UKRAINIAN CATHOLIC UNIVERSITY

MASTER THESIS

Real-time simulation of arm and hand dynamics using ANN

Author:
Mykhailo MANUKIAN

Supervisor:
Dr. Sergiy YAKOVENKO

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Mykhailo MANUKIAN, declare that this thesis titled, "Real-time simulation of arm and hand dynamics using ANN" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Victorious warriors win first and then go to war, while defeated warriors go to war first and then seek to win."

Sun Tzu, The Art of War

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Master of Science

Real-time simulation of arm and hand dynamics using ANN

by Mykhailo MANUKIAN

Abstract

The physics of body dynamics is a complex problem solved by the nervous system in real-time during the planning and execution of movements. The human arm and hand have complex mechanics involving hundreds of muscles that actuate over 30 degrees of freedom (DOF). To date, the problems of this complexity remain unsolved in engineering; yet, the nervous system computes control signals in a robust, accurate, and time-efficient manner. Neuroprosthetics require similar computations for the decoding of intent and encoding of sensory feedback. The trade-off of required computational accuracy and latency is hard to resolve with classical physics; thus, this research aims to develop "good-enough" approximations of these computations using machine learning methods, such as artificial neural networks (ANN). The kinematic and kinetic temporal computations that rely on the diverse number of terms within the equations of motion are consistent with the recurrent neural network (RNN) architectures. This study will test the general hypothesis that the inverse dynamics of arm and hand can be captured with RNN formulation and explore the utility of different architectures: i) simple Recurrent ANN, ii) Gated Recurrent Unit (GRU) ANN, and iii) Long Short-Term Memory (LSTM) ANN. The inverse problem is the mapping from joint kinematics (position, velocity, acceleration) to joint kinetics (torque). The training and testing datasets were derived from the physical model of arm and hand performing point-to-point movements between realistic postures arranged in a grid within the physiological range of motion. Lastly, we assessed the execution latency of the machine learning solutions in the context of real-time requirements for prosthetic applications.

Keywords: Inverse Dynamics, RNN, joint torques, joint kinematics, joint kinetics, hand dynamics, arm dynamics, locomotion, motion control.

Acknowledgements

First of all, I would like to express my deepest gratitude to Sergiy Yakovenko for bringing up such an exciting topic for the research and all of his support and help along the process. Also, I thank him for sharing with me an exciting world of biomechanics and neuroscience.

The general idea and study design were conducted based on Sergiy Yakovenko's support of NSF STTR Phase I 19-555 "Biomimetic Solution for Gesture-Based Human Machine Interactions".

Next, I express my deepest gratitude to Serhii Bahdasariants for his valuable contributions of ideas, the implementation of physical model simulations in Simulink (MathWorks Inc.), and the creation of kinematic and kinetic datasets. Also, I thank him for answering tons of my questions about problem background.

Furthermore, I would like to thank Andrey Lyubonko for his valuable support for all technical-related questions about RNN models and valuable tips along the process. Also, I thank him for staying up late in the evening for sync calls with Sergiy and Serhii.

I want to thank Oleksii Molchanovskyi for his support and help throughout the whole duration of this program. I thank him for the exciting challenges and interesting projects, especially including this research.

My deep gratitude also goes to UCU professors, lecturers, and teaching assistants for keeping gears running and helping us to get precious knowledge.

Finally, I would like to thank all of the other students in our group without whom these two years of study would not be so great.

Contents

List of Figures

List of Tables

List of Abbreviations

DOF	**D**egree **O**f **F**reedom
ANN	**A**rtificial **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
GRU	**G**ated **R**ecurrent **U**nit
LSTM	**L**ong **S**hort-**T**erm **M**emory
EMG	**E**lectro**m**yo**g**raphy
MT	**M**usculoskeletal **T**ransformation
LD	**L**imb **D**ynamics
ML	**M**achine **L**earning
ID	**I**nverse **D**ynamics
NEL	**N**eural **E**ngineering **L**ab
WVU	**W**est **V**irginia **U**niversity
NLP	**N**atural **L**anguage **P**rocessing
APE	**A**rtificial **P**hysics **E**ngine
ReLU	**Re**ctified **L**inear **U**nit
MGU	**M**inimal **G**ated **U**nit
EDA	**E**xploratory **D**ata **A**nalysis
UoM	**U**nit **o**f **M**easurement
MSE	**M**ean **S**quare **E**rror
HDF	**H**ierarchical **D**ata **F**ormat
FC	**F**ully **C**onnected
GPU	**G**raphical **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
LR	**L**earning **R**ate
RMSE	**R**oot **M**ean **S**quare **E**rror
CPU	**C**entral **P**rocessing **U**nit

List of Symbols

q	joint positions	(rad)
$\dot{q}$	joint velocities	$(\frac{\text{rad}}{\text{s}})$
$\ddot{q}$	joint accelerations	$(\frac{\text{rad}}{\text{s}^2})$
v	velocity	$(\frac{\text{m}}{\text{s}})$
a	acceleration	$(\frac{\text{m}}{\text{s}^2})$
f	force	$(\text{N} \cdot \text{m})$
t	time	(s)
I	inertia	
S	motion freedom	
W	weights	
x	input data	
h	hidden state	
c	cell state	
b	bias	
r_t	reset gate	
z_t	update gate	
n_t	new gate	
i_t	input gate	
f_t	forget gate	
g_t	cell gate	
o_t	output gate	
τ	joint torques	$(\text{N} \cdot \text{m})$
σ	sigmoid function	

This research is dedicated to my family, who supported me during two years of my study. Also, I dedicated this work to my friends, who are waiting for me for the same two years. Finally, I would like to dedicate this research to all people living with prosthetic devices. I sincerely hope that someday this or similar works will change their life for the better.

Chapter 1

Introduction

1.1 Motivation

The problem of decoding control signals for arm and hand movements has not yet been fully solved, even though several approaches show promise. The human hand has 24 degrees of freedom (DOF): 4 in each finger, 3 for extension and flexion, and one for abduction and adduction; the thumb is more complicated and has 5 DOF, leaving 3 DOF for the rotation of the wrist (Agur, 1999). In addition to this, the arm itself also has 4 DOF for elbow and shoulder joints. Due to the complexity of hand structure and functions, we need a complex biological "computer" in our head to control it. As a result, the most significant part of our motor cortex responsible for hand motion control. So, solving a task of controlling arm and hand dynamics from control signals to the limb's exact position in space amongst the most challenging tasks in human locomotion simulation.

Many pieces of research focus on pattern recognition of EMG signals to classify a limited amount of gestures. Usually, the number of gestures is not high and lies within interval 4-12 gestures. Larger data sets are rare. However, despite promising results with high accuracy reported, pattern recognition usage in real-life applications could be complicated due to the small number of gestures. When the number of gestures increases, the accuracy of pattern recognition decrease as highlighted in Atzori, Cognolato, and Müller, 2016.

Solving the outlined problem will lay a foundation for future prosthetic limb improvements and enhance its range of supported movements. In addition, we could improve the robustness of human-machine interactions by solving limb dynamics problem with the help of Artificial Neural Networks (ANN) formulation. Finally, high performance and minimal response time of prosthetics in daily activities will improve amputees' quality of life by reducing their disability.

1.2 Problem Background

The mapping of control signals recorded from cortex, nerves, or muscles during contractions and relaxations to precise limb position in space is a non-linear one. Considering this, any model which tries to map control signal to arm and hand position in space directly needs to solve a highly complex relationship. Thus, it could not be solved as a time series problem and approached as a pattern recognition problem with fewer gestures. As a result, the practical application of such solutions is limited also. The required model should solve two parts of the system - musculoskeletal transformation (MT) and limb dynamics (LD). Existing Machine Learning approaches attempt to solve both MT and LD parts all together - without breaking them apart.

In the current research, the idea is to use the approach described in Sobinov et al., 2020, where the musculoskeletal transformation was solved already. Hence this work will focus only on the kinematic part - find a model based on ANN formulation to approximate limb dynamics. This problem is known as Inverse Dynamics (ID) problem in robotics and biomechanics.

Inverse Dynamics is the problem of finding the torques to produce a required acceleration in the rigid-body model (Featherstone, 2007). In a nutshell, the ID problem could be described by the equation:

$$\tau = ID(model, q, \dot{q}, \ddot{q}) \tag{1.1}$$

where q, $\dot{q}$, $\ddot{q}$, and τ are vectors of joints positions, velocities, acceleration, and torques, respectively, and a *model* is a rigid-body model (Featherstone, 2007). Relation between vectors q, $\dot{q}$, $\ddot{q}$ could be described by formulas $\dot{q} = \frac{dq}{dt}$ and $\ddot{q} = \frac{d\dot{q}}{dt}$.

One possible way to solve the ID problem for body i is to use the Recursive Newton-Euler algorithm, which consists of the following steps.

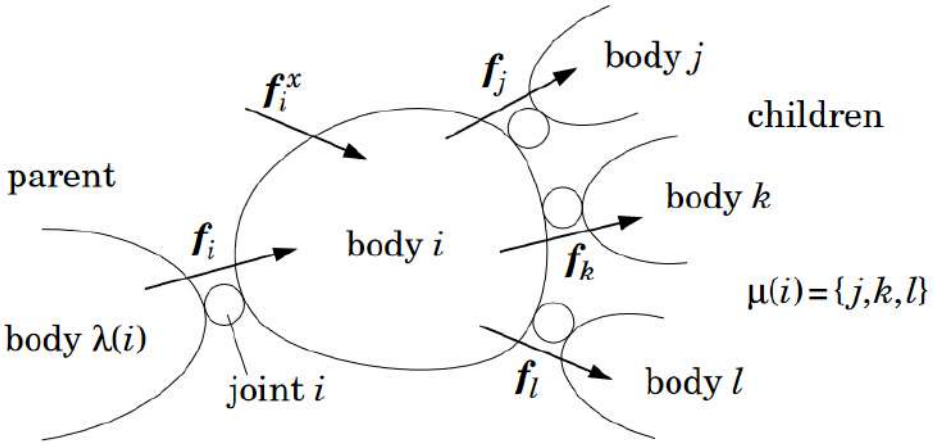

FIGURE 1.1: Forces acting on body i by Featherstone, 2007

Step 1:

$$v_i = v_{\lambda(i)} + S_i\dot{q}_i \tag{1.2}$$

where S_i is a motion freedom matrix, v_i is the velocity of body i and could be defined recursively as the sum of the velocity of its parent $\lambda(i)$ and the velocity across the connecting joint (Featherstone, 2007). Initial value $v_0 = 0$. By differentiating equation 1.2, recurrent relation for acceleration could be obtained:

$$a_i = a_{\lambda(i)} + S_i\ddot{q}_i + S_i\dot{q}_i \tag{1.3}$$

where S_i is a motion freedom matrix and a_i is an acceleration of body i by Featherstone, 2007. Initial value $a_0 = 0$.

Step 2: Net force action on body i is f_i^B and calculated by equation:

$$f_i^B = I_i a_i + v_i \times^* I_i v_i \tag{1.4}$$

where I_i is inertia of body i and $\times^*$ dual cross-product operator $\times$ by Featherstone, 2007.

Step 3: From figure 1.1 f_i is the force transmitted from body $\lambda(i)$ to body i through joint i, and f_i^x is the external forces applied to body i (Featherstone, 2007). The net force on the body i calculated by the formula:

$$f_i^B = f_i + f_i^x - \sum_{j \in \mu(i)} f_j \tag{1.5}$$

which could be rearranged to recurrence relation of the joint forces:

$$f_i = f_i^B - f_i^x + \sum_{j \in \mu(i)} f_j \tag{1.6}$$

as explained by Featherstone, 2007. Generalized torques at the joints could be found by the equation:

$$\tau_i = S_i^T f_i \tag{1.7}$$

where S_i is a motion freedom matrix by Featherstone, 2007.

Also, there is a temporal relation in this problem. For example, at time step $t+1$ position of the limb in space will depend on two factors: distance covered within a time step $t+1$ and the limb's initial position at time step t. The proposed model should consider such a temporal relationship where the final result on step $t+1$ also depends on step t.

1.3 Goals of Research

To achieve desired results, it is also essential to have some physical model used during ANN training. As a possible option, we will use the musculoskeletal model created by colleagues from the Neural Engineering Lab (NEL) at West Virginia University (WVU) to provide input and target data for network training. The baseline model is described in detail in section 3.1. Finally, to allow such ANN model application in some real-life scenarios, there is also an important constraint. The resulting model should work in real-time and have a latency as small as possible for the forward propagation.

The main goal is to find suitable recurrent neural network (RNN) models designed to work with sequential data in Natural Language Processing (NLP) domain. Next, we suggest applying the resulted model to find a robust mapping between joint kinematics and torques to identify a simulated limb's precise motion in space. In general, we would like to find an ANN model to solve the Inverse Dynamics problem. Moreover, such an ANN model, in addition to joint kinematics-torques mapping described above, should learn temporal dependency between resulted trajectory on each time step from a previous one. Afterward, we plan to integrate such a model into the Artificial Physic Engine (APE) tool, which will approximate physical processes with ANN instead of direct calculations.

1.4 Structure of Master Thesis

Further in this thesis following parts are included:

- chapter 2 contains information about other researches in this field and a review of related works;
- chapter 3 describes available data itself, how data was generated for this research, and its analysis;

- chapter 4 is about the research approach used and solution structure;
- chapter 5 covers details of conducted experiments;
- chapter 6 summarize obtained results, discuss research limitations and outline future work directions.

Chapter 2

Related Work

2.1 Inverse Dynamic Problem and ANN

In the past, there were successful attempts to approximate human arm and hand motion with the help of Artificial Neural Networks (ANN) of type Recurrent Neural Network (RNN), results are described by Draye et al., 1995. In this work, the authors used Electromyography (EMG) signals as model input to approximate a trajectory when drawing figure "eight" with a straight arm. For such tasks, feed-forward ANN is not a good fit since such networks do not learn temporal relationships between samples (Draye et al., 1995). In our study, EMG signals are mapped to joint kinematics (joint positions, velocities, and accelerations) as described by Sobinov et al., 2020.

In general, the recurrent internal structure of RNN models gives the latest enough computational power to approximate dynamic systems, as highlighted by Ogunmolu et al., 2016. Also, RNN could solve Inverse Dynamics (ID) problem in robotics for robot tracking control of flexible joints robot Baxter (Chen and Wen, 2019), which indicates that RNN can cope with such problems. Finally, in Hartmann et al., 2012 RNN was applied to a musculoskeletal robot arm with 7 DOF, comparable with a human arm in structure and complexity. In the current research, we use a more complicated model, which includes a hand as well. Hence the total number of DOF is equal to 23, thus increasing problem complexity.

2.2 Recurrent Neural Network Types

To solve the mapping of joint kinematics to torques for a simulated realistic 27 DOF limb, we consider the following list of suitable recurrent neural network types: the "vanilla" RNN as baseline one, Gated Recurrent Unit (GRU), and Long Short-Term Memory (LSTM). We described all mentioned RNN types in detail below. We choose Recurrent Neural Networks (RNN) as the best fit for this problem because those are well-suited for time-related data and can solve ID problems in general, as mentioned in section 2.1. We plan to determine exact ANN architecture based on series of experiments for architecture search and hyperparameters tuning.

2.2.1 Simple Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) is a type of ANN where for each element in the input, each recurrent layer computes the following:

$$h_t = ReLU(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh}) \tag{2.1}$$

where h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$, W_{ih} and W_{hh} are the input-hidden and hidden-hidden weights matrices, b_{ih} and b_{hh} are the input-hidden and hidden-hidden bias terms.

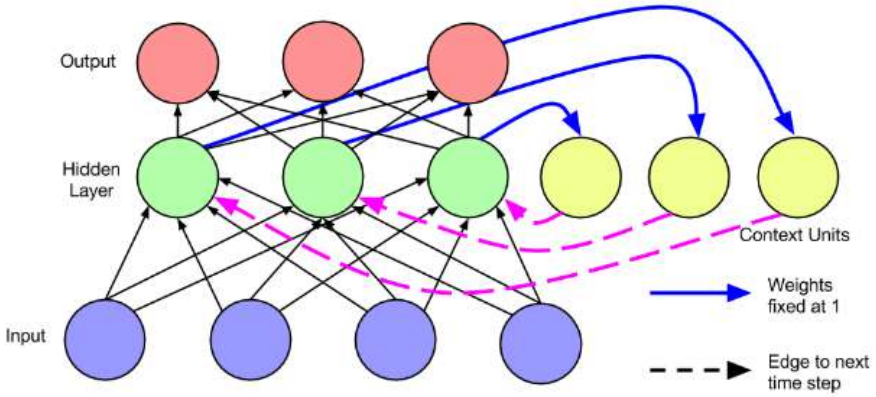

FIGURE 2.1: Simple or Elman RNN reviewed by Lipton, Berkowitz, and Elkan, 2015

Figure 2.1 shows an overall structure of simple or Elman RNN. This network was introduced by Elman, 1990 and reviewed by Lipton, Berkowitz, and Elkan, 2015. However, simple RNN models might suffer from vanishing or exploding gradient problems during training as described in Bengio, Simard, and Frasconi, 1994. With *ReLU* function used as nonlinearity activation function exploding gradient issue is more likely to happen (Lipton, Berkowitz, and Elkan, 2015).

2.2.2 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) is a type of ANN where for each element in the input sequence, each recurrent layer computes the following functions:

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \tag{2.2}$$

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \tag{2.3}$$

$$n_t = tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \tag{2.4}$$

$$h_t = (1 - z_t) * n_t + z_t * h_{(t-1)} \tag{2.5}$$

where h_t is a hidden state at time t; x_t is the input at time t; $h_{(t-1)}$ is the hidden state of a layer at time $t-1$; r_t, z_t, n_t are the reset, update, and new gates, respectively. Thus, $W_{i\_}$ and $W_{h\_}$ are the input-hidden and hidden-hidden weights matrices, $b_{i\_}$ and $b_{h\_}$ are the input-hidden and hidden-hidden bias terms. σ is the sigmoid function, and $*$ is the Hadamard product.

This type of RNN was initially proposed by Cho et al., 2014 and displayed on figure 2.2. In Chung et al., 2014, authors compared two gated RNN types - GRU and LSTM - with simple RNN. Results indicate that both GRU and LSTM models outperform simple RNN on different datasets. However, there is no way to determine which gated RNN is better than the other, and the selection of the best model depends on the task and dataset itself (Chung et al., 2014). While in Zhou et al., 2016,

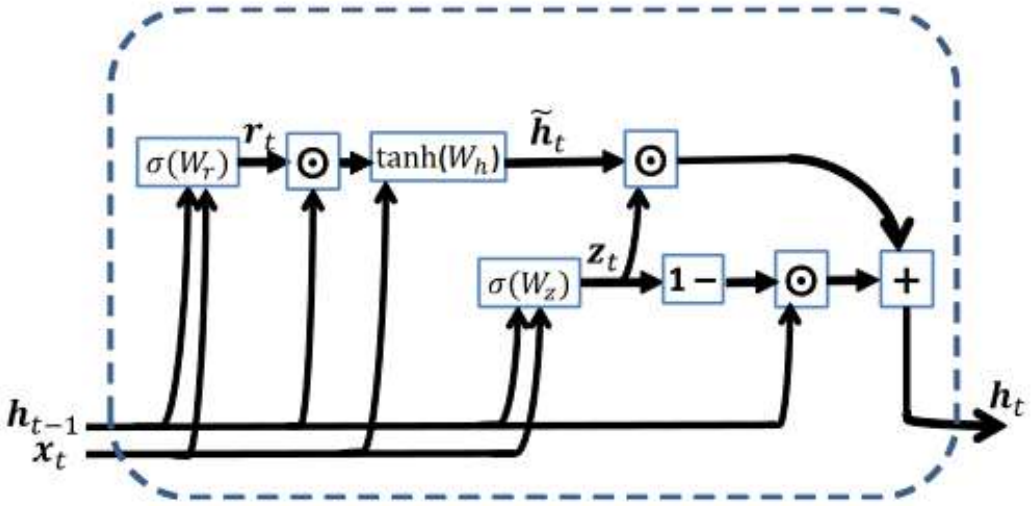

FIGURE 2.2: GRU block re-printed from Zhou et al., 2016

authors proposed a new simplified version of GRU - Minimal Gated Unit (MGU) with only one gate and no peephole connections, making it faster to train and less computationally intensive with the same level of performance. Hence MGU could be an excellent alternative to GRU models considering real-time response requirements and edge device limitations. Finally, authors in Jozefowicz, Zaremba, and Sutskever, 2015 have found some permutations of GRU architecture that outperform both standard GRU and LSTM models on multiple tasks. However, the improvement margin was not significant for GRU architecture.

2.2.3 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of ANN where for each element in the input sequence, each layer computes the following function:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \tag{2.6}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \tag{2.7}$$

$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \tag{2.8}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \tag{2.9}$$

$$c_t = f_t * c_{(t-1)} + i_t * g_t \tag{2.10}$$

$$h_t = o_t * tanh(c_t) \tag{2.11}$$

where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. Thus, $W_{i\_}$ and $W_{h\_}$ are the input-hidden and hidden-hidden weights matrices, $b_{i\_}$ and $b_{h\_}$ are the input-hidden and hidden-hidden bias terms. σ is the sigmoid function, and $*$ is the Hadamard product.

Based on results obtained in Greff et al., 2017, it makes sense to start with what was called the "vanilla" LSTM block setup in that paper (see figure 2.3). Such commonly used LSTM block architecture shows good performance on various data sets, and none of the improvements tested in Greff et al., 2017 significantly improved this result. However, considering the real-time model response requirement, simplifying

the LSTM block to reduces its computational cost makes sense. Following experiments from Greff et al., 2017 most significant simplifications are coupling of input and forget gates and removing peephole connections because both of these changes do not decrease performance significantly. In work by Jozefowicz, Zaremba, and Sutskever, 2015, authors have found some permutations of GRU architecture that outperform the standard LSTM model on multiple tasks. However, the LSTM model with dropout or increased forget bias might have better results, but this depends on the particular dataset and task at hand (Jozefowicz, Zaremba, and Sutskever, 2015). Finally, Chung et al., 2014 showed that gated RNN, like GRU and LSTM, outperforms simple RNN architecture for different tasks, but results were not that clear to decide which one is better, and the decision should be taken based on the task to solve.

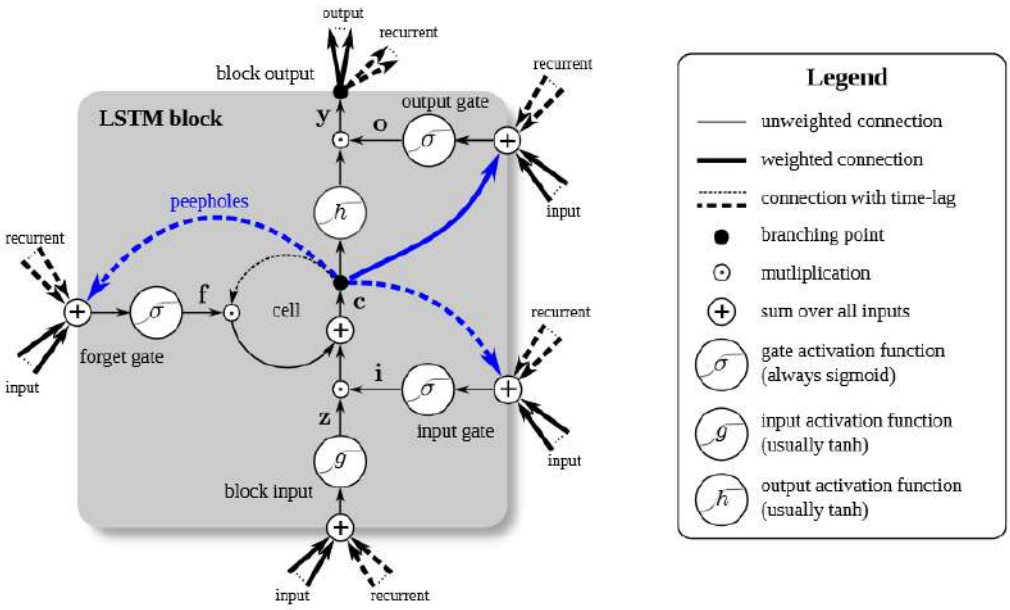

FIGURE 2.3: "Vanilla" LSTM block setup as described in Greff et al., 2017

The most relevant hyperparameters are the learning rate and size of hidden layers. It is confirmed by analysis done in Greff et al., 2017 with the help of the fANOVA framework for assessing hyperparameter importance. So, it makes sense to tune only these two hyperparameters while searching for the best-performing model. Lastly, authors in Greff et al., 2017 prove that we could treat hyperparameters as independent for the sake of tuning simplification. The measured interaction between the two mentioned hyperparameters is insignificant. In Greff et al., 2017, it also suggested tuning the learning rate with the help of a smaller network to save time. However, exact LSTM network architecture remains an open question since even in Greff et al., 2017, authors used different network architectures for different datasets.

Chapter 3

Data

3.1 General Information

This research arm and hand model is a simplified model of the real bio-mechanical system. Hence the number of DOF is reduced down to 23. Colleagues from Neural Engineering Lab (NEL) at West Virginia University (WVU) generated data for this research using a Simulink model in MATLAB. Figure 3.1 depicts the internal structure of this model, which is similar to the human hand in structure, and figure 3.2 shows this model in action. Additional details of the internal model design are shown on figures B.1 and B.2 in appendix B. We did not use data from real subjects in this research. List of all DOF used in this research provided in table 3.1 with a description of which joint it corresponds to.

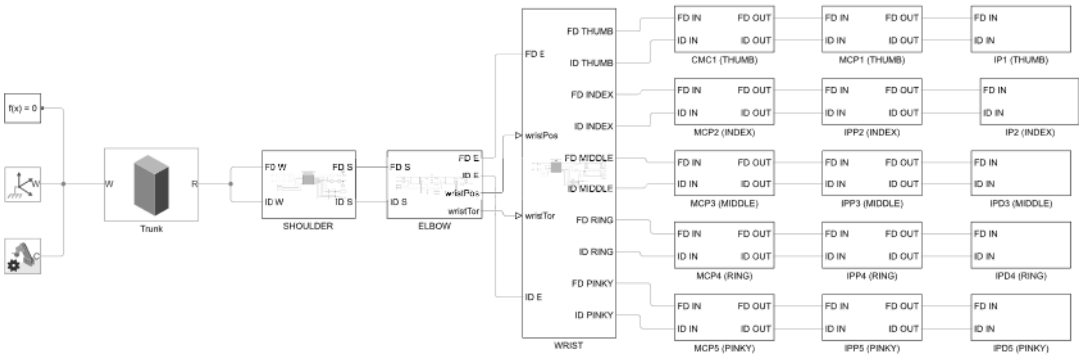

FIGURE 3.1: Simulink model internal schematics by NEL at WVU

We organized all data into 1053 files, recording a particular movement of simulated hand between starting and ending positions. Each file name follows a pattern - {movement type}_{start position}_{end position}_{movement duration in s}.

All possible movements between the selected postures were created using the bell-shaped velocity constraint with zero starting and final velocities. The grid was defined by $3 \times 3 \times 3$ postures spanning positions on both sides of the body midline and covering the central area of space between shoulders. The total number of movements, defined by the combinatorial combination $C(n,k) = \frac{n!}{[k!(n-k)!]}$, where n is the number of postures and k is the number of postures in a selection $(k = 2)$, was 351. The shape of the Gaussian curve defined the velocity maximum and three movement durations (0.5 s, 1.0 s, 2.0 s) to evaluate the impact of varied dynamics. Thus, this dataset captured a diverse subset of dynamic repertoire (reaching, defense, and manipulation) typically examined in primate research of limb control (Graziano and Aflalo, 2007).

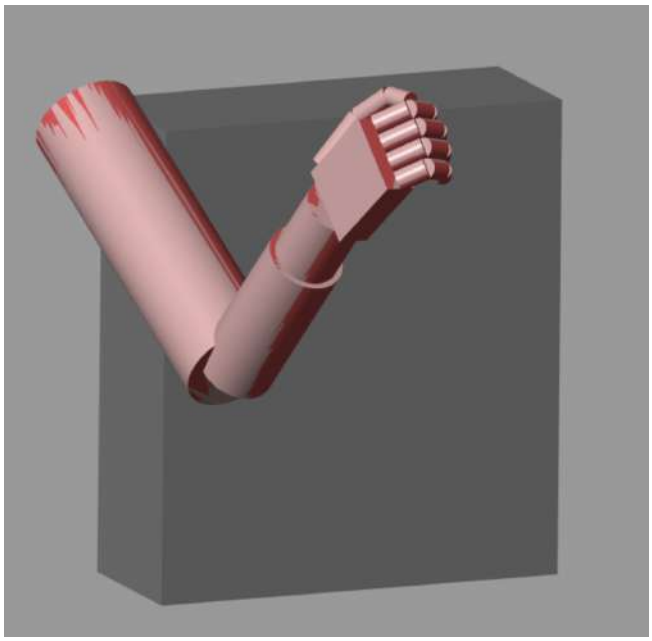

FIGURE 3.2: Simulink model visualization used for data generation by NEL at WVU

DOF Name	Description
ra_wr_s_p	hand rotation motion; supination is negative; pronation is positive
ra_wr_e_f	wrist flexion/extension motion; flexion is positive
ra_cmc1_ad_ab	thumb proximal abduction/adduction motion
ra_cmc1_f_e	thumb proximal flexion/extension motion
ra_mcp1_f_e	thumb central flexion/extension motion
ra_ip1_f_e	thumb distal flexion/extension motion
ra_mcp2_e_f	index proximal flexion/extension motion
ra_pip2_e_f	index central flexion/extension motion
ra_dip2_e_f	index distal flexion/extension motion
ra_mcp3_e_f	middle proximal flexion/extension motion
ra_pip3_e_f	middle central flexion/extension motion
ra_dip3_e_f	middle distal flexion/extension motion
ra_mcp4_e_f	ring proximal flexion/extension motion
ra_pip4_e_f	ring central flexion/extension motion
ra_dip4_e_f	ring distal flexion/extension motion
ra_mcp5_e_f	pinky proximal flexion/extension motion
ra_pip5_e_f	pinky central flexion/extension motion
ra_dip5_e_f	pinky distal flexion/extension motion
ra_sh_ab_ad	shoulder abduction/adduction motion
ra_sh_e_f	shoulder flexion/extension motion
ra_sh_rot	shoulder rotation motion
ra_el_e_f	elbow flexion/extension motion
ra_wr_ad_ab	wrist abduction/adduction motion; radial deviation is negative

TABLE 3.1: DOF details

The resulting data was organized as the structure containing $[q, \dot{q}, \ddot{q}, \tau]$, where q is the vector of joint positions, $\dot{q}$ and $\ddot{q}$ are its angular velocity and acceleration, and τ is the joint torque. For the 23 DOF model, each vector contained 92 signals sampled at 10

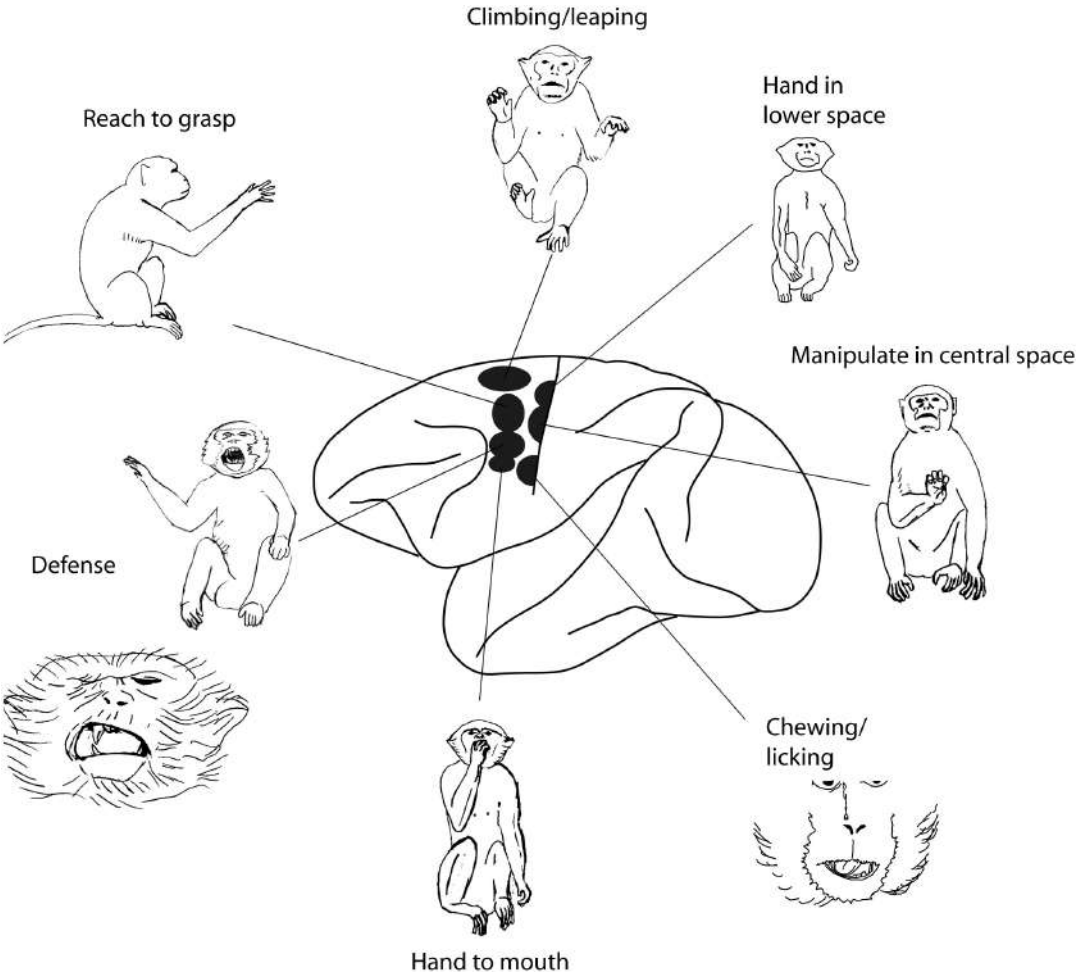

FIGURE 3.3: Action Zones in the Motor Cortex of the Monkey from Graziano and Aflalo, 2007

kHz - 10 cycles per millisecond (ms). For Inverse Dynamic (ID) problem positions, velocity and acceleration represent input variables, resulting in 69 input features - 3 features per DOF. While torques data represents target variables, thus there are 23 target features the model needs to predict.

The final name of any feature consists of a prefix concatenated with the respective DOF name with the help of the underscore symbol "_". Throughout this work, we are using the following prefixes:

- *pos* for position data per DOF, for example, *pos_ra_el_e_f*;
- *vel* for velocity data per DOF, for example, *vel_ra_el_e_f*;
- *acc* for acceleration data per DOF, for example, *acc_ra_el_e_f*;
- *tor* for torque data per DOF, for example, *tor_ra_el_e_f*.

3.2 Exploratory Data Analysis

As part of the research, we performed Exploratory Data Analysis (EDA) on given data to check any abnormalities or specifics. As a result, we identified that 42 input features out of 69 have a constant value of 0 for all recorded movements in the whole

dataset. Such specific is known and related to the movement type which is used to collect the data. We do not include the intended movement of hand DOF in the current dataset on purpose. Hence the majority of velocity and acceleration features for those DOF are not used and equal to 0. Since training on zero-valued features does not yield any meaningful results, those features were excluded from further usage in all 1053 files leaving only 27 meaningful features for training, validation and testing. Such exclusion also helps to reduce dataset size and speed up processing. Table 3.2 contains an EDA summary of non-zero input features. If we include the intended movement of hand into the dataset, those features will be essential and could not be excluded anymore. However, such movements are not in the scope of this research and will be part of future work.

Feature name	**Min**	**Max**	**Mean**	**UoM**
pos_ra_wr_e_f*	0.0873	0.0873	0.0873	rad
pos_ra_cmc1_ad_ab*	0.43635	0.43635	0.43635	rad
pos_ra_cmc1_f_e*	0.43635	0.43635	0.43635	rad
pos_ra_mcp1_f_e*	-0.3927	-0.3927	-0.3927	rad
pos_ra_ip1_f_e*	-0.7854	-0.7854	-0.7854	rad
pos_ra_mcp2_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_pip2_e_f*	1.0472	1.0472	1.0472	rad
pos_ra_dip2_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_mcp3_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_pip3_e_f*	1.0472	1.0472	1.0472	rad
pos_ra_dip3_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_mcp4_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_pip4_e_f*	1.0472	1.0472	1.0472	rad
pos_ra_dip4_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_mcp5_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_pip5_e_f*	1.0472	1.0472	1.0472	rad
pos_ra_dip5_e_f*	0.7854	0.7854	0.7854	rad
pos_ra_sh_e_f	-0.383972	2.356194	0.724635	rad
pos_ra_sh_rot	0.000000	1.431170	0.548809	rad
pos_ra_el_e_f	0.087266	2.234021	1.231427	rad
pos_ra_wr_ad_ab*	0.085	0.085	0.085	rad
vel_ra_sh_e_f	-12.96415	9.24831	-0.710687	rad/s
vel_ra_sh_rot	-6.523361	6.771084	-0.000928	rad/s
vel_ra_el_e_f	-9.743755	10.156626	0.157892	rad/s
acc_ra_sh_e_f	-89.151843	89.151843	-5.895056e-13	rad/s^2
acc_ra_sh_rot	-46.563383	46.563383	-1.490752e-14	rad/s^2
acc_ra_el_e_f	-69.845074	69.845074	1.542024e-13	rad/s^2

TABLE 3.2: EDA results for input features

In the table 3.2, we marked with an asterisk "*" those features which have constant value across the whole dataset. As was mentioned in section 3.1, the current dataset consists only of actions where there is no intended movement of the hand. Because of this reason, position features for the wrist and fingers do not change during the action and remain constant. However, as could be seen from table 3.3, a minor amount of torques is still required to keep hand DOF in a stable position during movement. It happens due to other forces applied to arm and hand segments during movement, including gravity force. Also, the content of table 3.3 confirms the initial observation

that the current dataset is focused around elbow and shoulder joints only - the most significant amplitude of torques applied to elbow and shoulder DOF to execute a movement. Therefore, we kept all of the input features with constant values in the model since those are still relevant for final results predictions.

Feature name	**Min**	**Max**	**Mean**	**UoM**
tor_ra_wr_s_p	-0.737575	0.288719	-0.069673	N · m
tor_ra_wr_e_f	-1.021415	1.711340	-0.008283	N · m
tor_ra_cmc1_ad_ab	-0.089691	0.244647	0.022555	N · m
tor_ra_cmc1_f_e	-0.104478	0.151575	-0.005417	N · m
tor_ra_mcp1_f_e	-0.045148	0.072021	0.007732	N · m
tor_ra_ip1_f_e	-0.013673	0.009171	0.001053	N · m
tor_ra_mcp2_e_f	-0.045963	0.142117	-0.000329	N · m
tor_ra_pip2_e_f	-0.017351	0.060641	-0.000069	N · m
tor_ra_dip2_e_f	-0.002087	0.007087	0.000002	N · m
tor_ra_mcp3_e_f	-0.050628	0.155408	-0.000345	N · m
tor_ra_pip3_e_f	-0.017948	0.062216	-0.000070	N · m
tor_ra_dip3_e_f	-0.002110	0.006693	0.000002	N · m
tor_ra_mcp4_e_f	-0.034543	0.106487	-0.000231	N · m
tor_ra_pip4_e_f	-0.012739	0.043525	-0.000052	N · m
tor_ra_dip4_e_f	-0.001679	0.004946	3.297148e-07	N · m
tor_ra_mcp5_e_f	-0.021682	0.066597	-0.000144	N · m
tor_ra_pip5_e_f	-0.007879	0.026493	-0.000034	N · m
tor_ra_dip5_e_f	-0.001121	0.003064	-4.562871e-07	N · m
tor_ra_sh_ab_ad	-12.861373	32.386611	3.198925	N · m
tor_ra_sh_e_f	-48.406797	49.133783	5.980571	N · m
tor_ra_sh_rot	-32.820674	39.781555	1.569562	N · m
tor_ra_el_e_f	-21.420735	41.259662	6.176702	N · m
tor_ra_wr_ad_ab	-2.679080	0.984739	-0.246989	N · m

TABLE 3.3: EDA results for output features

Chapter 4

Methodology and Research Approach

4.1 Methodology

For this research, we decided to go with a methodology that is inspired by the scientific method. Hence our research consists of five phases: problem understanding, data analysis, data preprocessing, model selection, and results evaluation. However, this process is not linear, and we iteratively repeated phases multiple times.

4.1.1 Problem Understanding

As part of this phase, we defined and set the goals of this research. We captured those goals in section 1.3. Also, we studied the problem background of Inverse Dynamics (ID) itself and how it is addressed by usual means. We outlined this information in section 1.2. Next, according to defined goals, we reviewed related work. We reviewed similar researches about solutions to ID problems in the area of robotics. In addition, we reviewed researches where arm dynamic was approximated from surface Electromyography (EMG) signals. In all of the reviewed cases, successful results were achieved with the help of Recurrent Neural Networks (RNN), albeit on more simplified models, if compared with current research. Also, using EMG signals to approximate hand dynamics is a different approach than the ID problem. However, it indicates that RNN models are well suited for dynamic systems approximation. Subsequently, we also review related work regarding different RNN model types, their performance on different tasks, and potential simplifications, which might be useful considering edge device limitations and real-time response requirements. Section 2 contains a summary of the review results. As the last step of this phase, we prepared an execution plan for this research.

4.1.2 Data Analysis

In this phase, we studied available data and performed Exploratory Data Analysis (EDA). Chapter 3 describes what kind of data we had to approach the ID problem, who and how generated the data for this research, and summarizes the results of EDA. This step is essential for the successful execution of this work. EDA of the previous dataset version reveals severe data abnormalities that force to discard dataset. Those abnormalities are cases when joint positions were beyond the physical capabilities of a real joint, for example, 11 radians. Flaws in the model used for data generation were a root cause of mentioned abnormalities. To avoid such issues and re-collect a new dataset, we adjusted our model and reduced the number of revolute joints connected in series. Those were replaced with a spherical joint (for shoulder)

and a combination of universal joint and revolute joint (for wrist and thumb). Such changes reduce models' complexity and numerical error that arise due to having many separate revolute joints and rotation between them. Afterward, we performed the same analysis for the new dataset, and data quality was acceptable - no more severe abnormalities and data corresponds to the task specifics.

4.1.3 Data Preprocessing

This phase of research focuses on how to modify and process available data for the upcoming models. During this phase, we split all available data into train, validation, and test sets. We used the train and validation set during the model selection phase, while the test set we kept aside till the results evaluation phase. Due to the enormous size of the initial dataset, we decided not to perform cross-validation on the train set and instead have a separate validation dataset. As a result, we performed model training and validation cycles faster and iterated rapidly on different model architectures.

Another important aspect, which comes from data understanding, is the necessity to scale the data. We decided to scale input data to make training more robust and stable by shifting the scale of data closer to the ANN weights scale. It is essential if we consider the vulnerability of simple RNN models to vanishing or exploding gradient issues. Also, considering that model of hand and arm used for this research consist of 23 DOF, the Inverse Dynamic (ID) problem could be treated as a multi-target non-linear regression problem with 23 target variables or features, each with its scale. Due to this reason, it might be complicated to optimize with the help of Mean Squared Error (MSE) loss. In such a case, optimization happens for features with the most significant scale and ignores others. We found two options for dealing with such cases - either scale target features to make them more equal or use weighted MSE. In weighted MSE, some weights are assigned to one or another target feature. Since weighted MSE requires expert knowledge to correctly weight each target feature and adjustments of weights might be required based on new data, we decided to discard this option and scale target features.

As the last step of this phase, we prepared data for RNN model consumption. This process consists of two parts. Firstly, RNN models work with sequences of data. Hence we executed sequencing for train, validation, and test datasets. Secondly, due to the enormous size of available data, we had to take only its part. It also makes sense from the future practical application of the resulted model. However, even the reduced dataset was still too big to fit into memory, requiring special handling for big datasets. We solved this issue with the help of Hierarchical Data Format (HDF) files, which allow storing and reading datasets from disk. As a result, scaled and sequenced data was persisted on the disk for utilization during the next phase. We recorded all detailed information about this step in section 4.2.1.

4.1.4 Model Selection

We split the model selection phase into two parts: to find the most suitable architectures with hyperparameter tuning and to evaluate the relative performance of different architectures after final model training. We described all details for both parts in chapter 5 while capturing all details of the training pipeline itself in section 4.2.2.

The main goal of our model development is to use computationally light models that provide a "good-enough" description of limb dynamics, considering the requirements of real-time model response and potential edge device limitation. Previously, we have demonstrated how musculoskeletal dynamics could be described with high-quality approximations using our new formulation that relies on the information theory to determine the composition of the approximating power polynomial functions (Sobinov et al., 2020). We have taken this approach further to leverage machine learning for the solution of not only the musculoskeletal dynamics (Sobinov et al., 2020), but also the limb dynamics that is typically accomplished by solving the equations of motion. Furthermore, this approach based on artificial neural networks (ANN) can simplify the application of our models in movement control and assessment problems.

The mapping from posture to torque $\tau(t) = ANN\{q(t), \dot{q}(t), \ddot{q}(t)\}$ was performed by several types of ANNs: 1) Recurrent Neural Net (RNN); 2) Gated Recurrent Unit (GRU); and 3) Long Short-Term Memory (LSTM). The subjective choice of hyperparameters was based on error magnitude and rate during training with a diverse set of parameters in the preliminary training session for a limited number of epochs evaluating the entire training dataset. All models were assessed by MSE score on the scaled validation dataset. We kept the test dataset as a hold-out dataset during this phase and did not use it in model assessment. In general, we considered relatively simple architectures with one or few recurrent layers and a fully connected layer as the output one. We avoided more complex architectures and parked those for later if simple ones will fail.

Overall, we selected five candidate architectures - one type of RNN architecture as a baseline and two types for each GRU and LSTM architectures using the same general hyperparameters. We trained all five candidate models with a more sophisticated pipeline and without the temporal limit. We used the mean squared error of each model on the original non-scaled validation dataset with random initial setup, the models trained until no further improvement was registered.

4.1.5 Results Evaluation

In this phase, we executed the final evaluation of model performance with the help of a hold-out test set. In addition to calculating MSE scores for the test dataset, we will also evaluate selected model performance on all test set files directly to investigate which files are most challenging to predict and visualize those results. Also, we will validate model average time for forward propagation of one sample, assess model performance on noisy data and data with different lengths of sequence. Finally, together with colleagues from the Neural Engineering Lab (NEL) at WVU, we agreed to evaluate the selected model by connecting it to the musculoskeletal model used for data generation. However, this only possible after the selected model will be transferred to MATLAB. It is out of the scope of this thesis; hence, this evaluation will take place later and be part of future work. More details about model evaluation could be found in sections 5.2 and 5.3.1.

4.2 Solution Structure

We executed all experiments and data processing with the help of the Google Colab Pro service using GPU-enabled runtimes with enhanced RAM.

4.2.1 Data Processing Pipeline

As a first step, we split all 1053 files into training, validation, and test sets with a ratio of 0.7, 0.15, and 0.15, respectively. As a result, the training set consists of 737 files of different movement duration. Validation and test set both have 158 files, each with a different movement duration as well. Therefore, the distribution of movement duration approximately equals for training, validation, and test sets.

Available data first needs to be split into multiple sequences of fixed length to feed it into models of type RNN. We chose a sequence length of 100, equal to a sequence of 10 ms length in the original dataset. We decided not to use padding. Hence, for each file, sequences were only created from sample 100 and above. After sequencing the training dataset, it became evident that such an enormous amount of data does not fit into memory. To cope with such an issue, we decided to store data on disk in *.h5 files, a data file stored in Hierarchical Data Format (HDF). With this approach, it was possible to slice all training set files and store data on a disk. For ANN model training, we utilized the PyTorch framework. Hence, we used a custom-built PyTorch Dataset, which reads data from the *.h5 file on the fly when queried based on the given index. When dealing with *.h5 files, it is essential to know how data will be stored on a disk to optimize writing or reading.

For this work, reading performance was essential; hence, we discarded automatic chunking of the *.h5 file, leading to long-running batch reading from such dataset. Instead, we handpicked chunk sizes for input and target datasets as 3*100*27 and 300*23, respectively. This minor change improves the batch's reading speed, leading to faster model training and validation. Afterward, we built a simple custom PyTorch Dataset object intended to read data from *.h5 files and return a tuple of input and corresponding target arrays. Data for training supplied by PyTorch DataLoader in batches with shuffling enabled to ensure that it will be unbiased. As a result, data for training was supplied in randomly sampled batches from all files of the training dataset. The same prepossessing we also applied to validation and test sets - splitting into sequences, storing in *.h5 file, and shuffling when consumed via DataLoader. Shuffling is not required for validation and test sets, but the same pipeline was used for all sets.

However, usage of the original dataset for training is inconvenient due to its vast size, which leads to prolonged training. Also, from the future application point of view, there is no practical usage in the model, which needs to estimate predictions ten times per 1 ms. Therefore, we decided to take each tenth time step from the original dataset and collect a new reduced dataset. Now, the sequence of 100 timesteps corresponds to 100 ms of duration. Thus, the amount of data was reduced significantly, which positively impacted both - speed of model training and potential model practical application in the future.

As a next step, we have created a simple RNN model as a proof of concept model consisting of one RNN layer and one Fully Connected (FC) layer on top. RNN layer parameters are: hidden_size = 23, nonlinearity = 'relu' and FC layer parameter is in_features = 23. We used the FC layer as the last step without any activation function because the model could predict values that lie outside of the range for the ReLU function used in the RNN layer, such as negative torque values. After training such a model, we assessed its performance on some files from the test set. We noticed an obvious gap in results visualizations related to the usage of MSE as a loss function for optimization. We have very different amplitude among all 23 target features, as shown in table 3.3, with the biggest range for elbow and shoulder joints and a minor

range for finger joints. As a result, the model pays maximum attention during gradient descent to target features related to elbow and shoulder joints and completely ignores others. To address this issue, we decided to apply either standardization or normalization to input and target data. With the help of the simple RNN model described above, we have assessed three possible preprocessing scenarios. First, apply standardization to input and target features. Second, apply normalization to input and target features. Third, the mixed scenario with normalization applied to input features and standardization applied to target features. Afterward, we assessed the MSE score between true and predicted target features with inverse scaling for each scenario. Scenario with standardization of both input and target features yields the lowest MSE score. Hence, we recollected all datasets adding standardization as a preprocessing technique and the sequencing of data.

We used the already available implementation of Scalers in the Python scikit-learn library for both standardization and normalization. MinMaxScaler [1] from scikit-learn is an implementation of the normalization technique, while StandardScaler [2] is a class implementing standardization. We chose the scikit-learn implementation because there is a possibility to fit Scalers on streaming data in smaller chunks. With the help of this feature, we fit it on our data, which does not fit into memory as a single chunk. Instead, we used all of the files, which comprise a train set and fit Scaler on those files. We kept files from validation and test sets aside to avoid bias in the model and simulate a real-life application where not all data might be known upon training, including Scaler's fitting. We preserved fitted Scalers to use those as part of the pipeline and execute the inverse transformation of model predictions in both cases.

As a result, with the sequence of length 100, we collected and preserved a training dataset of size 804037 samples each of shape 100*27 for the input part and 1*23 for the target part. The validation and test datasets sizes are 153858 and 166358 samples, respectively, with the same sample shapes. Both datasets were also preserved to save time during training.

4.2.2 Model Training Pipeline

As a next step, we created an overall pipeline for the hyperparameters tuning, which will be used during our experiments described in chapter 5. This pipeline consists of model training for 15 epochs with mini-batch gradient descent optimization across the whole dataset. For gradient descent, we used PyTorch implementation of Adam optimization algorithm, which Kingma and Ba, 2015 initially proposed. Also, it includes model validation after each epoch. Finally, to assess model quality, we used the MSE loss metric between model predictions and scaled target features.

Furthermore, we set up a pipeline for final model training for the best-found set of hyperparameters. This pipeline is more complicated than the one we used for hyperparameters tuning and includes the Learning Rate (LR) scheduler, early stopping feature, and checkpoint saves after each epoch. In this pipeline, we validated trained models on the validation dataset only, and the test dataset is only used after training is over.

When optimization happens in the vicinity of optimum point initial value of learning rate for model training might not be optimal, despite showing good results initially. It happens because gradient descent jumps over an optimum due to the right direction but high length. To avoid such issues and to make optimization

[1] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

more precise in the vicinity of optimum value, we included an LR scheduler in our pipeline. As LR scheduler, we used the out-of-the-box class ReduceLROnPlateau [3] from the PyTorch library. This scheduler reduces LR by a chosen factor if there are no improvements for a certain number of epochs for the observed metric. As an observed metric for the LR scheduler, we decided to use MSE loss on the validation dataset for original non-scaled data. We perform inverse transformation with previously persisted Scaler object of scaled target values and model predictions to achieve this, then only MSE is calculated. Thus, we ensure that LR reduction will happen if there is no improvement of MSE loss on original data and not on scaled one. Other parameters of the LR scheduler are factor = 0.5, which will reduce LR by half, and patience = 5, which will trigger LR reduction only after five unsuccessful epochs.

We did not limit the number of epoch for training so that all models could converge as best as possible. However, to save time and avoid inefficient training after model optimization reaches optimum, we build an early stopping feature as a custom class. It allows checking if the network does not show any improvements for the observed metric over a particular number of epochs and stops further training as inefficient. Furthermore, to allow the LR scheduler to execute its role, we set the patience parameter of the early stopping class to 16, allowing the LR scheduler to reduce the rate twice in a row before training is interrupted.

Lastly, we were using a Google Colab Pro environment for this research, where any runtime has a timeout regardless of whether training is over or not. Considering that, we have implemented a save of model checkpoint after each epoch. It allows to resume training from any point in time or revert to the best model based on the original MSE score on the validation dataset. We did not use the test dataset during training.

After training is over for all models, we evaluated the results by using a hold-out test dataset. Pipeline for evaluation is relatively lean - best performing model according to original validation MSE score is loaded from the respective checkpoint and evaluated by two metrics - scaled test MSE and original test MSE. Original test MSE metric obtained by using persisted Scaler, which we fit initially on train data only. In addition, we also evaluated models on series of experiments to check how different models behave on data with added noise or data collected with different lengths of sequence. Furthermore, we perform measurements of averaged time required for model forward propagation of one sample to compare different architectures. Lastly, we visualized some of the experiments to improve the understanding of model performance. For all mentioned experiments, no additional pipelines were created, except for the visualization part. In all of the cases, we used existing pipelines for model testing or data preprocessing tweaked and tuned with the help of input parameters.

[3] https://pytorch.org/docs/master/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html

Chapter 5

Experiments

5.1 RNN Architecture Search and Hyperparameters Tuning

To search for the best set of ANN parameters for optimal training, we chose to perform a grid search for three model parameters - LR, hidden layer size, and the number of layers. Therefore, a similar grid search was executed for all three types of models - simple RNN, GRU, and LSTM. Based on the results of research by Greff et al., 2017, we know that for LSTM most important hyperparameters are the learning rate and size of the hidden layer, which could be tuned independently from each other. However, no such research confirms the same for simple RNN and GRU model types. Hence, we used the same grid for all three model types and treated all hyperparameters as interdependent. We did this for fair competition between different model types. Table 5.1 shows the exact values of hyperparameters we choose for tuning. In summary, for each architecture, we executed 18 experiments based on a defined hyperparameters grid.

Speaking about LR hyperparameter, initially, we planned to tune one more value which is 0.005, but training with such a rate was inefficient and did not converge. As a result, we discard this value and end up with only two possible values for LR. We did not choose any smaller values since we would use LR Scheduler for final training, which will reduce the rate based on training progress. As for the hidden size hyperparameter, we choose its values based on the number of target features equal to the number of DOF in our model. Hence, the hidden layer sizes are $1\times$, $3\times$, and $5\times$ by number of DOF. Lastly, we have chosen values for the number of layers hyperparameter in the same way. We did it by taking multiplication factors for model hidden size hyperparameter with step 2 to enable more diverse models. However, considering the requirement we have about the real-time model response, we kept the overall architecture relatively simple. We intentionally did not go above five recurrent layers in the model.

Hyperparameter	Grid Values		
Number of Layers	1	3	5
Hidden Size	23	69	115
Learning Rate	0.001	0.0005	-

TABLE 5.1: Grid used for hyperparameter tuning of ANN models

We have collected results of hyperparameter tuning of simple RNN model in the table 5.2. Also, we collected results for GRU and LSTM models in tables 5.3 and 5.4, respectively. In bold, we have highlighted the lowest achieved MSE score based on the standardized validation dataset for all three model types. In general, we could conclude that for all models LR parameter plays an important role. Lower values of LR are preferable to achieve faster convergence and make training more stable. As

for the number of layers hyperparameter, with the increase of its models, shows a worse validation MSE score. This rule holds in general except few cases with small hidden feature sizes. Such behavior on validation data is probably related to model overfitting to training data due to the high amount of parameters when the number of layers is higher. Lastly, higher values of hidden features size hyperparameter lead to lower validation MSE score in general, which indicates that such models can generalize better on previously unknown data. As a result, we could conclude that models, which have a higher amount of hidden features and smaller amounts of recurrent layers, are powerful enough to generalize better on validation data and avoid overfitting.

Learning Rate = 0.001		Hidden Size			Learning Rate = 0.0005		Hidden Size		
		23	69	115			23	69	115
Number of Layers	1	0.02120858	0.00766305	0.00715601	Number of Layers	1	0.02349921	0.00645291	**0.00376442**
	3	0.01211087	0.00763333	0.00838280		3	0.01528858	0.00481066	0.00434209
	5	0.01350686	0.01054625	0.01346354		5	0.01150493	0.00759974	0.00796293

TABLE 5.2: Validation MSE loss score of hyperparameters tuning for RNN model

Learning Rate = 0.001		Hidden Size			Learning Rate = 0.0005		Hidden Size		
		23	69	115			23	69	115
Number of Layers	1	0.00684158	0.00330278	0.00481885	Number of Layers	1	0.00455494	**0.00268362**	0.00307620
	3	0.00723735	0.00709713	0.00802608		3	0.00308833	0.00441637	0.00576102
	5	0.00733653	0.00956555	0.01528571		5	0.00482277	0.00432470	0.00819317

TABLE 5.3: Validation MSE loss score of hyperparameters tuning for GRU model

Learning Rate = 0.001		Hidden Size			Learning Rate = 0.0005		Hidden Size		
		23	69	115			23	69	115
Number of Layers	1	0.00368882	0.00221740	0.00237145	Number of Layers	1	0.00274252	0.00172109	0.00171244
	3	0.00284546	0.00205254	0.00285071		3	0.00270155	**0.00158603**	0.00172317
	5	0.00353208	0.00338600	0.00418648		5	0.00266886	0.00190042	0.00189096

TABLE 5.4: Validation MSE loss score of hyperparameters tuning for LSTM model

As a next step, we trained the most promising architectures for each model type after successfully identifying the most optimal parameters. Finally, we have added one additional architecture for GRU and LSTM types with the same hyperparameters identified for simple RNN. This time we performed training using a more complicated pipeline compared to the one used for hyperparameters tuning. It is described in detail in section 4.2.2. The following section will give more details about the results of the final model training.

5.2 Final Models Training

To train the final models, we used a pipeline described in section 4.2.2. After successful training, we will evaluate each model on a hold-out test dataset. Since training for final models consists of multiple epochs, we chose only models from epoch with the lowest MSE score based on the original non-scaled validation dataset for the final evaluation. To calculate such an MSE metric, we used persisted Scaler object, which was fit initially on train data only. We collected all training results in table 5.5,

including epoch number on which best score for original validation MSE achieved. Also, training for all models was executed with a learning rate = 0.0005. In this table, models RNN_1_115, GRU_1_69, and LSTM_3_69 are found by hyperparameters tuning. In contrast, we added models GRU_1_115 and LSTM_1_115 to the list to compare final results on equal with baseline model RNN_1_115. In table 5.5, we also highlighted the best performing model in bold if to judge by scaled and original validation MSE scores. The most successful model is LSTM_1_115, according to those metrics. Its training took a moderate number of epochs - 34.

Model Name	Number of Layers	Hidden Size	Scaled Validation MSE	Original Validation MSE	Epochs to Train
RNN_1_115	1	115	0.00327048	0.00433565	27
GRU_1_69	1	69	0.00114785	0.00180974	90
LSTM_3_69	3	69	0.00144839	0.00174503	47
GRU_1_115	1	115	0.00112260	0.00146464	455
LSTM_1_115	1	115	**0.00108824**	**0.00116210**	34

TABLE 5.5: Candidate models training results

5.3 Final Models Evaluation

After training is over for all candidate models, we evaluated all of them on a hold-out test dataset in a series of experiments. We did not use this data before, neither during model training nor as criteria to make a model selection. If not stated otherwise, we used the MSE score on the original non-scaled test dataset as a metric in all of the experiments. This metric is calculated with the help of previously persisted Scaler, which was fit on train data only.

Experiment 1: First, we assessed model performance in general on the test dataset by calculating scaled and original MSE scores for each model. Original test RMSE metric is calculated simply as $\sqrt{Orig.Test.MSE}$.

Model Name	Scaled Test MSE	Original Test MSE	Original Test RMSE
RNN_1_115	0.00383323	0.00657948	0.08111398
GRU_1_69	0.00262797	0.00364660	0.06038707
LSTM_3_69	0.00427745	0.00333634	0.05776109
GRU_1_115	0.00228220	**0.00144345**	**0.03799275**
LSTM_1_115	**0.00184871**	0.00167773	0.04096008

TABLE 5.6: Candidate models evaluation results on test dataset

Results: We have organized the results of the evaluation in table 5.6. In this table, we highlighted in bold the lowest scaled and original test MSE scores. Interesting to note that those scores belong to two different models. If to judge by scaled test MSE metric, then the most performing model is the same as per validation metrics - LSTM_1_115 (see table 5.5). However, if the original test MSE metric is to judge, which is much more important for practical application of the model, then the most performing model is GRU_1_115. Also, the less performing model is simple RNN_1_115, if to judge by the same criteria - original MSE score.

Experiment 2: Next, we checked how models perform on individual files from which the test dataset was composed. To check this, we executed all models on each

file and measured the RMSE score per file. Also, we calculated averaged RMSE score across all models.

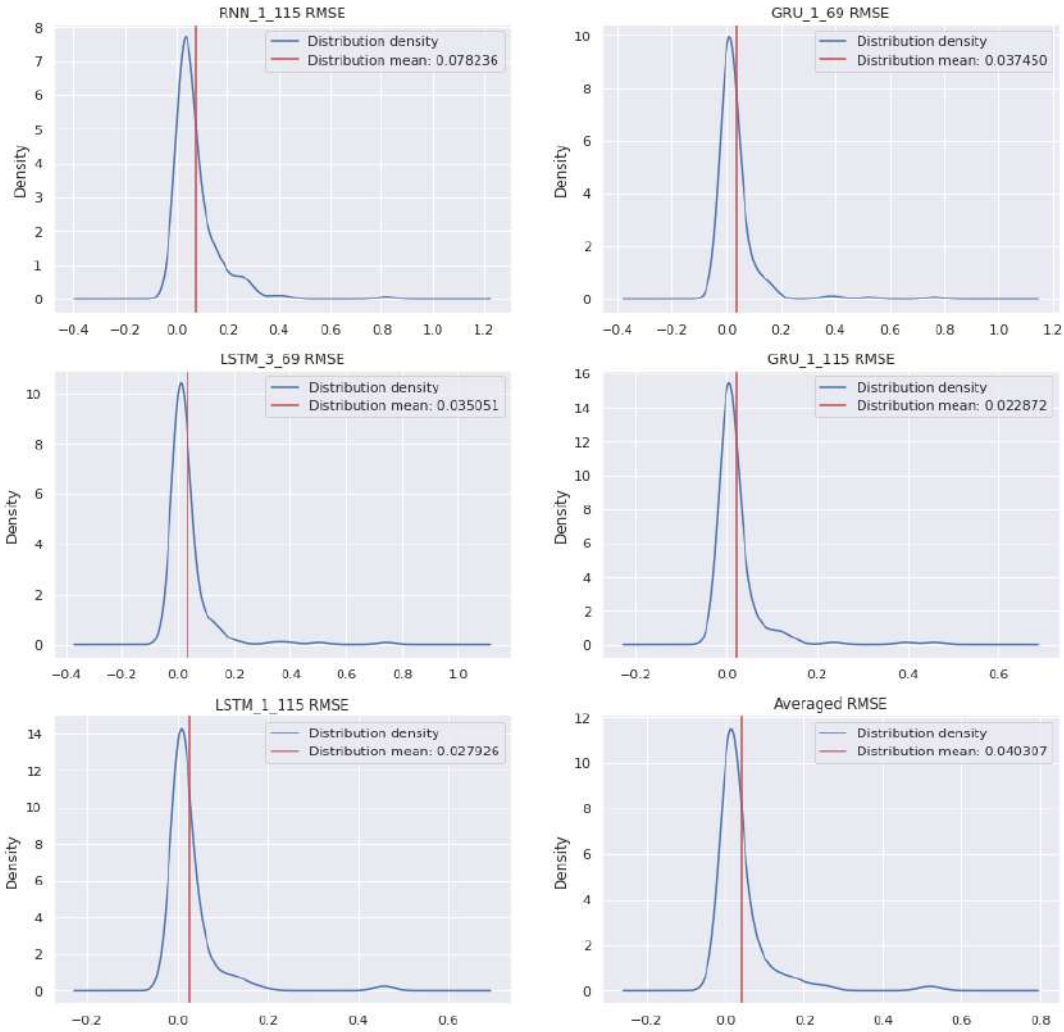

FIGURE 5.1: Test files RMSE density plot

Results: We collected the results of this experiment on figure 5.1. There are a density plot of RMSE values for each model separately and averaged RMSE score across all models. Each plot visualizes the distribution of RMSE values across all 158 files in the test set estimated with kernel smoothing. From this figure, we could conclude that model GRU_1_115 has the highest peak with the lowest mean value, indicating the lowest RMSE score for a broader list of files. It is closely followed by model LSTM_1_115, which shows the second-best performance. Models LSTM_3_69 and GRU_1_69 show similar results, while model RNN_1_115 shows the highest RMSE scores per file among all models. It is indicated by the low height of the peak coupled with the highest mean value and some outliers on the right side of the density plot further away than for any other model.

Experiment 3: Next, we would like to measure and assess the average time all models require to execute forward propagation of one sample (batch size equals one) of size sequence length $\times$ the number of input features. When the model is applied in practice, it needs to predict just one sample on each time step. Here we only measure how much time the model requires to calculate its prediction without backward propagation and optimization. It allows us to judge how fast the model response is.

To do this, we took all 158 test files and performed prediction in batches of size one on both devices - CPU and GPU. By doing this, we calculated the average time of forward propagation per model and device. All measurements were performed on a Google Colab Pro High-RAM environment powered by GPU.

Device	RNN_1_115	GRU_1_69	LSTM_3_69	GRU_1_115	LSTM_1_115
CPU	1.971	5.219	13.886	5.459	5.025
GPU	1.354	1.356	3.337	1.354	1.379

TABLE 5.7: Average timing of forward propagation for one sample in milliseconds

Results: We summarized the results of this experiment in the table 5.7, all timings provided in milliseconds (ms). According to these results, when the model is executed on a GPU device, there is no noticeable difference between different RNN types models with only one recurrent layer. However, model LSTM_3_69 with three-layered architecture is almost three times longer to perform a forward propagation of one sample than one-layered models. When a model is executed on a CPU device, the results are different. The fastest model on CPU is RNN_1_115, which has a timing slightly below 2 ms. On the other hand, gated RNN models with one recurrent layer take around 5-5.5 ms for forward propagation, two and a half times slower than simple RNN. Furthermore, same as with GPU device, the longest forward propagation time belongs to the LSTM model with three layers - around 14 ms, which is almost three times slower than other gated RNN models.

Experiment 4: Next, we would like to assess model performance on sequences with a length less than 100, which we used for initial model training. To do so, we choose to assess models' performance similar to in experiment 1. This time, all models were assessed on the initial test dataset for reference and three other test datasets with sequences of 50, 20, and 10. This experiment should indicate if models could generalized well or "memorize" initial distribution from train data due to similar movements.

Model Name	Sequence 100 MSE	Sequence 50 MSE	Sequence 20 MSE	Sequence 10 MSE
RNN_1_115	0.00657948	0.00700876	0.01623526	0.08437408
GRU_1_69	0.00364660	0.01486074	0.09703008	0.20517447
LSTM_3_69	0.00333634	0.01608654	0.20361219	1.40760101
GRU_1_115	0.00144345	0.00482661	0.03463733	0.12392941
LSTM_1_115	0.00167773	0.00276157	0.00834123	0.07718295

TABLE 5.8: Original MSE score on test dataset with different sequence length

Results: We have gathered results on this experiment in the table 5.8. Overall, for all models, performance decrease with the decrease of sequence length. However, a pattern is different for each model. The three-layered model LSTM_3_69 showed the worst results on all sequences of shorter length, despite intermediate results on the initial sequence of length 100. Performance of model GRU_1_69 also decreases significantly, but not as fast as for model LSTM_3_69. Model GRU_1_115 showed the second-best MSE score on sequence length 50, but shorter sequences were more challenging to predict precisely. Model RNN_1_115 showed the second-best result on the shortest sequence of length 10, which was not expected since this model had the highest MSE score on the original sequence of length 100. Also same simple RNN model showed the second-best score on the sequence of length 20. The most

performing model on all sequence lengths was model LSTM_1_115, which has the second-best MSE score on the initial sequence of length 100 and best scores on all shorter sequences.

Experiment 5: Following up experiment 4 above, we also measured the average time required for forwards propagation of one sample using sequences of shorter length. We again used sequences of length 50, 20, and 10 to assess timings for each of them. The overall setup of this experiment is the same as in experiment 3. This experiment aims to measure how much time the model requires to calculate its prediction when input sequences have a shorter length than the one used for training. As in experiment 3, measurements does not include backward propagation and optimization step.

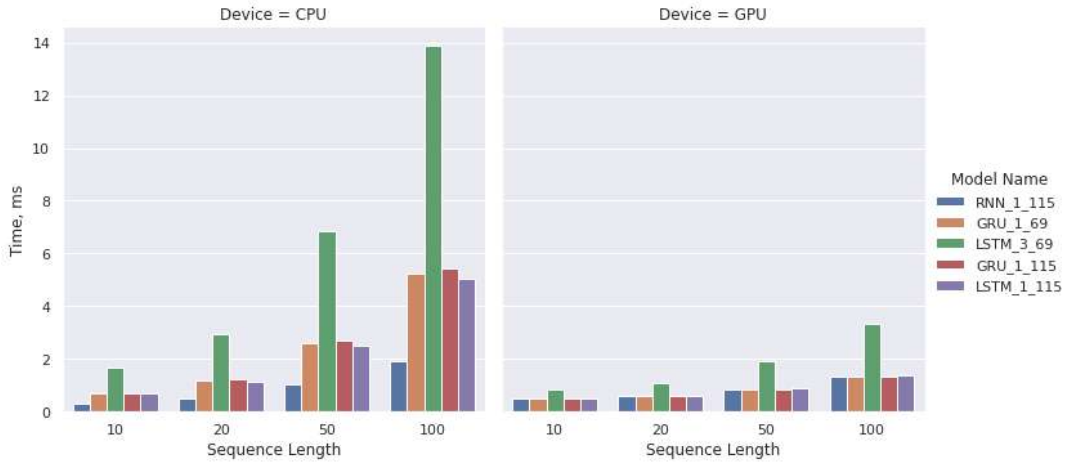

FIGURE 5.2: Average timing of forward propagation for one sample with different sequence length in milliseconds

Results: A figure 5.2 shows the results of this experiment, all timings recorded in milliseconds (ms). Results for the sequence of length 100 were added here for reference based on table 5.7. In general, there is nothing unexpected in the results. With the reduction of sequence length, forward propagation time also decreases. For example, with the sequence of length 50, the timing of forward propagation approximately equal to half of the timing for the sequence of length 100. The same rule applies for the sequence of length 20 if compared with the sequence of length 50. For the sequence of length 10, a reduction in time still happens, but the factor not that big anymore, especially for GPU devices. Interesting to note that for model RNN_1_115, execution on CPU device is faster than on GPU for sequences of lengths 10 and 20. Due to some administrative costs applied to manage asynchronous GPU execution, calculations are simple enough and executed faster on the CPU for short sequences. It is an important observation considering edge device limitations of computational power.

Experiment 6: Last but not least, we checked candidate models' performance on test data with added noise. We added noise by sampling it from random Gaussian normal distribution with mean and standard deviation from data. We added noise on a per-feature basis and to input features only. We used two levels of noise for this experiment 1% and 5% of noise. This experiment will show how resilient models are to the noise, which could appear in practice due to faulty sensors or other reasons.

Results: We collected the results of this experiment in table 5.9. Original MSE

Model Name	No Noise Original MSE	Noise 1% Original MSE	Noise 5% Original MSE
RNN_1_115	0.00657948	0.01176509	0.09137829
GRU_1_69	0.00364660	0.00525671	0.03442241
LSTM_3_69	0.00333634	0.24412315	1.47980811
GRU_1_115	0.00144345	0.00247943	0.02095914
LSTM_1_115	0.00167773	0.00669904	0.07798817

TABLE 5.9: Candidate models evaluation results on test dataset with added noise

metric without noise was added for reference from experiment 1 table 5.6. According to these results most resilient to noise is model GRU_1_115. Its MSE score degraded by approximately factor 2 only with 1% of noise and by factor 15 with 5% of added noise. Model GRU_1_69 achieved second best results on noisy data - MSE score degraded by factor 1.5 and 9.5 for noise levels 1% and 5% respectively, but absolute values still lower than for model GRU_1_115. Finally, model LSTM_1_115 performance degraded by factors 4 and 46 with the addition of noise. Three-layered model LSTM_3_69 performance degradation with noise was the highest among all other models.

5.3.1 Results Visualization

It also makes sense to visualize some of the predictions for files included in the test set to assess the mentioned models' results. In our case, the target feature space has a size of 23. As a result, such visualizations are too big to be included in the body of this thesis. Hence we have included all of them in appendix A. In this section, we will only provide a brief description and a reference to the respective figure. Since it is impossible to visualize all 158 files of the test set, we instead decided to check which files all models perform best or worth. Results of this check are collected in table 5.10. Out of 158 test files, we choose the top 5 files with the highest and lowest average original MSE score across all models. All figures mentioned below are plotted in original non-scaled values.

Filename	RNN_1_115 MSE	GRU_1_69 MSE	LSTM_3_69 MSE	GRU_1_115 MSE	LSTM_1_115 MSE	Average MSE
Highest average original MSE on test set files						
reach_1_12_0.5	0.66973072	0.27498683	0.15527545	0.21080871	0.21323116	0.30480658
reach_3_19_0.5	0.18227339	0.58483177	0.25480074	0.15544073	0.20417245	0.27630382
reach_2_26_0.5	0.06152614	0.00662388	0.55219090	0.01066955	0.00389768	0.12698163
reach_10_16_0.5	0.07737701	0.16701087	0.11544502	0.01230989	0.02224136	0.07887683
reach_1_19_0.5	0.14101827	0.01648147	0.01007973	0.05558154	0.03505260	0.05164272
Lowest average original MSE on test set files						
reach_13_23_2	0.00006896	0.00000944	0.00000692	0.00000019	0.00000660	0.00001842
reach_23_26_2	0.00012363	0.00000503	0.00000688	0.00000028	0.00000672	0.00002851
reach_4_5_2	0.00019248	0.00000345	0.00000494	0.00000043	0.00000447	0.00004116
reach_18_27_1	0.00018506	0.00000645	0.00000609	0.00000024	0.00001350	0.00004227
reach_23_24_2	0.00019856	0.00001117	0.00000212	0.00000075	0.00000809	0.00004414

TABLE 5.10: Top-5 files by highest and lowest average original MSE score on test set

Based on the content of table 5.10, we decided to visualize file reach_1_12_0.5, which turns out one of the most challenging for all models. During visual inspection of figure A.1, it is obvious why averaged MSE score is high - model RNN_1_115 predictions are very volatile and not stable. The same happens with model LSTM_1_115 but on a smaller scale. Despite this volatility, all models are still good at capturing the overall dynamics of the model - the overall trajectory is correct.

We also visualized file reach_13_23_2 to show all models' performance on file with the lowest averaged MSE score. It could be found on figure A.2. From this figure, it is clear that all models dealt very well with predictions, except for the baseline RNN_1_115 model, which experiences some glitches predicting the tor_ra_sh_ab_ad feature. However, except mentioned feature, even the baseline model performs well on this file.

The next visualization on figure A.3 demonstrates the ability of model LSTM_1_115 to generalize when dealt with sequences of shorter lengths. According to figure A.3, there is almost no difference between model predictions with a sequence length of 100 and a sequence length of 50. For the majority of features, those two plots coincide or very close to each other. The sequence of length 20 predictions is not that precise anymore and usually shows the same shape but at some distance from the correct prediction. However, for some features, predictions are precise even with such a short sequence. Finally, the model rarely predicts close to correct results with the sequence of length 10. Nevertheless, it can still capture overall system dynamics by predicting the correct shape almost for all features.

Lastly, figures A.4, and A.5 shows model GRU_1_115 performance on data with added noise. Noise is added per feature basis from Gaussian normal distribution. From figure A.4, we could conclude that 1% of normal Gaussian noise added to data does not prevent the model from predicting this specific file. For all features, the overall trend is closely following true values. As for figure A.5, 5% of normal Gaussian noise makes model prediction much more volatile. However, the model can still capture the overall trend in data, and noisy predictions follow it. For many features visually, it looks like model predictions are far away from true values, but this is due to the tiny scale of those features.

Chapter 6

Results

6.1 Conclusion

Based on the results described in sections 5.2 and 5.3, we could conclude that RNN networks can capture recurrent relation of Inverse Dynamic problem and approximate successfully joint torques by using history of joint kinematics data (joint positions, velocities, and accelerations).

As part of this research, we have identified and evaluated five different RNN model architectures that could be used to solve Inverse Dynamics problem. Those models are RNN_1_115, GRU_1_69, LSTM_3_69, GRU_1_115 and LSTM_1_115. Table 5.5 contains additional details about these models. Initially, all models were evaluated on a hold-out test dataset (table 5.6 for reference) to assess their performance on unknown data. In this research, we organized joint kinematics data in sequences of length 100 for initial model training. However, we also evaluated all models on sequences of shorter length to validate the ability of models to generalized well to data and not to "memorize" it. Also, we did measurements of the time required for models forward propagation of one sample. Such measurements, shown in table 5.7 and on figure 5.2, will help to judge which model is faster or slower than others, which is vital for future practical application of models. Lastly, we evaluated all models for resistance to noise added to the data, which might occur in practice due to faulty hardware. Table 5.9 shows information about how different models behave with noise added to the data.

Based on table 5.6, we could conclude that using RNN architectures with more than one recurrent layer does not bring any significant benefits from the performance point of view. Moreover, table 5.7 and figure 5.2 indicate that RNN models with more than one layer are significantly slower than models with one layer only. Hence the usage of a multi-layered RNN model for real-time approximation of ID problems will be complicated. In addition, considering results in table 5.8, we could also conclude that multi-layered RNN models poorly perform on sequences different from those used during training. It might indicate that model is not generalized well and remembers the trends in data due to the high amount of parameters. Finally, according to table 5.9, the three-layered LSTM model is very vulnerable to the noise in data, and its performance degrades significantly even with minor noise added.

In this research, gated RNN models (LSTM_1_115 and GRU_1_115) with a higher number of hidden features demonstrated the most successful results, as shown in table 5.6. However, it is worth noting that such models are relatively slower than simple RNN models when executed on CPU devices (table 5.7 and figure 5.2 for reference). Also, when we tested such models on data sliced in sequences of shorter length than the one used during training, their performance degrades less than for other models, except the simple RNN model. Results of this experiment could be

found in table 5.8. Hence we could conclude that a higher amount of hidden features allow a model to generalize better even on different sequence lengths than the model was trained on. Lastly, models LSTM_1_115 and GRU_1_115 demonstrate higher noise resilience that all other models (table 5.9 for reference). It is a good trait if considering model application in practice, where noisy data might occur. It is also worth highlighting that other GRU model GRU_1_69 demonstrate high resilience to noisy data, which assumes a specific feature related to GRU architecture.

We used a simple RNN model as the baseline model in this research. Its overall performance was lowest among all models, as shown in table 5.6. Also, based on table 5.9, model RNN_1_115 demonstrated a low level of noise resilience among other models. However, the simple RNN model has other qualities, which worth mentioning here. First of all, according to table 5.7 and figure 5.2 simple RNN model has the lowest timing for forward propagation of one sample. It is expected since such a model is less computationally expensive than other models. In section 2.2, we showed why simple RNN models are easier to calculate. Secondly, the simple RNN model also demonstrated good generalization capabilities when evaluated on test data of shorter sequence lengths. At this point, it is not completely clear how model RNN_1_115 outperformed model GRU_1_115 on short sequences, and further experiments might be required to answer this question.

In summary, we could recommend applying gated RNN models with a single layer and a high number of hidden features to solve Inverse Dynamics problems if model performance and noise resilience play an essential role. In particular, we would recommend using Gated Recurrent Unit (GRU) RNN type in such cases over LSTM and simple RNN types, according to the results of our experiments. However, suppose the time required for the forward propagation step of one sample on a CPU device is more important than a model performance. In that case, simple RNN architecture could be an excellent alternative to more complicated GRU and LSTM RNN types. Such a model should have a single recurrent layer, ReLU non-linearity function, and a high number of hidden features. Gated RNN models require much more time to be calculated on CPU, which could be a critical decision point due to the absence of GPU on edge devices.

6.2 Discussions

There is one major limitation for this research setup - no ability to evaluate model response time of hardware close to edge device specifications. All experiments we have recorded in table 5.7 and figure 5.2, we executed on Google Colab Pro environment. Consequently, we could not use any obtained measurements to derive conclusions about the real-time model response for the forward propagation of one sample. Because of this reason, we could only compare models between each other based on results in mentioned tables. However, the conclusion about how real-time model response is only possible on the edge device. Hence, such a conclusion is not possible in the scope of this research, but we could indicate which models forward propagation is faster or slower than for others.

Another limitation, which worth mentioning, is that the dataset for this research consists only of movements without the intended movement of the hand as described in section 3.1. Such movements focus on elbow and shoulder joints, and hand joints are not fully covered. Because of this reason, many features in the dataset have a constant value of 0, as we have documented in section 3.2. However, despite this dataset specifics, all models still predicting associated joints torques for hand

DOF (figure A.2 shows the model prediction for wrist and fingers DOF). It indicates that RNN models can capture recurrent relations of the Inverse Dynamic (ID) problem.

Lastly, we choose Recurrent Neural Networks (RNN) as a possible solution approach for the ID problem in this research. We made such a choice for two reasons: the recurrent internal structure of ANN itself, which allow approximating dynamics system as shown by Ogunmolu et al., 2016, and because RNN was used in the past to approximate ID problem in different settings by Chen and Wen, 2019; Hartmann et al., 2012. Also, Draye et al., 1995 apply RNN to approximate arm dynamic from surface EMG signals. However, this does not mean that other ANN types could not be used as an alternative to RNN. For example, we explored neither Convolution Neural Networks (CNN) nor Multilayer Perceptron (MLP) nor any other ANN types in this work. Hence, the results of this research do not discard any other ANN types as a possible alternative to RNN for ID problem approximation.

6.3 Future Work

Due to the limited timeline of this thesis, it was not possible to cover all aspects of the research. Initially, together with colleagues from the Neural Engineering Lab (NEL) at West Virginia University, we plan to evaluate the final RNN model in MATLAB by linking it to the musculoskeletal model used for data generation. However, to achieve this final model need to be implemented and trained in a MATLAB environment. Hence, one of the future research directions is to find a way to import-export chosen model from Python to MATLAB environment or train it from scratch in MATLAB itself. Also, if the model will be trained in MATLAB, this raises other questions like performance assessment between models trained on two completely different ecosystems. As a result, a successful model in the Python environment able to solve the ID problem might be lacking behind in the MATLAB environment.

Another important aspect of this research was the length of the sequence. We obtained all of the results of this research described in the chapter 5 with a sequence of length 100, except specific experiments with shorter sequences. However, the choice of the sequence length 100 might not be optimal, and the sequence length should be evaluated as another hyperparameter. Considering the possible application of this model in practice and the result captured in table 5.7 and figure 5.2, optimal sequence length is crucial. It is a trade-off between faster execution of forward propagation and model performance. Hence, another direction of future work might be to evaluate the same pipeline on a sequence of shorter length and compare obtained results with this research.

In this research, we used a dataset initially collected with a sampling rate of 10 kHz, which means ten samples per 1 millisecond (ms). In section 4.2.1, we described why we have to down-sample it to the one sample per 1 ms. However, this choice was somewhat arbitrary, and we did not validate other sampling rates like one sample per 2 ms or 5 ms. Furthermore, based on results in table 5.7, more complicated gated RNN models require up to 6 ms for the forward propagation, which makes real-time usage of such models impossible with a sampling rate of one sample per 1 ms. One possible solution could be to sample data more seldom if this has no significant impact on model performance. As a result, we see another direction of future work evaluating different sample rates on overall model performance and comparing results with a sampling rate of one sample per 1 ms.

As an alternative to RNN models, we propose to consider Universal Transformer (UT), introduced and fully described in Dehghani et al., 2018. As highlighted in Dehghani et al., 2018, UT is "a parallel-in-time recurrent self-attentive sequence model that can be cast as a generalization of the Transformer model" and combines the benefits of both feed-forward and RNN models. Authors claim that UT outperforms Transformers and LSTM models on various sequence-to-sequence tasks and confirmed this with conducted experiments in the scope of Dehghani et al., 2018. Considering UT models' complexity, we assume that an overall model response time could be high. However, since UT models are easy to parallelize, as mentioned in Dehghani et al., 2018, one possible measure could be to run many models in parallel. Concrete decisions about UT model fine-tuning or any other Transformer model adaptation for ID problem is another direction of future work. However, it worth mentioning that due to the complexity of Transformer architecture, it might be challenging to use such models on edge devices.

As already mentioned in section 3.1, the dataset used in this research is limited by movements without intended hand motion. Hence, future collaboration with the Neural Engineering Lab (NEL) at West Virginia University is required to enrich the dataset with additional movement types. Once enriched, it would be interesting to evaluate current models on previously unknown data of different movement types to assess overall models' generalization capabilities. It also makes sense to train the same or new models on enriched datasets to assess possibilities to improve overall results despite evaluation. Since models trained on the current dataset already available, transfer learning could be used as one option on another dataset, or datasets themselves need to be merged and blended.

In addition to Inverse Dynamics (ID) problem, in robotics and biomechanics, when mapping between joint kinematics and joint torques needs to be found, Forward Dynamics (FD) problem also exists. In a nutshell, it is a problem to find a mapping between joint torques and joint kinematics on step t knowing joint kinematics of step $t-1$. Hence, FD problem approximation with the help of RNN models represents another promising direction of future research.

Lastly, there is an initiative at West Virginia University (WVU) to create a tool called Artificial Physics Engine (APE). The solution to the ID problem is a fundamental part of this tool. Hence, we could identify the last direction of future work by integrating currently obtained results for the ID problem solution as part of the APE tool.

Appendix A

Results Visualization

In this section following visualization are added:

- The figure A.1 shows a performance of all models on the file reach_1_12_0.5, which has the highest averaged MSE score;
- The figure A.2 shows a performance of all models on the file reach_13_23_2, which has the lowest averaged MSE score;
- The figure A.3 shows a performance of the model LSTM_1_115 on the file reach_13_23_2 when a prediction happens with sequences of different lengths;
- The figure A.4 shows a performance of the model GRU_1_115 on the file reach_13_23_2 with 1% of a normal Gaussian noise added to the data;
- The figure A.5 shows a performance of the model GRU_1_115 on the file reach_13_23_2 with 5% of a normal Gaussian noise added to the data.

Due to many target features and making figures readable, each figure occupies a separate page.

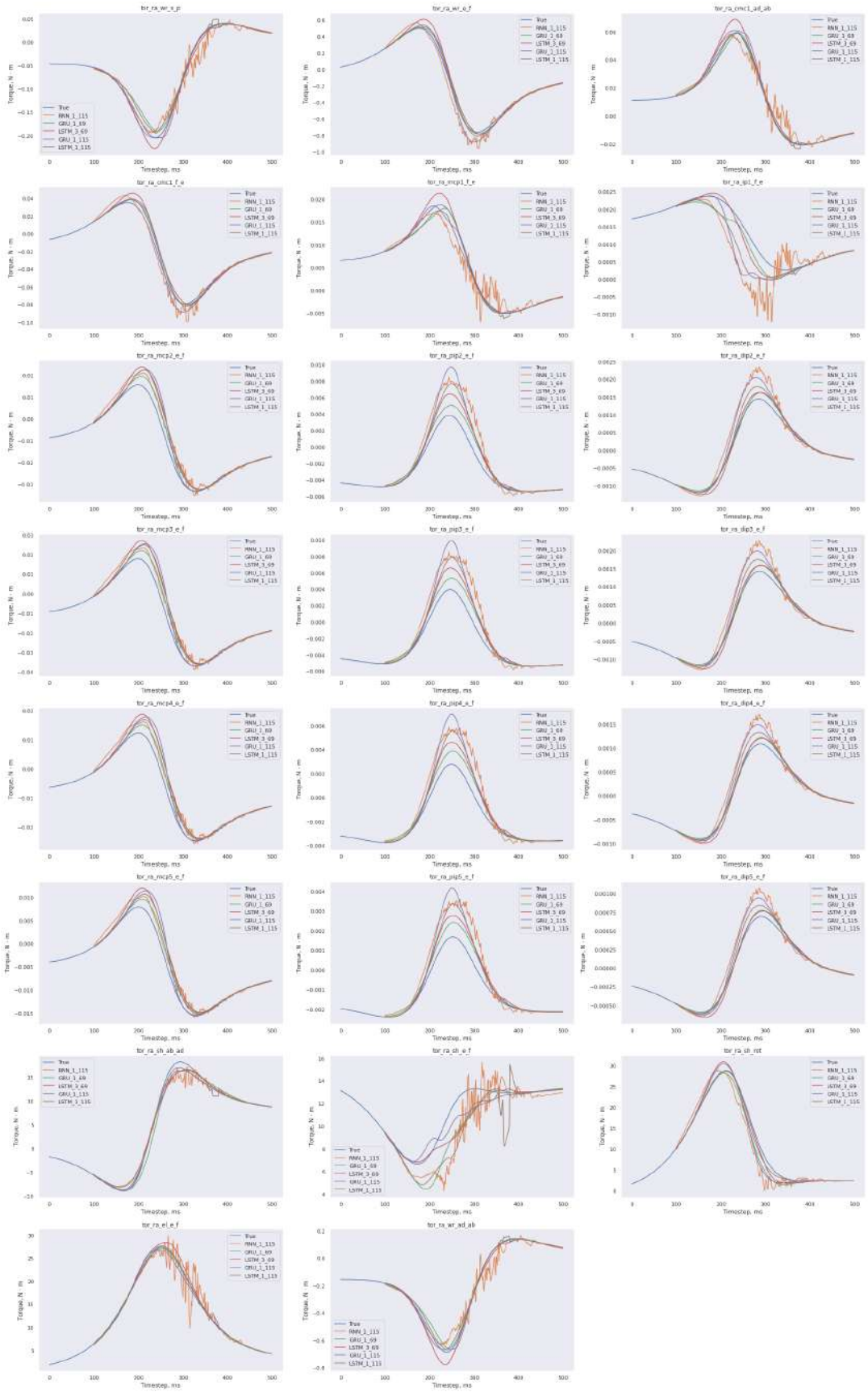

FIGURE A.1: File reach_1_12_0.5 predictions visualization with highest averaged MSE score

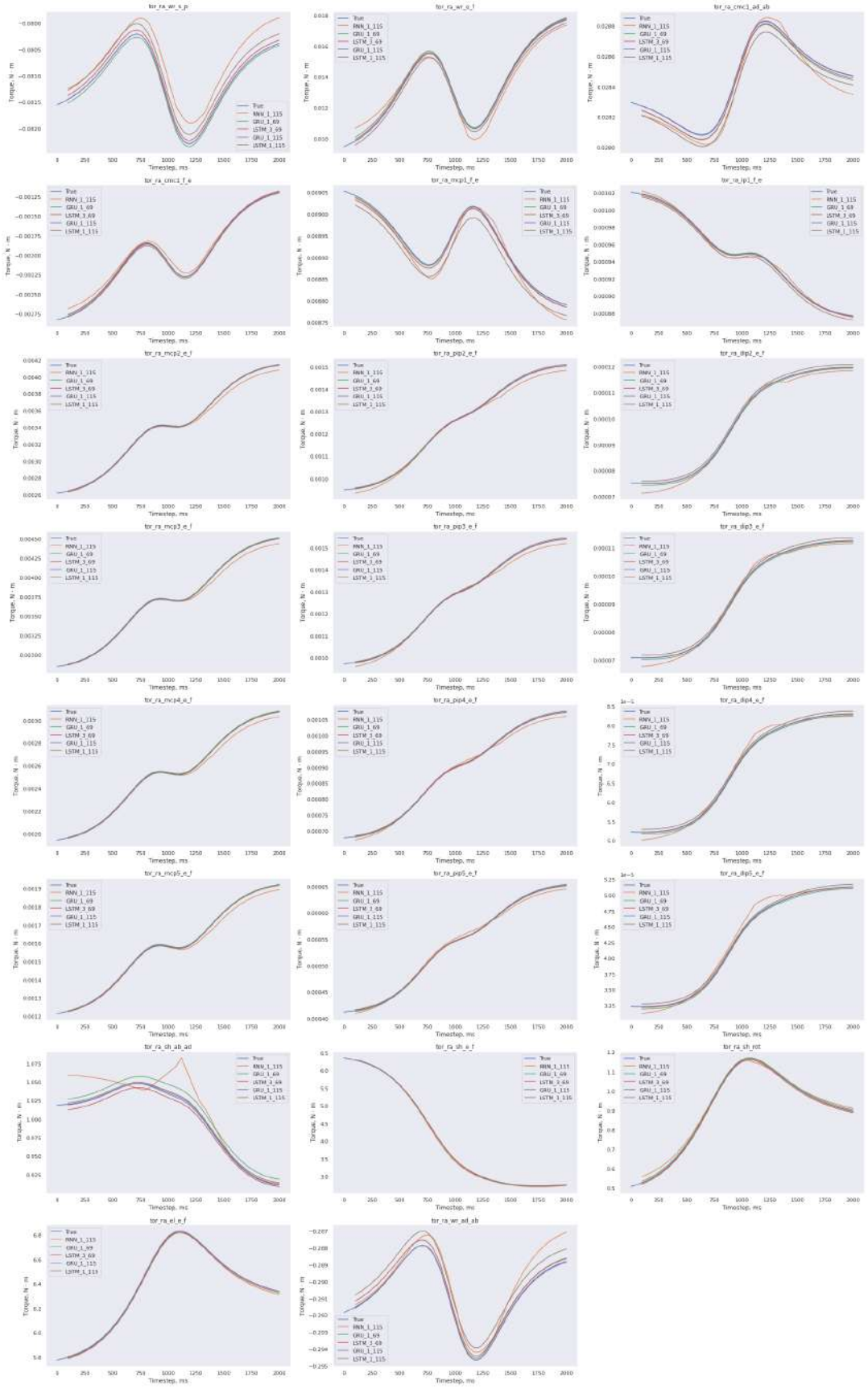

FIGURE A.2: File reach_13_23_2 predictions visualization with lowest averaged MSE score

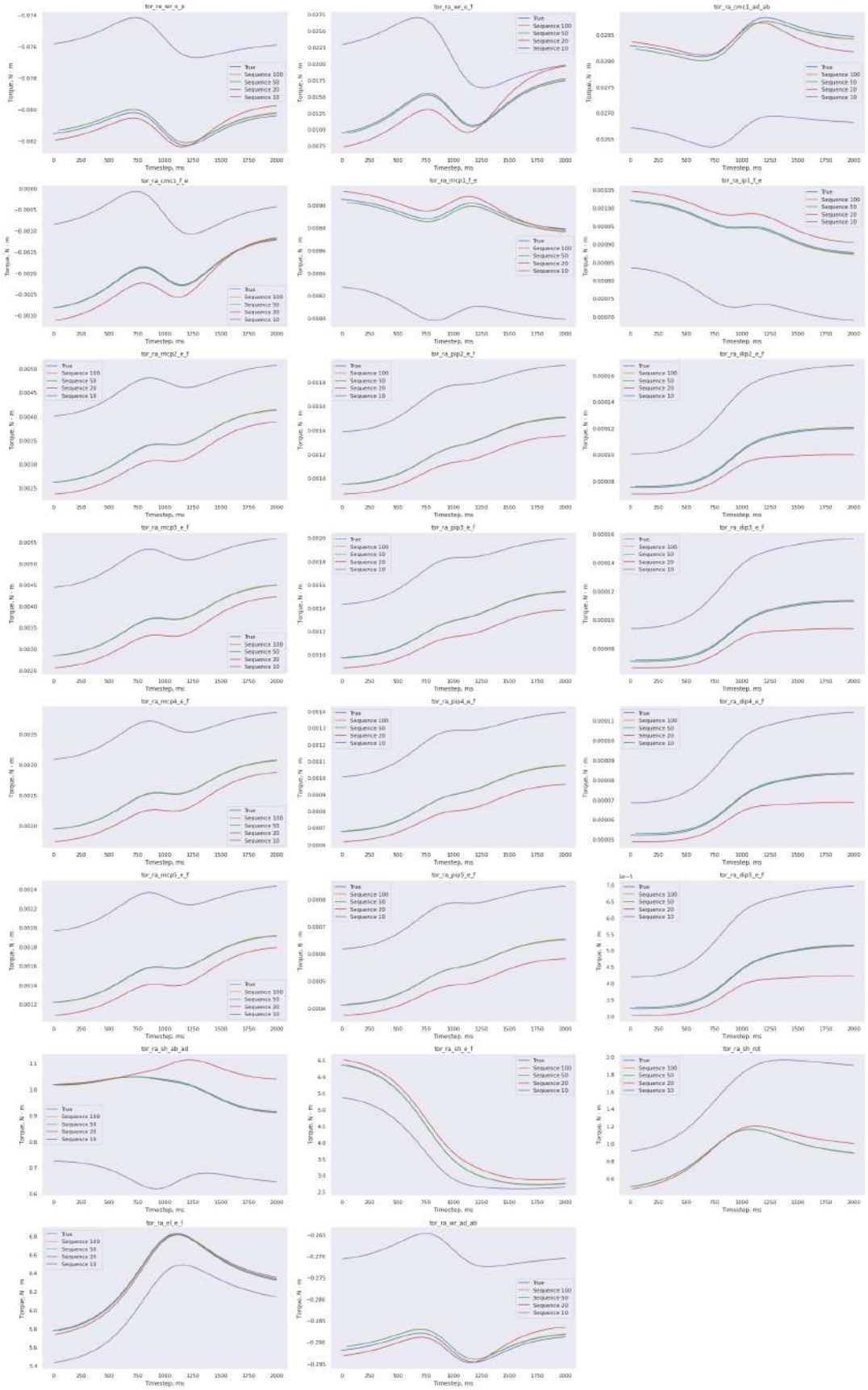

FIGURE A.3: File reach_13_23_2 predictions with LSTM_1_115 model and different sequences

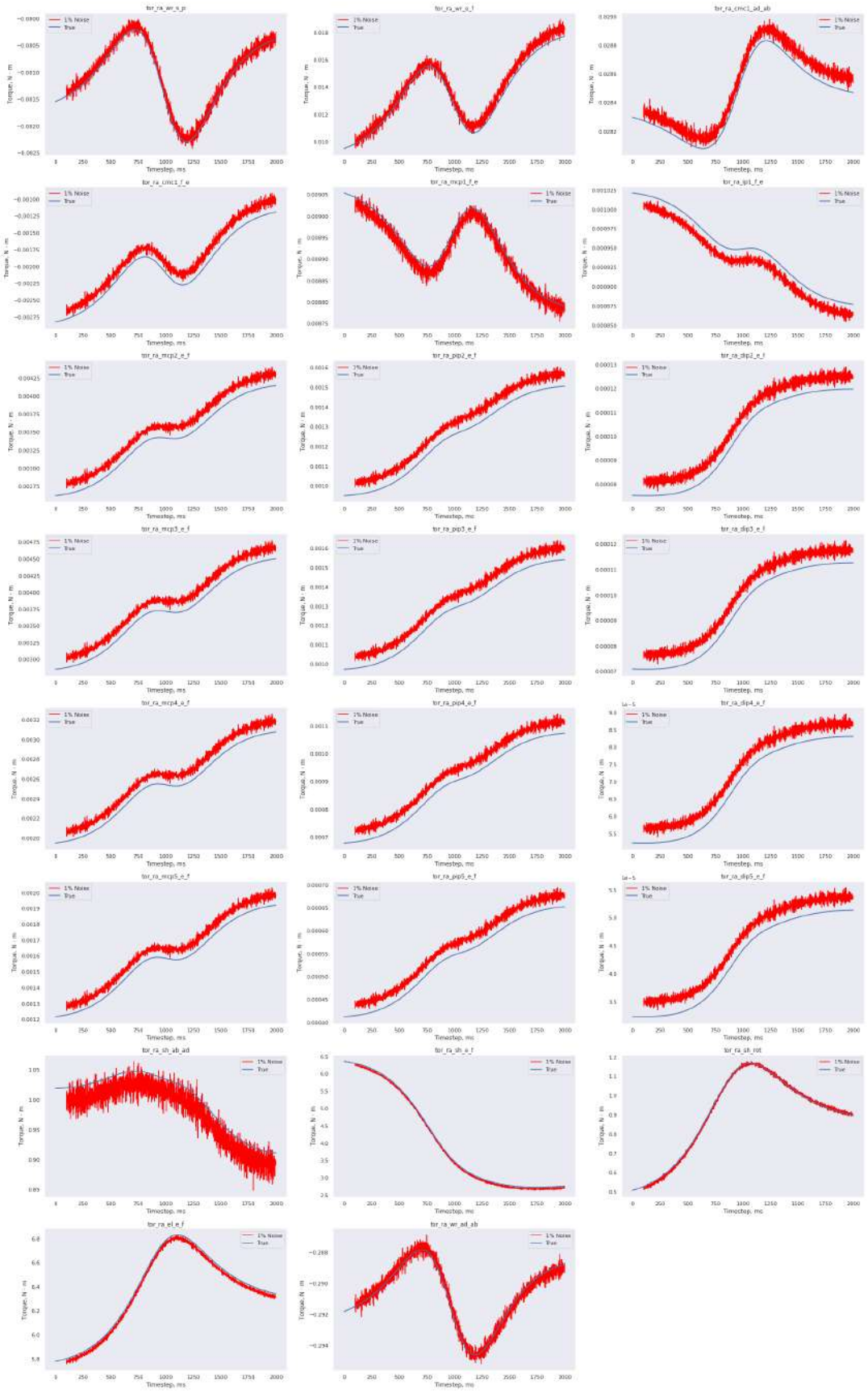

FIGURE A.4: File reach_13_23_2 predictions with GRU_1_115 model and 1% of noise

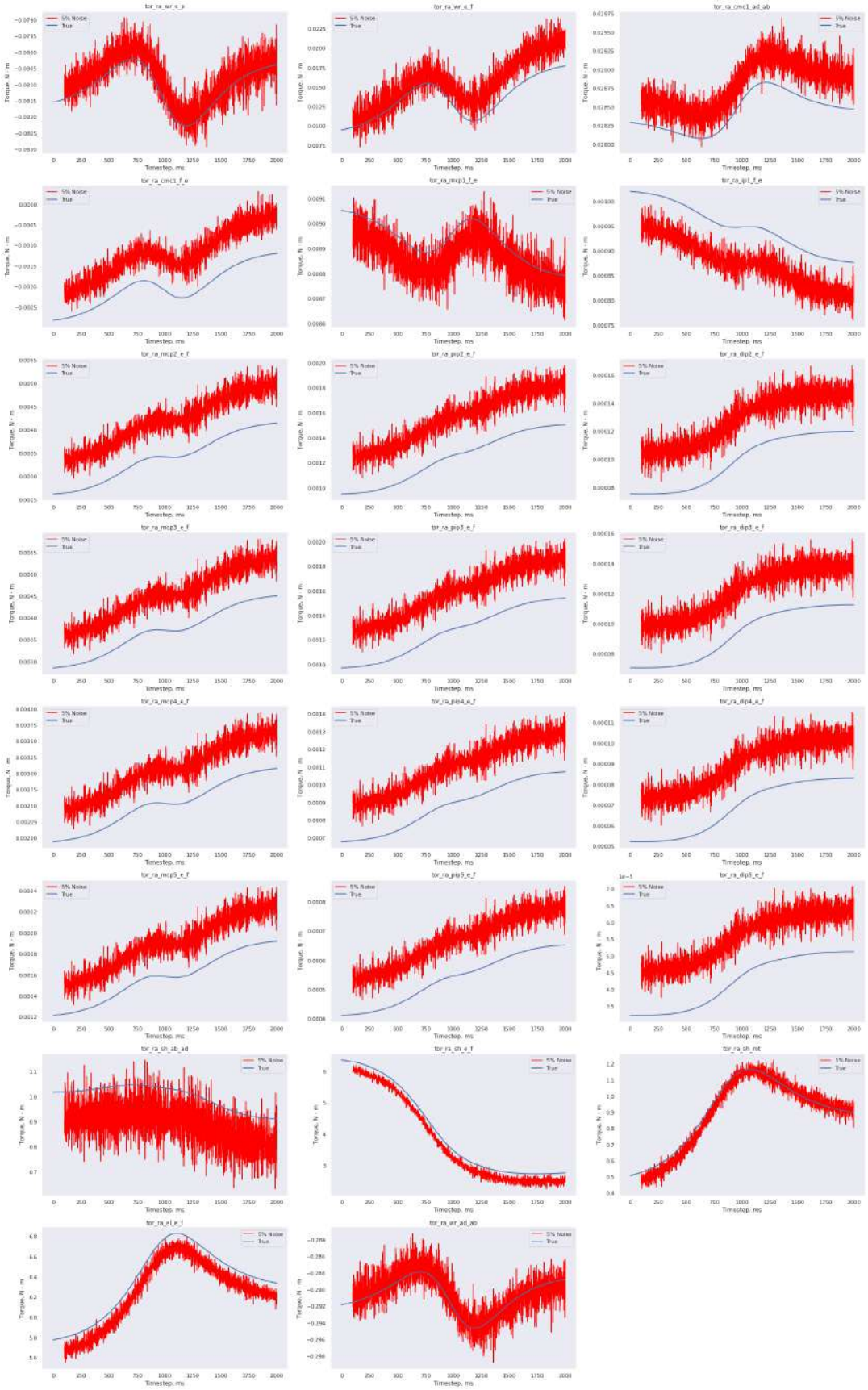

FIGURE A.5: File reach_13_23_2 predictions with GRU_1_115 model and 5% of noise

Appendix B

Simulink Model Details

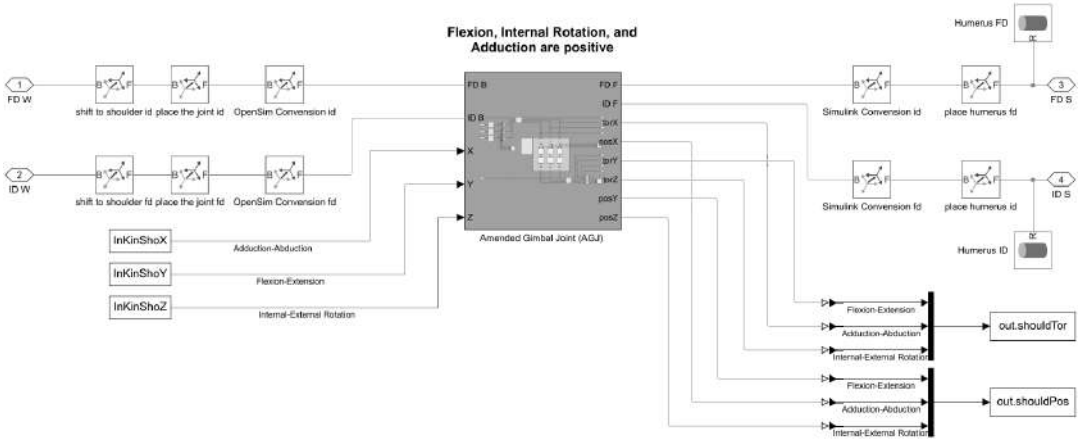

FIGURE B.1: Typical 3DOF joint used in arm and hand model by NEL at WVU

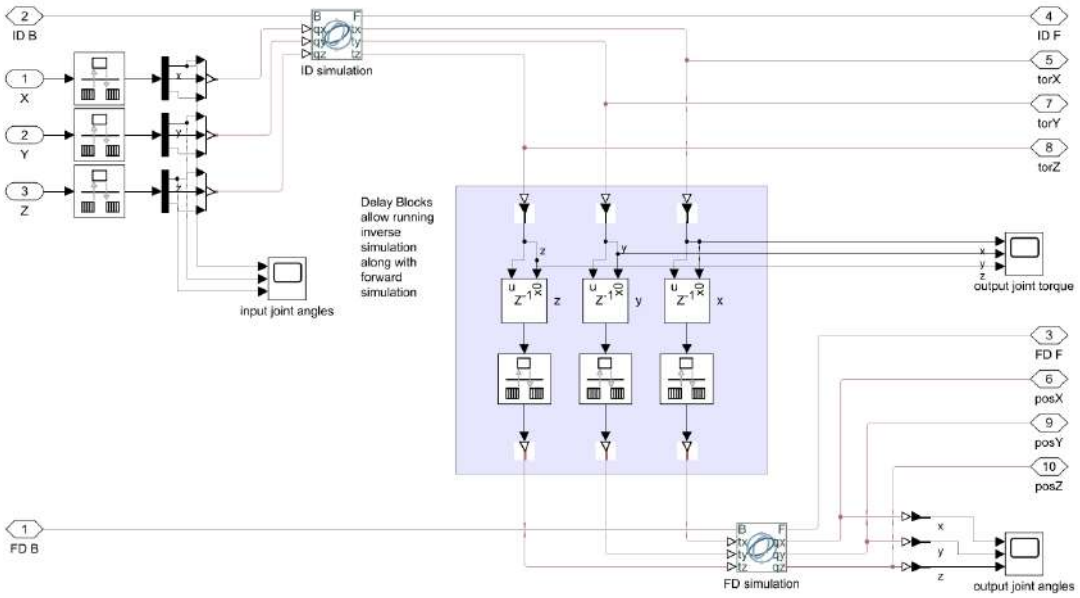

FIGURE B.2: Internal schematic of 3DOF joint by NEL at WVU

Appendix C

Source Code

We collected all of the code to execute this research in the following GitHub repository. However, we do not provide access to data since it is owned by Neural Engineering Lab (NEL) at West Virginia University (WVU).

Bibliography

Agur, Anne M. R. (1999). *Grant's Atlas of Anatomy, 10th Edition*.

Atzori, Manfredo, Matteo Cognolato, and Henning Müller (2016). "Deep Learning with Convolutional Neural Networks Applied to Electromyography Data: A Resource for the Classification of Movements for Prosthetic Hands". In: *Frontiers in Neurorobotics* 10, pp. 9–9. DOI: 10.3389/FNBOT.2016.00009.

Bengio, Y., P. Simard, and P. Frasconi (1994). "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166.

Chen, Shuyang and John T. Wen (2019). "Neural-Learning Trajectory Tracking Control of Flexible-Joint Robot Manipulators with Unknown Dynamics". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 128–135.

Cho, Kyunghyun et al. (2014). "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches". In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111.

Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555*.

Dehghani, Mostafa et al. (2018). "Universal Transformers". In: *arXiv: Computation and Language*.

Draye, Jean-Philippe et al. (1995). "Identification of the human arm kinetics using dynamic recurrent neural networks." In: *ESANN*.

Elman, Jeffrey L. (1990). "Finding Structure in Time". In: *Cognitive Science* 14.2, pp. 179–211.

Featherstone, Roy (2007). *Rigid Body Dynamics Algorithms*.

Graziano, Michael S. A. and Tyson N. S. Aflalo (2007). "Mapping Behavioral Repertoire onto the Cortex". In: *Neuron* 56.2, pp. 239–251.

Greff, Klaus et al. (2017). "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks* 28.10, pp. 2222–2232.

Hartmann, Christoph et al. (2012). "Real-Time Inverse Dynamics Learning for Musculoskeletal Robots based on Echo State Gaussian Process Regression". In: *Robotics: Science and Systems VIII, 9-13 July 2012, University of Sydney, Sydney, NSW, Australia*. Vol. 8, pp. 113–120.

Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever (2015). "An Empirical Exploration of Recurrent Network Architectures". In: *Proceedings of The 32nd International Conference on Machine Learning*, pp. 2342–2350.

Kingma, Diederik P. and Jimmy Lei Ba (2015). "Adam: A Method for Stochastic Optimization". In: *ICLR 2015 : International Conference on Learning Representations 2015*.

Lipton, Zachary C., John Berkowitz, and Charles Elkan (2015). "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *arXiv preprint arXiv:1506.00019*.

Ogunmolu, Olalekan P. et al. (2016). "Nonlinear Systems Identification Using Deep Dynamic Neural Networks." In: *arXiv preprint arXiv:1610.01439*.

Sobinov, Anton et al. (2020). "Approximating complex musculoskeletal biomechanics using multidimensional autogenerating polynomials." In: *PLOS Computational Biology* 16.12.

Zhou, Guo-Bing et al. (2016). "Minimal gated unit for recurrent neural networks". In: *International Journal of Automation and Computing* 13.3, pp. 226–234.