.11 Tracking performance along x axis 67
6.12 Tracking performance along y axis 68
6.13 System Performance with feedback controller 69
6.14 System Performance with LSTM based feedforward control 69
6.15 System Performance with TCN based feedforward control 70
6.16 Error Plot during the three scenarios 71

A.1 Artificial Neuron 79
A.2 Feeforward neural network 80

C.1 Single Kinematic Chain [4] 83

D.1 Feedforward evaluation for a random trajectory 85

List of Tables

4.1 Design Parameters 22
4.2 Maxon motor and encoder specifications 23

5.1 loss function comparison 51
5.2 Feedforward Evaluation 55

6.1 loss function comparison (Real time environment) 63
6.2 Feedforward Evaluation from real time system 71

D.1 Feedforward Evaluation on random trajectory 86

Contents

Acknowledgement iii

Abstract v

List of acronyms xv

1 Introduction 1

1.1 Motivation . . . 2
1.2 Literature Review . . . 3
1.3 Goals and Research Questions . . . 4
1.4 Report organization . . . 5

2 Background and State of the Art 7

2.1 Manipulator Dynamics . . . 7
2.2 Inverse Dynamics . . . 9
2.3 Control Scheme . . . 9
2.4 Control Allocation . . . 10
2.5 Feedforward Control for the Manipulator . . . 11
2.6 Machine Learning for Inverse Dynamics Prediction . . . 12

3 Machine learning Driven Control 15

3.1 Long short term memory networks (LSTM) . . . 15
3.2 Convolutional LSTM Network (CNN-LSTM) . . . 17
3.3 Temporal Convolutional neural network (TCN) . . . 19

4 Modelling and Experimental Setup 21

4.1 Simulink Model of the Manipulator . . . 21
4.2 Experimental setup . . . 22
4.3 Data Acquisition . . . 23
4.3.1 Data Acquisition from the Simulation Environment . . . 24
4.3.2 Data Pre-processing for simulation model . . . 27
4.3.3 Data Acquisition from the Real-time Environment . . . 29

4.4 Data Pre-Processing for real time environment 32

5 Results from Simulation Environment 35
5.1 Data Visualization 35
5.2 Setting up the learning environment 35
5.3 LSTM Network Design 36
5.3.1 Hyperparameters tuning and experimental results 37
5.4 TCN network Design 42
5.4.1 Hyperparameters tuning and experimental results 43
5.5 1D-CNN LSTM network Design 48
5.5.1 Experimental Results 49
5.6 Results Comparison 51
5.7 Feed Forward Control and Evaluation 52

6 Results from Real-time Environment 57
6.1 Data Visualization 57
6.2 Setting up the learning environment 57
6.3 LSTM Network Design 58
6.3.1 LSTM Network Experimental Results 59
6.4 TCN Network Design 61
6.4.1 TCN Network Experimental Results 62
6.5 Results Comparison 63
6.6 Feedforward Control and Evaluation 64

7 Conclusions and recommendations 73
7.1 Conclusions 73
7.2 Future Recommendations 74

References 75

Appendices

A Feedforward Neural Networks 79

B Back Propagation Through Time 81

C Kinematics of the Manipulator 83

D Additional Results 85

List of acronyms

DOF Degrees of freedom

SISO Single Input Single Output

MIMO Multi Input Multi Output

FNN Feedforward Neural Networks

RNN Recurrent Neural Networks

LSTM Long-Short Term Memory

CNN Convolutional Neural Networks

TCN Temporal Convolutional Networks

EE End Effector

FB Feed Back

FF Feed Forward

ILC Iterative Learning Control

RC Repetitive Control

TBPTT Truncated Back Propagation Through Time

Chapter 1

Introduction

With the world moving towards automation steadily over the past century, the demand for high precision technology has increased. This is true especially in the case of robotic manipulators, which are pervasive throughout many industries such as manufacturing, automotive, aerospace and so on. High precision manipulators are typically used for manufacturing components with nanometric details and assembly of optical circuits. Manipulators with flexure joints have joints based on flexible elements instead of conventional bearings. Thereby, backlash or friction is virtually non-existent in this mechanism. One such manipulator is considered in this study, and it is shown in Figure 1.1. The manipulator is redundantly actuated as the number of actuators are higher than the number of DOF of the end-effector. Over-actuating the manipulator opens up some extra possibilities. Few advantages of a redundant manipulator include, more power for actuating the end-effector, and assistance in traversing singular points in the work area of the end-effector.

Due to the absence of any hysteresis accurate manipulation is possible while using this redundantly actuated manipulator [5]. The achievable accuracy then depends heavily on the quality of force or torque feedforward control that can be applied for a specific motion profile. Applying a feedforward control requires a good knowledge of the inverse dynamics of the system. The optimal feedforward control for a system is the inverse of the plant model itself [6]. Several classical approaches for designing the feedforward control exists. In this study, a data-driven approach is considered for designing a feedforward controller for this manipulator.

With the capabilities of modern computer to process huge amounts of data, machine learning and neural networks are increasingly making their way into practical robotics applications [7]. Hence, in this study feasibility of applying machine learning techniques for learning the inverse dynamics of the manipulator is investigated, and a feedforward control is developed based on the learned inverse dynamic model for improving the tracking performance of the manipulator.

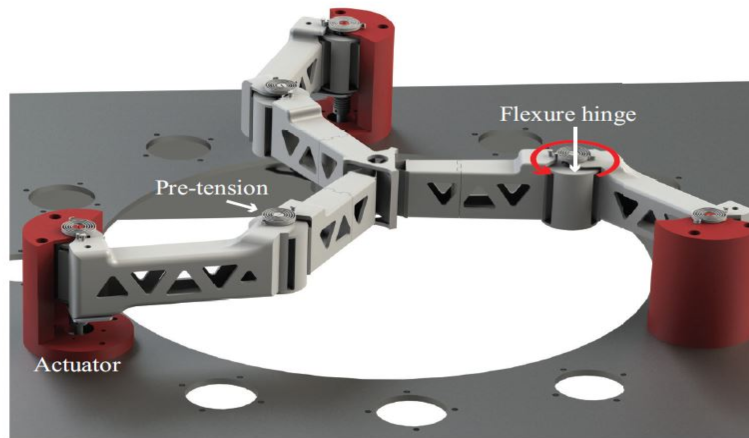


Figure 1.1: Redundantly actuated 2-DOF manipulator with flexure joints [1]

1.1 Motivation

Typically in a robot control, a feedback controller and a feedforward controller are present. Usually, the feedback controller maintains stability of the system while ensuring a good tracking performance. However, in any motion control systems the closed loop controller bandwidth is always limited, e.g. due to delay, higher order dynamics, controller performance etc. Thus, achieving precise and smooth motion are not possible with just the feedback controller alone [8]. Therefore, feedforward control becomes particularly important. A possible method to determine the feedforward follows a model-based approach, where a dynamic model is developed for the system based on rigid body dynamics formulation. Such models rely on precise determination of physical parameters of the rigid bodies (in this case, manipulator's flexure joints), composing the robot. Often times, these parameters have to be identified and then employed in model-based control and state estimation schemes [9]. The disadvantages of such parametric model is that they are only crude idealizations of the actual robot dynamics. In many cases these parameters cannot be identified precisely. These models do not account for missing dynamic behaviours as well, rendering such models to be inaccurate [10].

This motivates for finding an alternative modelling approach which can enable an efficient and accurate implementation of the feedforward. One of the possibilities is to make use of experimental data to approximate the input-output relationship of the system, using machine learning techniques, making it a data-driven model. Such models do not require the knowledge of the physical properties of the system compared with the classical methods. With a suitable learning architecture designed, the machine learning models can be more flexible to use and are powerful in capturing higher order nonlinearities that cannot be considered by a parametric model [10]. Thus, this could lead to a more precise manipulation of the robot.

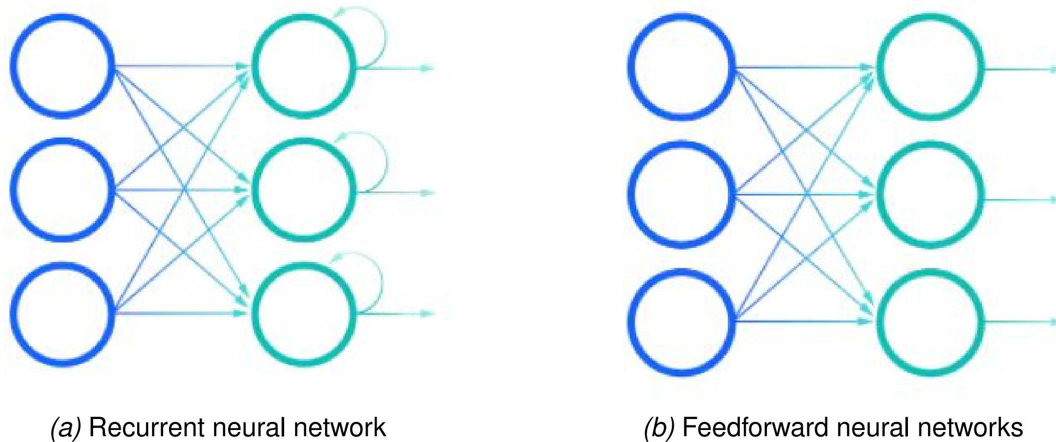
1.2 Literature Review

With the advancements in the field of data science, various machine learning techniques such as neural networks have been developed for addressing various classification and regression problem in practice. From the machine learning perspective, predicting the feedforward control comes under the regression problem, as the feedforward control requires continuous updating of the end effector forces, to improve the tracking performance of the manipulator. Since this study deals with a non-linear system, neural networks are an ideal choice for addressing such regression problems, as they are quite flexible when it comes to modelling non-linearity [11].

A subfield of machine learning is deep learning, which has gained immense popularity over the last decade. A typical deep learning network consists of multiple layers of neural layers stacked one after the other (in other words multiple hidden layers) forming a deep architecture before a final prediction is made. Compared to shallow neural networks, deep learning neural networks can extract hidden natural structures and inherent abstract features of the data better. Thus, deep learning methods could be promising in modelling inverse dynamics of a manipulator. The work done in [12] proposes a Feedforward Neural Network (FNN) for modelling the inverse kinematics and inverse dynamics of a 6 DOF Yasakawa Robot. The trained model is used for updating velocity and torque feedforward control on the robot. In [13], a comparison between the state of the art inertial parameter estimation to a purely data-driven approach using feedforward network for constructing a feedforward control on a humanoid robot (Apollo) is studied. The results showed that neural-network based approach yielded accurate results on real-time data compared to the former. Although, the neural networks proposed in these papers showed reasonable results, the feedforward neural networks do not take advantage of the time dependencies or auto-correlation within the time-series data, thus limiting the performance of such networks.

In [14], a Recurrent Neural Network (RNN) is proposed for modelling dynamic system. In the Figure 1.2 a comparison between a RNN and a FNN is displayed, as can be seen RNNs have recurrent connections in their hidden layer while the FNNs do not. The RNNs are distinguished by their 'recurrent' unit as they take information from previous inputs to model the current input and output [2]. Often times, RNN experiences vanishing gradient problem while training [15]. Hence, Long Short Term Memory(LSTM) network based architecture was proposed in [16] for modelling the inverse dynamics of a 6 DOF collaboration robot (UR5). LSTM is a more sophisticated recurrent network, which can handle long term dependencies in

a time series data and does not experience any vanishing gradient problem. The LSTM architecture proposed in [16] showed good results for predicting the inverse dynamics of the UR5 robot, and the effects of different hyper-parameter settings were evaluated. But, their work does not include application of feedforward control using the network, and the LSTM network is not trained over large iterations, which could improve the prediction accuracy.



(a) Recurrent neural network *(b)* Feedforward neural networks

Figure 1.2: Comparison between RNN and FNN [2]

Most of the prior works [12], [13] done for predicting inverse dynamics use feed-forward neural networks, which showed reasonable results, but it could be improved when using RNNs. The works that did consider RNN [10], [16] do not evaluate the performance of the feedforward controller making use of the learned model. Moreover, the redundant manipulator considered in this study exhibits different dynamics compared to the robots that were used in the existing works, this means that, different neural network architectures with different hyper parameters settings have to be experimented to obtain an optimal learned model that can be used for feedforward control.

1.3 Goals and Research Questions

The main focus of this study revolves around the following research questions: *"are machine learning techniques suitable for approximating the inverse dynamics of the redundant manipulator?, if so which techniques could be suitable?"* and *"How well the trained model can enable an implementation of feedforward control, that can improve the tracking performance of the redundant manipulator?"*

For this, a kinematic and a dynamic model capturing the redundant manipulator's behaviour is developed, for testing in the simulation environment. Investigations are made on the type of machine learning techniques that can be potentially used for

feedforward control. The models are trained, first with the data gathered from the simulation environment, and later with the data gathered from the real-time system. A feedforward controller is developed based on the trained model for both the simulation environment and the real-time system. Experimental results are gathered and the performance of the feedforward controller is evaluated.

1.4 Report organization

The remainder of this report is organized as follows. Chapter 2 details the background for this study and details about the state of the art. The Chapter 3 provides an overview of the machine learning techniques that were considered during this study. In Chapter 4, the simulation model and the existing test setup is introduced, followed by the explanation about the data acquisition process. Chapter 5 details out the architectures of the machine learning techniques implemented on the simulation environment and the results obtained thereof. Similarly, chapter 6 delves into the architectures implemented on the real time system and the results obtained thereof. In Chapter 7 the conclusions from the study and the recommended future works are presented.

Chapter 2

Background and State of the Art

This chapter provides the necessary background knowledge and the overall scope of this study. Special focus is put on the description of the model problem, and the formulation of an optimal feedforward control in terms of machine learning techniques.

2.1 Manipulator Dynamics

This study deals with a redundantly actuated manipulator. Redundantly actuated basically means, that the manipulator has more actuators than the degrees of freedom. In this case, the end-effector of the manipulator can move in *x* and *y* directions in the cartesian coordinates(2 DOF), while it is being actuated by three joint motors.

The manipulator has seven rigid bodies. These include the three upper arms, three lower arms and the wrist hinge which is the end effector, see figure 2.1. The motion of these rigid bodies in planar 2D can be described by a set of 21 generalized coordinates. These generalized coordinates showcase the translation in xy plane and rotation around the z-axis for every rigid body in the manipulator, and they are represented in the vector $\vec{q}$ as,

$$\begin{aligned}\vec{q} = [&x_{up1}\, y_{up1}\, \theta_{s1}\, x_{low1}\, y_{low1}\, \theta_{e1}\, x_{up2}\, y_{up2}\, \theta_{s2}\, x_{low2}\, y_{low2}\, \theta_{e2} \\ &x_{up3}\, y_{up3}\, \theta_{s3}\, x_{low3}\, y_{low3}\, \theta_{e3}\, x_{ee}\, y_{ee}\, \theta_{ee}]\end{aligned} \tag{2.1}$$

Here, the θ_{ee} represents the end-effector joint's angle of deformation. The equation of motion for the unconstrained system are derived using the well established Euler-Lagrange method. To reduce the unconstrained equations of motion to a constrained 2DOF equation of motion, the minimum coordinates formulation presented in [17] is adopted. The minimal coordinates are formed by applying geometric constraints to the manipulator (refer Appendix A in [5]). By applying these constraints the gener-

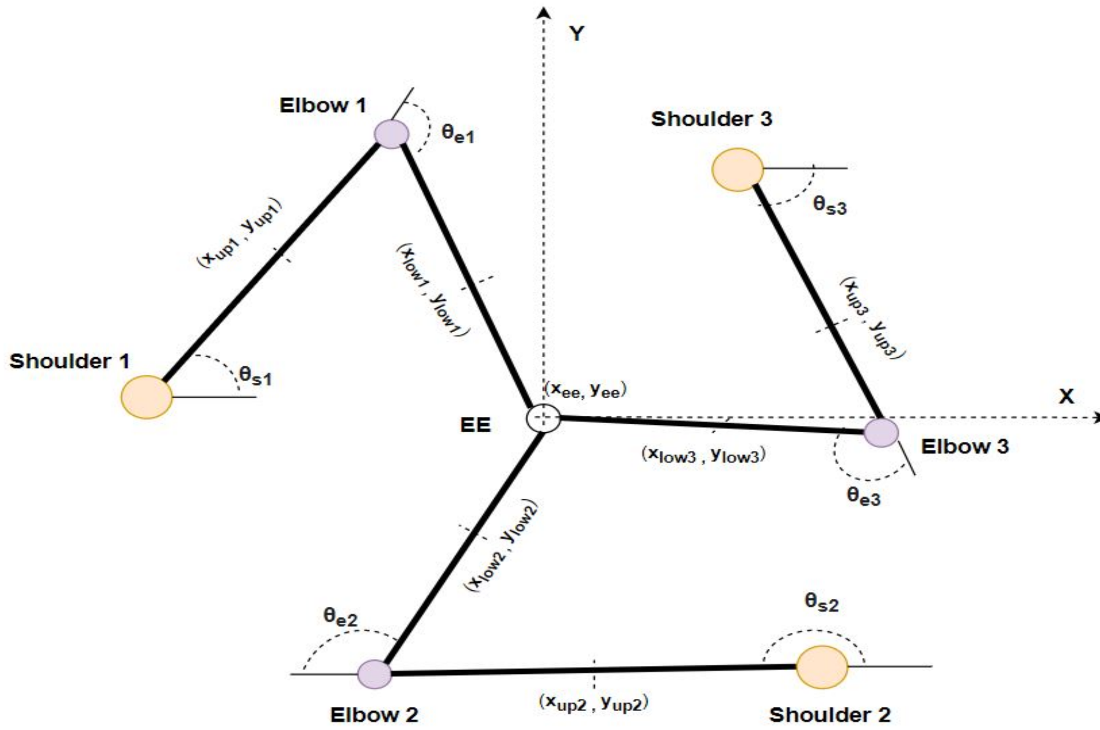


Figure 2.1: Manipulator rigid body diagram

alized coordinates can be split into dependent and independent coordinates. The independent coordinates here are chosen to be the end effector positions x_{ee} and y_{ee} and the rest of the generalized coordinates are set to be dependent. The resulting equations of motion can be expressed by the following equation (for detailed derivation refer [5]),

$$\bar{M}(\vec{q})\ddot{\vec{q}}_i + \bar{C}(\vec{q},\dot{\vec{q}})\dot{\vec{q}}_i + \bar{Q}(\vec{q}) = A(\vec{q})^T\vec{\tau} \tag{2.2}$$

where, $\bar{M}$ represents the inertia matrix function, $\bar{C}$ is the coriolis matrix function and the $\bar{Q}$ is the stiffness matrix function. The $\vec{q}_i$ represents the independent coordinates $[x_{ee}, y_{ee}]$, while $\vec{\tau}$ is the vector with input torques provided by the actuators at the shoulder hinges. The $A(\vec{q})^T$ is the jacobian matrix function that transforms the joint torques to the forces experienced by the end effector of the manipulator in x and y direction. This is expressed by the following equation,

$$A(\vec{q})^T\vec{\tau} = \vec{F_{ee}} \tag{2.3}$$

where, $F_{ee} \in \mathbb{R}^{i\times 1}$, is a vector containing the effective end effector forces of the manipulator along two independent coordinates $\vec{q}_i$, i.e, along the x and y direction in the cartersian coordinates.

This forms the forward dynamics relationship of the manipulator under an ideal

situation, as in, with the knowledge of input Forces $\vec{F_{ee}}$ applied and the position $\vec{q}$ of the system, the state variables [$\dot{\vec{q}}, \ddot{\vec{q}}$] (velocity and acceleration) of the manipulator can be found out.

2.2 Inverse Dynamics

The inverse dynamics of the system, is essentially the opposite of forward dynamics, wherein, the state variables [$\dot{\vec{q}}, \ddot{\vec{q}}$] and the link positions $\vec{q}$ of the system are known, and the end effector forces $\vec{F_{ee}}$ is under question. Such inverse dynamic relation can be expressed as follows,

$$\vec{F_{ee}} = \bar{M}(\vec{q})\ddot{\vec{q}}_i + \bar{C}(\vec{q}, \dot{\vec{q}})\dot{\vec{q}}_i + \bar{Q}(\vec{q}) \tag{2.4}$$

where, the terms $\bar{M}$, $\bar{C}$ and $\bar{Q}$ forms the rigid body dynamics of the manipulator under ideal situation which was discussed in previous section. However, the real time system is bound to higher order non-linearities due to various reasons such as missing dynamic behaviour, backlash and frictions. These non-linearities cannot be modelled precisely by rigid body formulation, making this model crude idealization of the real system. Thus, for a more general case in a non-ideal situation, the inverse dynamics of the manipulator can be represented by the following equation,

$$\vec{F_{ee}} = \bar{M}(\vec{q})\ddot{\vec{q}}_i + \bar{C}(\vec{q}, \dot{\vec{q}})\dot{\vec{q}}_i + \bar{Q}(\vec{q}) + \epsilon(\vec{q}, \dot{\vec{q}}, \ddot{\vec{q}}) \tag{2.5}$$

here, the term $\epsilon(\vec{q}, \dot{\vec{q}}, \ddot{\vec{q}})$ is added to the rigid body formulation. This term denotes the modelling errors arising due to unmodeled dynamics, ideal joint assumptions such as no friction and clearance, and some inaccuracies in the model parameters.

2.3 Control Scheme

As mentioned in the previous chapter, robot control consists of a combination of feedback and feedforward controller, see figure 2.2. The $Reference\ trajectory$ is the motion profile a user can give as input to the system. The $controller$ here is the feedback controller which essentially keeps the manipulator stable and minimises the tracking error up to a certain level. The feedforward controller is a function of motion profile that ensures precise tracking performance. Typically, the input to the $plant/manipulator$ for controlling it can be additively decomposed as,

$$\mathbf{T} = \mathbf{T}_{fb} + \mathbf{T}_{ff} \tag{2.6}$$

where, $\mathbf{T}_{fb}$ and $\mathbf{T}_{ff}$ are the feedback and feedforward component respectively. The feedback term can be established by a linear controller such as a PD controller. The feeforward component can be predicted by a known inverse dynamics model given the positions, velocities and accelerations of the joints of a manipulator.

The scope of this study is to develop this feedforward controller that can enable a precise tracking performance. As discussed in the previous chapter, the perfect feedforward controller is the inverse dynamics of the manipulator which was briefed in previous section. Most of the classical approaches for developing a feedforward control, use the inverse dynamics represented in equation 2.4 neglecting the unmodelled errors (the term $\epsilon(\vec{q}, \vec{\dot{q}}, \vec{\ddot{q}})$ in equation 2.5) in their computations. But often times, constructing a feedforward control neglecting these unmodeled errors could lead to poor tracking performance and unsafe operation in a human-centered environment [13]. Thus, with the machine learning approach, the study aims at approximating the inverse dynamics of the manipulator that takes into account these unmodelled errors as well.

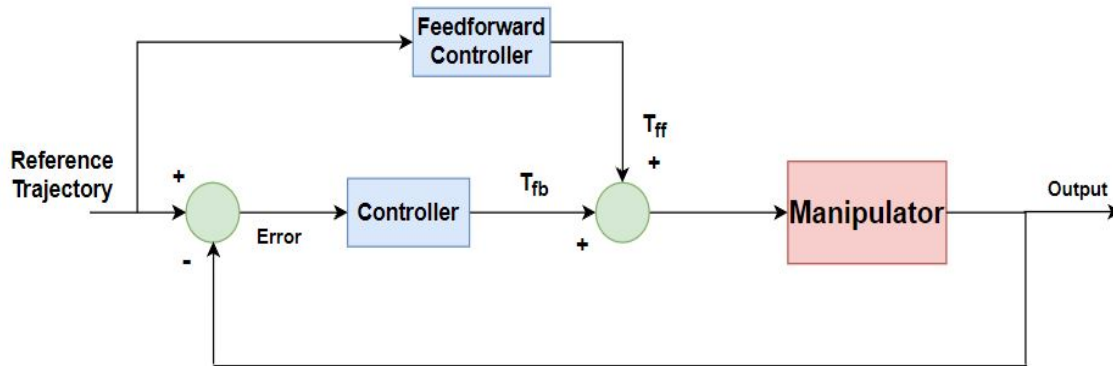


Figure 2.2: Control Schematic

2.4 Control Allocation

For this manipulator the feedback controller regulates the forces applied on to the end-effector ($\vec{F_{ee}}$). Since the manipulator under study is redundantly actuated the end-effector forces have to be translated to motor torques for actuating the system. This technique is called control allocation, wherein, the feedback controller regulates the end-effector forces, while the control allocation translates the end-effector forces to joint torques.

Due to the redundancy of the system, the number of actuator torques are more than the number of end effector forces. Consequently, there exists infinite sets of

joint torques that could apply same set of reaction forces on the end-effector. To obtain an optimal set of joint torques that could satisfy the equation 2.3, the Moore-Penrse pseudo inverse of the jacobian transfer function is used:

$$\vec{\tau} = pinv(A^T(\vec{q}))\vec{F_{ee}} + \lambda.null(A^T(\vec{q})) \tag{2.7}$$

Given the end effector forces that are to be applied any linear combination with a null space of $A^T(\vec{q})$ in equation 2.7 is a valid solution. Thus, a possible way to determine an optimal set of joint torques would be to optimize the torque distribution equation with respect a certain performance index which is a function of λ. Such performance indices can be 2-norm which minimizes the kinetic energy required for actuation or ∞-norm which minimizes the maximum driving torque, to prevent actuator saturation. In this study, the 2-norm was selected as the optimization criterion which is given as,

$$L_2 = \sqrt{(\tau_1^2 + \tau_2^2 + \tau_3^2)} \tag{2.8}$$

Applying this on equation 2.7, the 2-norm torque distribution is reached with λ =0.

2.5 Feedforward Control for the Manipulator

As discussed earlier, applying feedforward control could improve the accuracy of the tracking performance of a system. In this regard, many advance motion control concepts that apply feedforward control are well established in the field of control engineering. Genearlly, the feedforward control can be classified into an online feedforward control and an offline feedforward control. Online feedforward control is when the feedforward is updated during real-time such as Repetitive Control (RC) [18]. An offline feedforward control is when the feedforward that has to be updated is established offline before running it on real-time such as Iterative Learning Control (ILC) [19]. These control algorithms work well for SISO systems subjected to single reference path. However, these are not quite flexible when changing the reference path.

In this study, An offline feedforward control is developed with the use of machine learning model. For applying the feedforward control the inverse dynamcis is of the importance. Ultimately, the inverse dynamics model could be described as the mapping from joint positions, velocities and accelerations to the forces applied on the end effector,

$$(\vec{q}, \vec{\dot{q}}, \vec{\ddot{q}}) \mapsto \vec{F_{ee}} \tag{2.9}$$

It is this relationship that the machine learning algorithm should try to learn, in order to obtain an optimal feedforward control.

The established feedforward is then used for controlling the manipulator. Since, the learning model is trained with different trajectories, the algorithm should be flexible for different reference paths that the manipulator is subjected to. An online feedforward control is possible with machine learning techniques such as forecasting models, but, they are out of scope of this study.

2.6 Machine Learning for Inverse Dynamics Prediction

By collecting experimental data from the sensors for the positions, velocities, accelerations of all joints and the end-effector forces, over time, supervised machine learning techniques [20] can be employed for the learning the inverse dynamics of the manipulator. Basically, the supervised machine learning technique should be capable of learning the following mapping,

$$g : (\vec{q}, \dot{\vec{q}}, \ddot{\vec{q}}) \mapsto \vec{F}_{ee} \tag{2.10}$$

In machine learning terms, the positions, velocities and accelerations of the joints forms the inputs for training, while the end-effector forces forms the labels which the trained model attempts to predict.

One has to keep in mind that the joint positions, velocities and acceleration are the state variables of a dynamics system, and they evolve over time forming the intricacies of a dynamic system. Harnessing the time dependencies of these inputs in the learned model could improve the prediction accuracy [16]. The Neural networks are a good choice to start with for modelling the inverse dynamics of a system. But, timeseries data usually follow a pattern or trend which Recurrent neural networks can exploit better compared to vanilla feed forward networks (for more information on FNNs refer Appendix A). The FNNs usually only considers a single input to output relation, while RNNs also considers a complete input sequence to output relation. A single RNN cell is presented in figure 2.3. An input X_t at time step t passes through a hidden state h_{t-1} that contains information from the previous time step $t-1$. The cell returns an output Y_t and new hidden state h_t which are calculated using an activation function such as hyperbolic tangent ($Tanh$) as shown in the figure. Other activation function such as logistic sigmoid function can also be used. The equations describing the neurons of RNN are as follows,

$$\begin{aligned} a_t &= b + \mathbf{V}h_{t-1} + \mathbf{U}X_t \\ h_t &= tanh(a_t) \\ Y_t &= c + \mathbf{W}h_t \end{aligned} \tag{2.11}$$

In this case, the input X_t to the network will be the position, velocity and acceleration data $(q_t, \dot{q}_t, \ddot{q}_t)$ at a time step t. The output Y_t will be the end effector forces the network tries to predict (F_t) at time step t. The hidden state h_{t-1} contains the ouput predicted by the network at previous time step $t-1$. The **V**, **U** and **W** are the weight matrix corresponding to the hidden neuron, input neuron and the output neuron respectively. b and c are the biases.

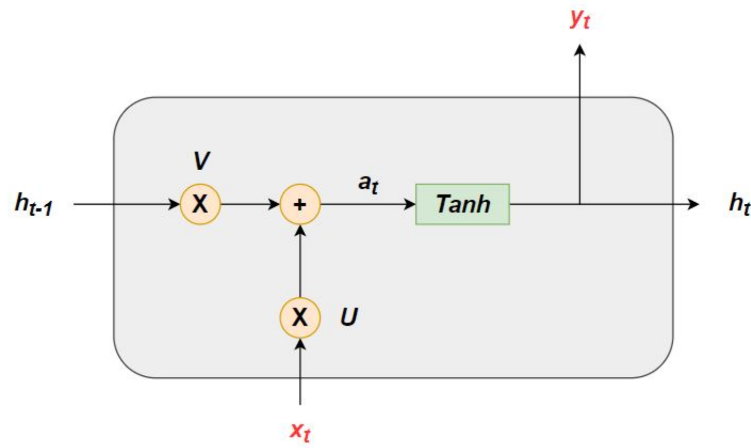


Figure 2.3: Recurrent Neural Network Cell

Basically, RNNs have sets of the RNN cells which are interconnected along a temporal sequence to form a deep neural network as shown in figure 2.4. The input to RNN cells from previous hidden states h_{t-1} as shown in this figure forms a recurrent or feedback connections which ensures that the network learns from previous time steps. These recurrent connections help the network to pass information back to themselves, thus, acting like a memory component. The number of hidden neurons can vary depending on the sequence data and the task the network is subjected to. Like any neural networks, the RNNs use forward and backward propagation for the learning process. The backpropagation in RNN stems from the chain rule, where weights between the neurons are updated through the gradient descent algorithm [21]. In RNNs, back propagation is also called as back propagation through time (refer Appendix B), as the weights between current time steps and previous time steps are learned.

In this study, three deep learning techniques are proposed and evaluated. First, the LSTM network which is a special case of RNN, that showed promising results in

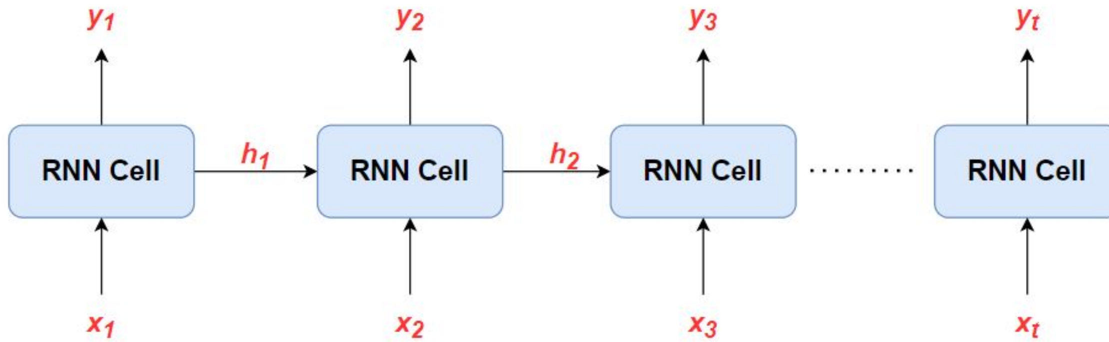


Figure 2.4: Recurrent Neural Network Structure

[16] is considered. The key advantage of LSTM is that it can comprehend long term dependencies better compared to the simple RNN. The drawbacks of LSTM is that the number of parameters to be trained increases as the number of input features increases. Thus, for huge amount of dataset the training time would be large. To mitigate this, the Convolutional-LSTM network (CNN-LSTM) [22] is adopted as the second technique. CNNs are good at exploiting spatial features from time domain data, while LSTMs are good at exploiting temporal patterns in the time domain data. Therefore a hybrid model combining these two is proposed. With the hybrid model, it is aimed at reducing the number of features by extracting higher level features with CNN, and using these features to train an LSTM network. This could significantly reduce the training time of the network. As a third technique, Temporal convolution network (TCN) [23] is proposed. TCNs are special type of CNNs that are designed to exploit time series data. Recently, TCNs have gained popularity among the deep learning community because of the advantages it imposes over LSTM networks [23]. One of the famous examples in which TCNs are used includes google's speech recognition system [3]. So, in this study TCNs are also put to use for learning the inverse dynamics of the manipulator. More background of these three proposed networks are detailed out in the subsequent chapter.

Chapter 3

Machine learning Driven Control

This chapter gives the overview of the three proposed machine learning techniques considered during this study.

3.1 Long short term memory networks (LSTM)

The LSTM are a popular recurrent neural networks for modelling long time series. LSTMs were designed by adding special gating mechanism to a classical RNNs in order to avoid the vanishing gradient problem during back propagation through time. A single LSTM unit is shown in figure 3.1. The LSTM unit consists of a cell state C_t which is the horizontal line running on top of the LSTM unit, it undergoes minor linear interactions and the information running through this doesn't change much. The LSTM unit also consists of gates which regulates the flow of information into and out of the cell state, and they consist of multiple activation layers interacting with each other. First, the f is the forget gate which determines how much the previous state C_{t-1} affects the current cell state. This is subjected to a sigmoid activation layer that output a value between 0 and 1, 0 indicates 'completely get rid of this information', while 1 indicates 'completely keep this information'. The second is the input gate i and g which regulates the new information that flows into the cell state. The gate i is subjected to a sigmoid activation layer while gate g is subjected to a $tanh$ activation layer, both of these are combined to create an update to the cell state. Finally, an output gate o combines the information flow from forget gate and the input gates and generates the output of the cell indicated by h_t or y_t. The

principle behind LSTM can be expressed by the following,

$$
\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t]^T + b_f) \\
i_t &= \sigma(W_i[h_{t-1}, x_t]^T + b_i) \\
g_t &= \tanh(W_c[h_{t-1}, x_t]^T + b_c) \\
C_t &= f_t \odot C_{t-1} + i_t \odot g_t \\
o_t &= \sigma(W_o[h_{t-1}, x_t]^T + b_o) \\
h_t, y_t &= o_t \odot \tanh(C_t)
\end{aligned}
\tag{3.1}
$$

The $\sigma()$ represents the sigmoiod activation function, while the $\tanh$ represents the hyperbolic tangent activation function. The $\odot$ is the Hadamard or element wise product. The W_f, W_i, W_c and W_o represents the weights of the gates $[f, i, g, o]$ respectively. similarly, the b_f, b_i, b_c and b_o represents the biases added to the gates $[f, i, g, o]$ respectively.

By combining multiple LSTM cells over a temporal sequence a LSTM network is created like in the case of simple RNN discussed in previous chapter. This structure enables LSTM network to remember information over long time intervals in the past without running into vanishing gradient problem [24]. An appropriate network architecture can be designed using these LSTM units, such that the network learns the relationship shown in equation 2.10. The architectures designed for this study are presented in later chapters.

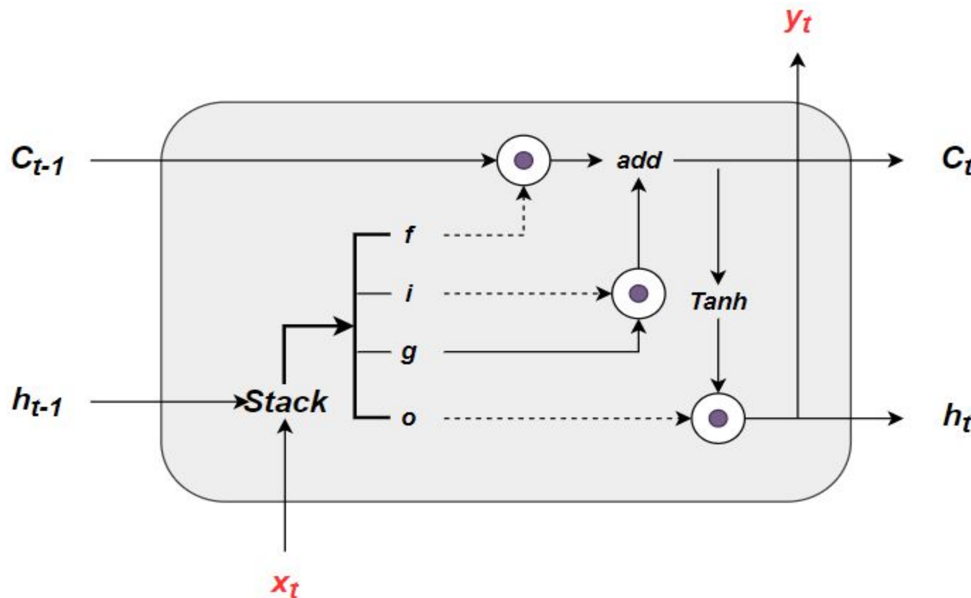


Figure 3.1: LSTM Cell Architecture

3.2 Convolutional LSTM Network (CNN-LSTM)

LSTMs are quite accurate at modelling long time sequences. However, as discussed earlier, in the presence of a large dataset and a large number of features to train, the number of parameters to train increases proportionally. Thus, often LSTM networks exhibits slow convergence and take more time to train. One of the ways to mitigate this would be to reduce the number of features for training. Approaches like principle component analysis (PCA) for model reduction is possible, but, for a time series data, applying such a method could be much more complex. Thus in this study, a convolution neural network(CNN) based LSTM is designed.

The CNN is a type of feedforward neural network which is most widely used in the field of computer vision. CNNs have the ability to extract higher level features from a sample efficiently [22]. For time series data, 1D CNNs are predominantly used to extract higher level features. A simple convolution operation in 1D CNN is shown in figure 3.2. In this case, the input features can be position, velocity or acceleration data collected over a certain timesteps. 1D CNN employs multiple filters to move along the timeline to extract features. The filters also called as kernels in CNN are basically windows of weights that the network tries to learn. For example a filter of size 4 is considered in the figure 3.2, which consists of 4 weights that takes in 4 inputs at a time and performs a convolution operation to generate an output. A simple convolution operation can be represented by the equation,

$$o[t] = (x * w)[t] \tag{3.2}$$

Similarly, this window of weights slides along the timeline to generate new outputs. These outputs that the CNN extracts are called as feature maps. However, due to the kernel size, the 1D CNN outputs a sequence of data smaller than the input sequence by default. For example, in this case when performing convolution operation with kernel size 4 along a sequence of say 15 timesteps, the output feature map will be of size 12. This leads to a loss in temporal sequence which is not desirable. Thus a zero-padding is usually added to the input data before convolution operation is performed. zero-padding is to basically add zeros at the start and the end of the data depending on the sequence length to obtain the same sequence length in the output feature map.

Several 1D CNN layers can be stacked one after the other to form a deep neural network which can extract higher level features. The idea here is to train the CNN model with position, velocity and acceleration data to extract these higher level features. These extracted features can then be used to train a LSTM network that was

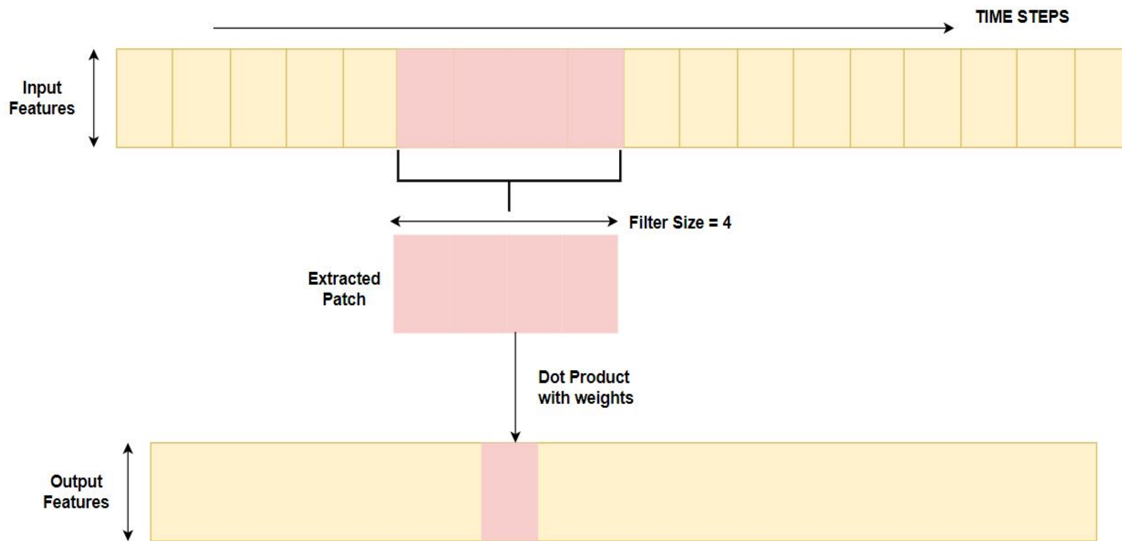


Figure 3.2: 1D Convolution operation

shown in previous section, which is good at exploiting temporal sequences. A typical CNN-LSTM network is presented in figure 3.3.

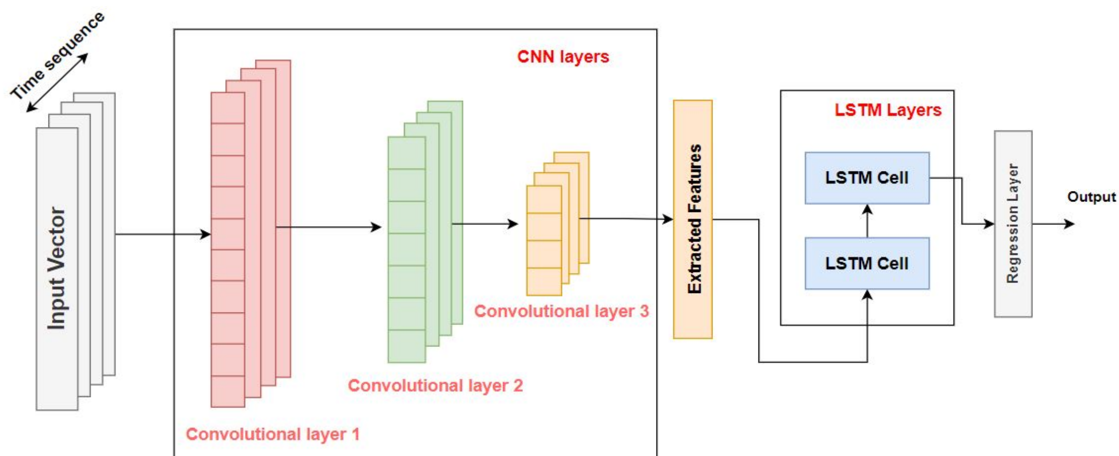


Figure 3.3: CNN-LSTM Network

here, the input vector is the position, velocity and acceleration data collected over time. This is passed into CNN layer, the number of convolutional layers can be set according to user's wish depending on the feature map to extract. The extracted features are then given as input to LSTM layer which can also contain arbitrary number of LSTM cells according to the user. This LSTM layer outputs the desired predictions, which in this case are end effector forces.

3.3 Temporal Convolutional neural network (TCN)

In deep learning community, most of the sequence modelling tasks are pervasively associated with recurrent neural networks such as LSTM which is discussed in previous sections. Recent study by S. Bai [23] has shown that convolutional neural networks with some simple adaptation can become a powerful tool for sequence modelling tasks. A Temporal Convolutional network, or simply TCN, is a variation of Convolutional neural networks for sequence modelling tasks, by combining aspects of CNN and RNN architectures.

The 1D CNNs discussed in the previous sections employ a moving window of arbitrary size along a temporal sequence. However, 1D Convolutions consider future values as well into their computation, for example, a window of size 3 would consider one past value, the present value and one future value for a single computation, see figure 3.4. In TCNs a causal filter is used rather. A causal filter essentially ensures that no future information is leaked into the past while performing convolution operations. This is achieved by zero-padding the data asymmetrically, i.e, instead of adding zeros in the beginning and end of the dataset like in a standard CNNs, zeros are added only in the beginning. This enables the network to look back two timesteps in the past, when considering a window size of 3 as shown in the same figure 3.4.

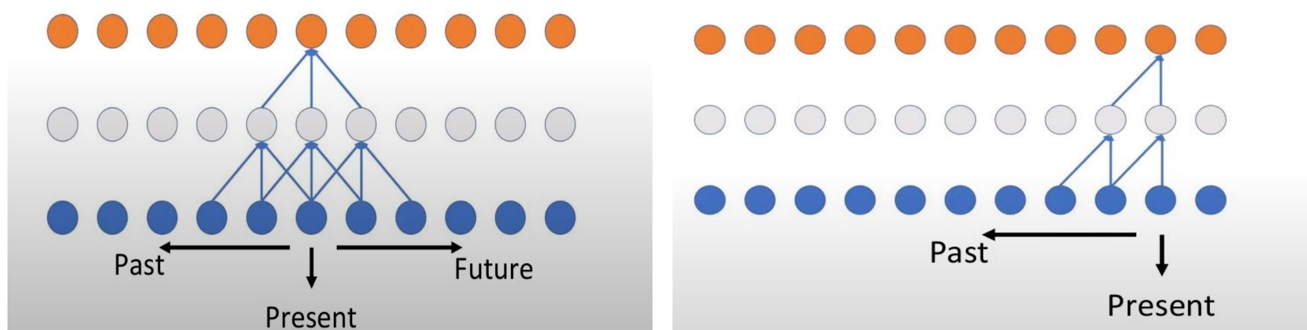


Figure 3.4: Standard Convolution (on the left) and Causal convolution (on the right)

Now, for long time sequence modelling, there is a need to look back in time through larger number of timesteps. The simple causal convolutions have the disadvantage to look back through a limited number of timesteps. One can increase the window size to over come this, but, it would increase the number of parameters to train exponentially and wouldn't be practical. To circumvent this, the TCN architecture employs dilated convolutions. The principle behind dilated convolution is to introduce a fixed step between every two adjacent filter taps as shown in figure 3.5.

For an input sequence $x \in \mathbb{R}^t$ and a kernel size k, the dilated convolution operation H on element x can be defined as [25],

$$H(x) = x *_d k(x) \tag{3.3}$$

where, d is the dilation factor given by $d = 2^v$ and v is the level of the network. By dilating a filter size of k with a dilation factor of d, the output at time t can be dependent on input values upto $d(k-1)$ timesteps back in time. These forms the basic building blocks of a Temporal Convolutional network. By stacking, these dilated network one after the other, the previous time steps the network can see increases, see figure 3.5.

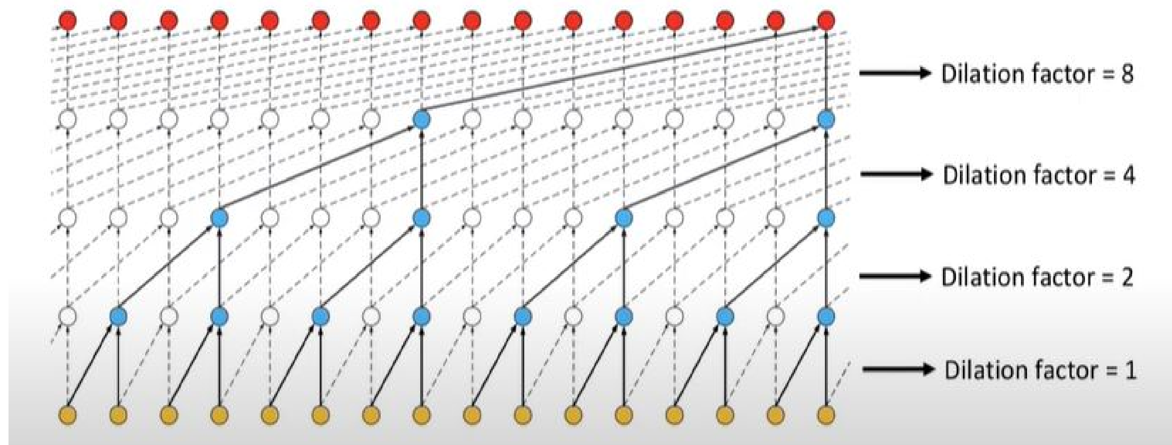


Figure 3.5: Temporal convolutional network Architecture [3]

The time-frame the network can look back in time is called as the ”receptive field”. With exponentially growing receptive field, one can observe all the elements within the receptive field of the entire network. With this dilation scheme, the receptive field of the TCN can be calculated as follows,

$$R = 2^l(k-1) \tag{3.4}$$

where, R is the receptive field of the network, l is the number of layers, and k is the convolution kernel or window size. In this figure the receptive field is 16 timesteps. The TCNs have different gradient flow compared to the recurrent neural networks, they do not use back propagation through time, this makes it possible to train the network in parallel, thus, it ensures optimal GPU usage and much more faster training compared to RNNs such as LSTM. These networks do not experience any vanishing gradient problem as well [26].

Chapter 4

Modelling and Experimental Setup

This chapter provides the overview of the Simulink model of the 2DOF redundantly actuated manipulator that is used for testing in simulation environment. Later, the existing test setup is briefed. Data gathering and data preprocessing techniques for training the machine learning algorithms are detailed out for both the Simulink model and for the test setup.

4.1 Simulink Model of the Manipulator

The model considered in this study was originally developed by Berendsen [5] based on minimal coordinates formulation described in [17]. The goal behind using a simulation model in this study is that, the model serves as a simplification of the real time system for testing the proposed algorithms.

Based on the equation of motion shown in equation 2.2, a Simulink model was developed and a PID feedback controller was designed to control the end effector forces in the x and y plane. The block diagram of the Simulink model is shown in figure 4.1. Here, the block $plant$ contains the dynamic equation of the manipulator. Reference trajectories for manipulating the end effector position along x and y plane are given as input to the system, based on the tracking error, the PID controller provides end effector forces that tries to improve the tracking performance. The control forces are then transformed to motor torques through control allocation, a second norm redundancy resolution is considered for this purpose (equation 2.8). A motor constant is defined in between, which converts the joint torques into motor current and saturation block is added to limit the maximum current passing into the plant. As the output, shoulder angles of each of the shoulder hinges are obtained from the encoders, this is then transformed to end effector positions by means of forward kinematics (refer Appendix C).

To simplify the feedback controller design, the MIMO system considered here are

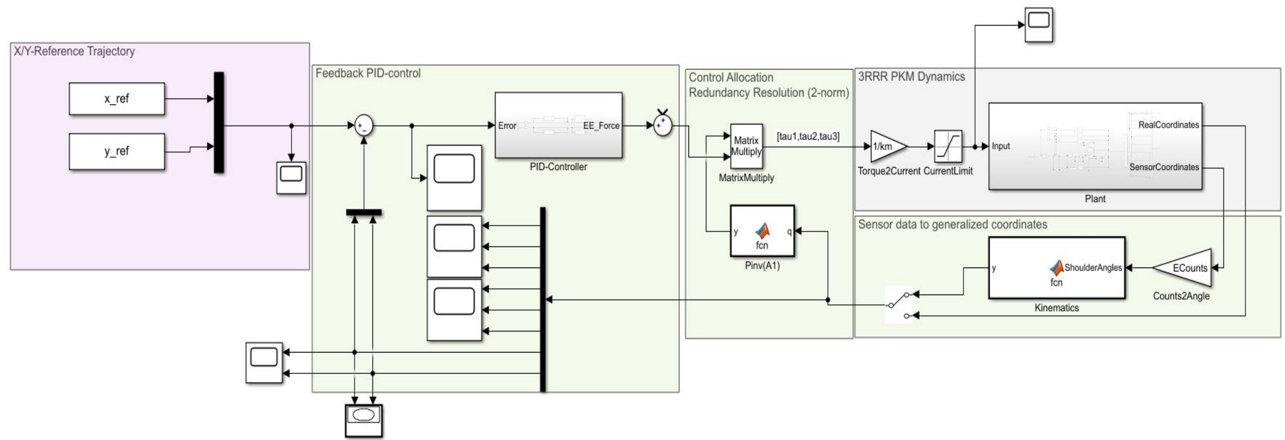


Figure 4.1: Simulink Block Diagram

treated as 2 SISO systems through model decomposition. It was verified that this concept is applicable for this manipulator by Berendsen [5]. So, two PID controllers for controlling these two SISO system was developed based on frequency domain loop shaping and the PID controller was tuned depending on a desired cross over frequency. For this Simulink model, the desired cross over frequency was set at 2.66 Hz.

4.2 Experimental setup

This section briefs about the existing test setup of the 2DOF redundantly actuated manipulator. The important design parameters of the manipulator is highlighted in table 4.1. The specifications of the motor components used for controlling the manipulator is shown in table 4.2. The test setup of the manipulator is shown in figure 4.2. The setup contains various electrical components that are responsible for signal processing and providing actuation power for driving the manipulator. A Maxon brushless DC motor is rooted at each of the three shoulder hinges of the manipulator, it provides the joint torques for these shoulder joints. The motor comes along with a built in encoders which have 512 lines per rotation. From these lines, a quadrature signal is evaluated such that the final encoder resolution becomes 2048 $counts/rev$.

Parameters	Values	Units
Upper arm length (l_{up})	0.3086	mm
Lower arm length (l_{low})	0.3086	mm

Table 4.1: Design Parameters

Parameters	Values	Units
Nominal Torque	0.128	Nm
Torque constant	36.9	Nm/A
Encoder resolution	2048	counts/rev

Table 4.2: Maxon motor and encoder specifications

The actuators are current controlled by ”Maxon Escon 50/5” motor controllers which requires a power supply of 24V. For manipulating the robot, Simulink Real-time running at sample rate of 1KHz is used on a target PC. The interface between the motor controllers, encoders and the target PC is provided by two NI PCI-6221 cards. The motor controller gives current to the motors based on an analog voltage signal generated by these cards. The encoder counts are also kept in track by the NI PCI-6221 cards.

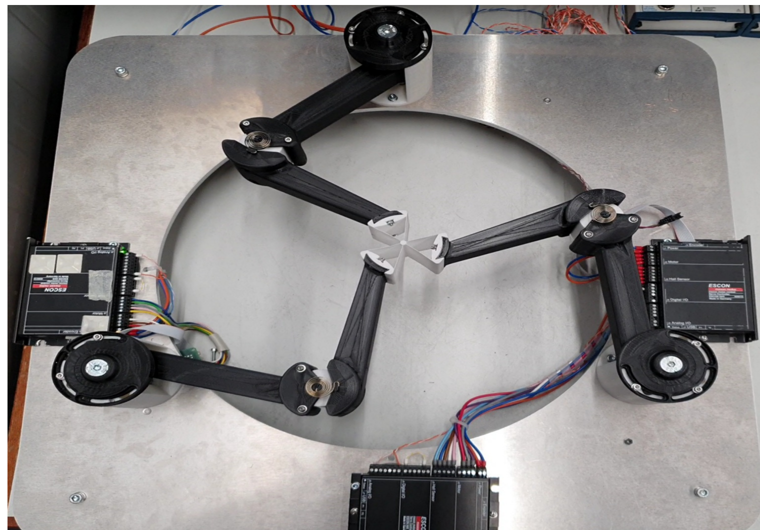


Figure 4.2: Redundant Manipulator test setup

The natural frequency of the system is found to be at approximately 11 rad/s (1.7Hz). For the test setup 2 PID controllers with a cross over frequency of 2Hz (above the natural frequency) are used for controlling the end effector in x and y direction. In addition, a notch filter was added to reduce the effect of second parasitic eigen frequency. The system is found to be marginally stable.

For the test setup similar to simulation environment, the control allocation method is also used to transform the end effector forces into joint torques.

4.3 Data Acquisition

This sections gives an overview of the data collected from the Simulink model and the real time system for training the machine learning algorithms. Initially, the dataset

are gathered with 2 main aims. The first one being, covering sufficiently a rich set of workspace area. The second one is to cover maximum range of possible performance of the manipulator, i.e, to cover maximum and minimum possible speeds the end effector can move without the manipulator becoming unstable. These two ensures that the acquired dataset captures a wide range of varying dynamics of the manipulator.

4.3.1 Data Acquisition from the Simulation Environment

The Simulation model discussed in previous section is considered for acquiring dataset. Although, the Simulink model was built with ideal conditions, it serves as a good starting point for testing the algorithms.

Different reference paths (x_{ref} and y_{ref} in the block diagram 4.1) are given as input to the model shown in figure 4.1. The paths that the system traces are collected for training the machine learning algorithm. 44 trajectories spanning most of the workspace is gathered from the model. Each of these trajectories are obtained by running the simulation with different reference paths for 15 seconds at a sampling rate of 1Khz. These trajectories consists of,

- Circular trajectories ranging from an angular frequency of 0.066Hz to 1.0Hz, with different magnitudes.
- Random trajectories within the workspace limit.
- Random trajectories and circular trajectories with different initial positions of the end effector.

The circular trajectory in this study is generated with a skew-sine based polynomial as the reference path, which is given by the following equations,

$$\begin{aligned} \theta_{ref}(t) &= \frac{\theta_m t}{t_m} - \frac{\theta_m sin(\frac{2\pi t}{t_m})}{2\pi} \\ x_{ref}(t) &= Rcos(\theta_0 \pm f \times \theta_{ref}(t)) \\ y_{ref}(t) &= Rsin(\theta_0 \pm f \times \theta_{ref}(t)) \end{aligned} \tag{4.1}$$

where, the θ_m is the total angle of rotation, $\theta_{ref}(t)$ is the angle of rotation at current time t. The $x_{ref}(t)$ and $y_{ref}(t)$ are the reference path along x and y coordinates at current time t. The θ_0 is the initial angular position of the end effector, varying this term leads to different initial positions. The R denotes the radius of the circle, varying this would lead to different magnitudes of the circular trajectory. Since the workspace area of the manipulator is in the range of 0m to 0.05m [5], the maximum range of

magnitudes of these circular trajectories are also within the range of 0m to 0.05m. The f here denotes the number of rotations the trajectory is subjected to within the total time of t_m. Varying this term varies the angular frequency of the trajectory. The $(\pm)$ in the equation denotes the clockwise and anti-clockwise rotations.

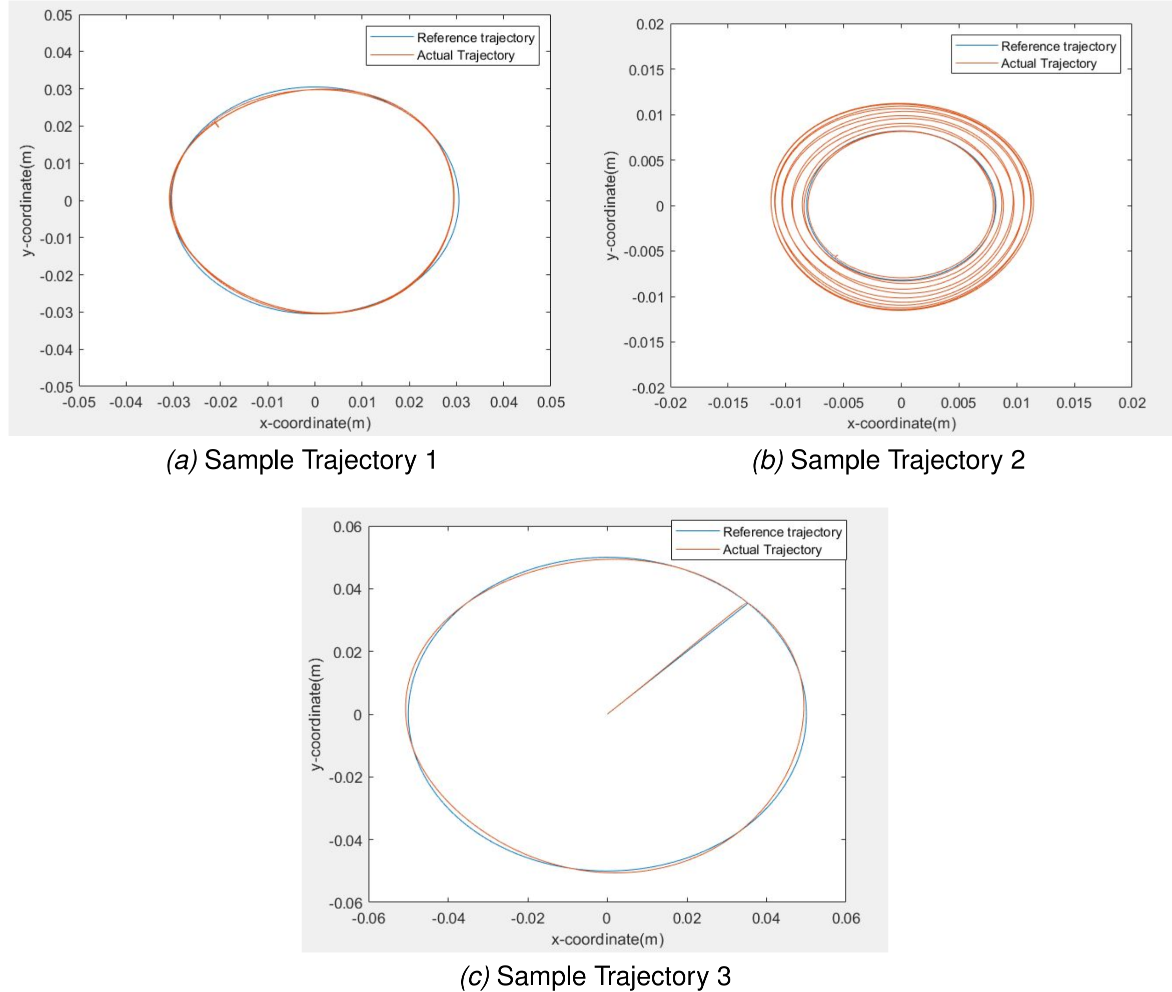
Out of the 44 trajectories that were collected, 30 trajectories are circular trajectories, with different combinations of angular rotation, magnitude, initial positions and the direction of rotation. The number of rotations f, the trajectory is subjected to ranges from 1 - 15 rotations in the span of 15 seconds, thus this corresponds to a angular frequency with a range of [0.66 1]Hz. Similarly the magnitudes or the radius of the trajectories spans from [0.007 0.05]m. The intial position of the end effector (x_{ee} and y_{ee}) can be set at the beginning according to users wish. Thereby, the initial angle θ_0 can be found by taking the inverse of the tangent between the y_{ee} and x_{ee} $(Tan^{-1}(\frac{y_{ee}}{x_{ee}}))$.

Few of the circular trajectories are displayed in figure 4.3. The sample trajectory 1 (figure 4.3a), is obtained using a circular reference path with the number of rotations f = 3 (0.2Hz), radius = 0.03m, initial angle $\theta_0 = \frac{3}{4}\pi$ and running in anti-clockwise direction. Similarly, the sample trajectory 2 (figure 4.3b) is obtained using a circular reference path with the number of rotations f = 11 (0.733Hz), radius = 0.0082m ,initial angle $\theta_0 = \frac{\pi}{4}$ and running in clockwise direction. It has to be noted that the model doesn't trace the reference path exactly. This is due to the fact that, the feedback controller alone cannot achieve precise tracking performance, especially at higher speeds (like in the case of sample trajectory 2). However, the trajectory that the model traces still contains the information about the dynamics of the system. Hence, the actual trajectory obtained is used for training the machine learning algorithm. The set of trajectories also include straight line motion profile followed by circular path as shown in figure 4.3c. This type of trajectory is gathered to replicate the original system, as the real time manipulator has an initial position set at origin of the workspace always. The straight line trajectory is again generated based on skew-sine polynomial, given by,

$$\begin{aligned} x_{ref}(t) &= \frac{h_x t}{t_m} - \frac{h_x sin\frac{2\pi t}{t_m}}{2\pi} \\ y_{ref}(t) &= \frac{h_y t}{t_m} - \frac{h_y sin\frac{2\pi t}{t_m}}{2\pi} \end{aligned} \tag{4.2}$$

where, h_x and h_y are total stroke or final setpoint to reach along x and y direction respectively. The trajectory shown in figure 4.3c contains a straight line motion path running for 2.5 seconds with the set-point to reach as h_x = 0.035m and h_y = 0.035m. Once it reached the setpoint, the system stays idle for 0.5 seconds, and then it

follows a circular trajectory for 12 seconds with number of rotations f = 1 (0.0833Hz), radius = 0.05m, initial angle $\theta_0 = \frac{\pi}{4}$, and running in clockwise direction. Thus a total trajectory time of 15 seconds is restored.



(a) Sample Trajectory 1

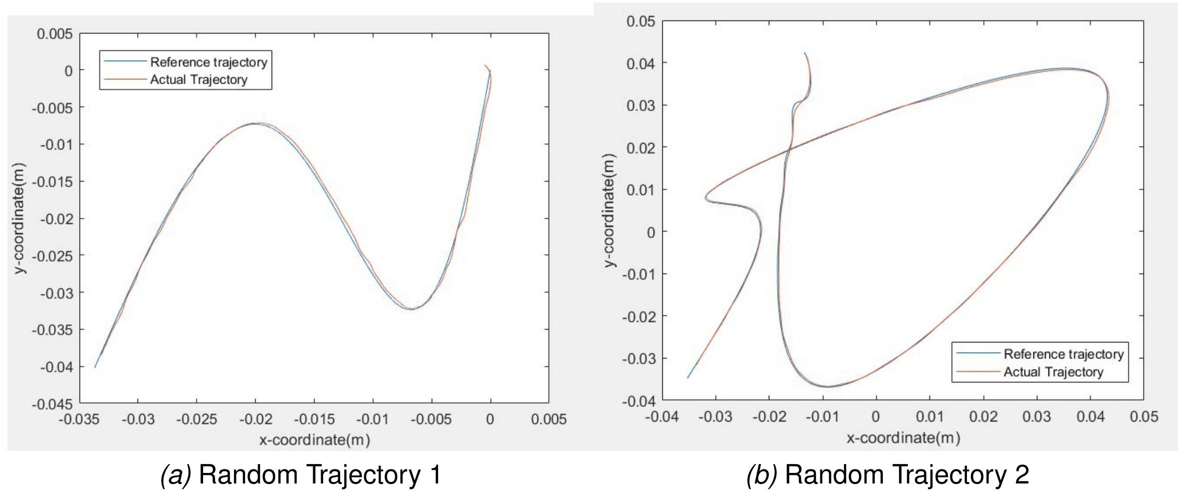
(b) Sample Trajectory 2

(c) Sample Trajectory 3

Figure 4.3: Circular Trajectories

The rest of the 14 trajectories are random trajectories that are generated using cubic spline function in MATLAB [27]. This function takes in the user defined points (in cartesian coordinates) the end effector has to pass through, and creates an interpolated periodic cubic spline curve. These interpolated data is collected over 15 seconds and given as reference trajectory for the model. Few of the random trajectories are shown in figure 4.4. Here, the number of points the end effector has to pass through and the actual points can be arbitrarily set by a user within the workspace limit including the initial positions. The random trajectory 1 shown in figure 4.8a passes through 4 user defined data points which are $[(0,0),\ (-0.006,-0.0321),\ (-0.02,-0.007)\ ,\ (-0.0334,-0.04)]$. The random trajectory 2 shown in figure 4.8b passes through 5 user defined data points which are

$[(-0.01275, 0.04292), (-0.035, -0.009), (0.038, 0.034), (0.0065, -0.0317), (-0.025, -0.0335)]$. Like wise other random trajectories were gathered with different sets of data points within the workspace limit.



(a) Random Trajectory 1

(b) Random Trajectory 2

Figure 4.4: Random Trajectories

4.3.2 Data Pre-processing for simulation model

As said earlier, the output of the simulation model are the three shoulder angles obtained from motor encoders. In the real time manipulator, these encoders output contains quantization noises and higher frequency noises. To replicate this real world scenario in simulation, the gathered shoulder angles from simulation model is contaminated with quantization noises and white noises manually. The quantization noises are added using the encoder resolution mentioned in table 4.2. As discussed earlier, the movement of all the joints in this manipulator can be described by a set of 21 generalized coordinates, shown in equation 2.1. With the use of the three shoulder angles, rest of the 18 coordinates can be found using the forward kinematics. These forms the positions of the joints in the manipulator. One of the trajectories after adding these quantization noises and white noises is shown in figure 4.5. The end-effector forces that are given as input to the system also contain high frequency noises in the real time system. Therefore, white noises were added for the end effector forces as well.

The velocities and accelerations of the joints in the manipulator contains useful information about the dynamics of the system. The velocities of each of the joints can be obtained by taking the first order time derivative of the position data in MATLAB. Similarly, the accelerations of joints can be obtained by taking the second order time derivative of the position data. However, taking time derivative of

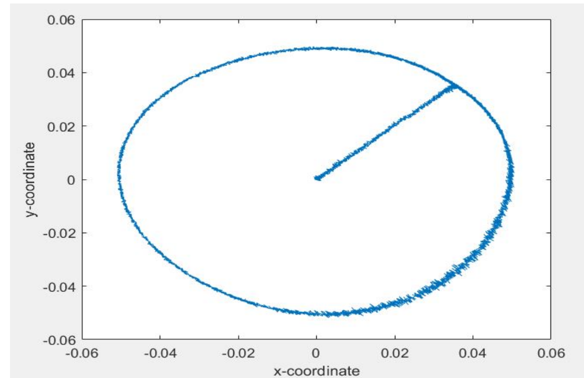


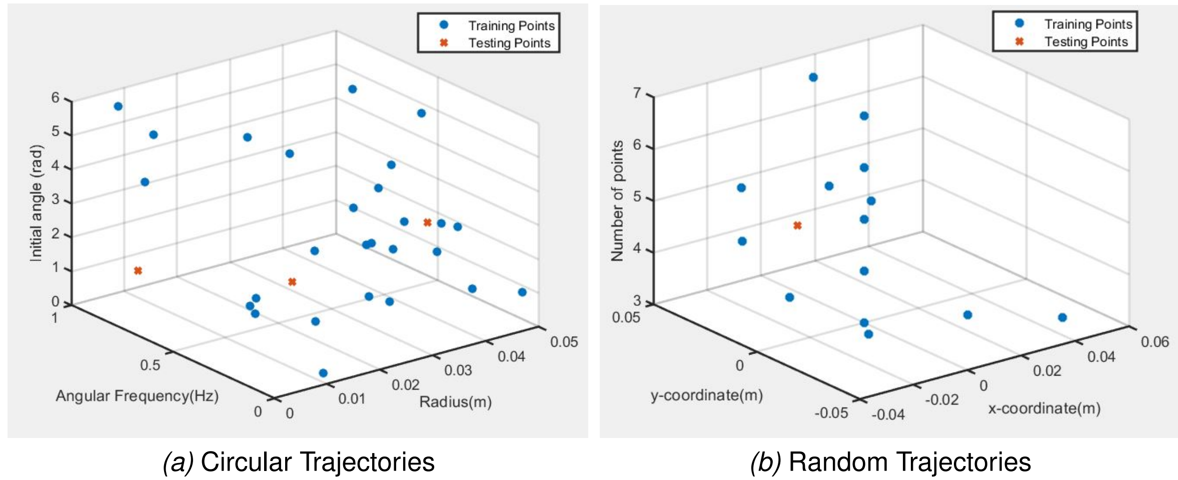
Figure 4.5: Sample trajectory with quantization and white noises

the position data which is contaminated with noises would lead to inaccurate estimations of the velocities and accelerations. Thus, before taking the derivatives, a 4th order low pass Butterworth filter is used for minimizing these noises. The cut-off frequency of the filter is set to 1.8 Hz, which is slightly above the natural frequency of the manipulator. This ensures that the dataset is free of high frequency noise while essential dynamics are not lost when filtering. The filter is designed using the MATLABs Butterworth function [28]. The filtering process is done using MATLABs filtfilt() command [29] which ensures that there is no phase lag while filtering. The same filtering process is used on end effector forces as well.

The resulting positions, velocities and accelerations for all the joints in the manipulator for each of these trajectories are gathered. The corresponding end effector forces are gathered. The combination of positions, velocities and acceleration data forms a vector of size 63, and this is used as input features for training. While, the corresponding end effector forces which are of vector size 2, forms the labels for training. Each of the trajectories span 15 secs, and the data are collected at a rate of 1Khz, which means, a single trajectory can be represented by a tensors of size $(1 \times 15000 \times 63)$ indput data and $(1 \times 15000 \times 2)$ output data.

The 44 trajectories gathered are split into training dataset and testing dataset. Nominally about 10 percent of the dataset is used for testing in machine learning algorithms. In this case, 40 trajectories are used for training and 4 trajectories are used for testing. The testing dataset contains three circular trajectories which are in between training points, i.e, one with a radius = 0.01m, angular frequency = 0.93Hz, the second one with radius = 0.043m and angular frequency = 0.367Hz, and the third one with radius = 0.02761 and angular frequency = 0.632Hz. The other trajectory is a random trajectory that passes through arbitrary data points. These testing trajectories are not leaked into training dataset, in order to evaluate the networks performance on an unseen dataset. The overview of the training and testing points of all the collected trajectories are presented in figure 4.6. The figure 4.6a provides

the training and testing data-points of the 30 circular trajectories, the three parameters namely, radius, angular frequency and initial angle that defines the circular trajectories are used for plotting this 3D graph. The figure 4.6b provides the training and testing data-points of the 14 random curve spline trajectories, the parameters used for plotting this graph are the initial end effector position in xy plane and the number of Cartesian points the spline trajectory is subjected to pass through.



(a) Circular Trajectories *(b)* Random Trajectories

Figure 4.6: Training and Testing Data-points (Simulation Environment)

Before starting with the training process, the data has to be normalized in order to match the consistency of the learning model. In this study, the normalization is done between the range of [0 1], the following is the normalization equation,

$$x_n = \frac{x_r - x_{min}}{x_{max} - x_{min}} \tag{4.3}$$

the normalization is applied only on the training dataset, and the resulting coefficients are used for normalizing the testing dataset separately. This is done in order to prevent any leakage of information between training and testing dataset. The normalization on input features, i.e, positions, velocities and accelerations are done separately and the normalization on labels, i.e end effector forces are obtained separately. After the prediction, original values are restored for comparison, by taking inverse of the normalization.

4.3.3 Data Acquisition from the Real-time Environment

Similar to the Simulation environment, from the real time system it is attempted to gather data that can cover rich set of dynamics of the manipulator. Upon few trials on the test setup, it is found out that the feedback controller couldn't handle very high

speed applications. It was first attempted to run the system with a circular reference path spanning 15 seconds similar to the simulation environment. It was found out that the the tracking performance of the manipulator was too poor within this time span, and testing with higher angular frequencies rendered the system unstable. Thus, the time span of trajectories are increased to 100 seconds, and the angular frequency ranges are also limited.

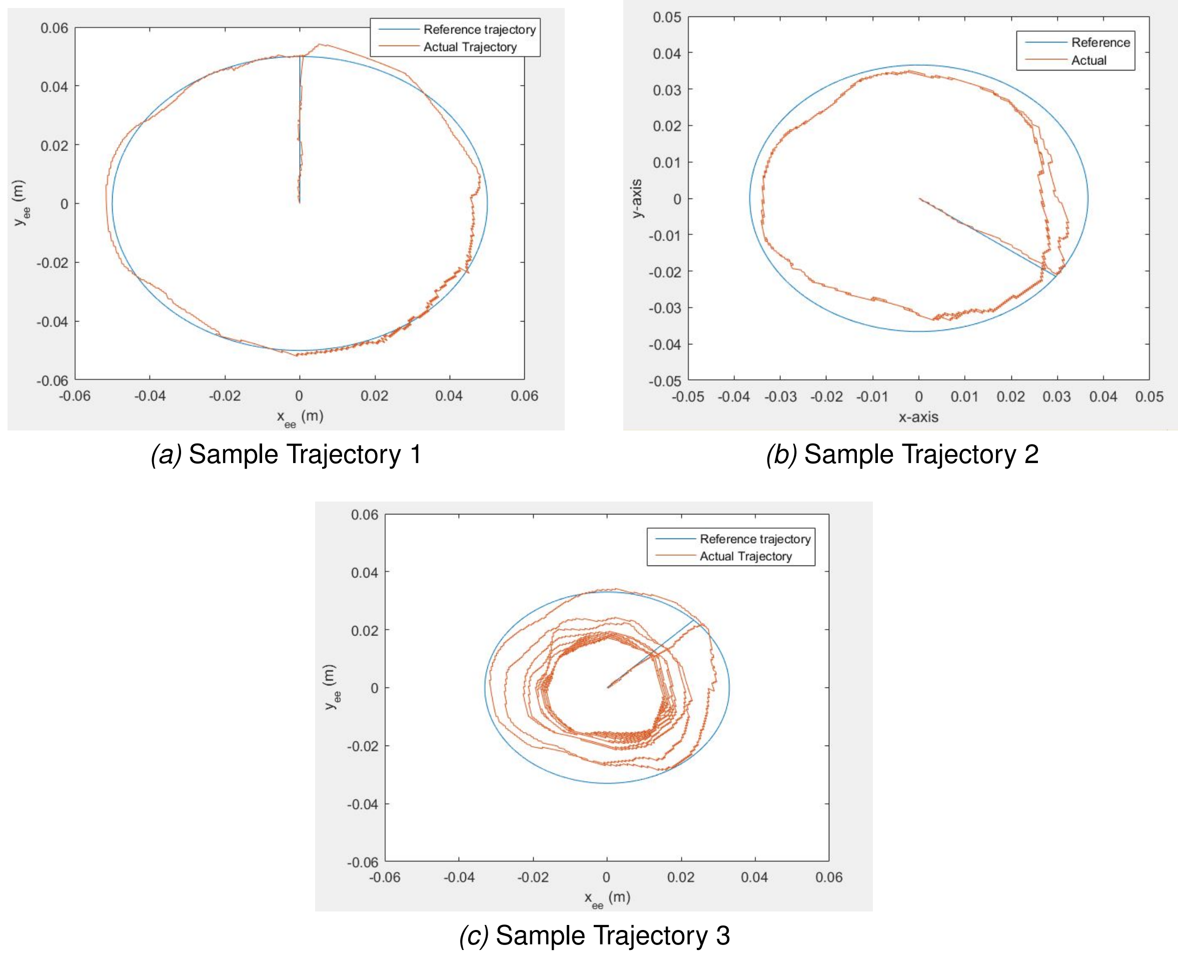
In order to cover a rich set of workspace and dynamics of the manipulator, different reference paths are given as input to the manipulator. Paths that the manipulator traced are gathered for training. 70 such trajectories spanning most of the workspace is gathered. Trajectories are obtained by running the test setup for 100 seconds at a sampling rate of 1KHz. These trajectories consists of,

- Circular trajectories ranging from a angular frequency of 0.0133Hz to 0.3Hz, with different magnitudes.
- Random trajectories within the workspace limit.
- Random trajectories and circular trajectories with different initial positions of the end effector.

The circular trajectory is again generated using a skew-sine based polynomial as reference path, similar to the simulation environment (equation 4.1). A straight line motion profile is also included in these circular trajectories which follows from equation 4.2. For the real time system, the initial end effector positions cannot be set manually like in the case of simulation environment. Instead, the manipulator is actuated with a straight line motion profile to reach the desired initial positions. Thus, most of the circular trajectories collected here are actuated for 21 seconds with a straight line motion profile, the system stays idle at this position for 1 second and then it follows a circular trajectory for 75 seconds, it stays idle at the final position for 3 seconds. Thus, the overall time span of the trajectory is 100 seconds.

Out of 70 trajectories, 45 trajectories are circular paths with different combinations of angular rotation, magnitudes, initial positions and different direction of rotation. The number of rotations f, these trajectories are subjected to a range from 1 - 22 rotations in the span of 75 seconds, thus this corresponds to an angular frequency with a range of [0.0133 0.3]Hz. The radius or magnitudes of these trajectories ranges between [0.007 0.05]m which is within the workspace limit.

Few of the circular trajectories collected are displayed in figure 4.7. The sample trajectory 1 (figure 4.7a) is obtained using a straight line motion profile traversing to 0.05m in y coordinate followed by a circular motion profile with number of rotations f = 1 (0.0133Hz), radius = 0.05m , initial angle $\theta_0 = \frac{\pi}{2}$ and it makes a clockwise rota-



(a) Sample Trajectory 1

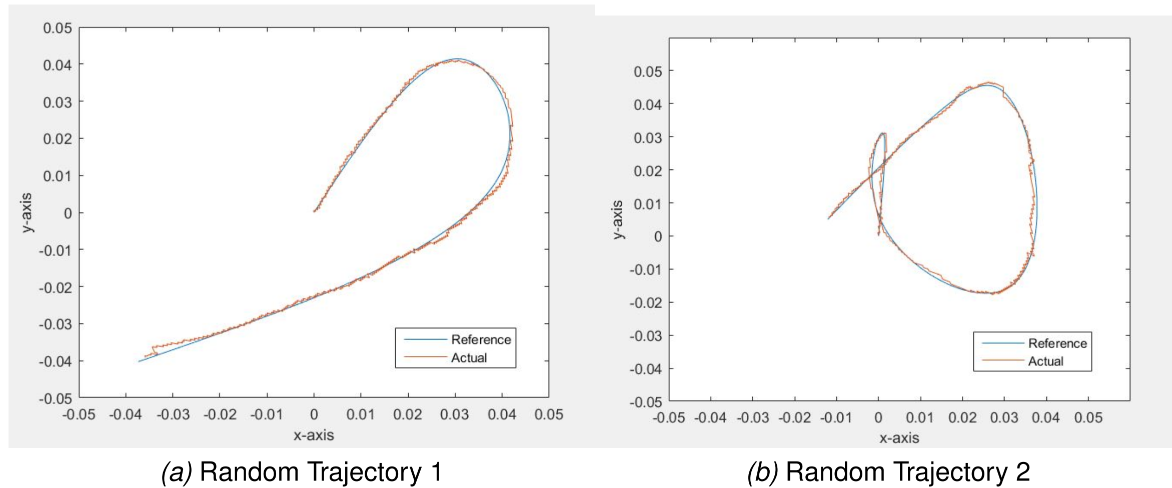
(b) Sample Trajectory 2

(c) Sample Trajectory 3

Figure 4.7: Circular Trajectories

tion. Similarly, the sample trajectory 2 (figure 4.7b) follows a straight line motion profile traversing to (0.0219m,-0.0219m) along x and y coordinates respectively. This is followed by a circular trajectory with number of rotations f = 2 (0.0266Hz), radius = 0.031m, initial angle $\theta_0 = -\frac{\pi}{4}$ and is running in anti-clockwise direction. The sample trajectory 3 again traverses a straight line upto (0.02334m,0.02334m) along x and y direction respectively and then performs a circular motion with number of rotations f = 14 (0.1866Hz), radius = 0.033m, the initial angle $\theta_0 = \frac{\pi}{4}$ and makes a clockwise rotations.

The rest of the 25 trajectories are random trajectories generated using cubic spline function in MATLAB as described in section 4.3.1 for the simulation environment. Two sample random trajectories are shown in figure 4.8. Again, it has to be noticed that the path the manipulator traces does not follow the exact reference path due to the feedback controller settings. Nonetheless, these trajectories still contain useful information about the dynamics of the manipulator.



(a) Random Trajectory 1

(b) Random Trajectory 2

Figure 4.8: Random Trajectories in Real time environment

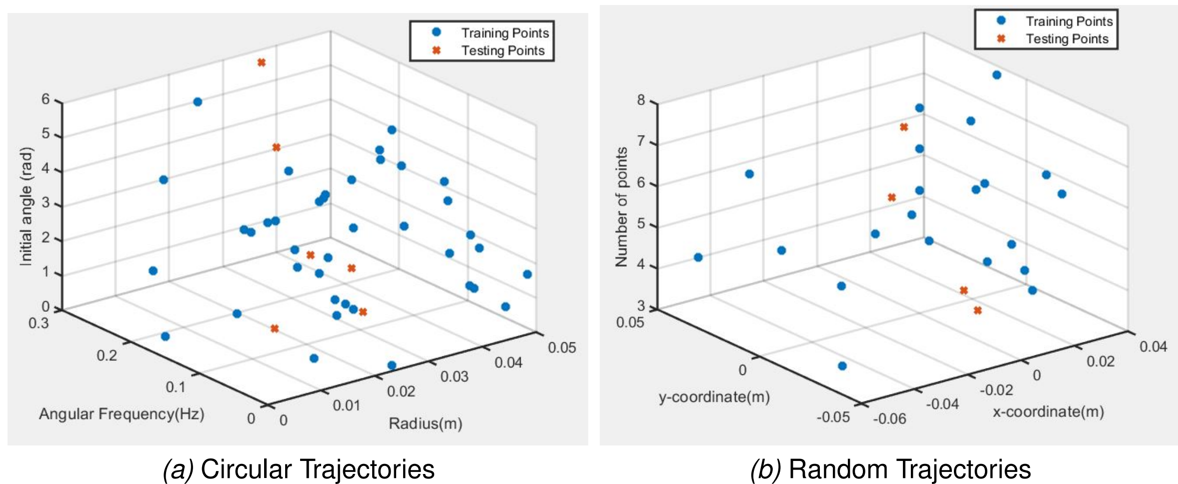
4.4 Data Pre-Processing for real time environment

The three encoders mounted in the motors gives the angular rotation of the each of the shoulder joints. It is then possible to find the rest of the joint positions of the manipulator through forward kinematics as discussed for the simulation environment. For the test setup, due to the encoder resolution, the obtained position dataset has quantization noises along with other unknown high frequency noises. These have to be filtered out before it can be used for obtaining velocities and accelerations data. For this purpose again a 4th order low-pass Butterworth filter is used which was discussed for simulation environment (section 4.3.2). The cut-off frequency of the filter is set to 1.8 Hz, which is slightly above the natural frequency of the manipulator. This ensures that the dataset is free of high frequency noise while essential dynamics are not lost when filtering. The velocity and acceleration of the joints are then found by taking the first order and second order derivatives of the position data respectively, similar to the simulation environment.

The corresponding end effector forces are collected from the feedback controller's output. The end-effector forces also contain high frequency noises, thus, these force data are also filtered using the same 4th order low pass Butterworth filter. This again ensures that the effect of filtering doesn't affect the learning process. Here, since each trajectories are obtained for 100 seconds at a sampling rate of 1KHz, a single trajectory can be represented by a tensor of size $(1 \times 100000 \times 63)$ indput data and $(1 \times 100000 \times 2)$ output data. Training such a high number of data points are computationally expensive. Thus, the obtained dataset is further down-sampled to 100 Hz to reduce the number of data for training. This reduces the number of data in a single trajectory from 100,000 data points to 10,000 data points. Essentially, a single trajectory contains tensors of $(1 \times 10000 \times 63)$ input data and

$(1 \times 10000 \times 2)$ output data.

The 70 trajectories collected from simulation environment are split into training dataset and testing dataset. In this case, 60 trajectories are used for training, while 10 trajectories are used for testing. 6 of the testing trajectories are circular paths which are in between the training points, i.e, the angular frequencies of each of these circular trajectories are 0.066Hz, 0.08Hz, 0.09Hz, 0.106Hz, 0.265Hz and 0.28Hz, the radius or magnitudes of these trajectories are less than 0.05m and these trajectories are not leaked into the training dataset. The rest of the 4 testing datasets are random trajectories passing through arbitrary points within the workspace limit. The overview of the training and testing data-points collected from real time environment is presented in figure 4.9. Similar to the simulation environment the figure 4.9a displays the training and testing points of 45 circular trajectories as a function of three parameters (radius, angular frequency and initial angle). The figure 4.9b again displays the training and testing data-points for rest of the 25 random curve spline trajectories similar to the case discussed for simulation environment.



(a) Circular Trajectories *(b)* Random Trajectories

Figure 4.9: Training and Testing Data-points (Real time environment)

The collected dataset are further normalized to match the consistency of the learning model. The normalization is performed between the range of [0 1] as discussed before (refer equation 4.3). The normalization on input features, i.e, positions, velocities and accelerations are done separately and the normalization on labels, i.e end effector forces are obtained separately. For the real time environment as well, the normalization is applied only on training dataset, the resulting coefficients are used for normalizing the testing dataset separately. This is again done to avoid leakage of information from testing dataset to training dataset. Inverse normalization is performed after prediction to retain the scale of the original values, for comparison.

Chapter 5

Results from Simulation Environment

This chapter details out the architecture of the proposed neural network models used for learning with the simulation environment and the results obtained thereby. The training process involved in each of the network is detailed out. An offline feed-forward controller is developed based on the learned model are evaluated and the performance of each of the networks are compared.

5.1 Data Visualization

Before starting the training process, it is attempted to visualize the auto-correlation properties of the collected data. Auto-correlation measures the relationship between a variable's present values to its past values. For this purpose the end effector forces that is collected along x and y direction of one of the trajectories is used to visualize. A lag plot is plotted in an attempt to see the auto-correlation properties of the data when there is a lag of 1, 100, 500 and 1000 timesteps. The lag here is a fixed time displacement. The lag plot is shown in the figure 5.1. The linearity in lag plots indicates that there is a strong autocorrelation in the dataset. It is observed that the linearity diminishes slowly after a lag of 200 timesteps. The developed machine learning model tries to exploit this autocorrelation property in the dataset, between the lags of 100 to 200 timesteps.

5.2 Setting up the learning environment

As discussed earlier, 40 trajectories are used for training, while 4 trajectories are used for testing in simulation environment. The training dataset are further divided into 87.5 percent for training and 12.5 percent for validating. The training and the prediction is performed in Python 3.7 environment with GPU compatible Tensorflow and Keras libraries. All the network models are built using the Keras libraries and are

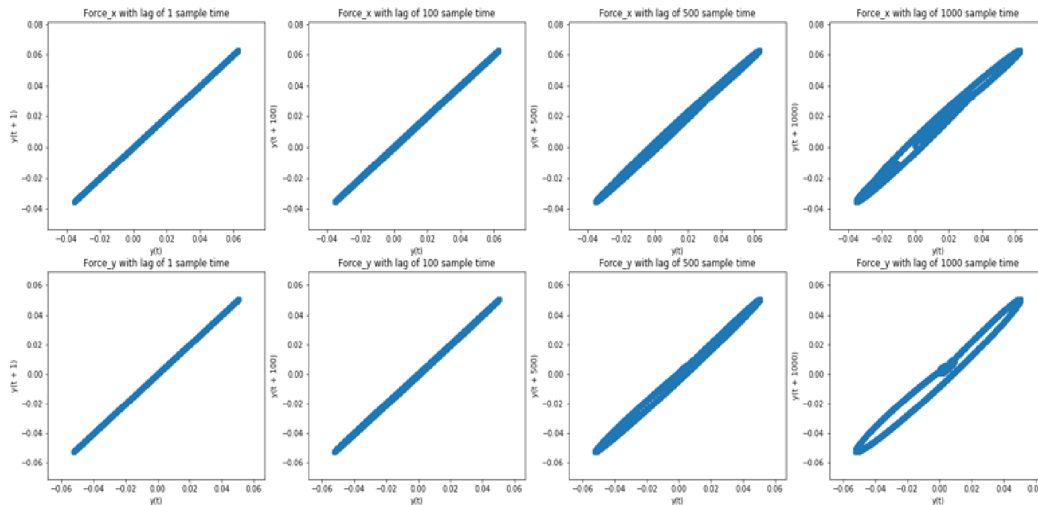


Figure 5.1: Lag Plots of End effector forces along x and y directions

trained with a NVIDIA GeForce GTX 1050 Ti GPU for an efficient training process. In the upcoming sections, the architectures of the proposed neural networks are presented and the results obtained from them are displayed.

5.3 LSTM Network Design

The LSTM network proposed here has one sequential input layer, two LSTM layers, followed by a dropout layer and a regression output layer, see figure 5.2. The input layer consists of 63 neurons which takes in 21 positions, 21 velocities and 21 acceleration data from 7 joints present in the manipulator. The two LSTM layers have 50 to 200 neurons each, different number of hidden neurons are experimented (these are discussed in upcoming sections). A dropout layer with a value of 0.05 is added in between each of these layers for regularization. Adding dropout layers minimises the risk of over-fitting on training datasets. The output regression layer has two neurons corresponding to the two end effector forces which the network tries to predict. $Tanh$ activation function is used in between the hidden LSTM layers and a linear activation function is used for the ouput regression layer.

Here, the Truncated Back Propagation Through Time(TBPTT) [30] is adopted for training the network. TBPTT is a modified back propagation through time, wherein, instead of back propagating through the whole timesteps of the data the TBPTT back propagates through a fixed number of timesteps. As indicated from the lag plots in the previous section, the data shows strong auto-correlation between 100 to 200 timesteps. Thus, TBPTT through 100 timesteps is adopted for this network.

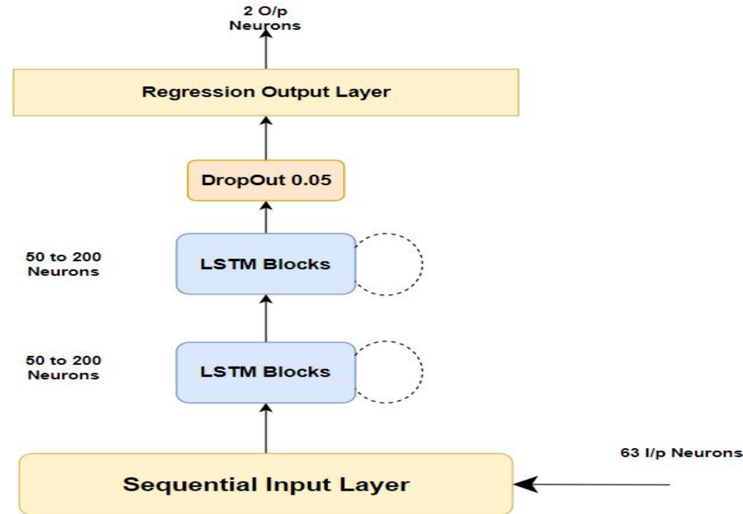


Figure 5.2: LSTM Network Design

The network is trained on 87.5 percent of the dataset collected the rest are used for validation. During each iterations while training, few trajectories from validation dataset are shuffled into training dataset. This is done, in order to improve the convergence rate of the learning process. Also, introducing new dataset to the network avoids the problem of over-fitting on a single sets of trajectories. The predictions on 4 unseen test dataset are evaluated to observe how well the network has learned the inverse dynamics of the manipulator.

As said earlier, predicting the inverse dynamics of the system comes under a regression problem. Therefore use of regression loss function is appropriate for fitting the data. Keras library provides with various regression loss functions. In this study, a Mean squared loss function (MSE) is used since this function penalizes larger errors more compared to small ones.

Keras library also provides with optimization functions for training the network. For training the LSTM network Adam optimizer [31] is used. The Adam optimizer is described as a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. It is memory efficient and combines most of the good parts of other optimizers in Keras.

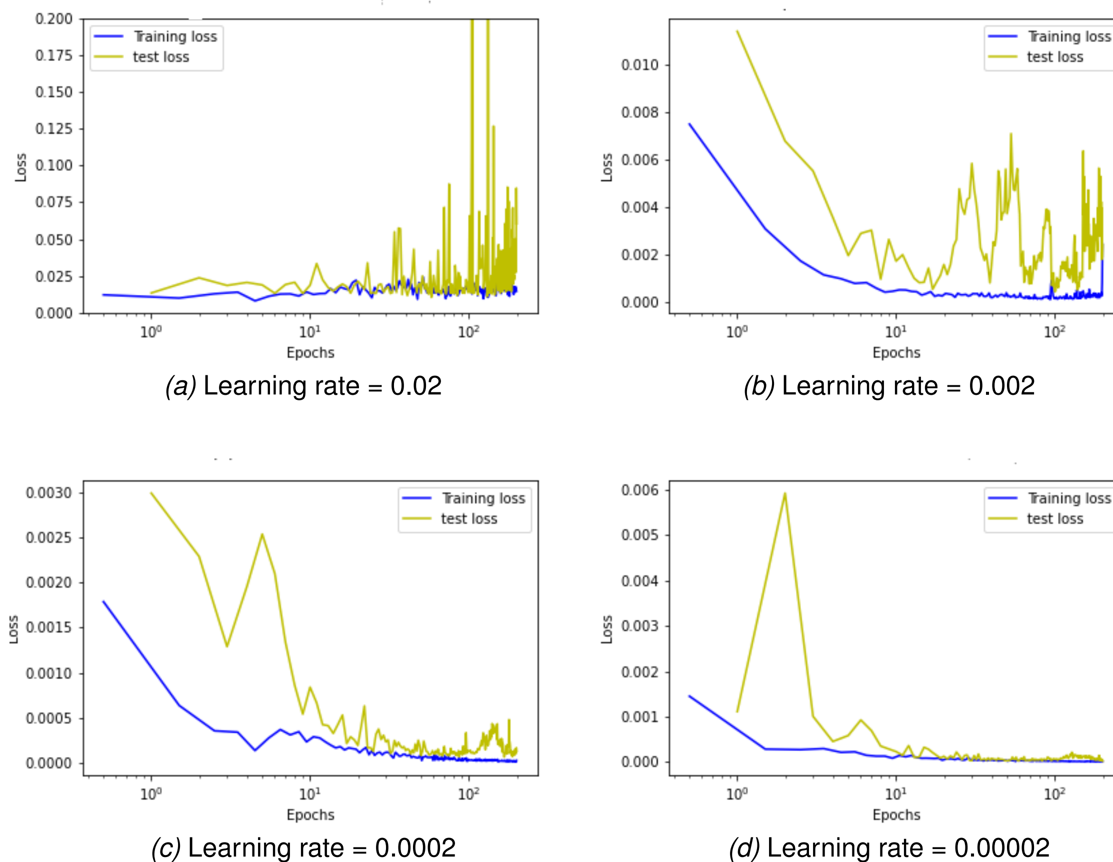
Often times, setting the initial weights (model parameters) also is crucial for the learning process. For this purpose, the Glorot uniform [32] provided by keras library is used, which is a good starting point for a dataset with uniform distribution.

5.3.1 Hyperparameters tuning and experimental results

The hyperparameters in machine learning are parameters whose value control the overall learning process. Often times, these hyperparameters are set before training

process and tuning these are essential for quality of the learning process. In this study, three main hyperparameters are considered for tuning. These are the number of hidden neurons in the LSTM layers, the learning rate of the network and the number of iterations or epochs the network is trained for.

To start with, the number of epochs is kept to 200, the number of hidden neurons are kept to 100 and the learning rate is first tuned. 4 different learning rates were tested, they are 0.02, 0.002 , 0.0002 and 0.00002. The learning rate basically determines the convergence rate of the learning process. The MSE while training the network with these different setting are plotted in a logarithmic plot and is shown in figure 5.11.



(a) Learning rate = 0.02

(b) Learning rate = 0.002

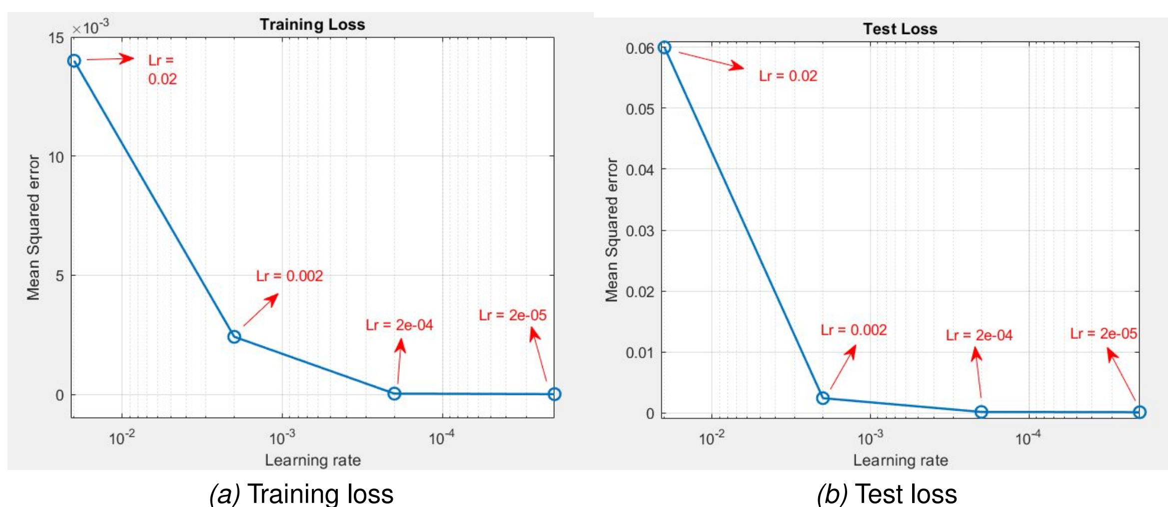
(c) Learning rate = 0.0002

(d) Learning rate = 0.00002

Figure 5.3: Loss Function plot - Comparison between different learning rate

It can be observed that when learning rate is 0.02 (figure 5.3a), the network hardly learns. The training loss doesn't converges and the test loss seems to be growing. When the learning rate is reduced to 0.002 (figure 5.3b), the training loss starts to converge better and the loss on test data also starts to converge slowly, but after 11 epochs or so, the test loss seems to be oscillating, while training loss doesn't seem to converge further. This could be due to the trained weights being stuck in

local minima. When the learning rate was set to 0.0002 (figure 5.3c) and 0.00002 (figure 5.3d) better convergence could be seen on both the dataset. To get a more clear view of the convergence, the MSE at the end of 200 epochs with different learning rates are compared and plotted in figure 5.4. As it can be observed from these plots, the MSE in training and testing dataset yielded good results when the learning rate are 0.0002 and 0.00002. The networks performance is slightly better when the learning rate is 0.00002. Thus, this learning rate is fixed as an optimal value.

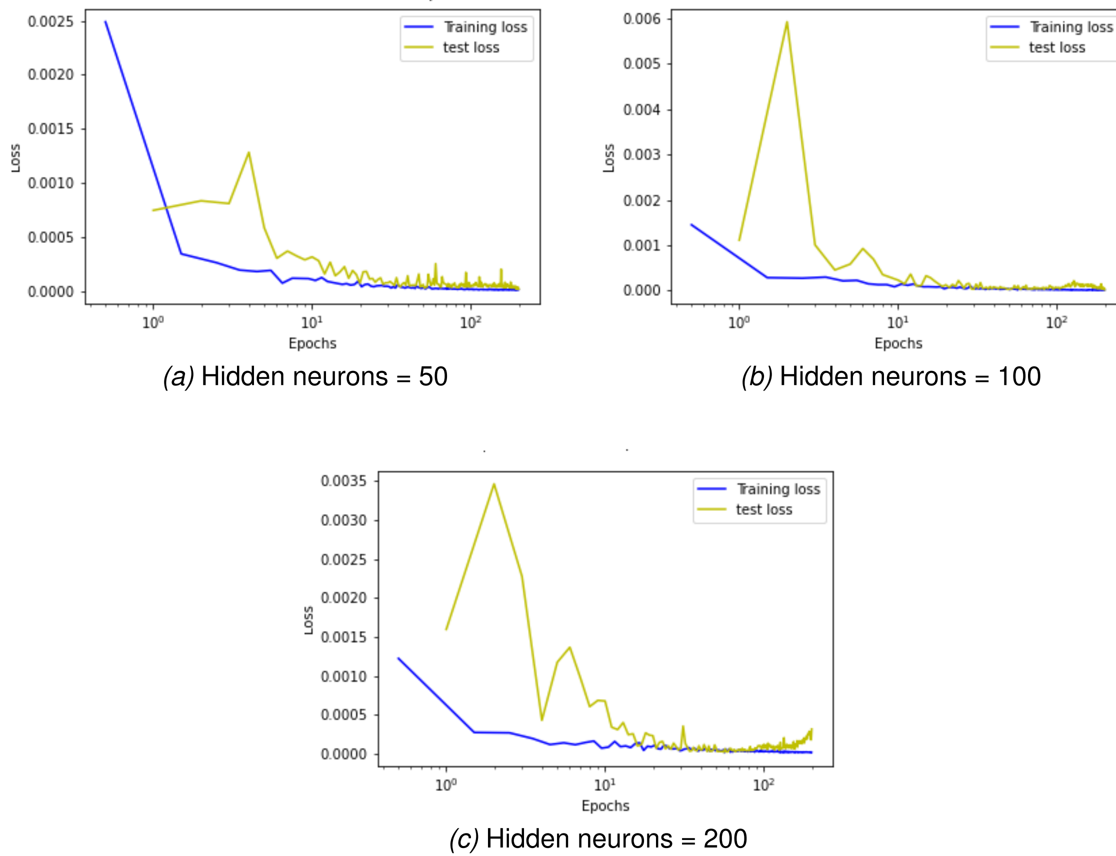


(a) Training loss *(b)* Test loss

Figure 5.4: Loss function against Learning rate

Next the number of hidden neurons are taken into consideration. The learning rate of 0.00002 and maximum number of epochs set to 200. The number of hidden neurons is varied between 50, 100 and 200, and the LSTM network is trained. The MSE loss function during each of these scenarios are plotted in logarithmic graph and is presented in figure 5.5. It is observed from figure 5.5a and 5.5b that, when the hidden neurons are 50 and 100, the network performed similarly, and the loss function converges to an optimal value. When the number of hidden neurons are 200, after approximately 130 epochs, the loss in test dataset increases, indicating that the network is overfitting on training data.

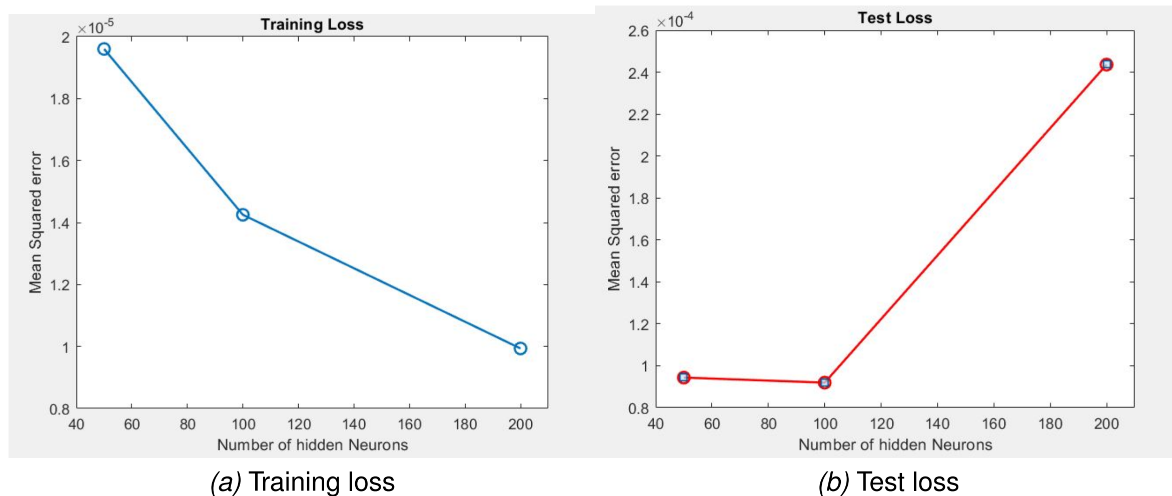
The MSE as function of the number of hidden neurons is plotted and displayed in figure 5.6. It can be clearly observed from this figure that the training and testing loss when the number of hidden neurons are 50 and 100 shows good convergence. The training loss when there are 200 hidden neurons is very much low, but on testing dataset the loss is far higher indicating that the model is overfit on training data. Thus, here the number of optimal set of hidden neurons was selected to be 100 neurons.



(a) Hidden neurons = 50 *(b)* Hidden neurons = 100

(c) Hidden neurons = 200

Figure 5.5: Loss Function plot - Comparison between different hidden neurons



(a) Training loss *(b)* Test loss

Figure 5.6: Loss function against number of hidden neurons

Next, to test the effect of number of iterations on the learning process, the maximum number of epochs is increased to 1000 with the learning rate of 0.00002 and

the number of hidden neurons set to 100. The training and testing losses obtained for 1000 epochs are displayed in figure 5.7. From the losses in the training and testing dataset, it was observed that the loss function converges further after 200 epochs and reaches an optimal value somewhere around at 600 epochs. There was no significant convergence noticed after this.

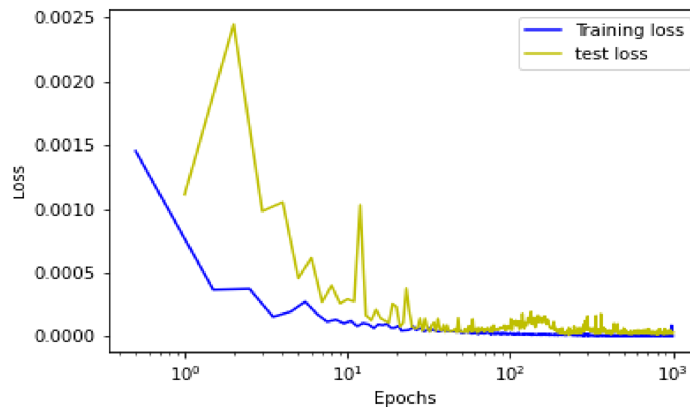
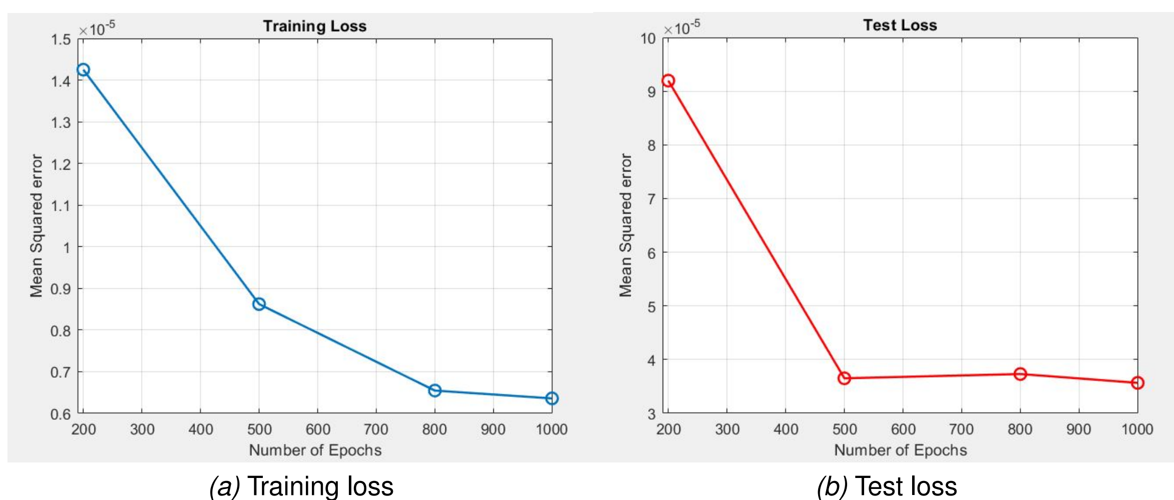


Figure 5.7: LSTM Training and testing loss over 1000 epochs

To get a more clear picture, the losses at 200 epochs, 500 epochs, 800 epochs and 1000 epochs are plotted separately for training and testing dataset. This is shown in figure 5.8. It can be observed from this figure that, there hardly any convergence in test dataset (figure 5.8b) after 500 epochs, while there is minute convergence in training loss (figure 5.8a). The most optimal weights were found at epoch 609, which had the least training and test loss (the values are shown in subsequent section while comparing with other networks).



(a) Training loss

(b) Test loss

Figure 5.8: Loss function against maximum number of epochs

The model weights during this epoch are used for predicting the end effector forces. The prediction from the network would correspond to a normalized value. For comparing the prediction with the actual values of the end effector forces, the predictions from the trained model are re-scaled by inverse transforming the normalized outputs. A comparison between the actual end effector forces in x and y direction to that of the predictions made by the network is shown in figure 5.9. The predictions from two out of the four trajectories used as an unseen test dataset are displayed in this figure. The units of the prediction after rescaling are in Newtons(N). It can be observed from this figure that the LSTM network obtains a good fit on unseen dataset.

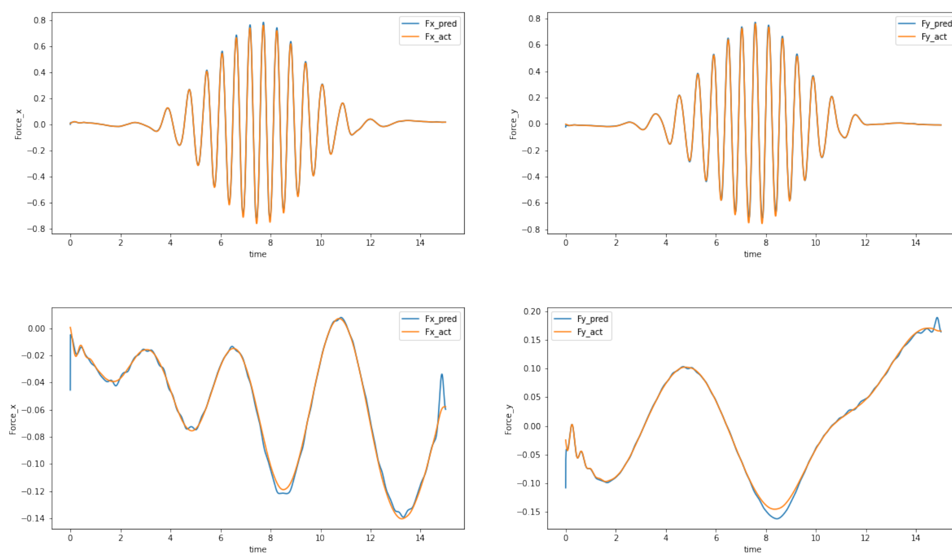


Figure 5.9: LSTM Predictions

5.4 TCN network Design

The TCN network designed in this study is shown in the figure 5.10. The network consists of an input layer with 63 neurons, followed by six convolutional layers stacked on top of each other. A $Relu$ activation function is used in between the convolutional layers. Each of the six convolutional layers have a hidden neurons of 50 to 200 (Different number of neurons are experimented which are discussed in upcoming section) and a kernel window of size 3. A dropout layer with a value of 0.05 is added in between each of these layers for regularization. The six convolutional layers have a dilation factors of 1, 2, 4, 8, 16 and 32 respectively. The choice of number convolutional layers, the dilation factor and kernel window size follows from the number of timesteps the network has to look back in time. Since the dataset shows

strong auto-correlation in the range of 100 to 200 timesteps, the network is designed such that the receptive field is between these ranges. With the chosen architecture, the receptive field of the TCN network is 128 timesteps (calculated from equation 3.4). At the end, a regression layer with a linear activation function is added, which consists of 2 neurons for predicting the end effector forces.

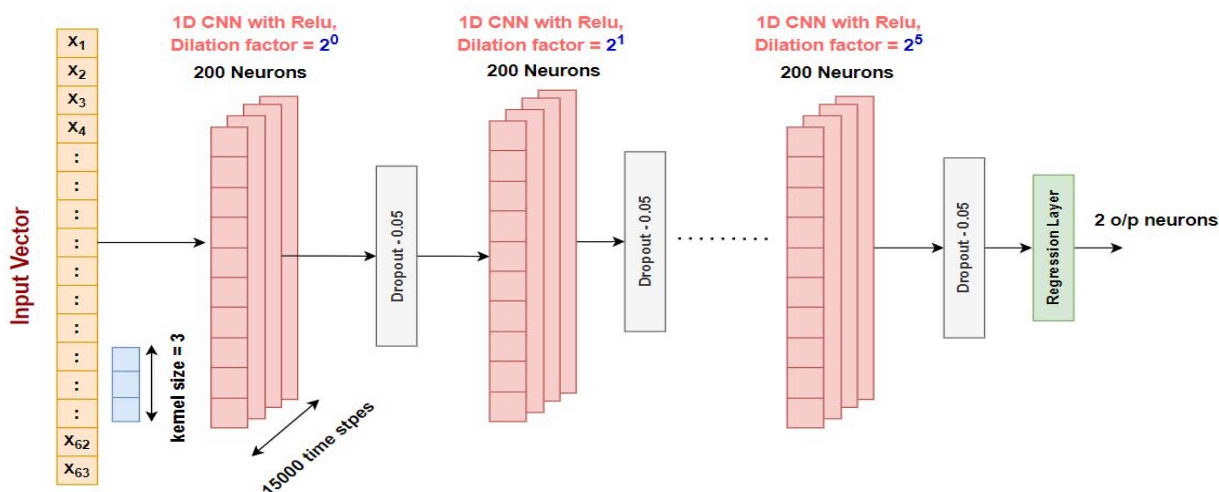


Figure 5.10: TCN Network Design

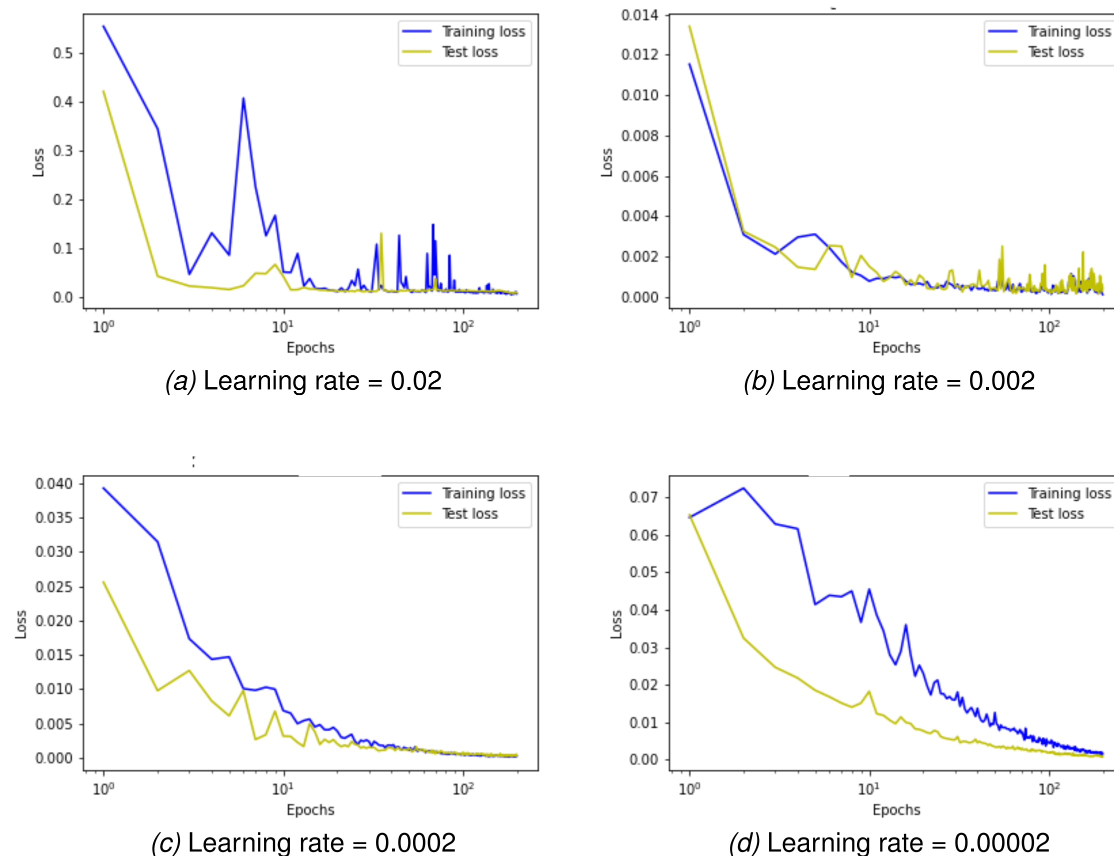
The network is trained with 87.5 percent of the training dataset collected, as in the case of LSTM network. The rest are used as validation dataset. Few trajectories from validation data are shuffled into the training data for improving the convergence and avoid over-fitting. The choice of loss function is considered as MSE again. Adam optimizer and Glorot uniform weight initializers are used for the training the network for similar reasons mentioned in section 5.3. The predictions on 4 unseen test dataset are evaluated later.

5.4.1 Hyperparameters tuning and experimental results

Similar to the LSTM network three hyperparamters : learning rate, number of hidden neurons and maximum number of epochs are considered for tuning.

To get started with, the learning rate is first tuned by keeping the maximum number of epochs to 200 and the number of hidden neurons to 100. Four different learning rates : 0.02, 0.002, 0.0002 and 0.00002 are experimented again. When the learning rate is 0.02 (figure 5.11a), the loss function in both the training and testing dataset fluctuates a lot, it was also noticed that the MSE did not converge to a lower value that is expected. The convergence improved when the learning rate is set to 0.002 (figure 5.11b), still some fluctuations can be seen in between the loss functions. When the learning rate was kept to 0.0002 (figure 5.11c), the loss

function converged to a lower value compared to the previous two case, although the convergence rate seems to be less, i.e the network might need more epochs to reach a much better optimal value. In the case when the learning rate was kept to 0.00002, the convergence rate is far more slower, which can be clearly seen from training loss, it was observed that within 200 epochs the network did not reach a lower MSE.



(a) Learning rate = 0.02

(b) Learning rate = 0.002

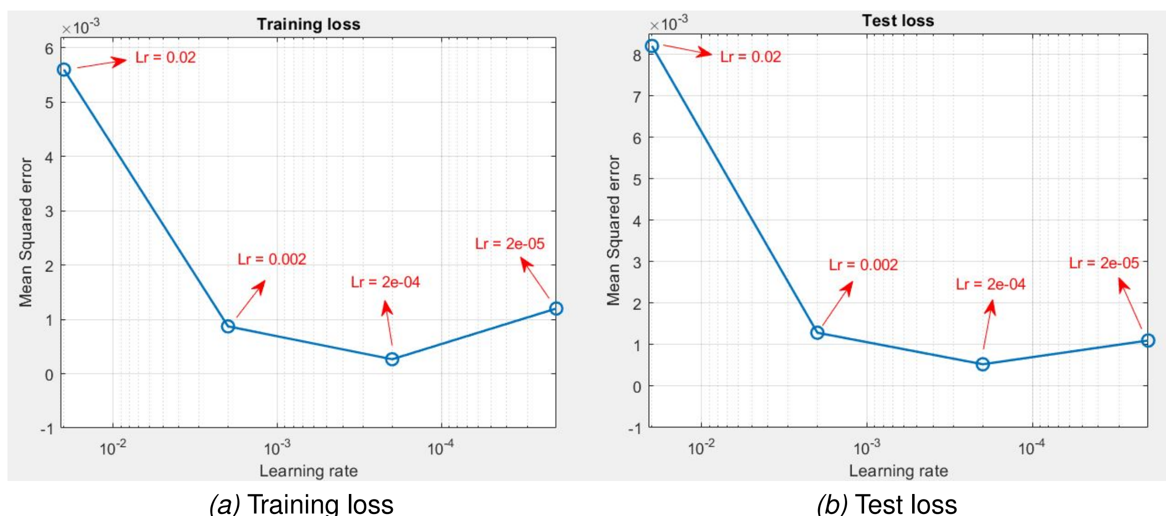
(c) Learning rate = 0.0002

(d) Learning rate = 0.00002

Figure 5.11: Loss Function plot - Comparison between different learning rate (TCN)

To get a more clear picture the MSE at the end of 200 epochs is plotted against the learning rate in logarithmic graph, see figure 5.12. It can be observed from this figure that the training and testing loss converged to minimum value when the learning rate is 0.0002 at the end of 200 epochs. Thus, for this network, the learning rate is set to 0.0002.

The number of hidden neurons are taken into consideration next. Again, the performance of the network is evaluated for 50, 100 and 200 neurons. The learning rate is set to 0.0002 and the maximum number of epochs are set to 200. The loss function during each of these scenarios are displayed in figure 5.13. The network with 50



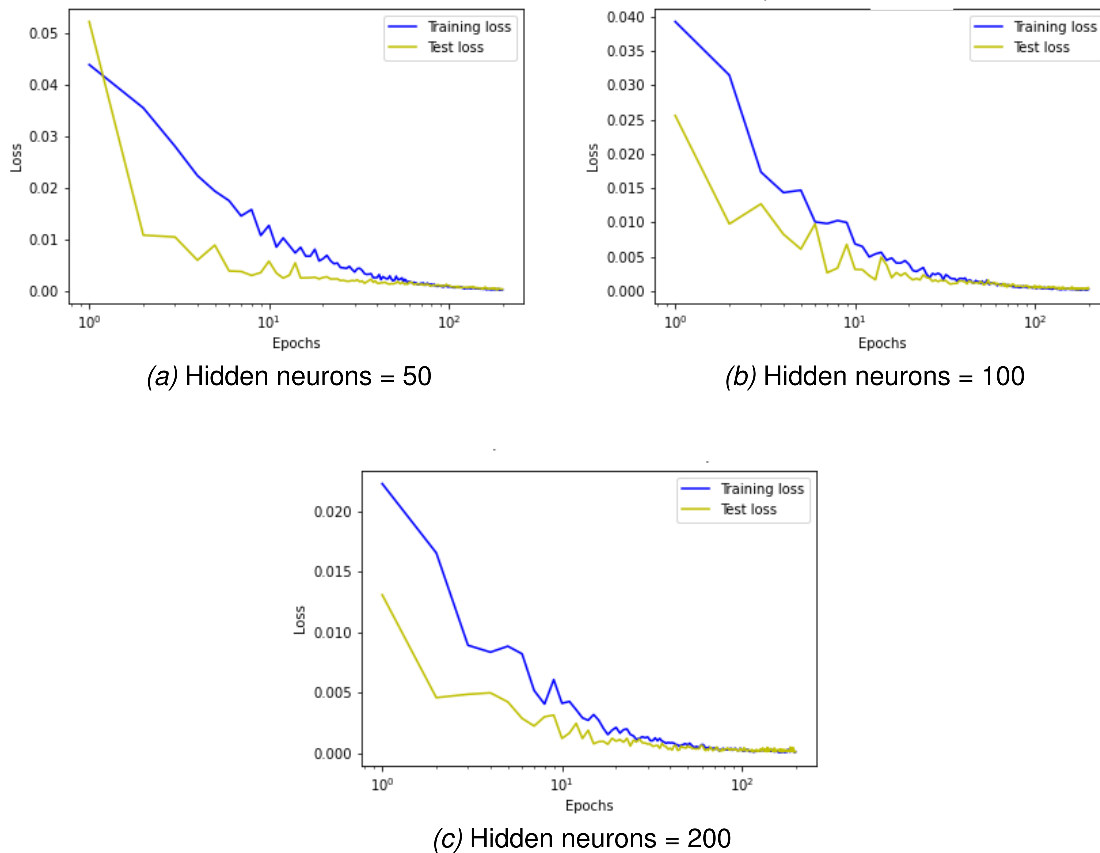
(a) Training loss *(b)* Test loss

Figure 5.12: Loss function against Learning rate (TCN)

hidden neurons (figure 5.13a) showed slower convergence. The loss function when the number of hidden neurons are set to 100 (figure 5.13b) and 200 (figure 5.13c) showed similar results. However, it was noticed that the network with 200 neurons converged to mildly lower value compared to the network with 100 neurons.

The MSE at the end 200 epochs is plotted against the number of hidden neurons for better visualization, see figure 5.14. It can be observed from this figure that the loss function in both the training and testing datset converges to a lower value when the number of hidden neurons are set to 200. Thus for this network further experiments were done by setting the number of hidden neurons to 200.

Next, the number of epochs the network is trained is taken into consideration. For this network it was observed that the loss did not converge to the extend the loss from LSTM network converged wihtin 200 epochs for both the training and testing datset. So, the TCN network is experimented with much more larger number of epochs. The maximum number of epochs is increased to 1500 with the learning rate of 0.0002 and the number of hidden neurons set to 200. The training and testing loss over 1500 epochs are plotted and presented in figure 5.15. The figure 5.15b presents the scaled up version of the loss function for better visualization. It was observed that TCN needed more number of epochs to converge to an optimal MSE compared to the LSTM network. After around 1200 epochs the loss in training and testing dataset reached its minimum level and was comparable with loss from LSTM networks. After 1200 epochs there were hardly any improvements in the convergence for both the training and testing dataset.

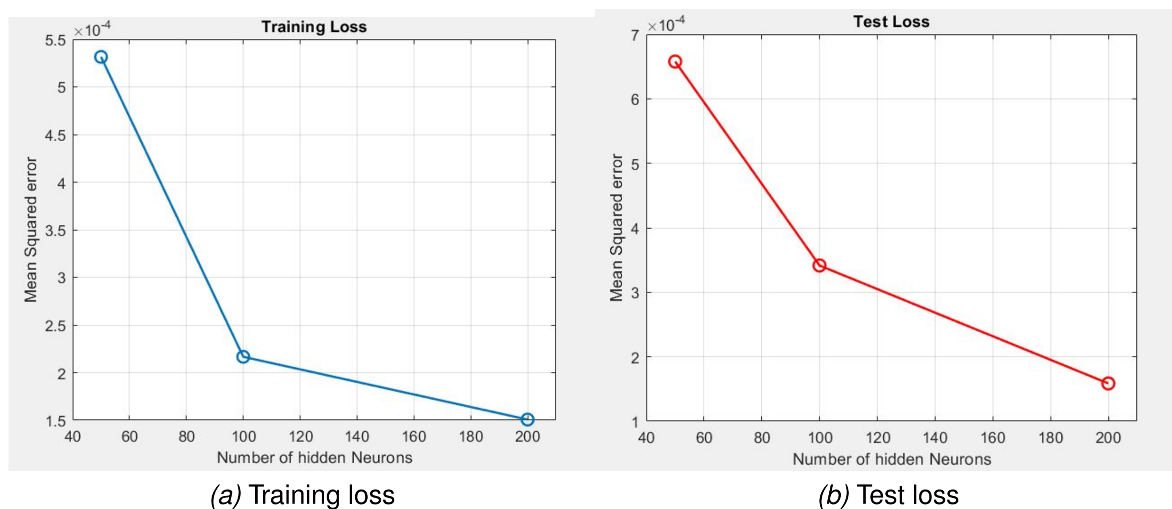


(a) Hidden neurons = 50

(b) Hidden neurons = 100

(c) Hidden neurons = 200

Figure 5.13: Loss Function plot - Comparison between different hidden neurons (TCN)

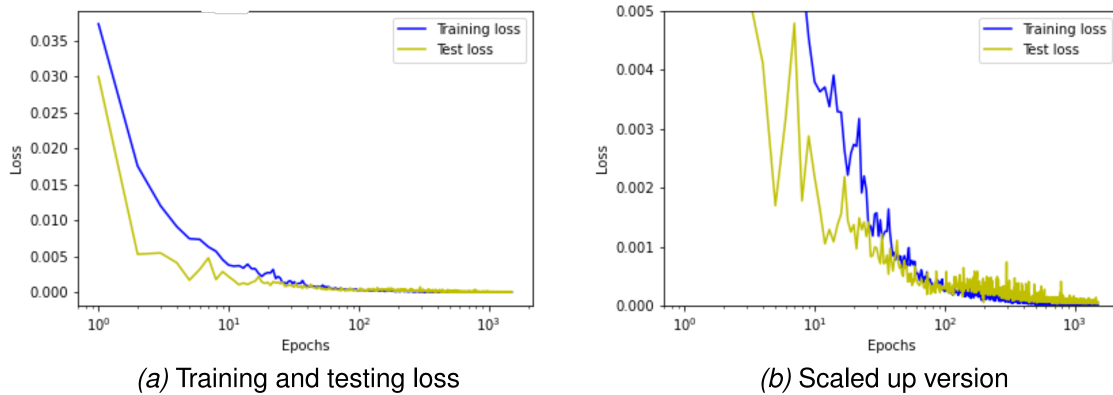


(a) Training loss

(b) Test loss

Figure 5.14: Loss function against number of hidden neurons (TCN)

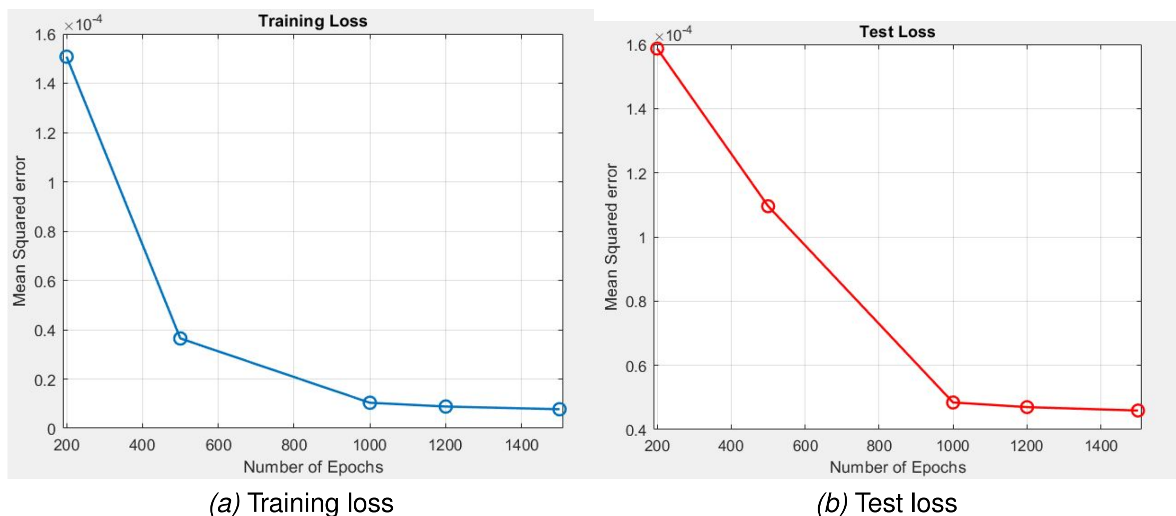
To get a more clear picture, the losses at 200 epochs, 500 epochs, 1000 epochs, 1200 epochs and 1500 epochs are plotted for training and testing dataset. This is



(a) Training and testing loss *(b)* Scaled up version

Figure 5.15: TCN Training and testing loss over 1500 epochs

shown in figure 5.16. It can be observed from this figure that there is no significant convergence after 1200 epochs for both the training and testing dataset. The most optimal weights were found at epoch 1467. It had the least training and test loss, and these model parameters were chosen for predicting the end effector forces.



(a) Training loss *(b)* Test loss

Figure 5.16: Loss function against maximum number of epochs (TCN)

The prediction from this network on two of the unseen test trajectories are shown in figure 5.17. The predictions from the network are in normalized form, so, they are re-scaled to the original values (as discussed in previous sections) by inverse transforming the normalized values. The units after rescaling are in Newtons(N). The predictions made by this network seems to obtain a good fit on the end effector forces, as can be seen from this figure.

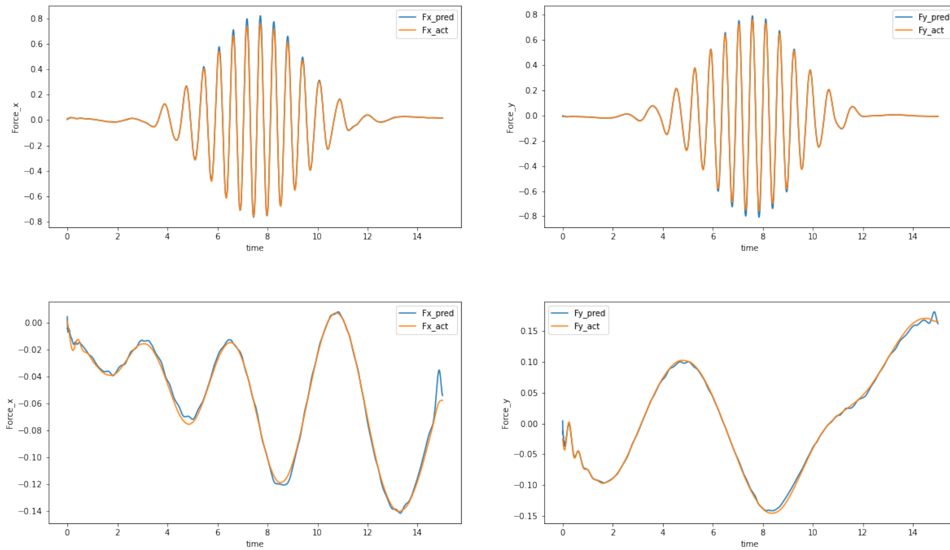


Figure 5.17: TCN Predictions

5.5 1D-CNN LSTM network Design

As mentioned earlier, 1D CNNs are good at exploiting higher level features from a timeseries data. For training the CNN-LSTM network, the 40 trajectories collected are further split into 20 trajectories for feature extraction from 1D CNN, and 20 trajectories for training the LSTM network. The designed network is shown in the figure 5.18.

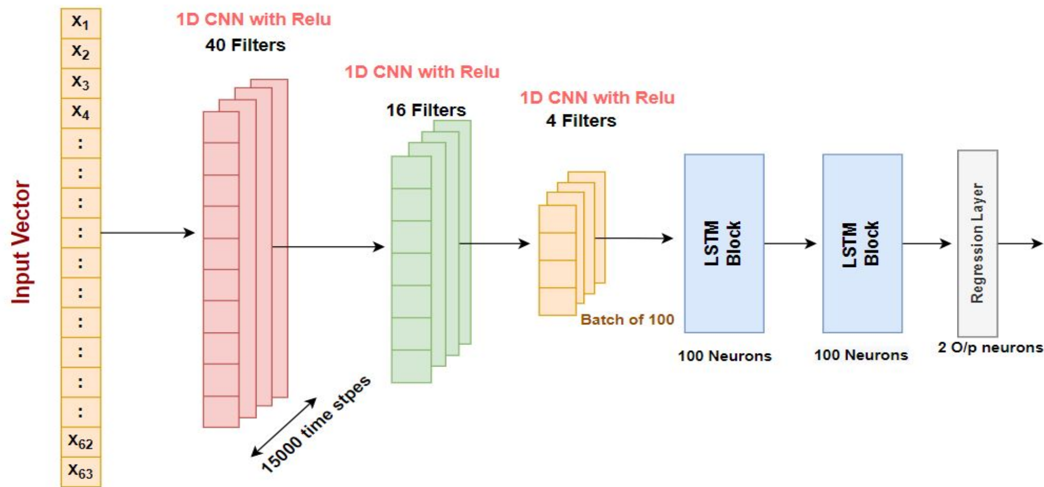


Figure 5.18: 1D CNN-LSTM Network Design

Again, the input layer consists of 63 neurons for the input data. Three 1D convolutional layers with output neurons of 40, 16 and 4 are stacked on top of each other. The kernel size of the first, second and the third CNN layers are 10, 4 and 2 respec-

tively. A $Relu$ activation function is used in between the convolutional layers. With the chosen design, the convolutional layers maps 63 input vectors to four higher level features (last layer corresponding to 4 features). This is called as feature extraction.Thus, the LSTM cells here contains 4 input neurons. Two LSTM cells each with 100 hidden neurons are stacked on top of each other, and it is followed by a regression output layer which outputs 2 neurons corresponding to the end effector forces. Similar to the previous network design, $tanh$ activation function is used between the LSTM layers and a linear activation function is used for the output regression layer.

First, the 20 trajectories are used for training the 1D convolutional network to extract 4 higher level features. The other 20 trajectories are then split into training (17 trajectories) and validation dataset (3 trajectories). This training dataset are first introduced to the trained convolutional layers to extract 4 features. The resulting higher level features are used for training the LSTM network with TBPTT of 100 timesteps similar to the one discussed in the previous section. The MSE metrics is set for evaluation, Adam optimizer and Glorot uniform kernel initializer are used in case of this network as well.

The hyperparamters for this network are set to optimal values that were found while training the LSTM network. Thus, the learning rate is set to 0.00002, the number of hidden neurons are kept as 100 and the maximum number of epochs are set to 1000. During each epochs, few trajectories from validation dataset are shuffled into the training dataset to avoid over-fitting. The predictions on 4 unseen test dataset are evaluated.

5.5.1 Experimental Results

The training and testing loss from CNN-LSTM network is shown in figure 5.19. The evolution of the MSE over 1000 epochs are plotted in a logarithmic plot. It can be observed from this plot that the training and testing losses do converge. However, both the training and testing loss did not converge to the same level compared to the other two networks. Moreover, the loss in testing dataset seemed to be increasing after 300 epochs approximately. This indicates that the network overfits on the training data. Another observation that can be made is that, there is a significant difference in the convergence of training and testing loss. This could happen in case the training dataset doesn't provide sufficient information to learn the problem. The huge gap between the training and testing loss can be attributed to the small number of datasets (20 trajectories) that are used for training the CNN-LSTM network. This could be seen as a trade-off between number of dataset required for extracting the features and number of dataset required for training the network which leads to an accurate prediction. The weights during the epoch 292 provided with the least

training and testing loss, thus, weights during this epoch is chosen as the optimal set of model parameters.

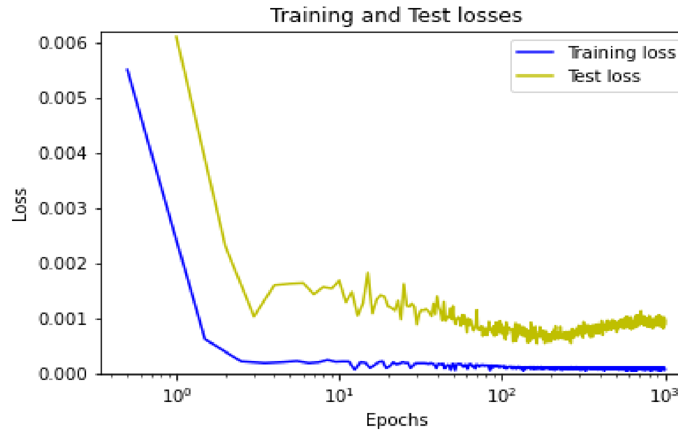


Figure 5.19: CNN-LSTM Training and testing loss

The prediction from this network on two of the unseen test trajectories are shown in figure 5.20. The network predicts normalized outputs which are inverse transformed to the scale of the original values, as discussed in the previous section. It is observed that the prediction from this network is not as accurate as the predictions from the other two networks. A detailed comparison in the results between each of these network are presented in the upcoming sections.

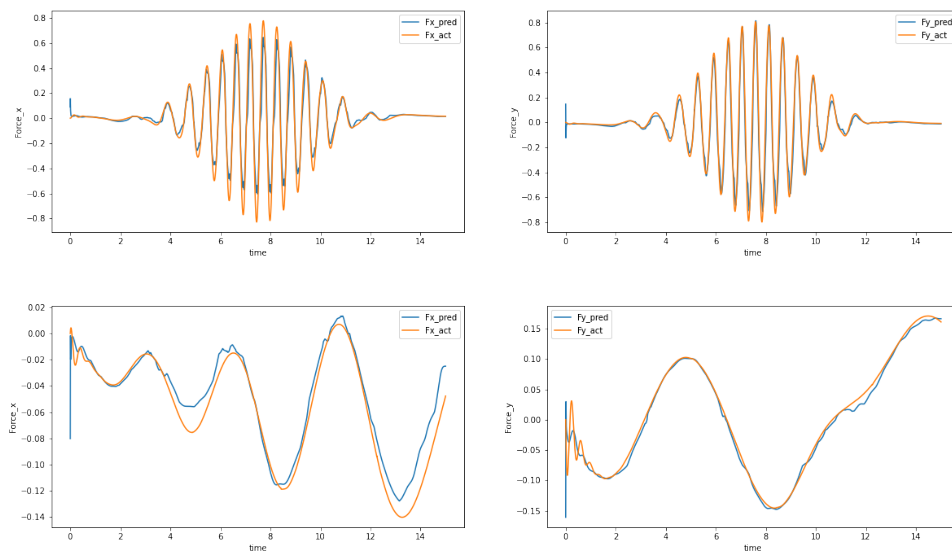


Figure 5.20: CNN-LSTM Predictions

5.6 Results Comparison

After the learned model converged to an optimal value, the trained weights during one of the epochs are used as the parameters that could predict the inverse dynamics of the manipulator precisely. The optimal loss function (MSE) obtained from all the networks are compared and tabulated in table 5.1. The losses are scaled by a factor of 10^{-6} for better readability and the most optimal results are highlighted in red ink. It can be observed that the CNN-LSTM performs poorly compared to the other two networks, the loss in test dataset is very much higher compared to the loss in the training dataset. This could be due to the extracted features from CNN not being generalized enough when applying on unseen dataset, another reason could also be due to some over-fitting on training dataset. It can also be inferred that, the training loss from LSTM and TCN networks are comparable, although, in the LSTM network the loss is slightly better compared to the TCN. On unseen dataset, again the losses from the LSTM and TCN are quite comparable, however, in this case the TCN outperforms the LSTM network slightly.

Mean Squared Error($\times 10^{-6}$)	**LSTM**	**CNN-LSTM**	**TCN**
Training loss	**5.727**	50.425	7.640
Test loss	23.234	687.482	**19.081**

Table 5.1: loss function comparison

Another two evaluation metrics that are considered in this study are the time taken for training per epoch and the number of parameters to train the network. The figures 5.21 and 5.22 presents the time taken per epoch and the number of trainable parameters of each of the networks respectively.

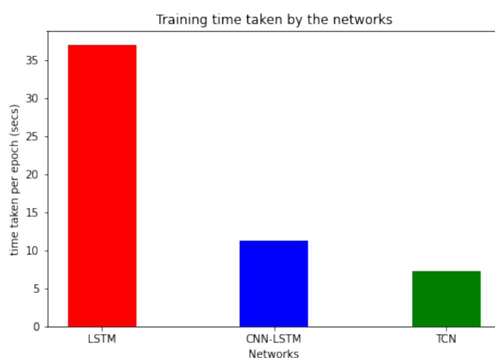


Figure 5.21: Time taken by networks

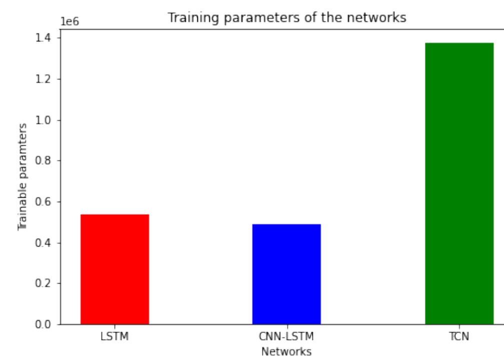


Figure 5.22: parameters to train by the networks

It can be observed that the LSTM takes a longer time to train compared to the

other networks. This can be attributed to the sequential nature of the TBPTT that is adopted while training the network, i.e the predictions in the later time steps waits for their predecessors to complete. With the number of features reduced using CNN-LSTM, this network took lesser time and lesser number of parameters to train comparatively. The TCN took the least amount of time since parallel computation is possible as same filter is used in each layer. Even though the number of parameters to train are higher in the TCN, due to parallelism the input sequence is considered as a whole rather than sequentially which is the case with LSTM. This ensures lesser training time in TCN.

5.7 Feed Forward Control and Evaluation

As discussed earlier, an offline feedforward control is developed in this study. First the learned networks are imported into MATLAB environment from python for testing. The predictions made by the network are used to implement a feedforward control along x and y direction separately. The overview of the control scheme implemented in Simulink is shown in the figure 5.23. The model is similar to the one discussed in the section 4.1, additionally a learned feedforward control is implemented along x and y direction as shown in this figure.

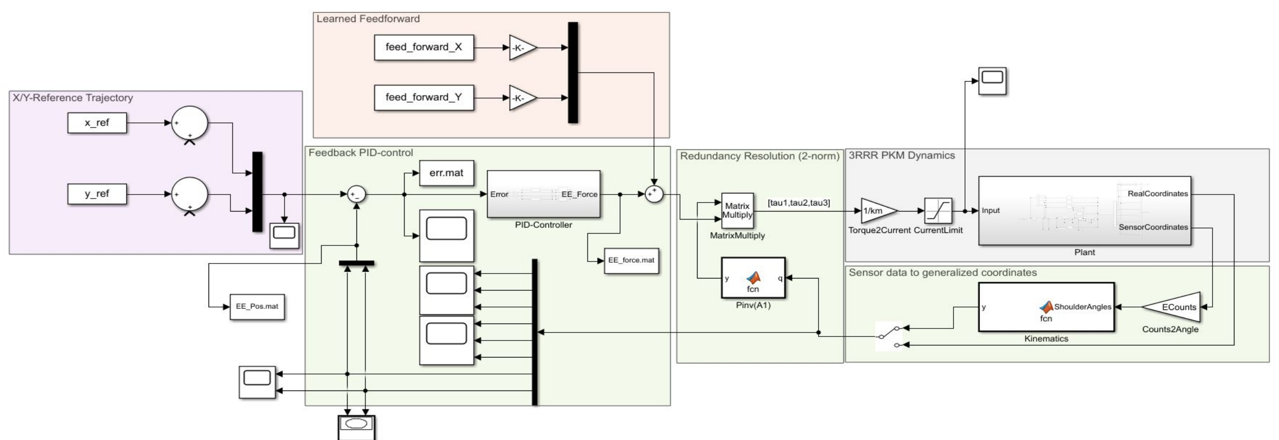


Figure 5.23: Simulink Block Diagram with feedforward control

For evaluating the feedforward, a straight line trajectory with a final set point to reach as h_x = 0.0098m and h_y = -0.0098m within 3 seconds, followed by a circular trajectory with an angular frequency of 0.958 Hz (11.5 rotations in 12 seconds) and a radius of 0.01400m is used as a reference path, see figure 5.24 (the scale of the trajectories along both the axis are in meters). The reference trajectory spans for a total time of 15 seconds. With the knowledge of reference trajectory, the inverse kinematics is applied to get the other 7 joints positions (19 dependent coordinates,

discussed in section 2.1). The positon data is then differentiated and double differentiated to obtain velocity and acceleration of all the joints respectively. These 63 input features collected over 15 seconds are given as the input to the network for predicting the feedforward control that needs to be applied for precise tracking.

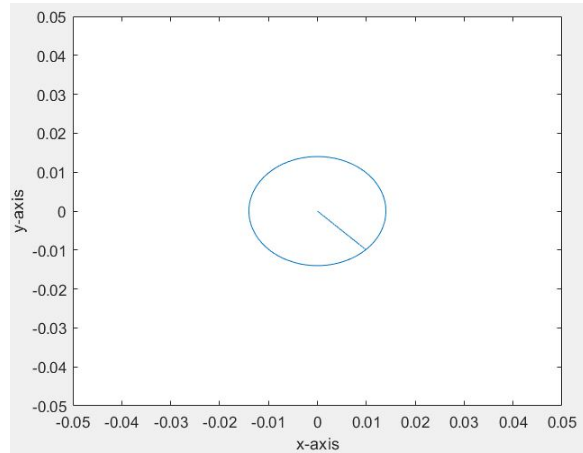
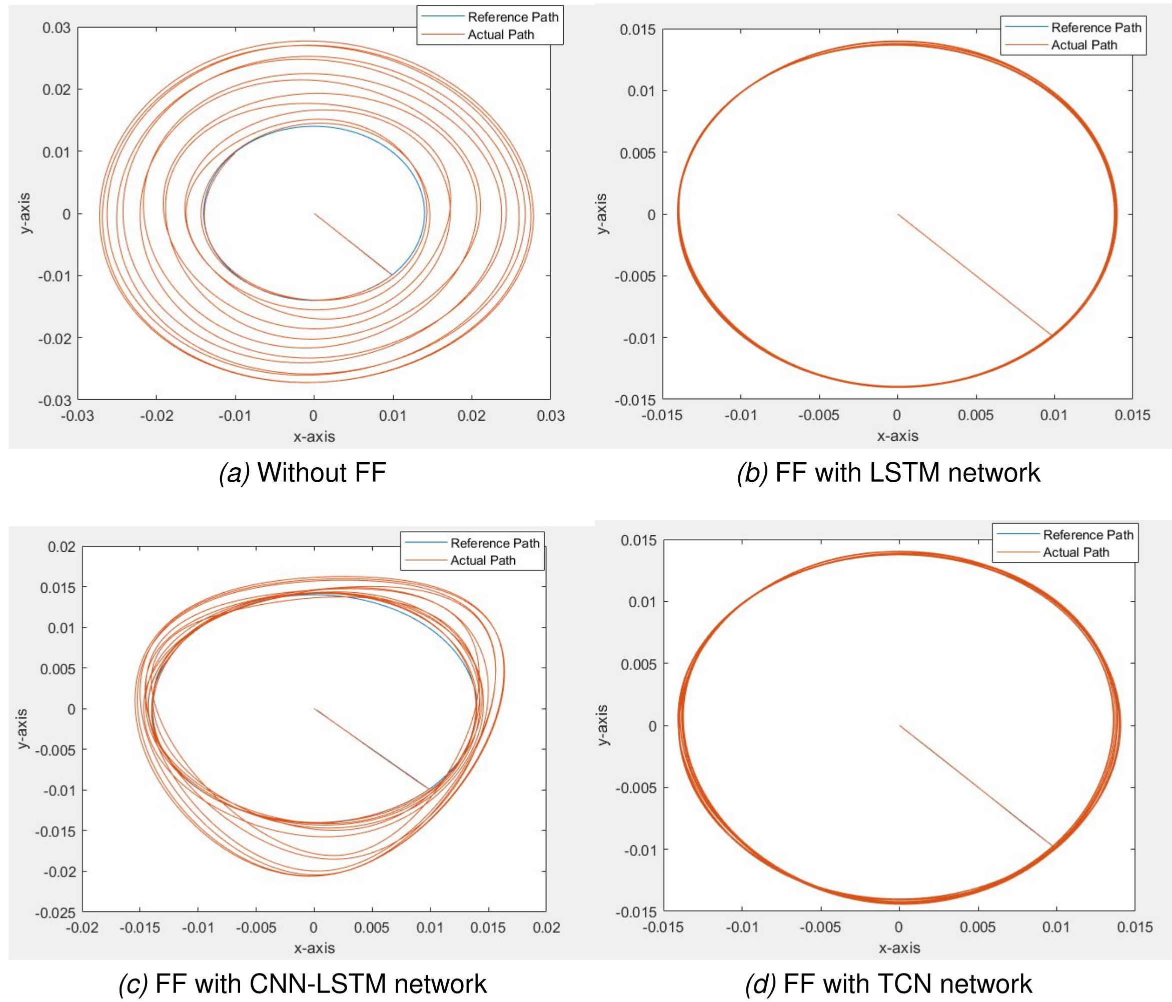


Figure 5.24: Reference Path

The tracking performance of the system are evaluated during different scenarios. First, the system is run without applying any feedforward and it is displayed in figure 5.25a, it can be observed from this figure that at high speeds the tracking performance with just the feedback controller alone is poor. Next, the learned feedforward control from the LSTM network is put to test and its tracking performance is provided in the figure 5.25b. It can be inferred from this figure that the tracking performance of the system has improved when applying the learned feedforward control. The figure 5.25c displays the system performance when CNN-LSTM model based feedforward control is applied, in this case the tracking performance has improved but not as much when using the LSTM network. Finally the figure 5.25d presents the tracking performance of the system when applying TCN model based feedforward control. In this case either, the tracking performance has improved similar to the LSTM network.

To get more insight into the tracking performance, the error between the reference path and the actual tracking of the system is observed and evaluated for all these scenarios. The error plots along x and y direction during each of these scenarios are presented in figure 5.26. The error plot without applying any feedforward resulted in highest error as can be seen from figure 5.26a, the error with CNN-LSTM based feedforward (figure 5.26c) has reduced quite a bit, but it is still not accurate. The error when applying LSTM and TCN based feedforward (figure 5.26b and 5.26d) showed the least error in both the direction compared to other cases. This plot indicates that the tracking performance of the LSTM is slighly better than that of TCN.



(a) Without FF

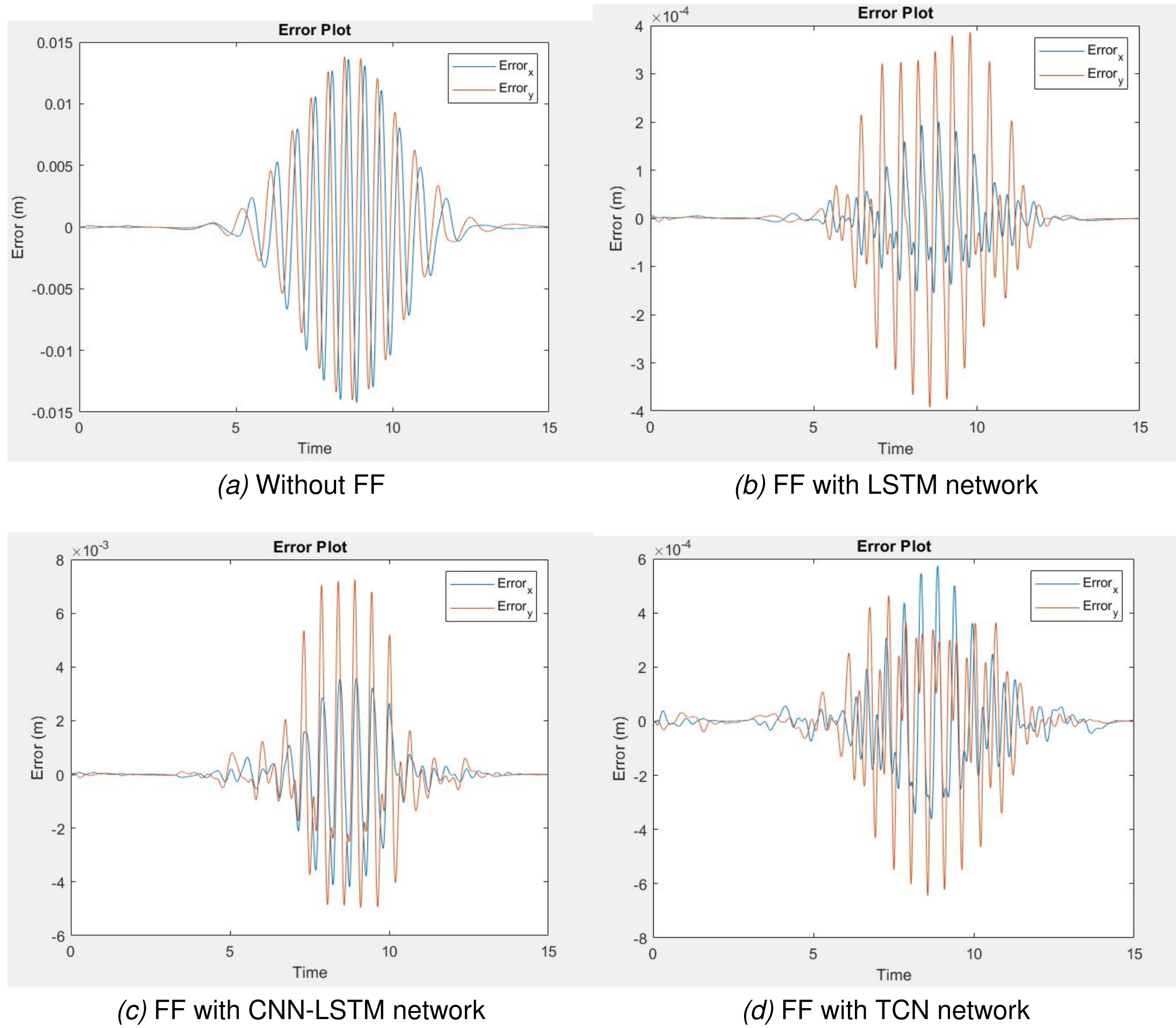
(b) FF with LSTM network

(c) FF with CNN-LSTM network

(d) FF with TCN network

Figure 5.25: Feedforward Evaluation

Further, the average second norm of the error and the infinite norm of the error over 15 seconds are calculated along both x and y direction. The evaluated results are showcased in the table 5.2. The $\|E_x\|_2$ and $\|E_y\|_2$ denotes the average 2-norm of the error over time calculated along x and y direction respectively (the units are in $\frac{m}{\sqrt{s}}$). Similarly, the $\|E_x\|_\infty$ and $\|E_y\|_\infty$ denotes the ∞-norm of the error calculated along x and y direction respectively (the units are in m). The error values are scaled with a factor of 10^{-4} for better readability and the most optimal results are highlighted in red ink. It is found out that the predictions from both the LSTM and TCN network performed the best in increasing the precision of the tracking performance of the manipulator, the LSTM based feedforward slightly yielded better results compared to the TCN based feedforward. The feedforward evaluated from CNN-LSTM did improve the tracking performance but not significantly.



(a) Without FF

(b) FF with LSTM network

(c) FF with CNN-LSTM network

(d) FF with TCN network

Figure 5.26: Error Plot

Error($\times 10^{-4}$)	Without FF	LSTM based FF	CNN-LSTM based FF	TCN based FF
$\|\|E_x\|\|_2$	42.930	**0.457**	10.550	1.257
$\|\|E_y\|\|_2$	42.940	**1.102**	16.648	1.367
$\|\|E_x\|\|_\infty$	142.23	**3.467**	42.146	5.861
$\|\|E_y\|\|_\infty$	140.00	**4.032**	72.34	6.372

Table 5.2: Feedforward Evaluation

Chapter 6

Results from Real-time Environment

This chapter provides the results obtained from the real time environment. The two networks LSTM and TCN that provided optimal results in the simulation environment are considered for learning the inverse dynamics of the manipulator. The architectures of these networks are also briefed in this chapter. An offline feedforward controller is developed for the real time system based on the learned models. The tracking performance of the developed feedforward controller is evaluated.

6.1 Data Visualization

Similar to the simulation environment, it is attempted to visualize the auto-correlation properties of the data collected from the real-time system. Again, the gathered end-effector forces along x and y direction of one of the trajectories is used for this purpose. A lag plot is plotted with a lag of 16, 100 and 500 timesteps, see figure 6.1. It is observed that the strong auto-correlation properties diminish after a lag of 16 timesteps. This makes sense, as the data collected from the real-time system is down-sampled to a 100 Hz. Therefore, the machine learning network developed here, exploits the auto-correlation property in the dataset, within a lag of 16 timesteps.

6.2 Setting up the learning environment

The gathered trajectories from the real time environment are split into training and testing dataset as discussed in chapter 4. The training dataset are further split into 90 percent for training and 10 percent for validating. The networks for training the data collected from real-time system had to be adapted a little bit compared to the simulation environment. These networks are trained in python 3.7 environment as well with GPU compatible tensor-flow and keras libraries. The training process runs

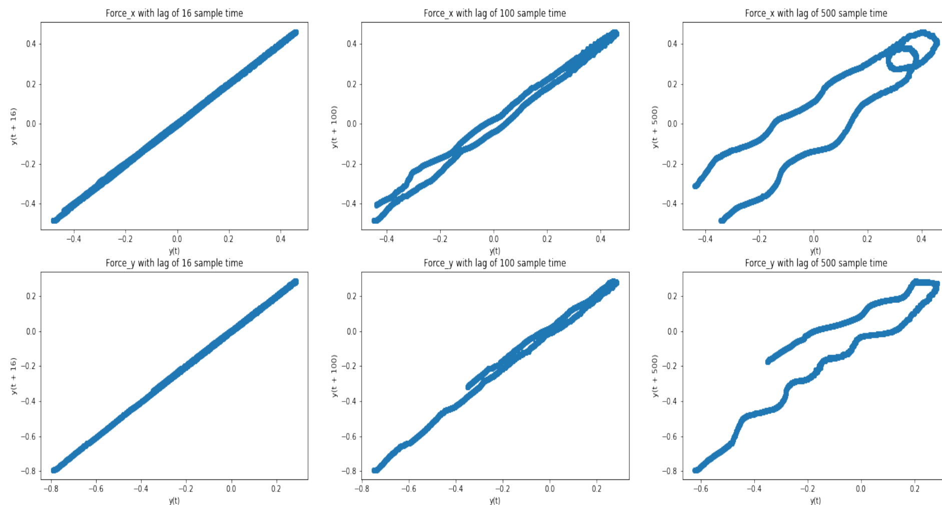


Figure 6.1: Lag Plots of End effector forces along x and y directions

on a NVIDIA GEForce 1050 Ti GPU for efficient computation. The two network architectures are presented and the obtained results are provided in the upcoming sections.

6.3 LSTM Network Design

The LSTM network designed here has a sequential input layer with 63 neurons that corresponds to the 21 positions, velocities and acceleration data gathered from all the 7 joints of the manipulator. Following this two LSTM layers are stacked on top each other, each with 100 neurons. $Tanh$ activation function is used between these layers. Following this, a dropout layer with a value of 0.05 is added to avoid overfitting. Finally, a regression output layer with 2 neurons and a linear activation function is added, which predicts the two end-effector forces. The designed network is shown in figure 6.2.

The network is trained with TBPTT similar to the simulation environment. As indicated in the lag plots, the collected data from real-time show strong auto-correlation property within 16 timesteps, therefore, the TBPTT over 16 timesteps is adopted for this network. For the real time system as well, the MSE is used as loss function for similar reasons discussed in previous chapter. The Adam optimizer and Glorot uniform initializer are adopted for training this network similar to the simulation environment. The network is trained using the 90 percent of the training dataset that are collected, rest 10 percent are used for validation as mentioned earlier. During each epochs, three trajectories from validation dataset are shuffled into training

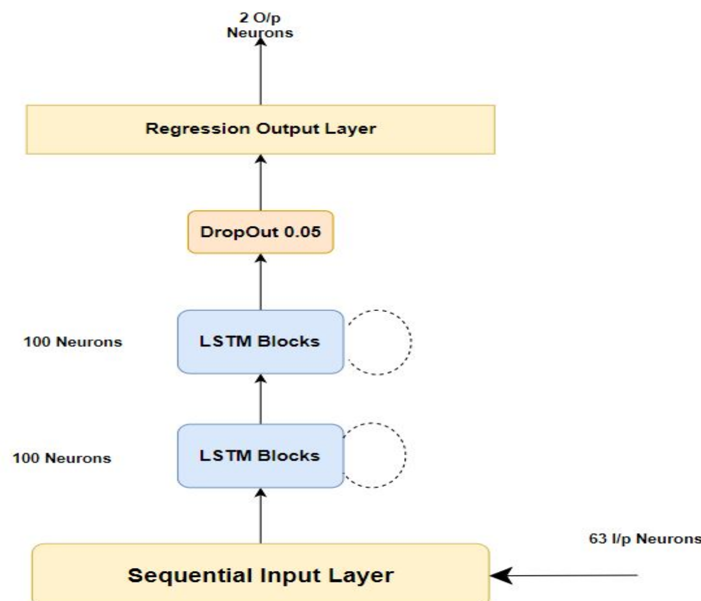


Figure 6.2: LSTM Network Design

dataset. The purpose behind this is similar to the simulation environment, which is to overcome over-fitting and achieve faster convergence rate.

The hyper-parameters are tuned in the similar way it was tuned for simulation environment in previous chapter. From simulation environment it was noticed that for LSTM network more number of maximum epochs resulted in good convergence, thus the maximum number of epochs is set to 1000. The number of hidden neurons is set to 100, since this resulted in an optimal convergence in the simulation environment. Also, increasing the number of hidden neurons results in higher number of parameters to train and the training became computationally expensive with 200 hidden neurons. Thus, the number of hidden neurons are set to a value of 100. The most challenging part is tuning the learning rate. The learning rate obtained for simulation environment did not give expected results, so, many learning rates were experimented between 0.00002 to 0.0000005. The most optimal results were found when setting the learning rate to 0.000001. With these hyperparameters setting the predictions on 10 unseen test dataset are evaluated to observe the accuracy of the inverse dynamics prediction for the real time system.

6.3.1 LSTM Network Experimental Results

The training and testing losses for the designed LSTM network are plotted on logarithmic scale and it is displayed in figure 6.3. It can be observed from this figure that the loss in test dataset fluctuates quite a lot during the first 200 epochs, after which the fluctuations reduces comparatively. These fluctuations could be due to

the varying noises or inaccuracies in the collected dataset. However, it is noticed that the loss function does converge after 900 epochs approximately.

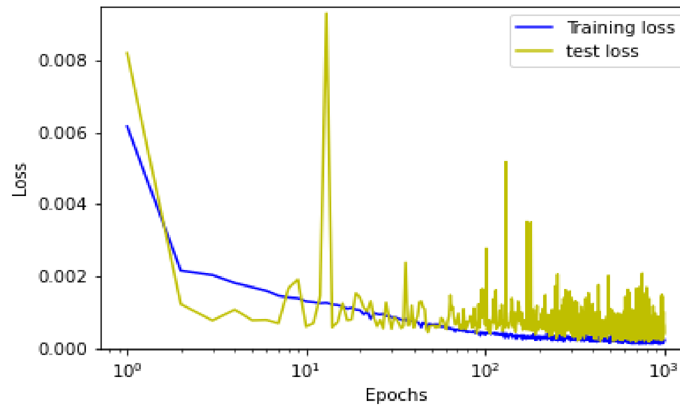


Figure 6.3: LSTM Training and testing loss

The weights during one of the epochs after convergence which showed optimal loss function is used to predict the end effector forces along x and y direction. The most optimal weights were obtained during epoch 983. The predictions are re-scaled to original values by inverse transforming the normalized values. The prediction from this network on two of the unseen test trajectories are shown in figure 6.4.

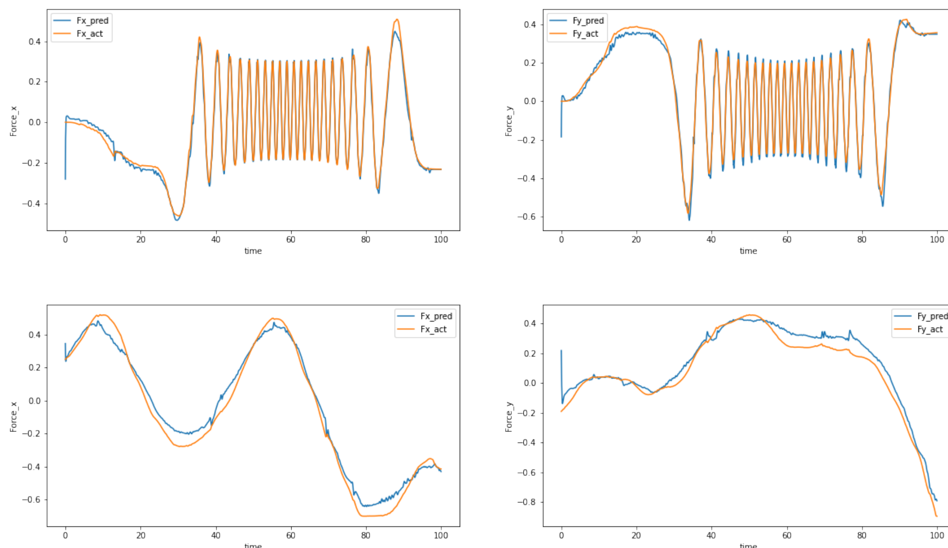


Figure 6.4: LSTMPredictions

It can be observed from this figure that the predictions from LSTM obtains a good fit on unseen trajectories, although, few inaccuracies are still present. Major flaw is that the LSTM network predicts false values at the initial timestep (A high or a low

value that shows up in the prediction), this could be attributed to the normalization that was implemented on the collected dataset. The presence of noises in the data or inaccurate filtering of the data might have an effect while normalizing the data within a range of 0 and 1. This could also be one of the reasons the test loss in figure 6.3 shows a fluctuating effect.

6.4 TCN Network Design

The designed TCN network is shown in figure 6.5. The network again consists of a sequential input layer with 63 neurons, followed by, 4 convolutional layers stacked on top of each other. A $Relu$ activation function is used in between the convolutional layers. Each of the 4 convolutional layers has a filter size of 200 and a kernel window of size 2. A dropout layer with a value of 0.05 is added in between each of these layers to avoid overfitting. The 4 convolutional layers have a dilation factors of 1, 2, 4 and 8 respectively. Finally, a regression output layer with a linear activation function is added, which consists of 2 neurons for predicting the end-effector forces. Again, the choice of the number of convolutional layers, kernel window size and the dilation factors follows from the number of timesteps the network has to look back in time. Since, the dataset collected from real time environment showed strong auto-correlation within 16 timesteps, the network is designed such that the receptive field of the network is about 16 timesteps. With the chosen design choices, i.e, the number of convolutional layers, the kernel window size and dilation factors, the receptive field of the TCN network is 16 timesteps (calculated from the equation 3.4).

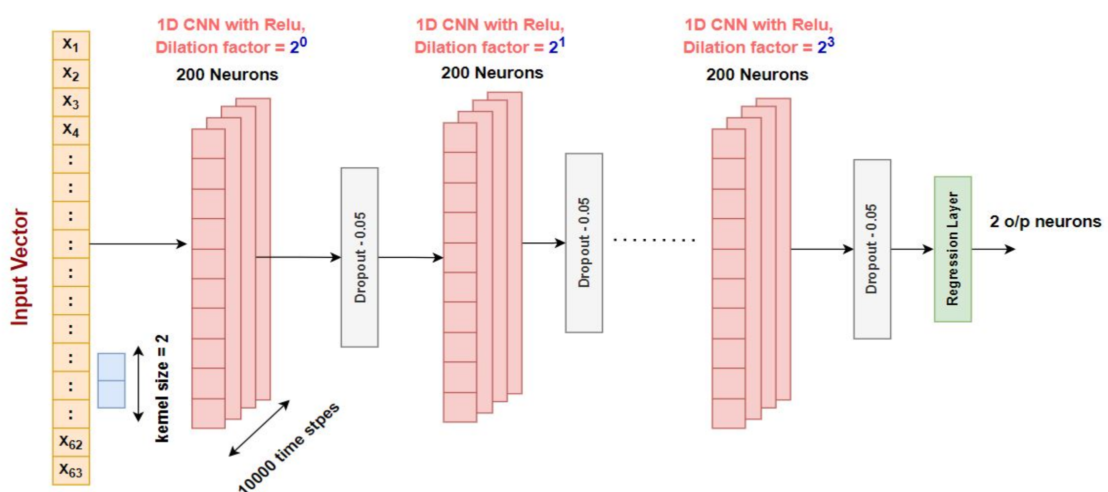


Figure 6.5: TCN Network Design

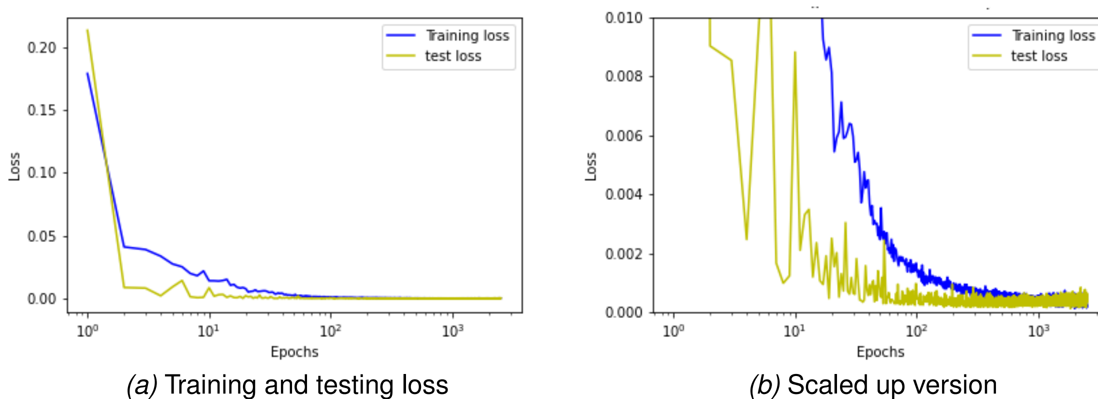
The network is trained with 90 percent of the training dataset and the rest 10

percent is used for validation, similar to the LSTM network. The evaluation metrics chosen is MSE again with adam optimizer and Glorot uniform initializer for the aforementioned reasons.

Again, the hyperparameters of the network are tuned based on simulation environment. The number of hidden neurons is set to 200, since it provided optimal results on simulation environment. For this network, tuning the learning rate and the maximum number of epochs were more challenging. The learning rate of 0.0002 did not provide optimal results like in the simulation environment. So, various learning rate was experimented in the range of 0.0002 to 0.000001. The most optimal results were obtained when the learning rate was set to 0.000045 within first 200 epochs. Next when the maximum number of epochs is set to 1500, it was noticed that the network did not converge to the optimal level as in the case of LSTM network. So, more epochs were considered upto 2500. An optimal level of loss function was obtained after 2100 epochs approximately.

6.4.1 TCN Network Experimental Results

The training and testing losses for the designed TCN network is plotted in a logarithmic scale and it is shown in figure 6.6. The network took more than 2100 epochs to converge to an optimal loss. It is observed that the test loss doesn't fluctuate like in the case of LSTM network. Although, the TCN network took more number of epochs to converge to an optimal level compared to the LSTM network.



(a) Training and testing loss

(b) Scaled up version

Figure 6.6: TCN Training and testing loss

The weights during one of the epochs after convergence are used to predict the end effector forces along x and y direction. The optimal weight with the least loss function on training and testing dataset was found at epoch 2466. The predictions

from the network are re-scaled to original values by inverse transforming the normalized values. The prediction from this network on two of the unseen test trajectories are shown in figure 6.7. It is observed that the predictions from TCN provide a good fit on unseen trajectories. The predictions do not run into the same problem that the predictions from the LSTM network experienced.

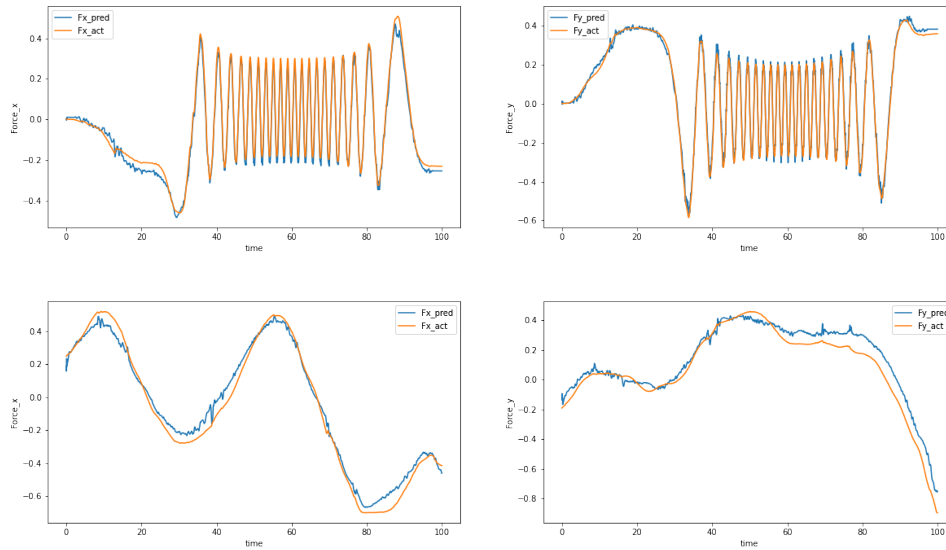


Figure 6.7: TCN Predictions

6.5 Results Comparison

To get more insights into the performance of the network, the optimal loss function (MSE) obtained for both of these networks are compared and tabulated in table 6.1. The loss function is scaled by a factor of 10^{-4} for readability and the most optimal results obtained are highlighted in red ink. It is found out that both the networks performed quite similarly on the training and testing dataset. The presence of varying noises and some inaccurate data could have affected the overall learning process of both the networks compared to the simulation environment. This results in a higher loss function in comparison to the simulation environment.

Mean Squared Error($\times 10^{-4}$)	**LSTM**	**TCN**
Training loss	2.540	**2.5158**
Test loss	**2.542**	2.614

Table 6.1: loss function comparison (Real time environment)

The time taken for training these networks and the number of parameters required to train these networks are also evaluated. The figure 6.8 shows the time

taken per epoch by the two networks. The figure 6.9 displays the number of trainable parameters of both the networks.

Similar to the simulation environment, the time taken by LSTM network per epoch is significantly higher than the time taken per epoch while using TCN. Again this is attributed to the sequential nature of LSTM, since the TBPTT over 16 timesteps is adopted in this case, the training took even more time compared to the simulation environment. The training time taken by TCN relatively didn't increase much even though the number of parameters to train are far higher in this case.

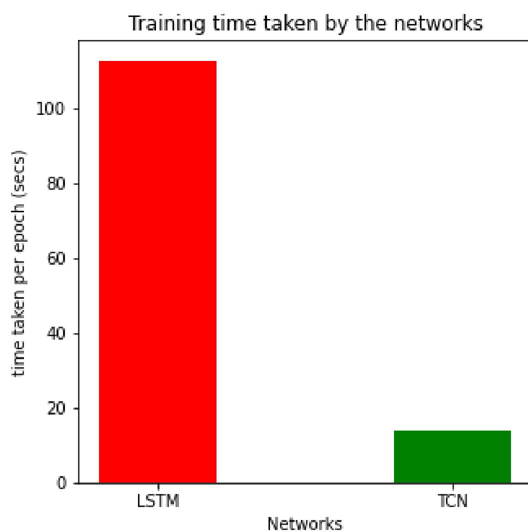


Figure 6.8: Time taken by networks

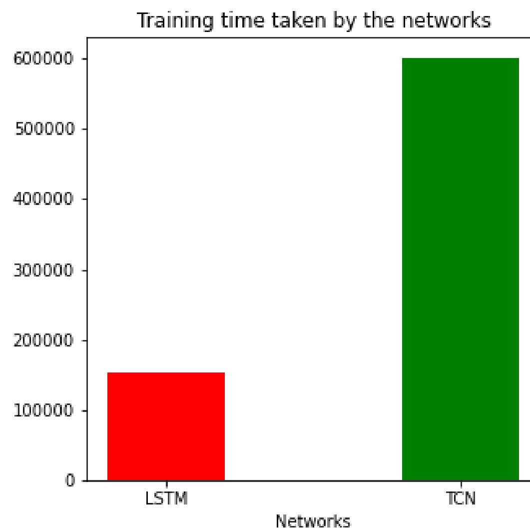


Figure 6.9: parameters to train by the networks

6.6 Feedforward Control and Evaluation

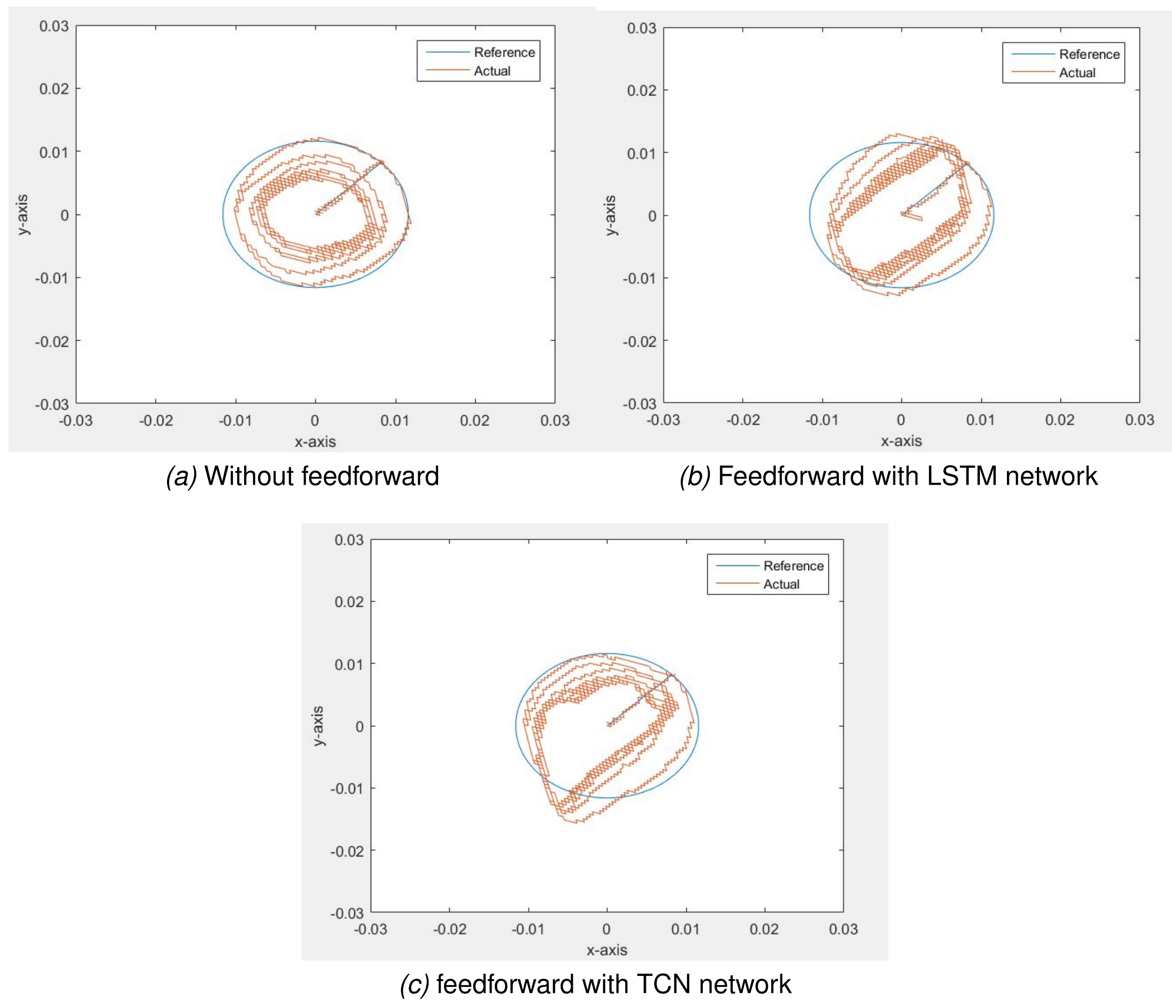
The manipulator is controlled on a MATLAB 2015B, Simulink real time environment. Thus, the learned model is imported into this environment from python for testing. An offline feedforward control is developed based on the predictions made by the learned model for the real time system. The control scheme is similar to the simulation environment, wherein, the predicted feedforward is used to control the manipulator along x and y direction separately.

For evaluating the feedforward, a straight line path with a set point of h_x,h_y = 0.0082m spanning for 21s, followed by a circular trajectory with a frequency of 0.12 Hz (9 rotations in 75s) and a radius of 0.0116m is used as a reference path. The reference path spans a period of 100 seconds at a sampling rate of 1Khz. With the

use of inverse kinematics and the reference path, the rest of the joint positions (19 dependent coordinates, discussed in section 2.1) of the manipulator is obtained. The velocities and the acceleration of each of the joints are obtained by taking the first and the second derivative of the position data respectively. Next the obtained dataset is down-sampled form 1KHz to 100Hz, to match the input tensor of the learned algorithm. The 63 input features collected over 100 seconds at a sampling rate of 100Hz is given as the input to the trained network for predicting the optimal feedforward forces along x and y direction. The forces predicted by the network are controlled forces with a sampling rate of 100Hz. To apply these predictions on the real time environment which is running at a sampling rate of 1KHz, the predicted feedforward forces are interpolated with a spline curve in MATLAB. This essentially up-samples the data from 100Hz to 1KHz which can be used on the real time system.

The tracking performance of the real time system is evaluated during various scenarios. First the tracking performance of the sytem without any feedforward is displayed in figure 6.10a. It can be observed from this figure that the tracking performance is vastly poor at high speeds. It can be noticed that the system tracks a concentric circular trajectory similar to the simulation environment. The tracking performance when applying LSTM based feedforward control is presented in figure 6.10b. It can be observed from this figure that the system tracks almost an elliptical trajectory, this could indicate that the tracking performance along one of the directions might have improved. One more observation that can be made from this figure is that there is a minor error during the initial point, this could be attributed to the inaccuracy of the prediction from LSTM network which was discussed in section 6.3.1. Similarly, the tracking performance of the system while applying a TCN based network is presented in figure 6.10c. In this case, the system seems to be following an elliptical trajectory as well but there are higher magnitude errors as it can be observed from this figure. The results show that there is a slight improvement in the tracking performance when applying the learned feedforward control. However, the improvements are not significant enough for a precise manipulation of the robot.

To verify the improvements in the tracking performance, the tracing along x and y directions are analyzed separately. The figure 6.11 presents the tracking performance along x axis. It can be observed from the figure 6.11a, that without feedforward control, there is a steady offset between the reference path and the actual path the system traces. This is usually due to the presence of coulomb friction in the system. When applying LSTM based feeforward control (figure 6.11b), the improvement in the tracking performance is clearly visible with the offset error vanishing to a good extend. There are visible errors in the magnitude of the tracking performance,



(a) Without feedforward

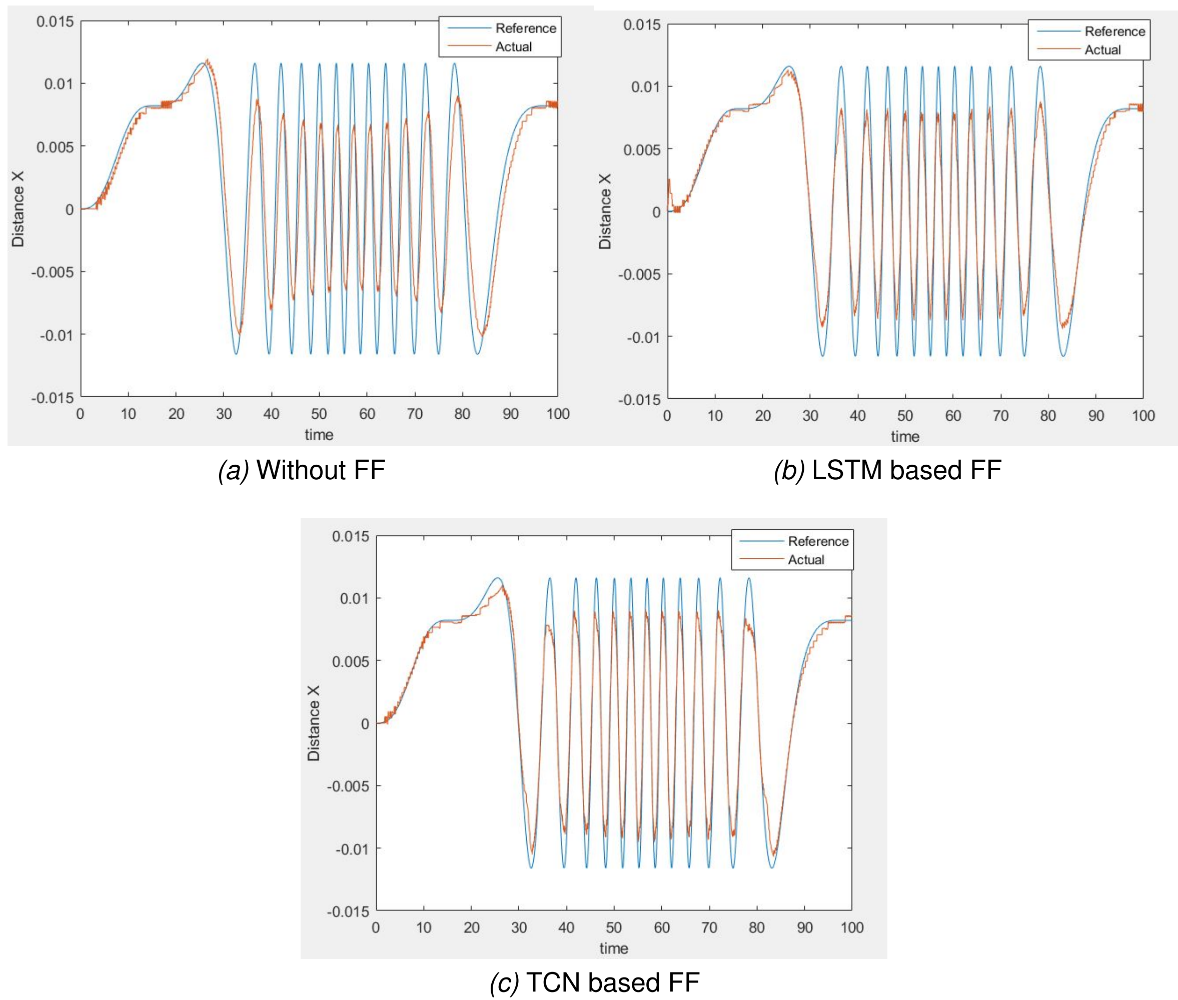
(b) Feedforward with LSTM network

(c) feedforward with TCN network

Figure 6.10: Feedforward evaluation on real time system

however, compared to the case without feedforward the magnitude has improved a bit. Similar observations can be made when applying the TCN based feedforward control, see figure 6.11c. The offset error has diminished and clear improvements in the magnitude of the tracing can be seen compared to the case without any feedforward.

The figure 6.12 presents the tracking performance along y axis during three scenarios. Again in the y direction as well, there are some offset and magnitude errors from the reference path when the feedforward control is not applied, see figure 6.12a. When applying LSTM based feedforward, the offset error has reduced but not quite as much as along the x direction, however, the error in magnitude has improved compared to the case without feedforward and also compared to x direction. This behaviour shows up as an elliptical shape in figure 6.10b. In case of TCN based feedforward shown in figure 6.12c, the tracking performance is poor compared to the other two cases. The predicted forces by this network seems to induce higher magnitude error, which makes the end effector go outside the bounds of the



(a) Without FF *(b)* LSTM based FF

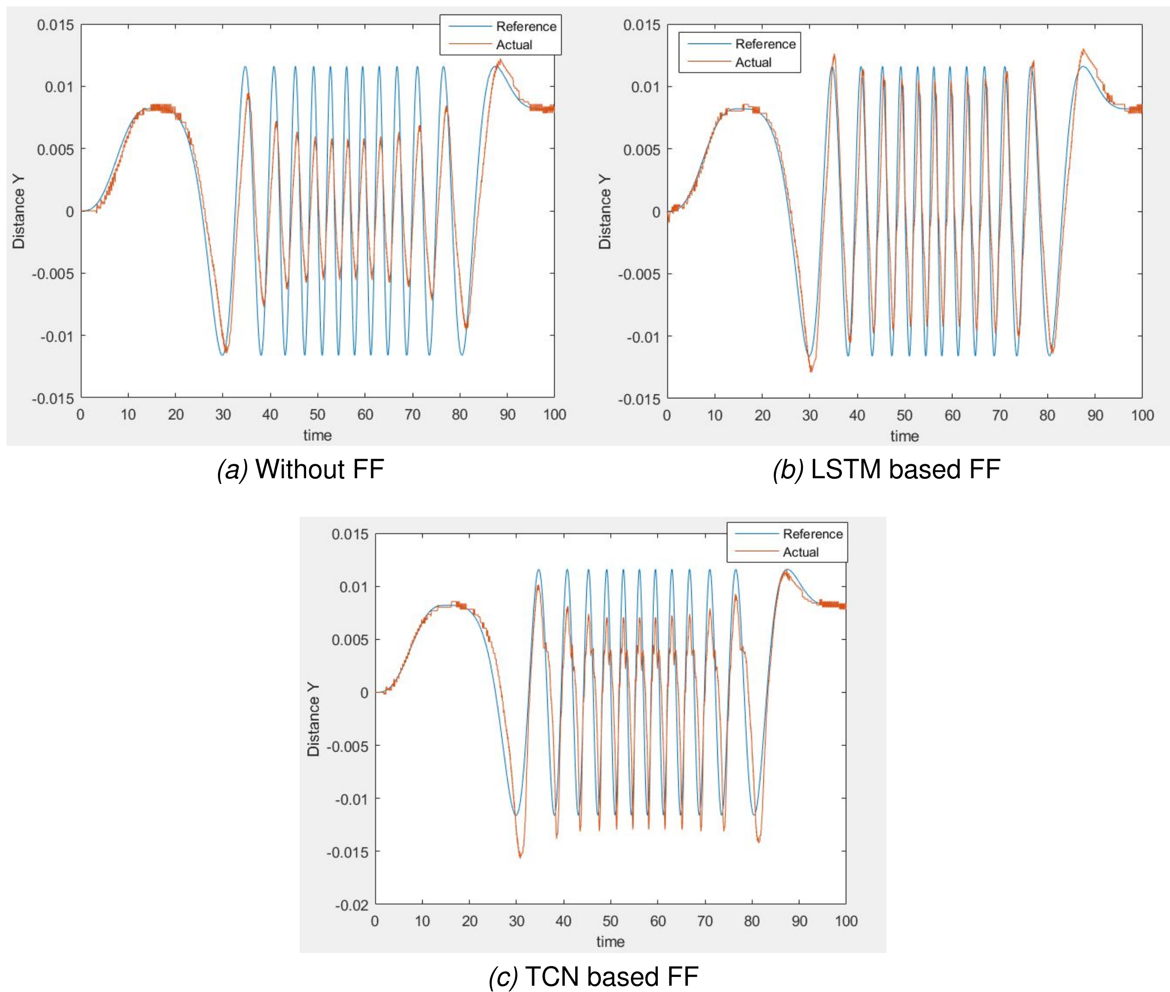
(c) TCN based FF

Figure 6.11: Tracking performance along x axis

defined reference trajectory.

To get more discernment on the system performance, the total force, feedback force and the predicted feedforward forces for each of the three scenarios are analysed and compared. First case, when the feedforward control is not applied, the total force injected into the system and the feedback force are equal, and it is displayed in figure 6.13. Ideally, when applying a feedforward control that is exactly the inverse of the system, the feedback controller applies no forces to the system. This property cannot be realised in the real world, however, with the application of feedforward control the force applied by feedback controller diminishes to account for residual errors alone, if the feedforward control is close to the inverse of the system. In this study, it is attempted to analyse this property of the learned feedforward controller.

The second case, when LSTM based feedforward controller is applied to the manipulator is analysed next. The system performance during this scenario is presented in figure 6.14. The figure 6.14a displays the predicted feedforward forces from LSTM network that is applied to the system. The figure 6.14b presents the



(a) Without FF

(b) LSTM based FF

(c) TCN based FF

Figure 6.12: Tracking performance along y axis

feedback forces applied on to the system. It can be inferred from this figure that the force applied by the feedback controller has diminished quite a lot along the x direction. Similarly along y direction the applied feedback force has diminished but not quite as much as along the x direction. This indicates that the LSTM based network has learned the inverse dynamics of the manipulator to a certain extend. This can be further verified with the total forces(summation of feedback and feedforward forces) applied on to the system which is presented in figure 6.14c. It can be observed that the total force injected into the system has improved in this scenario.

As the final case, the performance of TCN based feedforward controller is analysed next. The system performance during this scenario is presented in figure 6.15. The figure 6.15a displays the predicted feedforward forces during this scenario. The figure 6.15b provides the feedback force applied by the controller during this scenario. From this figure, it can be inferred that the feedback forces applied along x direction has diminished a lot similar to the previous case. However, the feedback forces along y direction seems to be increasing which indicates that this network

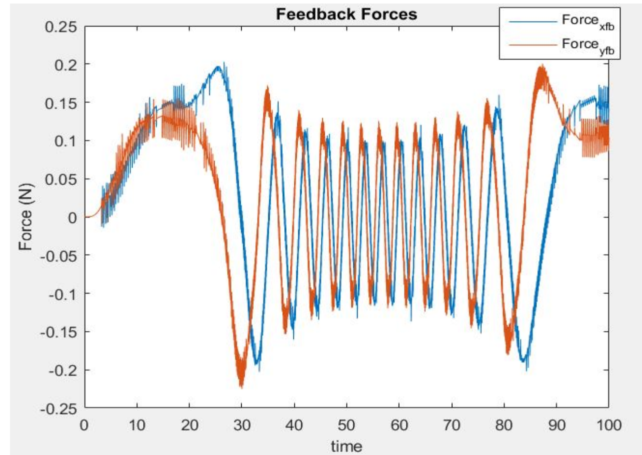
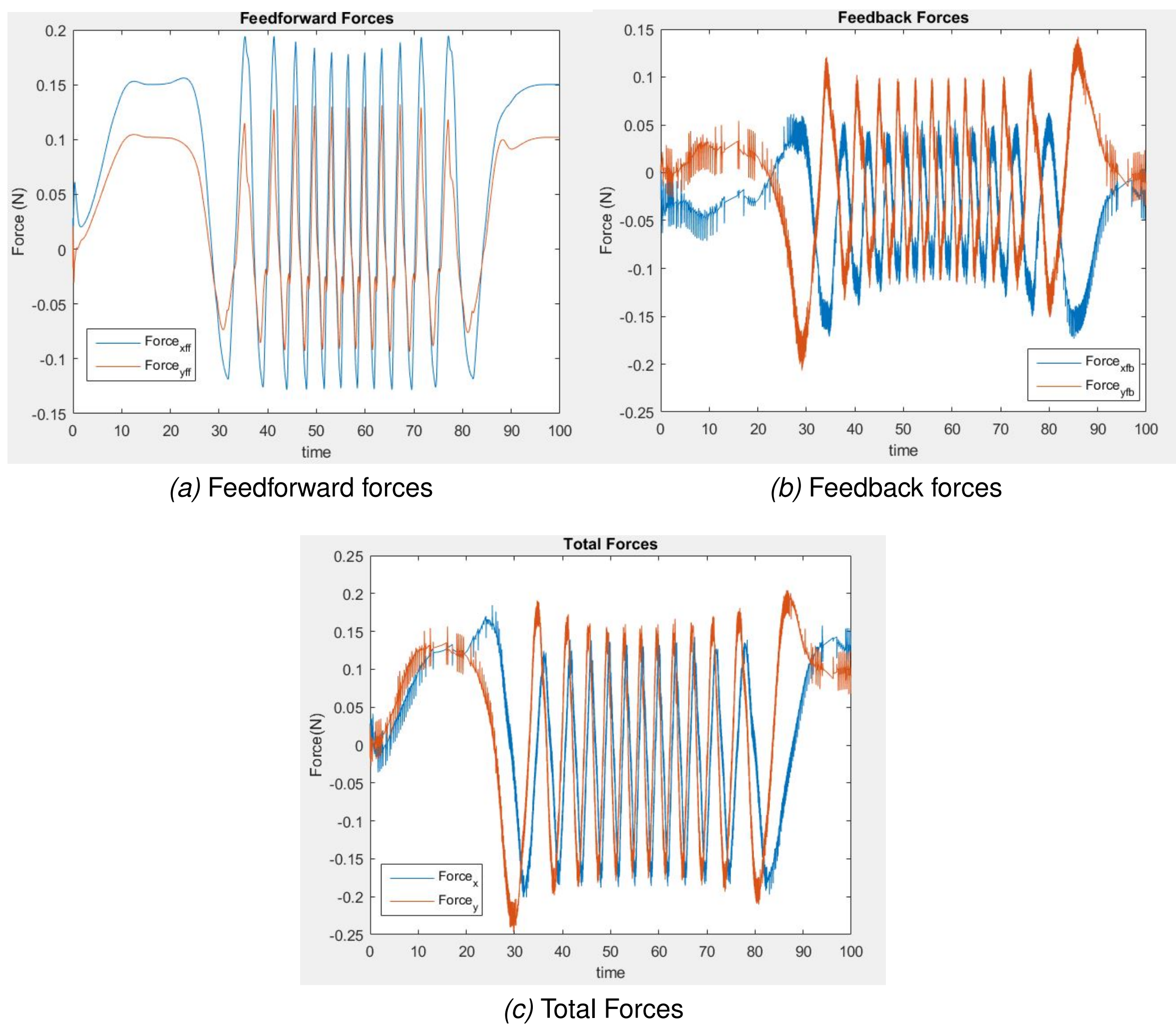


Figure 6.13: System Performance with feedback controller



(a) Feedforward forces

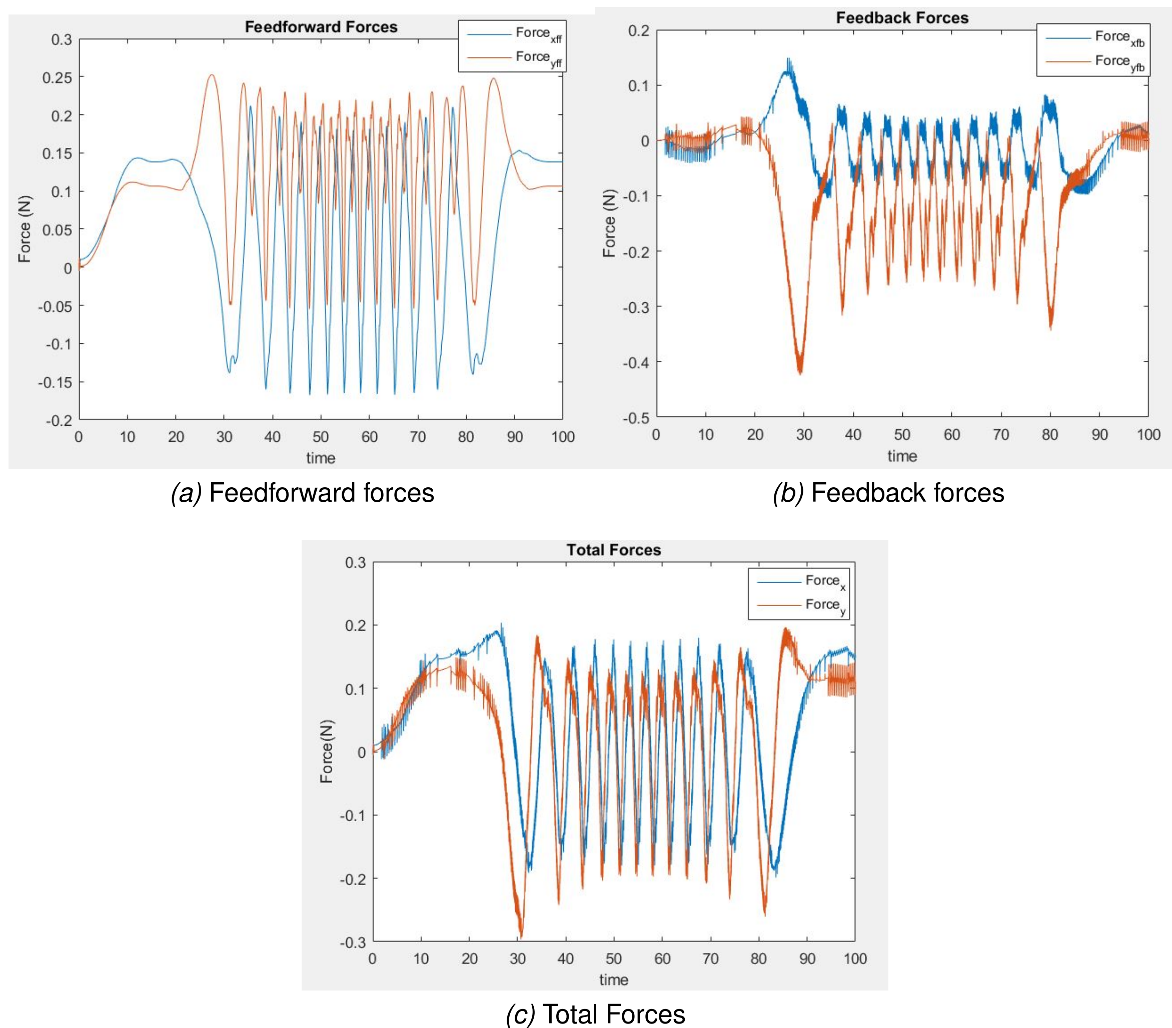
(b) Feedback forces

(c) Total Forces

Figure 6.14: System Performance with LSTM based feedforward control

has not learned the inverse dynamics along this direction quite well. Again, this can be verified from the total forces applied into the system during this scenario which

is shown in figure 6.15c. It can be noticed from this figure that there is a significant improvement in the forces applied along x direction but not along y direction.



(a) Feedforward forces

(b) Feedback forces

(c) Total Forces

Figure 6.15: System Performance with TCN based feedforward control

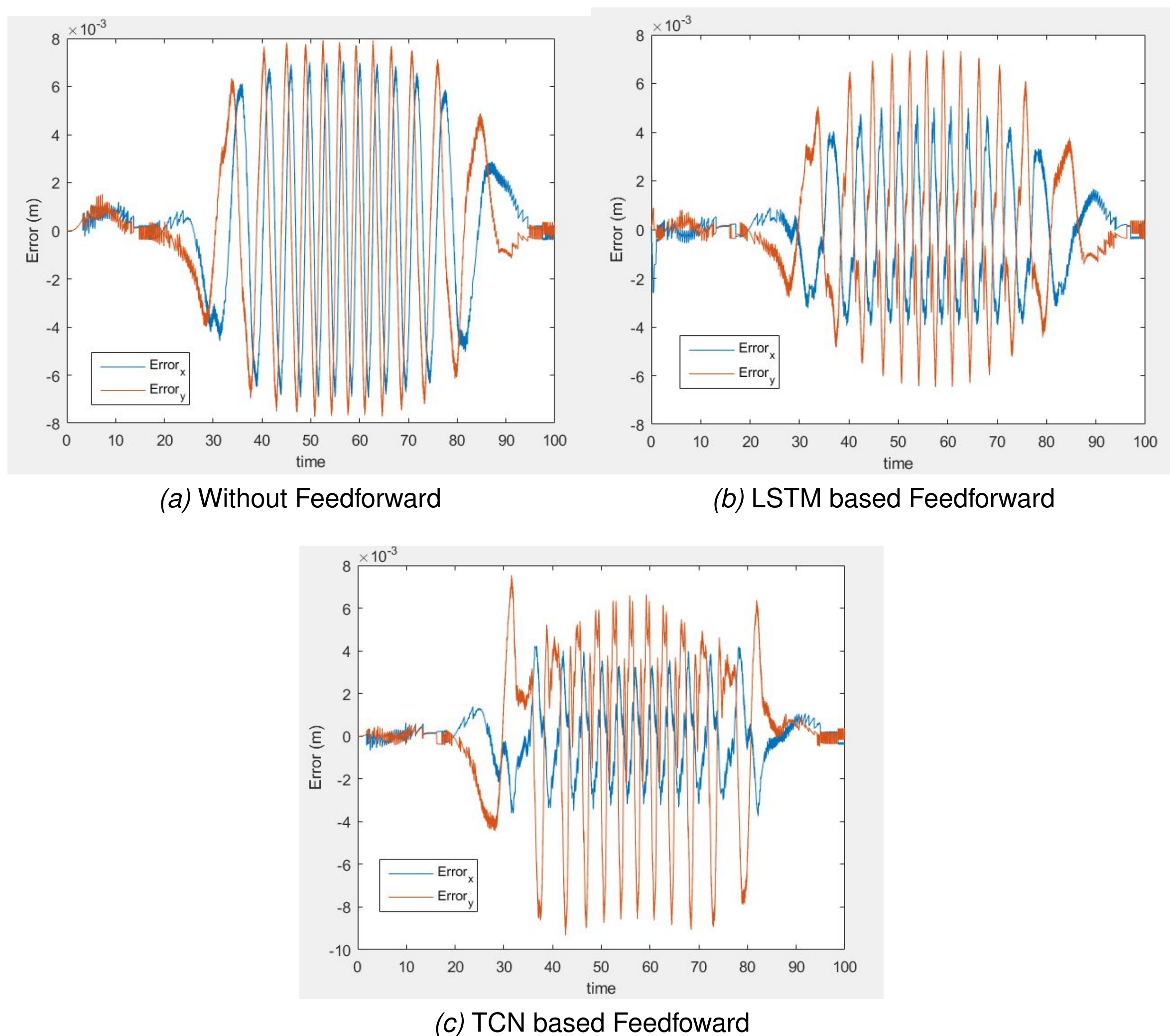
The observations from these results shows that the overall tracking performance when using LSTM based feedforward controller has improved in both the directions. The TCN based feedforward shows improvements along x direction but not along y direction. To get more insights into the tracking performance in these three scenarios, the second norm and the infinity norm of the error between the reference path and the actual traced path in both the directions are evaluated similar to the simulation environment discussed in previous chapter, see table 6.2. The values are scaled by a factor of 10^{-3} for better readability and the most optimal results are highlighted in red ink.

It can be verified from this table that the LSTM based feedforward provides optimal results along both the directions while the feedforward based on TCN performs well along x direction but not along y direction. This inference can be further verified

Error$(\times 10^{-3})$	**Without FF**	**LSTM based FF**	**TCN based FF**
$\|\|E_x\|\|_2$	3.585	1.985	**1.742**
$\|\|E_y\|\|_2$	3.517	**2.624**	3.446
$\|\|E_x\|\|_\infty$	7.82	5.15	**4.736**
$\|\|E_y\|\|_\infty$	8.24	**6.952**	9.361

Table 6.2: Feedforward Evaluation from real time system

from the error plots during these three scenarios which is shown in the figure 6.16. Similar results were obtained when evaluating a random trajectory and it is briefed in appendix D.



(a) Without Feedforward

(b) LSTM based Feedforward

(c) TCN based Feedfoward

Figure 6.16: Error Plot during the three scenarios

With the results obtained it is evident that the developed networks are able to learn the inverse dynamics of the real time system, although the predictions from both the networks could still be improved. Possible reasons the predictions from the network do not improve the tracking performance significantly could be due to the

fact that the 60 trajectories used for training the networks are not enough to learn the complete inverse dynamics of the system. It may also be the case that there are some biases in the collected dataset which hampers the learning process, this might be one of the reasons why significant improvements can be seen along x direction for both the networks but not along y direction especially in the case of TCN network. As mentioned earlier, the collected dataset from real time system is a subject of noise and inaccuracies. These inaccuracies can again affect the overall learning process, so a better filtering technique could be adopted to improve the learning process. Due to the redundancy in the system, only two encoders are used in the forward kinematics solution to obtain other joint positions. From Bendersen's [5] study this has proven to cause inaccuracies in the evaluation of joint positions, and this might also be one of the reasons the inverse dynamics of the system is not learnt accurately. One more plausible reason could be that, while down-sampling the data essential dynamics are lost, but chances of this happening are very mild.

Chapter 7

Conclusions and recommendations

7.1 Conclusions

The scope of this study was to develop a data-driven feedforward controller for a redundantly actuated manipulator with the use of machine learning techniques. In light to this, various machine learning algorithms were investigated. Ultimately, the aim with the feedforward controller was to improve the tracking performance of the manipulator. For this purpose three neural network based learning model were developed, and a comparison study between these networks were presented for the simulation environment. Two networks that performed the best in the simulation environment, were developed for the real time system, and the performance of these networks were compared and evaluated.

It is found out that the developed networks are capable of learning the inverse dynamics of the system. In the simulation environment the LSTM network and the TCN network provided optimal results. The clear advantage of TCN over LSTM network was its faster training time and the possibility of parallel computation while back propagation. However in most of the cases, the LSTM network took less number of iterations to converge to an optimal level. The CNN-LSTM network did not provide optimal results and it could further be fine tuned. An offline feedforward control was developed using the learned model, and the improvements in the tracking performance of the system was clearly visible in the simulation environment.

For the real time environment the developed networks (LSTM and TCN) were able to predict the inverse dynamics to an optimal degree. The evaluation of the feedforward control making use of the learned model revealed that there were some noticable improvements in the tracking performance of the system, although precise tracking performance could not be achieved like in the case of simulation environment. This is attributed to the nature of the dataset collected from the manipulator. It is expected that with more number of trajectories and accurate dataset the networks could predict the inverse dynamics better and thus, could improve the tracking per-

formance of the system.

7.2 Future Recommendations

During this study, it was noticed that the limited encoder resolution had a significant negative impact on the accuracy of the manipulator. This deteriorates the accuracy of the collected dataset as well while training, which is not desirable. Hence, it is recommended that the accuracy of the collected dataset is improved. Two ways of improving the accuracy of the dataset is foreseen. The first would be to increase the resolution of the encoder, the second would be to make use of the third encoder (redundant encoder) present in the test setup to reduce the quantization noises.

Another recommendation is to collect even more trajectories for training the networks. Although, care has to be taken while collecting these datasets in order to be fair and avoid any sort of biases while training.

It has to be stressed that the neural networks are always subjects of improvements. More experiments can be done with different hyper-parameter settings to improve the prediction accuracy of the networks. Investigations can be made about the feasibility of training a CNN-LSTM network on the real time environment.

An interesting direction of future research could be that, instead of using end effector forces as labels for training, one can directly use the three joint torques for prediction and to apply the feedforward control. This could come in handy, since the network could be capable of learning the redundancy resolution also along with the inverse dynamics of the system.

Bibliography

[1] R. Cornelissen, “Novel System Level Optimisation Method for Manipulators with Flexure Joints Applied to a 2-DoF 3RRR PKM,” Master’s thesis, University of Twente, 2019.

[2] “IBM Cloud Education - What are Recurrent Neural Networks.” [Online]. Available: https://www.ibm.com/cloud/learn/recurrent-neural-networks

[3] “Wavenet: A generative model for raw audio.” [Online]. Available: https://deepmind.com/blog/article/wavenet-generative-model-raw-audio

[4] A. Sridhar, “Workspace Optimization and Motion Control of a redundant flexural 2DOF 3RRR Parallel Kinematic Manipulator,” Master’s thesis, University of Twente, 2020.

[5] D. Berendsen, “2DOF Planar redundantly actuated parallel kinematic manipulator with flexure joints optimization and control,” Master’s thesis, University of Twente, 2021.

[6] M. Boerlage, R. Tousain, and M. Steinbuch, “Reference trajectory relevant jerk derivative feedforward control for motion systems,” 11 2021.

[7] E. Mollick, “Establishing moore’s law,” *IEEE Annals of the History of Computing*, vol. 28, no. 3, pp. 62–75, 2006.

[8] F. Lange, J. Langwald, and G. Hirzinger, “Predictive feedforward control for high speed tracking tasks,” in *1999 European Control Conference (ECC)*, 1999, pp. 4537–4542.

[9] R. Camoriano, S. Traversaro, L. Rosasco, G. Metta, and F. Nori, “Incremental semiparametric inverse dynamics learning,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 544–550.

[10] N. Liu, L. Li, B. Hao, L. Yang, T. Hu, T. Xue, S. Wang, and X. Shao, “Semiparametric deep learning manipulator inverse dynamics modeling method for smart city and industrial applications,” *Complexity*, vol. 2020, 2020.

[11] Y. Anzai, *Pattern recognition and machine learning*. Elsevier, 2012.

[12] O. A. Haidi, “Machine Learning Applications to Robot Control,” Ph.D. dissertation, University of California, Berkeley, 2018.

[13] K. Hitzler, F. Meier, S. Schaal, and T. Asfour, “Learning and adaptation of inverse dynamics models: A comparison,” in *2019 IEEE-RAS 19th International Conference on Humanoid Robots (Humanoids)*, 2019, pp. 491–498.

[14] C. A. Bailer-Jones, D. J. MacKay, and P. J. Withers, “A recurrent neural network for modelling dynamical systems,” *network: computation in neural systems*, vol. 9, no. 4, p. 531, 1998.

[15] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[16] N. Liu, L. Li, B. Hao, L. Yang, T. Hu, T. Xue, and S. Wang, “Modeling and simulation of robot inverse dynamics using lstm-based deep learning algorithm for smart cities and factories,” *IEEE Access*, vol. 7, pp. 173 989–173 998, 2019.

[17] A. Müller and T. Hufnagel, “Model-based control of redundantly actuated parallel manipulators in redundant coordinates,” *Robotics and Autonomous Systems*, vol. 60, no. 4, pp. 563–571, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188901100217X

[18] L. Yonghong, WuMin, and S. Jinhua, “Optimal repetitive control based on two-dimensional hybrid model,” in *2007 Chinese Control Conference*, 2007, pp. 89–92.

[19] J. van Zundert and T. Oomen, “On inversion-based approaches for feedforward and ilc,” *Mechatronics*, vol. 50, pp. 282–291, Apr. 2018.

[20] N. T. A. Singh and A. Sharma, “A review of supervised machine learning algorithms,” *IEEE/nternational Conference on Computing for Sustainable Global Development (INDIACom)*, 2016.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[22] M. Qiao, S. Yan, X. Tang, and C. Xu, “Deep convolutional and LSTM recurrent neural networks for rolling bearing fault diagnosis under strong noises and variable loads,” *IEEE Access*, vol. 8, pp. 66 257–66 269, 2020.

[23] V. K. Shaojie Bai, J. Zico Kolter, “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling,” 2019, arXiv:1803.01271v2[cs.LG].

[24] D. D’informatique, E. N, P. Esent, E. Au, F. Gers, P. Hersch, P. Esident, and P. Frasconi, “Long short-term memory in recurrent neural networks,” 05 2001.

[25] L. Zhang, Z. Shi, J. Han, A. Shi, and D. Ma, “Furcanext: End-to-end monaural speech separation with dynamic gated dilated temporal convolutional networks,” in *MultiMedia Modeling*, Y. M. Ro, W.-H. Cheng, J. Kim, W.-T. Chu, P. Cui, J.-W. Choi, M.-C. Hu, and W. De Neve, Eds. Cham: Springer International Publishing, 2020, pp. 653–665.

[26] “Temporal convolutional networks for sequence modeling.” [Online]. Available: https://dida.do/blog/temporal-convolutional-networks-for-sequence-modeling

[27] ““natural” or periodic interpolating cubic spline curve.” [Online]. Available: https://www.mathworks.com/help/curvefit/cscvn.html

[28] “Butterworth filter design.” [Online]. Available: https://www.mathworks.com/help/signal/ref/butter.html

[29] “Zero-phase digital filtering.” [Online]. Available: https://www.mathworks.com/help/signal/ref/filtfilt.html

[30] C. Aicher, N. J. Foti, and E. B. Fox, “Adaptively truncating backpropagation through time to control gradient bias,” *ArXiv*, vol. abs/1905.07473, 2019.

[31] S. Bock and M. Weiß, “A proof of local convergence for the adam optimizer,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[32] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

Appendix A

Feedforward Neural Networks

Neural Networks in machine learning are inspired by human brain, mimicking the way a biological neurons signal to one another. One such artificial neuron is displayed in figure A.1. Here, the input vector $X = [x_1, x_2, x_3...x_t]$ is multiplied with a weight vector $W = [w_1, w_2, w_3...w_t]$ and is passed as an input to an activation function as shown. The activation function can be Sigmoid or Tanh or any other activation functions according to the users wish.

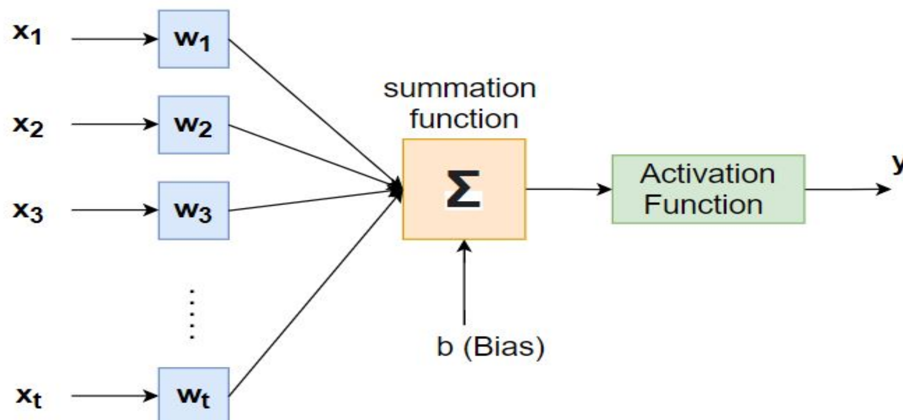


Figure A.1: Artificial Neuron

These artificial neurons are interconnected to form a deep fully connected feed-forward neural network as shown in figure A.2. The network learns through forward propagation and backward propagation. Forward propagation refers to the calculation and storage of intermediate variables including the outputs. A typical forward propagation can be expressed by the following equations,

$$\begin{aligned} \mathbf{h} &= f(\mathbf{W}^{(1)}\mathbf{X}) \\ \mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} \end{aligned} \tag{A.1}$$

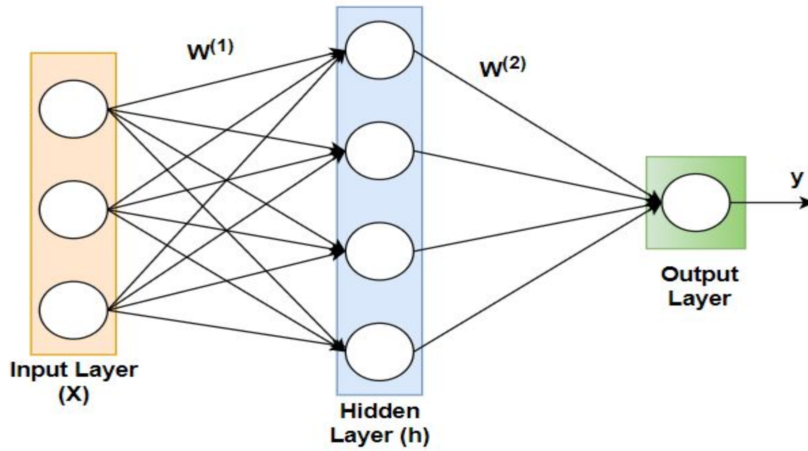


Figure A.2: Feeforward neural network

Backward propagation in FNN is the method of calculating the gradients of the parameters of the network and updating them to suffice the learning process. The equation for back propagation of weights at the input layer with a single hidden layer can be found by chain rule as follows,

$$\frac{\partial C}{\partial w^{(1)}} = \frac{\partial C}{\partial Y}\frac{\partial Y}{\partial h}\frac{\partial h}{\partial w^{(1)}} \tag{A.2}$$

where, C is the cost function or loss function defined according to the user. These derivatives are updates into the weights during the next iteration to facilitate learning.

Appendix B

Back Propagation Through Time

The forward propagation in RNNs are relatively straight forward and they were presented in equation 2.11. The back propagation in RNNs have more complications since the weights are to be updated throughout the previous time steps as well. Similar to FNNs, the gradient descent algorithm is used to update the weights of each layers in RNNs. The gradients of the hidden layers at time t can be computed from chain rule as follows,

$$\frac{\partial C}{\partial h^{(t)}} = \frac{\partial h^{(t+1)}}{\partial h^{(t)}} \frac{\partial C}{\partial h^{(t+1)}} + \frac{\partial Y^{(t)}}{\partial h^{(t)}} \frac{\partial C}{\partial Y^{(t)}} \tag{B.1}$$

where, C is again the cost function that can be defined according to the user. The gradients of the rest of the weights of the network can be computed similarly as well and they are displayed in the following equation

$$\begin{aligned} \frac{\partial C}{\partial V} &= \sum_t \frac{\partial C}{\partial Y^{(t)}} h^{(t)} \\ \frac{\partial C}{\partial W} &= \sum_t \frac{\partial C}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W^{(t)}} \\ \frac{\partial C}{\partial U} &= \sum_t \frac{\partial C}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial U^{(t)}} \end{aligned} \tag{B.2}$$

The gradients obtained are updated in the next iteration for learning. Since, the back propagation algorithm runs backward in time in case of RNNs, it is often called as Back propagation through time.

Appendix C

Kinematics of the Manipulator

The kinematics describes all the deflection angles and rigid body coordinates in terms of the end effector coordinates or the 2 DOFs $\vec{q}_i = [x_{ee}, y_{ee}]$. Figure C.1 showcases shoulder and elbow links of a single chain in the manipulator.

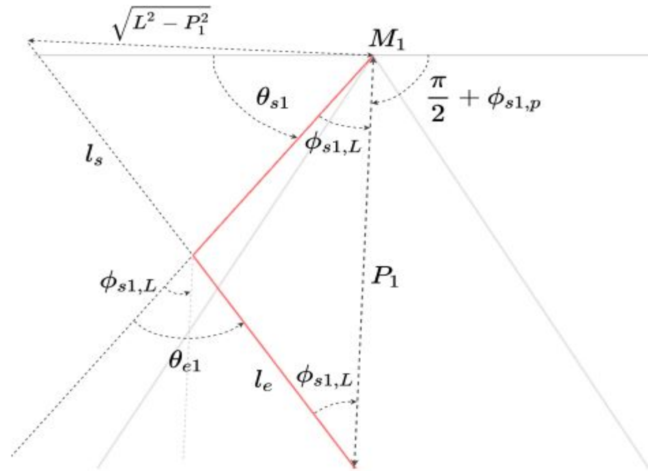


Figure C.1: Single Kinematic Chain [4]

The positions where motors are mounted are given by M_i:

$$M_1 = [0, R]^T, M_2 = [\frac{\sqrt{3}R}{2}, \frac{-R}{2}]^T, M_3 = [\frac{-\sqrt{3}R}{2}, \frac{-R}{2}]^T \quad (C.1)$$

The distance from the end effector position to the motor mounts are given by P_i for $i = [1, 2, 3]$. Based on the representation of the figure C.1, given the end effector position the shoulder angles can be found out as follows,

$$
\begin{aligned}
P_i &= \sqrt{(x_{ee} - M_{i,x})^2 + (y_{ee} - M_{i,y})^2)} \\
\theta_{s1} &= \frac{\pi}{2} + atan(\frac{x_{ee} - M_{1,x}}{-y_{ee} + M_{1,y}}) - atan(\frac{\sqrt{L^2 - P_1^2}}{P_1}) \\
\theta_{s2} &= atan(\frac{x_{ee} - M_{2,x}}{-y_{ee} + M_{2,y}}) - atan(\frac{\sqrt{L^2 - P_2^2}}{P_2}) \\
\theta_{s2} &= atan(\frac{x_{ee} - M_{3,x}}{-y_{ee} + M_{3,y}}) - atan(\frac{\sqrt{L^2 - P_3^2}}{P_3})
\end{aligned}
\tag{C.2}
$$

The elbow angles and wrist angle can be found from the end effector positions as follows,

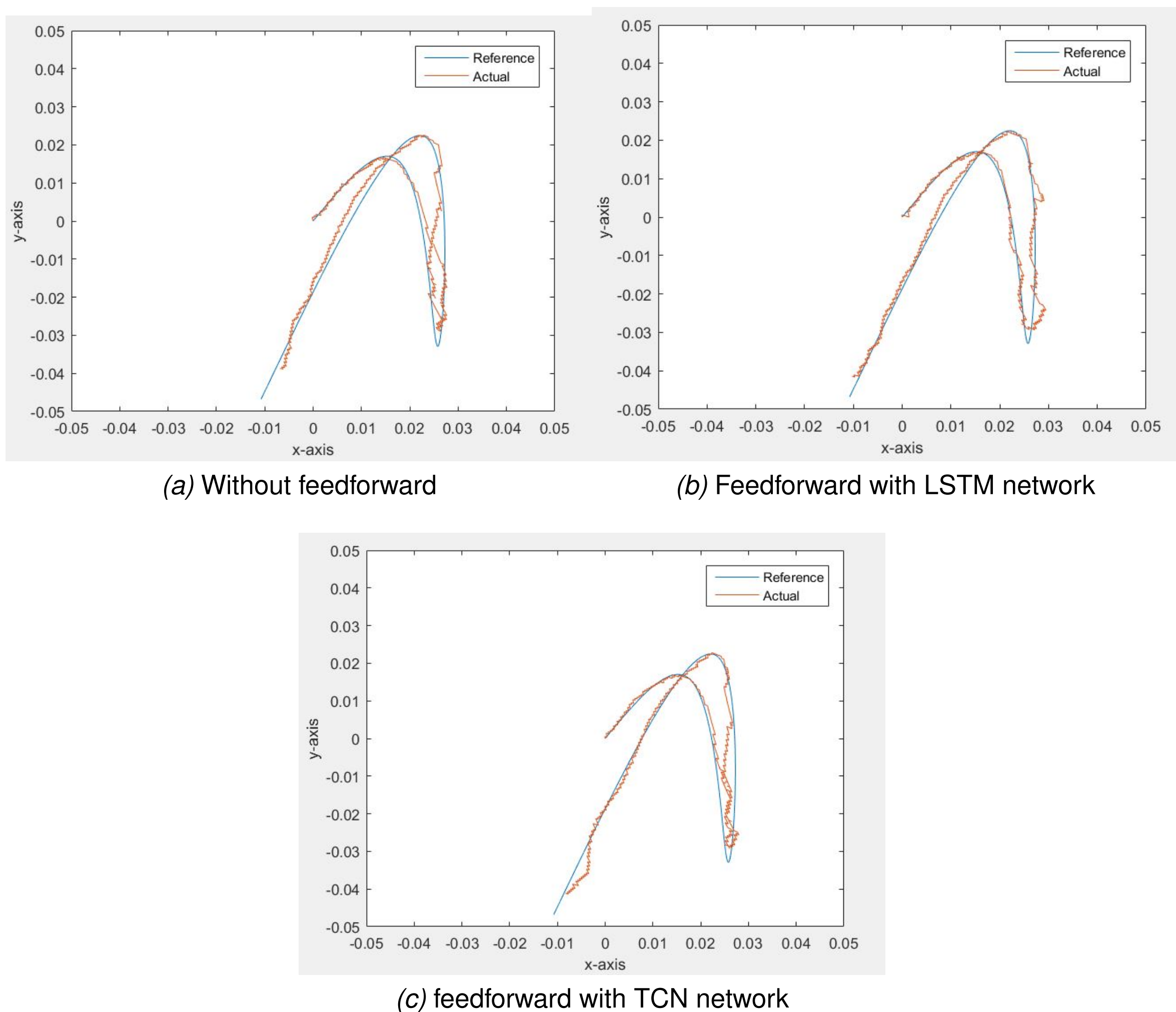
$$
\begin{aligned}
\theta_{ei} &= acos(\frac{P_i^2 - l_s^2 - l_e^2}{2 l_s l_e}) \\
\theta_{ee} &= \frac{1}{3}(\theta_{s1} + \theta_{s2} + \theta_{s3} + \theta_{e1} + \theta_{e2} + \theta_{e3})
\end{aligned}
\tag{C.3}
$$

The rest of the rigid body positions follows from these variables. Now, in the case of the actual manipulator, the three shoulder angles are obtained from the encoder. This means, that the end effector positions $[x_{ee}, y_{ee}]$ can be found from the equation C.2. However, the number of known variables (shoulder angles) are more than the number of DOFs (end effector positions). Therefore, only 2 shoulder angles are used to obtain the end effector positions in this study, which are the shoulder angles θ_{s1} and θ_{s2}.

Appendix D

Additional Results

The figure D.1 presents the tracking performance of the manipulator when it is subjected to a random trajectory.



(a) Without feedforward

(b) Feedforward with LSTM network

(c) feedforward with TCN network

Figure D.1: Feedforward evaluation for a random trajectory

The second norm and the infinite norm of the error during this evaluation is pre-

sented in table D.1. The values are scaled by a factor of 10^{-3} and optimal results are highlighted in red ink.

Error$(\times 10^{-3})$	**Without FF**	**LSTM based FF**	**TCN based FF**
$\|\|E_x\|\|_2$	1.363	**0.657**	1.214
$\|\|E_y\|\|_2$	4.227	**2.507**	2.826
$\|\|E_x\|\|_\infty$	4.70	**2.34**	4.44
$\|\|E_y\|\|_\infty$	13.42	**9.91**	10.7

Table D.1: Feedforward Evaluation on random trajectory