UKRAINIAN CATHOLIC UNIVERSITY

MASTER THESIS

Real-time inverse kinematics and inverse dynamics from motion capture

Author:
Kateryna ZABAVA

Supervisor:
Dr. Valeriya GRITSENKO

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, Kateryna ZABAVA, declare that this thesis titled, "Real-time inverse kinematics and inverse dynamics from motion capture" 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:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Master of Science

Real-time inverse kinematics and inverse dynamics from motion capture

by Kateryna ZABAVA

Abstract

This work applies machine learning to solving inverse dynamics and inverse kinematics tasks from the motion capture data. This approach may simplify the calculation process and help do scientific simulations as part of a physics engine that describes the neural control of human motion and decodes movement intent in individuals with neural damage. The existing algorithm has to be modified for every experiment and takes a significant amount of time to execute. It is also sensitive to noise and missing data, and it is not a real-time calculation. We propose a solution of inverse kinematics tasks with neural networks. Here we report accuracy results both on clean data and noisy data. We also apply a similar approach for the inverse dynamics task. The approach shows high accuracy on clean data, but this accuracy decreases if applied to the noisy data.

Keywords: inverse dynamics, inverse kinematics, motion estimation, motion capture, machine learning, real-time calculations, joint moments, dynamics, neural networks

Acknowledgements

We would like to thank Valeriya Gritsenko for supervising the project, providing the data and explanations. We give the gratitude to Artem Chernodub for many useful advises and technical help during the project.

Contents

List of Figures

List of Tables

List of Abbreviations

IK	**I**nverse **K**inematics
ID	**I**nverse **D**ynamics
DoF	**D**egree **o**f **F**reedom
mocap	**mo**tion **cap**ture
ReLU	**Re**ctified **L**inear **U**nit
AdaGrad	**A**daptive **G**radient **D**escent
Adam	**A**daptive **M**oment **E**stimation
SGD	**S**tochastic **G**radient **D**escent
ASGD	**A**synchronous **S**tochastic **G**radient **D**escent
MSE	**M**ean **S**quared **E**rror

Chapter 1

Introduction

The introduction of Computer Science into the world of sensorimotor systems uncovers many opportunities for human motion analysis. Not only can it have many applications in different fields, but it can also give insights into how the brain and muscles interact 'in-depth.' It is possible because mathematical modeling of the most complicated questions can create research insights.

The task of solving inverse dynamics and inverse kinematics is not new, but it still has many things to improve. There are different approaches to the problem's solution: geometric, algebraic, and iterative models. Furthermore, the task of efficiently solving the model inverse kinematics requires different strategies.

When applied to human modeling, more challenges arise as human kinematics is redundant and has many degrees of freedom (DoF) Rapetti et al., 2019, and multiple muscles move each joint Stanev and Moustakas, 2019.

The numerical solution is needed because the calculations or algorithms for inverse kinematics are used in different physics engines and robotics control tasks Rapetti et al., 2019. While used in simulations and robotics tasks, the solution of inverse kinematics requires real-time processing, unlike traditional non-linear optimizations. Some solutions like OpenSim Seth et al., 2010 generate discontinuity because they treat the movement as an array of separated movement snapshots and each of them has no information about the previous or the following snapshot. I should avoid these problems in our solution.

1.1 Domain overview

First, I would like to introduce the terminology and explain what domain I am doing this work in. Motion capture is the process of recording the movement of the physics body. The mocap approach involves putting markers on the person's body parts in the Neural Engineering and Rehabilitation Laboratory. Usually, there are three markers on each body segment, although this might vary. Then, cameras record the marker signals and send the position data for the computations and analysis. There is much existing technology available for scientists. For example, Microsoft Kinect incorporates several advanced sensing hardware: a depth sensor, a color camera, and a four-microphone array that provide full-body 3D motion capture Zhang, 2012.

There next thing to explain is inverse kinematics. It is a process of calculating the angles when I know the position of the end of the arm or robotic manipulator. The most common and accurate technique representing a body part - a kinematic chain: an assembly of joined segments that can perform the movement. When we know the parameters of joints, we can predict the location of the end of a kinematic chain. This process is called forward kinematics. What I am trying to solve in this project is the inverse problem. Speaking of the inverse kinematics task formulation, it is: the mapping between the Cartesian space and the joint space Tolani, Goswami, and

Badler, 2000. In the field of robotics, a common inverse kinematics problem consists of finding the mapping between the end-effector of a manipulator (task space) and the corresponding joint angles (configuration space) Rapetti et al., 2020.

The illustration of the process and the kinematic chain is in Fig 1.1. It is the representation of the inverse kinematics task and its relation to the forward kinematics. The black segmented object is the kinematics chain - a set of segments with joints in between. Each joint has to be bent at some angle so the end effector can reach the target. The end effector is usually a part of the kinematic chain that should reach the target. It has properties like location in space (usually 3-D) and orientation (angle) - these two parameters determine the pose of the end effector. When we know only the desired location of the end effector, we can calculate how to change every angle in the kinematic chain to reach the target. This process is called inverse kinematics. If we know the angles, we can calculate the resulting position of the end effector - solve the forward kinematics task.

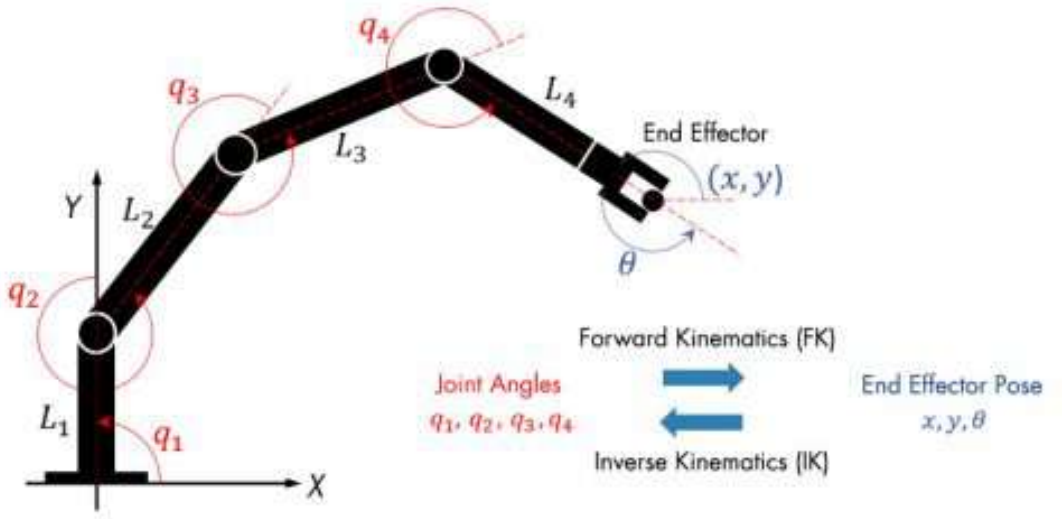

FIGURE 1.1: Inverse kinematics and forward kinematics

Source: MathWork article: Inverse Kinematics.

The inverse dynamics is calculating joint torques from the joint kinematics - acceleration, angles, velocity.

Both inverse kinematics and inverse dynamics allow us to model a 3D representation of the rigid body in the software. In the task, I have coordinates of the markers and some kinematics parameters (angular acceleration and velocity), and I need to solve the inverse kinematics and inverse dynamics.

An example of the usage of motion capture in medicine is in motion assessment clinics. Surgeons, neurologists, and physical therapists perform motion analysis tests to quantify movement deficits in different pathologies. Mocap is used for inverse kinematics to measure whether joint angles fall within the normal range during walking or other activities. [1] Therapists use mocap, usually from videos, to measure the range of motion. Also, it allows to calculate forces and determine possible weakness in the movement (like after the surgery or trauma).

[1] Here is an example from the OpenSim community https://simtk-confluence.stanford.edu/display/OpenSim/Tutorial+2+-+Simulation+and+Analysis+of+a+Tendon+Transfer+Surgery

1.2 Problem Setting and Approach to Solution

1.2.1 The problem outline

I am working with the Neural Engineering and Rehabilitation Laboratory, where scientists use mocap to study human movement. They use Matlab to run simulations. There is a solution to this task that can be done without machine learning, but it is slow, complicated, and requires manual work. In addition, there is a problem in the mocap experiments, where the markers can disappear from recording (like being obstructed by something). Extrapolation is used for the missing data, and the current calculations are not very good in working with missing data. The current IK solution is not accessible for people without a degree or much training. So it is not accessible for a vast community of users, and this should be changed. I also have some computations restrictions. As this solution will run in Matlab or even be executed on the end device, it should be fast. The goal of the simulation is to solve the task in less than 2ms. However, the hardware part and performance assessment of this is out of the scope of this task. I need to develop a solution that works as fast as possible, so it will be easier to optimize it later, but I will not test the speed on the end device.

1.2.2 Research questions and the specification of the problem

I need a solution that will solve all or at least most of the problems that occur in the NERL's discoveries. The main focus is to find out if the artificial neural networks can cope with the task and find the best one. We have some restrictions on computation speed. That is the forward path (the prediction) that should be fast. I do not include the training time in the measurements because I have no restriction on that. The training can be done offline, while the predictions should be real-time or close to it. I will have data from the NERL. It will be generated because it is easier to set up the Matlab experiments and gather data from those experiments. Moreover, the solution must be generic and not specific for an experiment setup; it should work well for all shoulder positions when measuring the IK and ID of the wrist movements.

So for this project, I have the following tasks:

1. Develop an ANN solution for the IK and measure its performance. One of the measurements will be MSE on train and test set, and another - I will visually plot the predicted angles and see where the model makes mistakes.

2. Test the solution on the noisy data. It is essential to do because real-world data is often noisy. I should know how the model performs on the noisy data. I also should test the solution on the unseen data as part of the methodology. In this work, I focus a lot on the inverse kinematics solution.

3. Test if the same architecture of the ANN can be successfully applied to the ID solution.

4. Test how predicted angles influences inverse dynamics error.

1.3 Structure of this work

The second chapter is an overview of the related work from the domain. The third chapter explains the data gathering process and describes the data. Chapter 4 includes the description of the pipeline, the methodology, and the results of the experiments. Chapter 5 gives an analysis of the results and conclusions. Chapter 6 is dedicated to the questions that remain open and future work.

Chapter 2

Related works

The following is a review of the most relevant articles that approach the problem from different angles and provide different solutions.

I want to start the review with a state-of-the-art musculoskeletal modeling framework - OpenSim (Pizzolato et al., 2017). This is the c++ library with API for many tasks, including inverse kinematics and inverse dynamics (Seth et al., 2010).

The Inverse Kinematics Tool from OpenSim separates the movement into timesteps and analyses the frames. For each frame, the Tool positions the model in a pose that "best matches" the experimental marker and coordinates data for that time step using a sum of weighted squared errors of markers and coordinates (*Getting Started with Inverse Kinematics*).

OpenSim uses a model that solves inverse kinematics and dynamics without using approximations or generic models (Pizzolato et al., 2017). To do so, the OpenSim team connected to the Vicon motion capture system and computed it in real-time. They also compared real-time and offline calculations and showed that the result is similar in joint angles and moments.

The authors of (Pizzolato et al., 2017) used different thread pools for computations and took advantage of multiprocessing in general.

OpenSim uses filtering that improves the results but adds to the delay. They used constraints in movement with many bodies to decrease degrees of freedom, leading to a closed-loop mechanism. Constraints increase the model's complexity, thus increasing the delay in solving inverse kinematics task and making it impossible to use in real-time. The hardware used allows 12 threads, but multi-threading did not show an increase in throughput.

Delays appeared in different stages of calculations: marker reconstruction and labeling, filtering, calculation of inverse kinematics, and data transmission. The delay that can be improved by tweaking the algorithm - is the delay in inverse kinematics and dynamics processing. The delay is affected by the chosen accuracy. Also, a crucial thing to consider in optimization is the time delay introduced by the filter. This one depends only on the selected cut-off frequency. This optimization approach only works for cyclical movements (walking) and cannot be used in other cases.

Considering all this, OpenSim created a generic solution that can be used to analyze any task. The application relies on streaming data in real-time.

There is another method of solving the task (Kim et al., 2009), where the team used a tri-axis accelerometer and gyroscope sensors (inertial sensors) and used 3d digital forearm for feedback tracking. The extended Kalman Filter was used to estimate three joint angles: pitch, roll, and yaw angle, and to denoise the signal and improve the estimation. Two different Kalman Filters were developed - one for the gyroscope signals and the other for the accelerometer signals. The created filter resulted in x, y, z-axis, pitch, and roll angles (angular velocities). The obtained data

was then used to control the digital version of the forearm. The control angles are expressed as a rotation matrix.

Another method used is the integration of differential kinematics using distance measurement on SO(3). SO(3) stands for the 3D rotation group - all rotations around the origin of three-dimensional Euclidean space. Paper (Rapetti et al., 2020) describes the development process of motion tracking algorithms that validate the inverse kinematics method performance on human and humanoid models in static and dynamic conditions. It is tested on a human-humanoid scenario that verifies the computed solution's usability for real-time robotics applications. The approach is evaluated according to the accuracy and computation load and compared to optimization algorithms simultaneously. The main focus was on time-critical applications of algorithms (Rapetti et al., 2020).

The (Rapetti et al., 2020) presents an approach for highly accurate humanoid models (including humans). Authors use a dynamic inverse kinematics optimization approach. The strategy is to use a rotation matrix parametrization for the angles of joints of the human model where none of the sub-bodies (called links) are constant. The convergence was proved by the method using Lyapunov theory. The tests were performed both on models and robots. The experiment's models were composed of 23 physical links connected by rotational joints with multiple DoFs. Additional components were performed on the iCub humanoid robot.

Two metrics described the accuracy of calculations: one for velocity targets (root mean squared error (RMSE)) and another for orientation targets (mean normalized trace error (MNTE).

The accurate results were achieved using optimizations, like the instantaneous optimization methods, to solve the inverse kinematics at each timestep. The idea of this approach is to assign a weight to each of the targets. This approach's performance decreased as the task becomes dynamical. Another optimization was dynamical optimization. Its orientation error is mostly comparable with instantaneous optimization. On the other hand, dynamical optimization keeps the computation time constant and has fast convergence. These approaches used here can be helpful in optimizing inverse kinematics models to achieve real-time computations.

The previous articles' review showed that modeling physics-based systems are hard or even impossible with traditional linear algebra methods. However, there is a potential workaround that involves machine learning, especially deep learning. The algorithm, presented in (Polydoros, Nalpantidis, and Kruger, 2015), uses the methods of self-organized learning, reservoir computing, and Bayesian inference. The result shows that the method can adapt to changes that happen in real-time significantly better than the other state-of-the-art algorithms. (Polydoros, Nalpantidis, and Kruger, 2015) To enable the adaptation of the model, it should use streams of data. The algorithm introduced is dubbed Principal-Components Echo State Network (PC-ESN). The approach used was similar to reinforcement learning that requires feedback from the action and the correction of the trajectory based on this feedback.

The model works as follows: the desired joints' location and velocities and accelerations are given to the algorithm, which estimates the required torques. Those torques are examined and corrected by the feedback controller. The feedback torques are a linear combination of the manipulator's actual position and velocity, weighted by error constants. And after these torques are applied, the resulting sensor measurements are used to train the algorithm further.

As a result, the proposed deep neural network uses only the sensory data to do

the action and train the algorithm simultaneously. The generalization ability of PC-ESN is similar to state-of-the-art. Its accuracy increases with the increase in update frequency. Moreover, the PC-ESN converges well in both noisy and noise-free environments.

The following article (Duka, 2014) describes the approach to the IK solution for a three-link robotic manipulator using neural networks. The manipulator has rotational joints. They have the desired position and need to calculate the joint angles. Those angles will position the robot's links in configuration, where it will reach the target. The authors of the article Duka, 2014 propose a feed-forward neural network. They use forward kinematics to compute the angles as a ground truth. So they record the position of the robot and then calculate the angles. After this, angles can be used during the training. The proposed feed-forward neural network takes data of shape 3 for input. Its hidden layer has 100 neurons. Finally, the output is a three-by-one vector of desired joint angles. The activation function on the hidden layer is hyperbolic tangent sigmoid. The dataset consisted of 1000 samples, and 15% of them were used for validation, and the same amount was used for testing. The training algorithm was the Levenberg-Marquardt algorithm (damped least-squares method). Mean squared error between target and output measured the performance of the network. The results showed that the neural network made it possible for the robot to track the desired trajectory almost perfectly.

Another article (Tompson et al., 2014) focuses on predicting human arm motion as well as elbow and wrist. There are three different parts for the task: RNN for wrist prediction, IK for full-arm motion, and it depends on the RNN predictions, modified Kalman filter (MKF) to adapt the model online.

For RNN, the approach is the Long Short-term Memory (LSTM) cell to control the memory either to remember or to forget. RNN takes in the N-step history and outputs the next step prediction

IK was solved using the Jacobian and matrix transpose to replace matrix inverse. Kalman filter was modified by adding a forgetting factor lambda to prevent the estimation from saturation.

The results outperformed state-of-the-art models by 14%; The solution generalizes well (checked on unseen humans), and the partial blocking of the wrist has no impact on prediction.

2.1 Conclusion from the literature review

The difference in our research is that I do not consider the target position of the effector. I have coordinates of markers, and I need to predict the angles of joints. Those joint angle predictions will later be used in inverse dynamics calculation.

As a starting guide, I will take this article (Duka, 2014). After trying simple architecture, I will increase the complexity of the model until it satisfies the desired accuracy. However, I will try to keep the model as simple as possible to explain where the results come from and satisfy other requirements.

Chapter 3

Data

3.1 Inverse kinematics data

The data I received is simulated data from Matlab using Simscape Multibody and Simulink. The data is gathered from the simulated movement of the forearm and palm. The palm is simplified and presented as one segment; the forearm model is also simplified (cylinder-like). We use this simplified process of gathering data to prove that the problem can be solved using machine learning.

On Fig. 3.1 you can see the model that was used for it. It is the simplified model of the arm, where each segment is represented by a cylinder (a rigid body). Then there are markers located on those rigid bodies. The joint we are studying here is the wrist joint. It is a joint that has three DoFs. Our task will be to predict the value of angles this joint can move in.

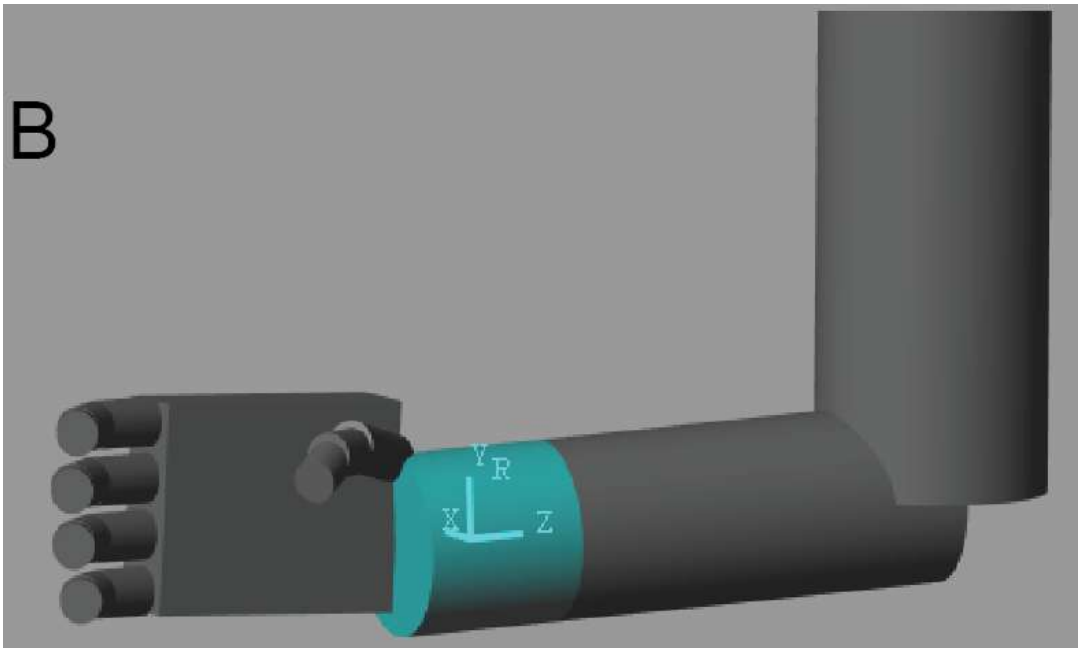

FIGURE 3.1: Simulink model used to generate the data
Source: Valeriya Gritsenko, NERL

Table 3.1 shows some abbreviations of the DoFs that I will use in the project.

In the Matlab experiment, I have three markers on the forearm, also called radius-ulnar, and another three markers on the palm (metacarpals). We run a simulation (using the NERL resources) and write down data about each degree of freedom angle. The movements we simulated (DoFs) are wrist supination-pronation, wrist flexion-extension, wrist abduction-adduction. If there is no movement in one of the

Abbreviation	Meaning
ra_wr_s_p	supination pronation degree of freedom
ra_wr_e_f	wrist flexion extension degree of freedom
ra_wr_ad_ab	wrist abduction adduction degree of freedom

TABLE 3.1: Abbreviations for degree of freedoms

degrees of freedom, the data is still written; only the values are constant. It is essential to have the data of the constant position of coordinates (and thus segments) as we also need our predictions to be continuous, and we need to be able to predict even the "no movement scenario.", meaning that there might be the length of time in an experiment, where the subject is not moving their hand, and we need to predict it as well as movement.

So, in the dataset, I have two segments joined by one joint. This joint can move in 3 degrees of freedom. I have three markers on each segment. The segments perform different movements, and Matlab records the position of the marker in 3-D space. It also measures the angle in each degree of freedom continuously (forward kinematics task). These measured angles are our ground truth.

In Matlab, we record the marker's coordinates in a global coordinate system. We call the global coordinate system the experiment set up. We have a single origin for all segments, and all marker coordinates are calculated according to this origin. When I receive the data, I have those marker coordinates in this coordinate system.

Before I receive the data for the training, it undergoes some preparation. We decided to convert markers' position into the local coordinates to have their position relative to the segment. The hypothesis is to check how this approach for data preparation will impact the results.

So, the coordinates are converted to the local coordinate system. This data will be called preprocessed. It was done for me, and I am only describing the process. I have the transformations of markers data into the local coordinate systems of segments. I have 1st, 2nd, and 3rd unit vectors of each basis in each segment in the dataset. I also have the origin vector for both segments. Now I can explain the dimensionality of data. Each marker has 3 unit vectors (1st, 2nd, and third); each segment has a basis vector that in the features is split into x, y, z value. As we have three markers on one segment, we have nine features for them; then, we have three additional features to describe the origin of this segment. It results in 12 features for one segment. We have two segments in the dataset. Thus the amount of features in the data is 24. It is the dimensionality of one type of data preparation, which requires the conversion to the local coordinate system.

I have another type of dataset that has raw marker coordinates. It contains only the x, y, and z positions of each marker. So again, we have three makers on each segment, and we have two segments. As a result, we have 18 features in this dataset.

There is a reason to prepare data by converting it to different coordinate systems. The marker position is highly dependant on the length of the segment. In the real world, people have different arm lengths. But conversion to the local system mitigates this problem. On the other hand, the conversion process is lengthy, and I should try both methods and see which one shows better results.

The unit of measurement for coordinates is a meter. In another file, I have data about the angle in each DoF. The unit of measurement for angles - radians. Every movement is a row in the dataset, and it also has a corresponding timestep measured in ms.

I had two stages of data for the experiment. The first one consisted of 4500 ms of experiments and contained different movements with a length of 500ms and 250 ms. I used this data for initial proof of concept and to ease the pipeline creation. This dataset has six movements that lasted 500 ms and six movements that lasted 250 ms. In this dataset, each movement belongs to a single DoF. In this experiment, I can see that each movement started from its own time, so there are no continuous simulation movements. It will not affect the model as I will not include the timestep in the training or testing data. However, such a representation allows us to easily count the number of completed movements in the received dataset. We can also see the range of movements for each movement in Fig.2 3.2. Here I plot the change in angle for each DoF of the wrist during the whole experiment. Here I can see that the simulated movement looks like a sigmoid, and there are four episodes of change in angle for every movement. I also notice that the abduction-adduction angle is smaller than the other two in this dataset. The drop to the baseline is where the movement was stopped and returned to the original base state. It is also possible to detect the length of the movement on this plot.

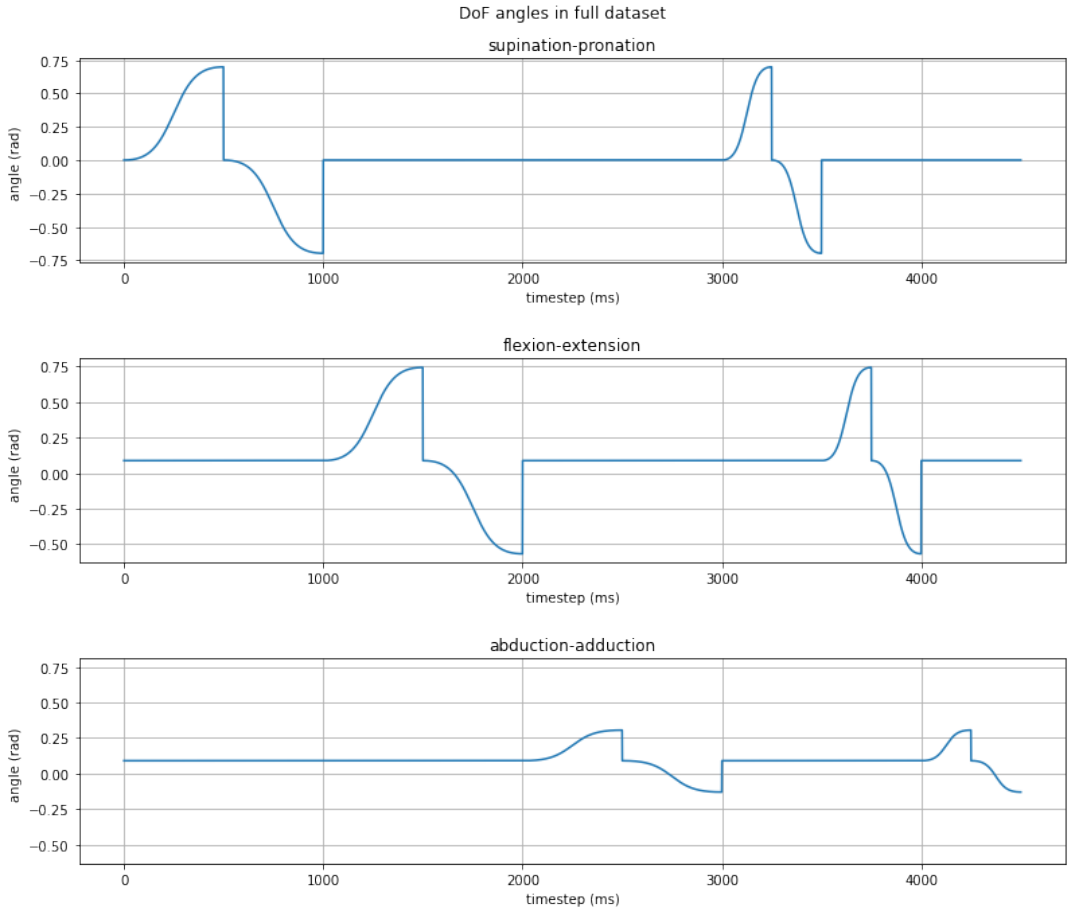

FIGURE 3.2: Angle change for each DoF in the initial dataset

As for the markers data, I can investigate both movements and the data itself. I have origins of local coordinate systems in the dataset, and I can plot those to investigate how they change during movements. In Fig.3 3.3, there is a plot of origin separately on the palm and the forearm. There also encodings of those origins: MRU or MMC respectively origin- on the forearm or the wrist(palm). Indexes 1, 2, 3 denote three coordinates x, y, z. From the plot, I can see that the origin on the

forearm does not move while the hand is moving. It does not mean that the forearm is stationary. Because if I take a look at the wrist origin - something moves it. Because the forearm and hand are connected, the movement in the forearm affects the change in origin.

I also have raw marker data. This is data without any preprocessing, and it represents the coordinates of markers in a global coordinate system.

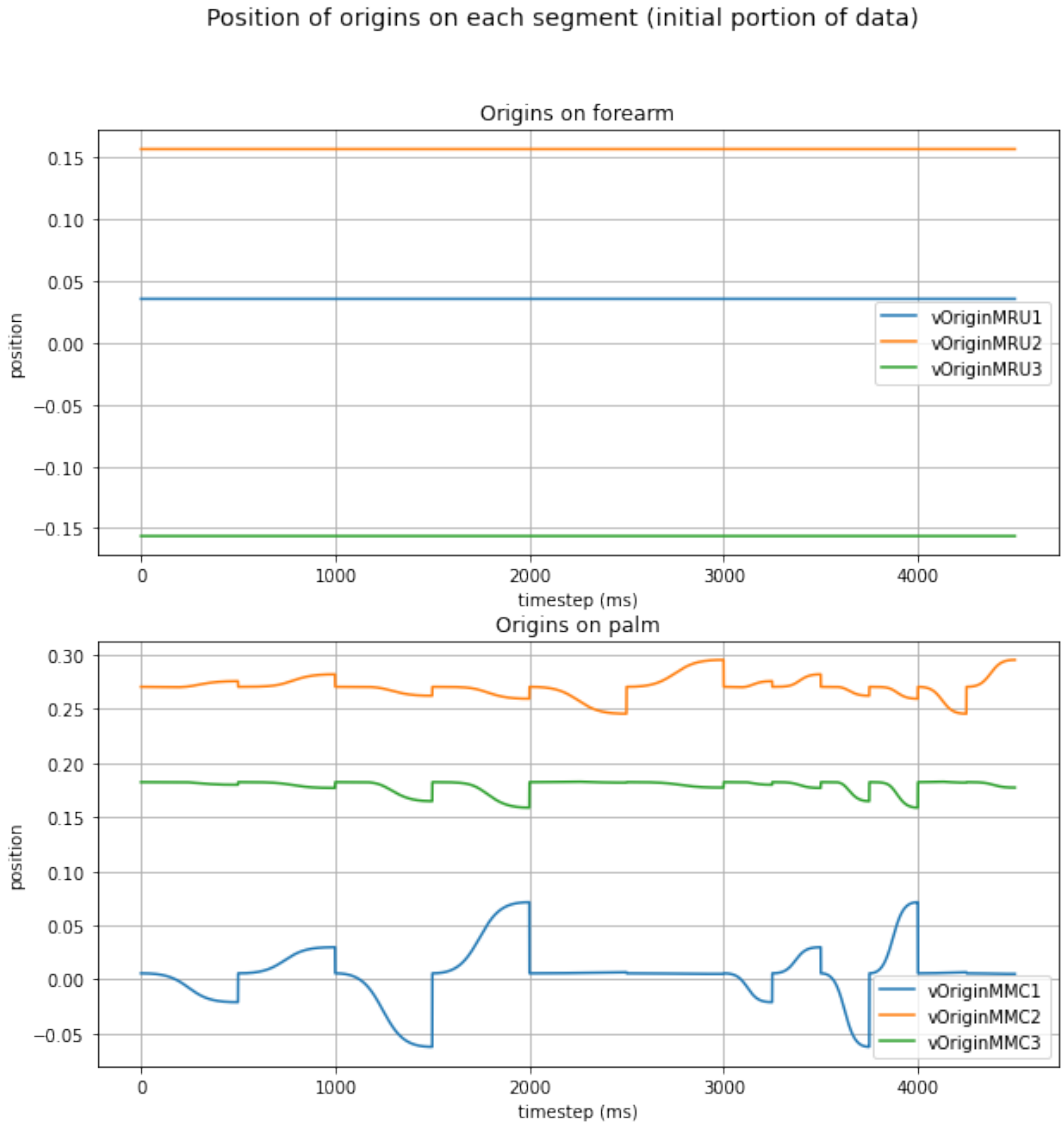

FIGURE 3.3: Position of origins on each segments in time in the initial dataset

In addition to the previous data, I received the same movements but added noise from a uniform distribution. I have 1 % of noise and 5% of noise. We will use this data for further testing of the model.

The data I received in the second iteration was more data with even more diverse movements, and I also got data about torques. The data for IK here consisted of three parts. The main difference is that the shoulder position is changed. Now I have data at 0 shoulder angle (arm by the side of the body); I also have 45 and -20 shoulder flexion angles. This change will change the orientation of local coordinate systems

of the forearm and hand. Also, the experiment in Matlab was done so that every next movement continues the previous. Each dataset consists of 42 movements.

Here we also can see how origins change in time (during movements). There are two origins, one on each segment, and I can plot those against time. It will show how the segments moved. The plot in Fig 3.4 shows concatenated data with different shoulder positions. The dips in plots are where the new dataset (in terms of different shoulder positions) begins. Both segments move, and there is no scenario that the one segment does not move.

I also ran some tests on the data to see if there any problems that can impact the performance. I checked the skewness of the coordinates data and angles data. I have some skewness in the data coordinates. To measure it, I use the unbiased skew function. In the results, I noticed that some unit vectors have high skewness. The most left skewness exists for the first marker on the hand. There is no skewness in the target (angles) data.

The range of coordinates for each dataset is between -1 and 1.

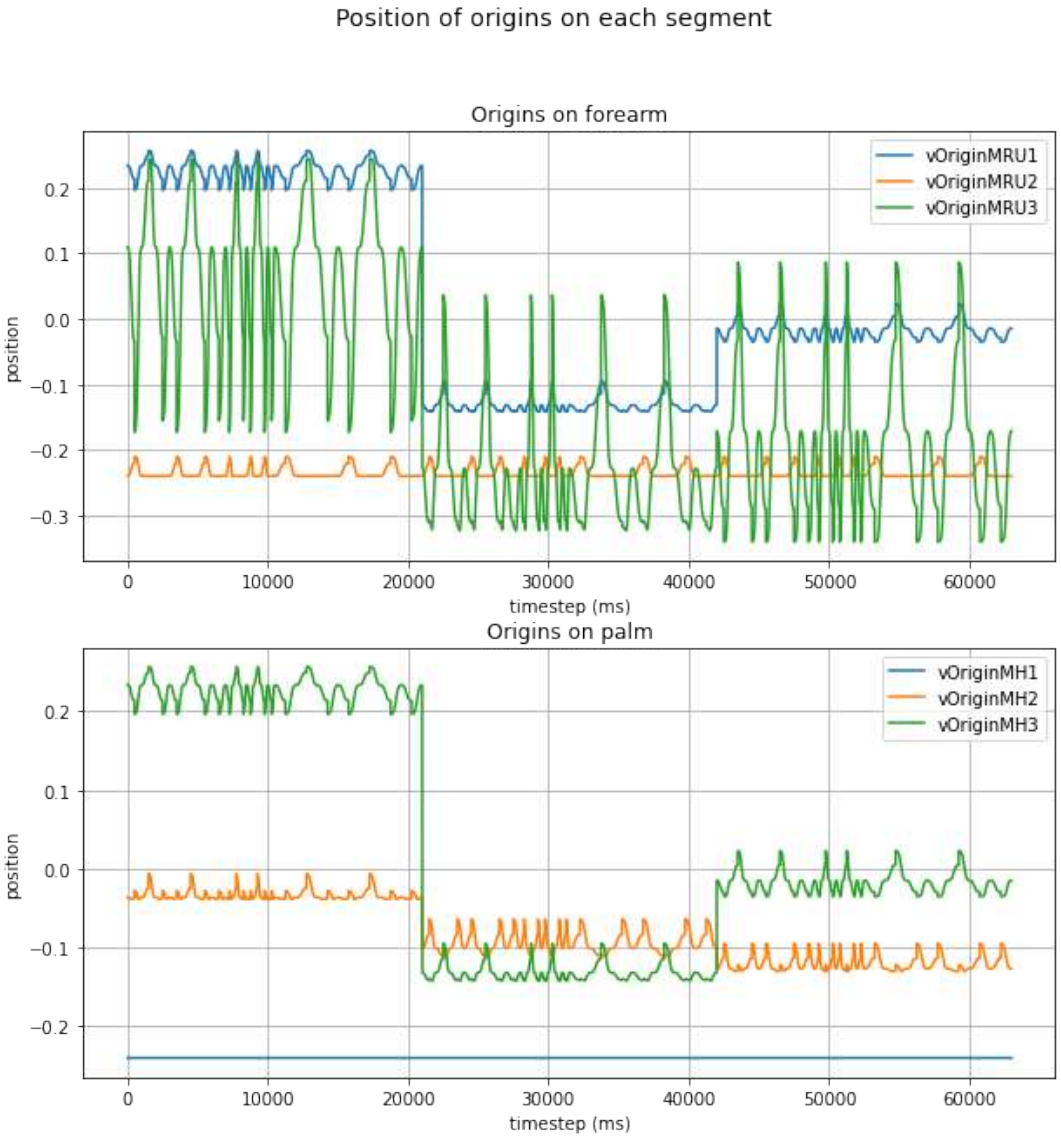

FIGURE 3.4: Position of origins on each segments in time in full dataset

3.2 Inverse dynamics data

This portion of data includes velocity, acceleration, and torque data for all movements with and without noise. Although I have three datasets generated for each shoulder position, the ID data is the same there. It is because I had the same movements in those experiments. For the inverse dynamics task, I have less data for the training than I had for inverse kinematics.

Velocity, acceleration, and torque data are present for every DoF I had in the IK task.

I looked into the ranges of values for each parameter. The results are in the Table 3.2

Parameter	DoF	min	max
Acceleration	ra_wr_s_p	-150.16800	213.41900
	ra_wr_e_f	-140.72300	200.08900
	ra_wr_ad_ab	-46.77230	66.45000
Torques	ra_wr_s_p	-1.82339	1.22100
	ra_wr_e_f	-1.20882	1.09586
	ra_wr_ad_ab	-1.84183	1.81413
Velocity	ra_wr_s_p	-13.21230	6.60615
	ra_wr_e_f	-12.38620	6.19311
	ra_wr_ad_ab	-4.11612	2.05806

TABLE 3.2: Range of values for inverse dynamics data

It is essential to know the range of the feature values because the bigger the feature's values, the more it will affect the result, which means that the big values will be "noticed" by the model more than the small ones.

The nature of this is in the MSE calculations. As I will use the MSE as the loss measure, our values should have the same scale. The bigger the feature value, the more it will contribute to the error. When the features have the same scale, there will not be a situation where one feature contributes more to the training just because it is bigger in scale.

Chapter 4

Experiments

4.1 Early work

In our early experiments, I focused on the IK task. Because only if I can solve the IK task, I could move to the ID task.

The goal of these experiments was to determine the vector of further experiments. The data used for the experiment is the data obtained in the first iteration. In this dataset, I have 4500 timesteps of movement. I have 500 ms movements and 250 ms movements, six of each duration. I use this small dataset to quickly iterate the experiments, see how the solution will work, and develop further hypotheses.

In all our tests, I measure MSE on the train and test set. I also measured MSE on it and compared it with train MSE.

To see how the model predicts, I also plotted the predictions against the reals angles. This approach allowed us to visually see how model estimation differs from actual angles and potentially see anomalies.

I started with a simple solution. I used a multilayer perceptron with three layers and two ReLU in between. I used stochastic gradient descent as an optimizer, and the learning rate was set to 0.01. I also tried using different learning rates in these experiments, but the 0.01 showed the best results. Finally, I trained the model for 15000 epochs. I shuffled data using a simple train test split with 80% training and 20% test data. I used data where the basis and unit vectors were present. Those unit vectors and origins calculations are coming from Matlab. The input was one row of coordinates: output - three angles in each DoF. I did not include time into training or testing features because this parameter is not consistent and should not affect the training. I will always have different timestep values in unknown experiments, and I do not want time to influence the prediction.

As for the number of neurons, I chose this approach: I had the first hidden size to have 15 neurons and the second to have 9 neurons. Thus, output was 3 neurons and input - 24 (amount of features).

In this set up I also tried using 100 neurons in the hidden size, and it showed smaller MSE and better predictions when I compared those with the actual data.

I compared those visually, and the results are on the plot. Fig 4.1 Although it looks good on clean data, the results worsen dramatically when I tested in noise. The MSE on 1% noise data is 0.240, and on 5% noise, it is 0.256.

In our second experiment, I wanted to split the data according to the moves. For this, I separated the movement data into 12 segments, where each segment contain one movement. I have six long movements and six short. I created the training dataset that consisted of 5 long and four short movements selected randomly. The last two short and two long go to test.

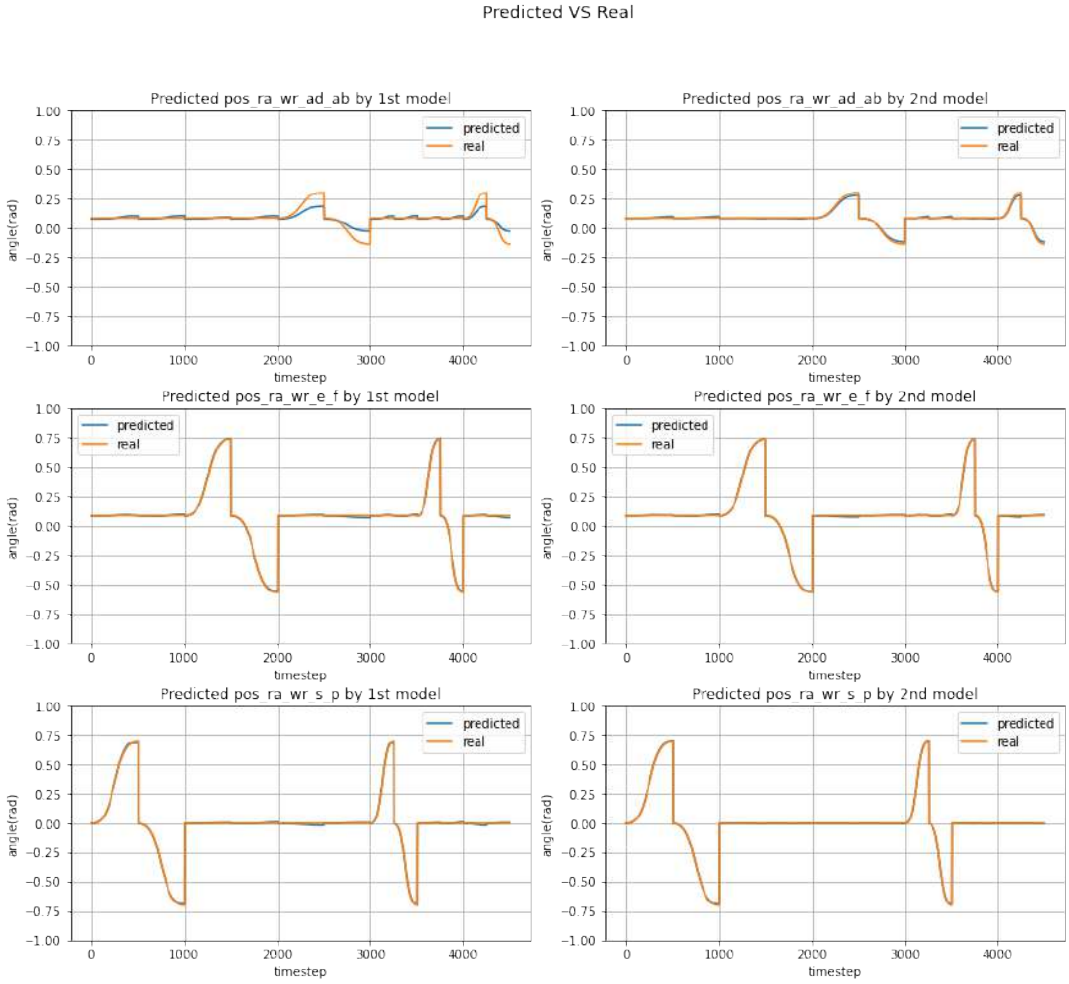

FIGURE 4.1: Comparison between two models

The model I use was the same as before. I also used the same parameters as in the previous experiment. At first, I had 15 neurons in the hidden layer, and on the second run, I had 100.

This experiment showed not the best result, although when I tested it on noise, I noticed an increase in MSE. I also see the overfit in both experiments.

When I used 15 neurons, I received a training MSE of 0.0127, and on the test, it was 0.1988. These results show that the model overfitted. The model predicts 1% noise data with MSE of 0.0558 and 5% results in 0.0572 of error.

When I used 100 neurons in the hidden layer, I received a training MSE of 0.0069 and a test MSE of 0.1189. The model is still overfitted. For this model, the result on 1% noise data is MSE of 0.2293 and 5% results in 0.2307 of error.

Both of our experiments did not show good results. Models were strongly affected by the noise. So I needed to use a different approach. I wanted to make the model history-dependent. For this, I do the following process: I feed into the model every five rows concatenated in one. This one will be included in our main pipeline, and I will use this in the following experiments.

4.2 General pipeline

Here I will describe the pipeline and approaches for the data preparation I use in the experiments.

4.2.1 Data Preparation

I have separate files for different shoulder positions. Also, the target and features are in different files. So I have three files of features (either markers coordinates or preprocessed basis data) and corresponding three targets. The first thing I do is reading the data in the data files.

The next step is introduced in one of the experiments. I take every five rows and merge them into one. I do this for every row, so it is like a sliding window in the data. For example, after this process, the first row has data from the first to the fifth row of the initial dataset. Then the second one - from the second to sixth, and I continue in this way. As for the target, I do not flatten it, but I cut the first 4, which means that the five merged rows in the result should correspond to the fifth angle in the target data frame. I do this process to each pair of target files - feature file separately.

There is also another approach for data preparation. For example, I have 1 experiment that uses convolutional neural networks, and I need different data representations. The approach I go for is cutting the dataset into segments and feeding those into the network. In this way, I receive the training data of shape Batch $\times$ Length of segment $\times$ Number of features.

After this, I do a Train/Test/Validation set splitting. First, I want to remove the last 10% movement from each file to create a test set. I do it this way because I do not want to shuffle it, but I want to keep the movement structure intact. This approach will allow us to see how the model predicts the angle and where it fails. After this, I remove the test set from the dataset and split the data into validation and train sets, where the validation takes ten percent of the dataset without the test. The rest goes into the train set. I do the split for each file. I do this in this way, so I ensure that all variants of data are present in all splits. After I split the data, I merge corresponding sets. In the result, in the IK data, I have 5670 entries for validation, 6297 entries for the test, 51021 for the train.

I had a change in the data split for the architecture search. I split data for 10% for the test and 10% for validation and left 80% for the training. I shuffled all sets.

As I have less data for the inverse dynamics, I do not use the validation set. I do this to increase the number of samples in the train set.

For the inverse dynamics task, I also scale the input data after I split the data. Again, I use MinMaxScaler that scales every feature to the range. The formula for scaling is:

$$X_{std} = \frac{X - X.min}{X.max - X.min}$$

$$X_{scaled} = X_{std} * (Range.max - Range.min) + Range.min$$

I fit it only on the train set, but I need to transform the test set and the unknown data.

For some experiments, I use batches in training if I notice that the training requires many epochs. This approach allows us to apply optimization more frequently, thus allowing us to reach the minimum in fewer epochs.

4.2.2 Early stopper

I implemented the early stopping functionality to overcome the overfit in some cases. This approach aims to stop the training if the validation loss worsens (increases) or does not change for a long time (many epochs).

I did a custom class with the following logic:

The class has a min_delta parameter that records the difference between the best loss on the validation set and the current. In addition, the class has a second parameter - patience, which allows the validation loss to not improve for some epochs.

The model runs the train set; after this, I apply it to the validation set and measure the loss. If the received loss is better than the best loss for more than min_delta, I record it as a new best loss and reset the counter. If there is no improvement, I start the counter. When the counter reaches the limit (patience), I stop the training.

4.2.3 Measure computations speed

I need to know how fast the model can predict one row of the new data. I should mention that measures are done in Google Colab, and numbers cannot tell us how long the computation will take on the end device or Matlab. I do this measure to compare several models that will be making predictions in similar environments. I also want to have a general idea about the time required for one forward pass.

To test the speed of the prediction, I will generate 100000 random variables of shape that the model requires. Then, I will make predictions for all those numbers. Once I get the time for the entire run, I will divide the resulting time by the number of records. The achieved result is an approximate time for one entry.

I measure the computation speed on the CPU.

4.2.4 Training Pipeline and Hyperparameter tuning

I created the training pipeline with the idea that I will need to tune the parameters. I started with saving the train test splint into the files. I do this to have the same data in every tuning run. Therefore our first method in the pipeline is to read the data from files into according variables.

I implemented custom hyperparameter tuning functionality in such a manner that I can control the experiments easily. Also, I did not need the out-of-the-box functionality; it was too much for our experiments. I used the Grid Search approach. It takes the grid of parameters, combines them in all possible ways, and returns the array of the experiment set up parameter. For tuning, I used a number of neurons in each layer; I also implemented it to try a different number of layers and learning rate for the optimizer.

I implemented a function that performed the model training and returned the results according to the training parameters. Also, it was essential to check how the model reacts to unseen noisy data. Then after I received the trained model, I tested it on noise and recorded the MSE on noisy data, both on 1% noise and 5% noise.

In all experiments the output is 3 angles - each is the separate wrist DoF angles. In the ID task the output has 3 torques - each represent the torque in wrists DoF.

4.3 Experiments on inverse kinematics

4.3.1 Experiment 1. History dependence

The initial experiments showed that the model had trouble predicting angles in the movements correctly. So I used the same model I had in the initial experiments. However, first, I performed the architecture search. I did this in a series of experiments to select the model that performs well enough. Our initial goal was to select the model that has a reasonable prediction rate on the clean data. So in the first experiments, I focused primarily on minimizing the test error on the clean data.

For our first experiment, I selected the new architecture. The model has four layers: input, first hidden and second hidden, and output. There is a ReLU activation function after the input and the same activation function after the hidden layer. There is no ReLU after the output of the range of activation functions - for example, I might need to predict the negative angles.

I use ReLU or rectified linear unit function as our activation function for several reasons. It has a range of values (0, x) and returns 0 if x is not positive and x otherwise. It overcomes the vanishing gradient problem (something sigmoid and hyperbolic tangent activation functions do not do). Both sigmoid and hyperbolic tangent tend to put large values to 1 or small to -1 (in case of hyperbolic tangent) or 0.5 (for sigmoid) Goodfellow, Bengio, and Courville, 2016

ReLU, on the other hand, has no such problem because it avoids easy saturation and is more straightforward for the calculation.

Taking those points into consideration, I decided to go with ReLU in our experiments. I prepared data according to our pipeline. The reason I merge every five rows into one is the idea of making the model history dependant, which means that the previous position will influence the calculations of the current angle. This method may solve the problem of discontinuity, which appears in some solutions, and it also might improve the predictions. Here I used the following number of neurons: the input size is 120 (this value comes from the data flattening - I have 24 features in one row, and I turn five rows into one, thus resulting in 120 features on the input); first hidden layer has 60 neurons and second is 24 neurons, the output is 3.

In this experiment, I used stochastic gradient descent for the optimization with a learning rate of 0.01.

I trained the model for 20000 epochs.

The training loss was 0.0035, and the test loss was 0.0036 as well. The validation loss was 0.0034

I also used the Cuda acceleration for this training, and the training time was significantly shorter. I also noticed that the validation loss was smaller than the training loss. Therefore, I concluded that the model could train for more epochs.

I also noticed that the results from this experiment are better than those I had in our early work. This means that I am going to keep this method of data preparation in the following experiments.

So our next iteration was to increase the number of epochs up to 40000. I left everything the same but changed only the number of epochs. The results were: train loss was 0.0033, validation loss was 0.0032, and test loss was 0.0034. Because I am talking about the radians as a unit of measurement, the error is deficient by itself. Although on the 40000 epoch, the model did not improve accuracy significantly, the training time increased twice. It was around 8 minutes for the first experiment and almost 17 minutes for the second. I also plotted the Loss VS Epochs plots for both experiments to compare the performance visually as well. 4.2 To make general

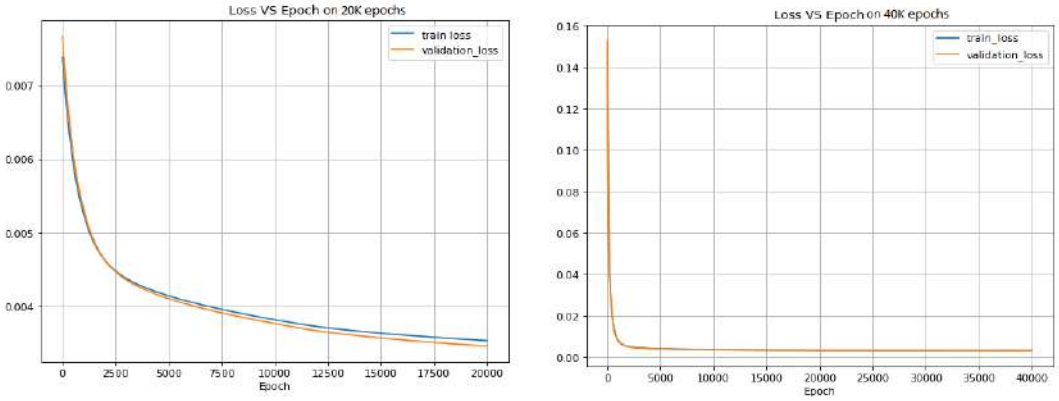

FIGURE 4.2: Comparison between two experiments on 20000 epochs and 40000 epochs

conclusions about these experiments, I need to see how the angles are predicted. For this, I fitted the model on all data. I had to check how the model treats the different angles. I had all the data shuffled in this experiment setup that disabled the possibility of consistent plots of the predicted vs. actual angles. I plotted the results of the model that trained for more epochs.

This plot represents three angles in 3 DoFs. I plot each DoF angle on its plot, so it is easier to see the performance of each of the targets 4.3. The blue line represents a prediction for each row, and the orange dotted line is the "trajectory" that the model is supposed to predict.

As I can see from the plot, there are significant errors in the abduction adduction movement. However, this movement is small, and it might be more challenging for this model to capture the values well enough. I also see that the model here tends to move even when there is no movement required. There is a particular bias towards movement in the model that I will need to fix in the following experiments. So far, those peaks in the DoFs on the segments will result in some twitching during the simulation.

4.3.2 Experiment 2. A more complex model

Model with more hidden layers

I hypothesized that a more complex model could find the solution faster than 20K epochs, and I wanted to test it. I added another hidden layer to the network, and I used the Leaky ReLU function after introducing more non-linearity. The Leaky ReLU is slightly different from the ReLU. If the value is below 0, Leaky ReLU multiplies it with a very little value (0.02 in our experiment). The function can output the negative values and help with the so-called "dying ReLU," which leads to the situation where some neurons always output 0.

For this experiment, I had the following number of neurons: input as before has 120, first hidden - 90, second hidden 60, third hidden is 24 and the outpus is 3 as in all experiments. The optimizer and the learning rate are the same as in the previous setup. The number of epochs was set to 20000 in the first iteration and increased to 40000 in the second iteration.

I did not receive an increase in accuracy in the first iteration: 0.0037 on the train set and 0.0038 on the test set. However, if I compare it to the previous results in Experiment 1, I can see a tiny increase in loss.

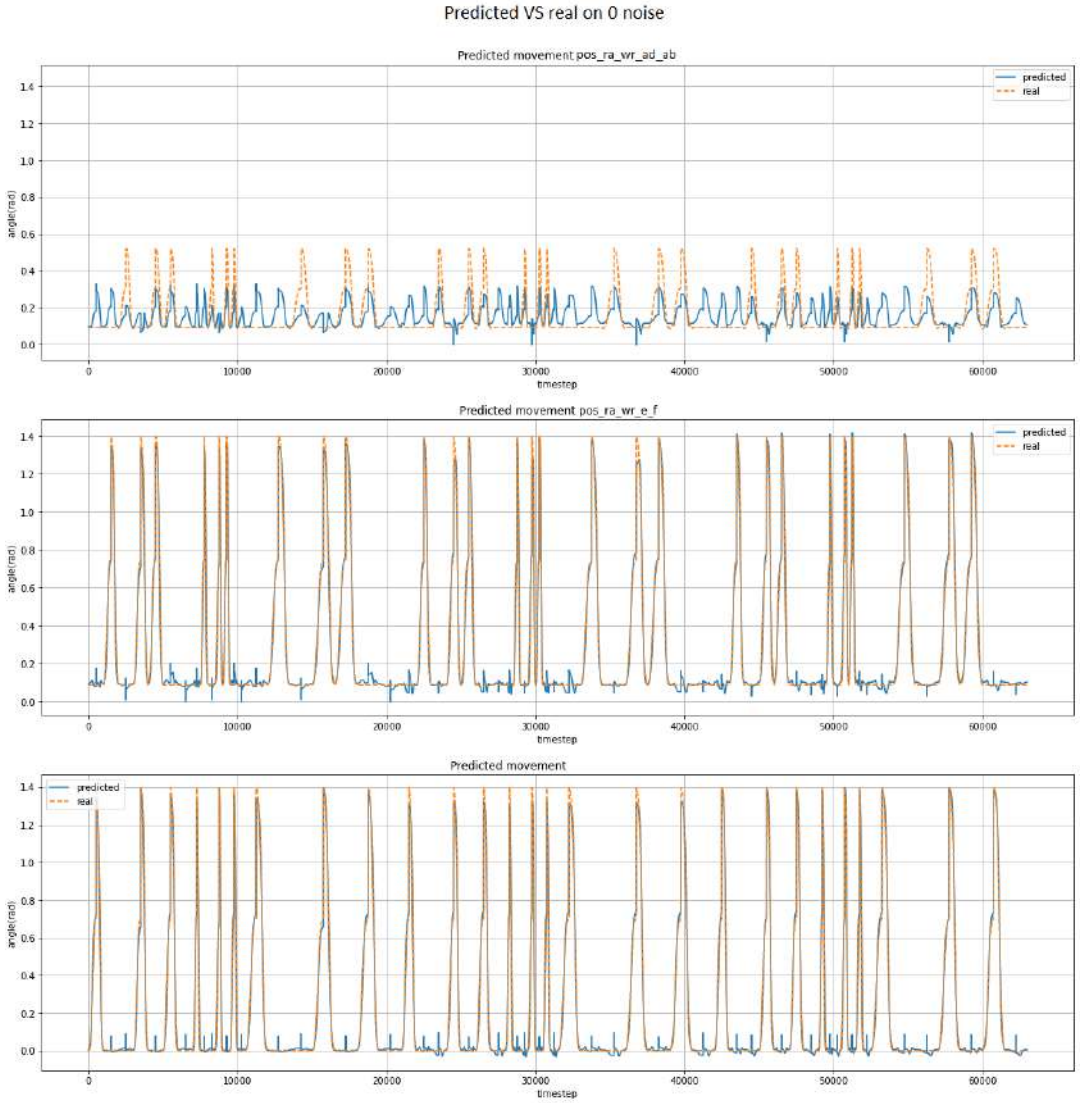

FIGURE 4.3: Visual representation of model results from Experiment1

When I increased the number of iterations up to 40000, I noticed similar behavior to Experiment 1. There was no significant decrease in the MSE. It was 0.0035 for test and train, although the training time increased.

Model with 4 hidden layers

As I did not see the increase in the performance in the previous experiment, I went one step further and added another hidden layer and another Leaky ReLU after it.

The number of neurons is these: 100 for the first hidden, 90 for the second, 60 for the third, and the 24 for the last and output is 3.

I decided not to iterate on 40000 epochs and left the training on 20000 only.

Everything else stays the same.

I received the following results:

1. There is no noticeable difference between losses in this experiment and the one with two hidden layers. It stays at 0.0038 on the test loss.

2. Learning time is significantly longer.

I can conclude that there is very little reason to increase the complexity of the model in the current setup.

4.3.3 Experiment 3. Experimentally determine the best optimizer

As I did not see any improvement in the previous experiments, I turned to the other thing that can influence training results.

Optimizers are methods whose goal is to change the weights of the neural network to minimize the losses.

As the previous experiment showed that increased complexity does not lead to a better model, I returned to the model I had in Experiment 1.

I also use an early stopping mechanism in this experiment.

In this experiment, I check the model performance while using different optimizers. There are many of them present currently. I will focus on these five:

1. SGD
2. ASGD
3. Adam
4. Adamax
5. Adagrad

SGD

It is a Gradient Descent that computes the gradient frequently - parameters update for each training example. The major disadvantage of this optimizer is that it may be stuck at local minima, and it is pretty slow (compared to others). The use of momentum speeds up the convergence. As I performed the experiment in the Experiment 1 section, I will not repeat the results here. I will compare all the following results to this one so that the SGD model will act as a baseline.

ASGD

ASGD stands for the Averaged Stochastic Gradient Descent. It is the case where the averaging was used for the acceleration of computations. When I ran the experiment with ASGD, I received 0.00366 for the test loss and almost no increase in execution time.

Adam

Adam stands for Adaptive Moment Estimation. This optimizer has two momentums: the exponentially decaying average of past squared gradients and an exponentially decaying average of past gradients. This approach allows Adam not to skip the minima during the search - it adapts the velocity according to the input. This allows Adam to be faster than SGD and avoid being stuck in local minima.

In this experiment, I needed to decrease the learning rate to 0.001 because a lower learning rate showed many fluctuations in the loss and did not allow the model to learn. The training stopped when the training loss was 0.00026, and the validation loss was almost the same (0.000266). The test error showed an MSE of 0.000268.

So this optimizer fits the task way better than the SGD and allows the model to learn better. I believe it is because the SGD tends to be stuck in local minima, while Adam is better at avoiding local minima.

I also plotted the plot to see how this model predicts all angles. I noticed that the abduction-adduction error decreased significantly, and the model predicted all angles very close to the real data. There was the same problem of bias towards movement, and most of it was in abduction-adduction predictions, while other DoFs showed almost perfect trajectory.

Adamax

While I achieved very good results using Adam optimizers, I still wanted to check how other optimizers change the model's performance. The next optimizer to test was Adamax. It is a variant of Adam where calculations are made using the infinity norm. In this experiment, I also needed a learning rate of 0.001 for the same reasons as it was done for Adam. I also disabled the early stopper for this experiment to see how far the optimizer can move the model. Although, when I run the same number of epochs as the Adam Experiment, Adam performed better.

When I plotted the angle predictions, I noticed that the number of errors in no-move areas decreased.

Adagrad

AdaGrad adapts the learning rate for each weight. The value of the update depends on the frequency of the feature. The more infrequent it is, the larger updates it receives.

Our experiment with AdaGrad showed the result of 0.0030 on training and validation and almost the same value for the test set. The difference was in the fifth digit. Moreover, this model showed significantly worse results in predicting angles; those were more similar to the SGD model predictions than Adam.

After looking at all the results of the experiments, I decided to continue our experiments with the Adam optimizer.

Before continuing to the next experiments I would like to sumarize the previous experiments in the one table. Table. 4.1

4.3.4 Experiment 4. Increased number of neurons in hidden layer

In all previous experiments, I have done a decreasing number of neurons in the hidden layers. Here I want to explore the influence of the number of neurons on the training results.

Here I set up the same model architecture as it was in Experiment 1. I use the best optimizer from Experiment 3 and 40000 epochs. There are first hidden layer with amount of neurons of 240 and the second with 120 neurons.

I received the MSE with a value in the fifth digits. Train loss was: 3.12e-05, Validation Loss was 3.21e-05, and the test loss was 4.15e-05. When I plotted the predictions of angles, I saw an almost perfect prediction of the movements and a slight twitchiness in the flat sections. Fig.4.5

Because I see quite a good result on the model, I decided to check its performance on the noisy data. To do so, I read the data with noise (1% and 5% noise accordingly) and prepare it (flatten by every five rows), and feed it into the neural network. I also measure the MSE on the noisy data and plot predicted VS real plot.

			train MSE	val MSE	test MSE
Experiment 1	hidden	120	0.0035	0.0034	0.0036
	hidden2	60			
	hidden3	24			
	hidden4	-			
	hidden5	-			
	learning rate	0.01			
	dataset type	basis			
	optimizer	SGD			
	epochs	20000			
Experiment 2	hidden	120	0.0033	0.0032	0.0034
	hidden2	60			
	hidden3	24			
	hidden4	-			
	hidden5	-			
	learning rate	0.01			
	dataset type	basis			
	optimizer	SGD			
	epochs	40000			
Experiment 2 with leaky ReLU	hidden	90	0.0037	0.0036	0.0038
	hidden2	60			
	hidden3	24			
	hidden4				
	hidden5	-			
	learning rate	0.01			
	dataset type	basis			
	optimizer	SGD			
	epochs	40000			
Experiment 3	hidden	100	0.0037	0.0036	0.0038
	hidden2	90			
	hidden3	60			
	hidden4	24			
	hidden5	-			
	learning rate	0.01			
	dataset type	basis			
	optimizer	SGD			
	epochs	20000			

TABLE 4.1: Results of first experiments

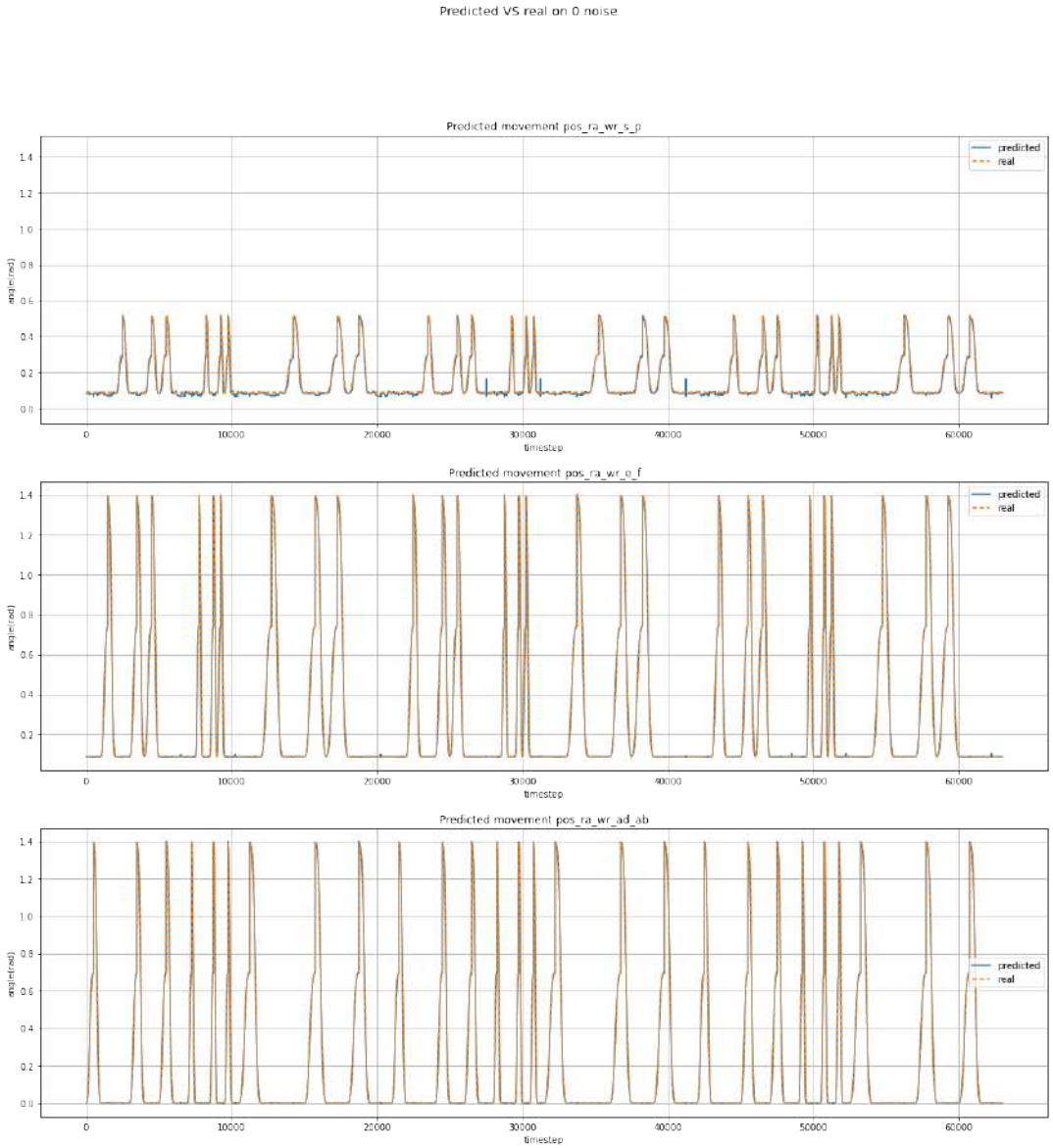

FIGURE 4.4: Visual representation of model results from Experiment 4

The MSE on the 60k of data with 1% of the noise is 0.0008, almost ten times bigger than on the clean training data, but still, it is quite low. The plotting of the predictions will tell us more. The MSE on the 60k of data with 5% noise is 0.0066.

When I take a look at the plot Fig.4.5, I can see that abduction adduction movement contributes to the error the most, while two other movements are traced almost correctly.

Computation time measurement

For this experiment, I also measured the time required to predict the output. We did this according to our methodology, described in section 4.2.3. The result was that the model requires 0.012 ms on CPU to compute one prediction. It is less than a 2 ms delay that I set for the task. It is also important to notice that this metric was measured in the lab environment and not a Matlab or end device. Therefore, this

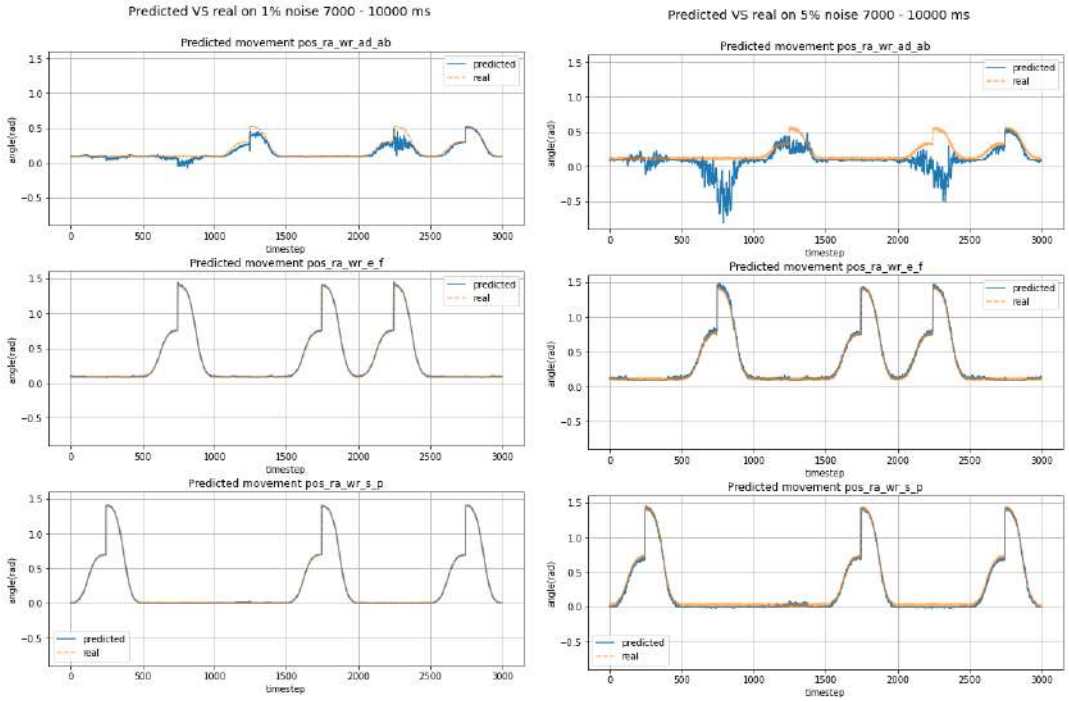

FIGURE 4.5: Prediction of the model trained on coordinates on noise of 1% and 5% in the sequence from 7000 ms to 10000ms

parameter can only be used to compare the speed of models, but not to say that this speed will be in the end device.

4.3.5 Use Markers Data for predictions

Because the task seems to be very simple for the model, I make it more complicated; I will take the marker data. The marker data is the data I would be recording directly, so it is the closest data to the real human motion capture recording. So wee took the markers data and tried the architecture in the previous experiment.

This approach might have some potential problems. One of those is subjects of different sizes. The basis data would not change between people of different sizes, but the marker data will change. Therefore, the model trained on marker data might not generalize to the real motion capture recordings with people whose arm size does not match the model size. But it is something to determine in the later experiments.

In this experiment, I prepared data in the same manner as I did in the previous iterations. There were two differences: I used raw marker coordinates as input, and I cut the test set from the dataset and did not shuffle it. I avoid using shuffle in this experiment to plot the model predictions and see where the errors occur. The target data stayed the same as before - 3 DoFs angles.

I utilized the model I determined was best for the task in Experiment 4.

I also cut the number of epochs to 15000 because it has the very little benefit of having more. Thus, the loss does not get significantly smaller, but the time required for the training increases.

I run hyperparameter tuning for this experiment. I tuned the number of neurons in the hidden layer and the learning rate for the optimizer. Because I focus a lot on the performance of the noise, I also included MSE measurement on noise (both 1% and 5%) for each model variant. When I received the result of the tuning, I could

select the best model by smallest validation loss or smallest MSE on the noisy data. In our selection, I prioritized the model that performed best on the 5% noise data.

I selected the model that has MSE of 0.0007 on 1% noise and MSE: 0.0037 on 5% noise. This model has 300 neurons in both hidden layers and a learning rate of 0.0002.

I also plot the result on both noise variants and select the range from 7000 to 10000. This range allows us to see the details for a small number of movements. Fig. 4.6

While the 1% data is predicted quite well, the 5% shows big errors on the extension-flexion movement, and the abduction-adduction predictions are highly inaccurate. I also see some peaks in the DoFs that should be stationary. Those peaks coincide with the movement in other DoFs, meaning that the data from one DoF somehow influence the prediction of others.

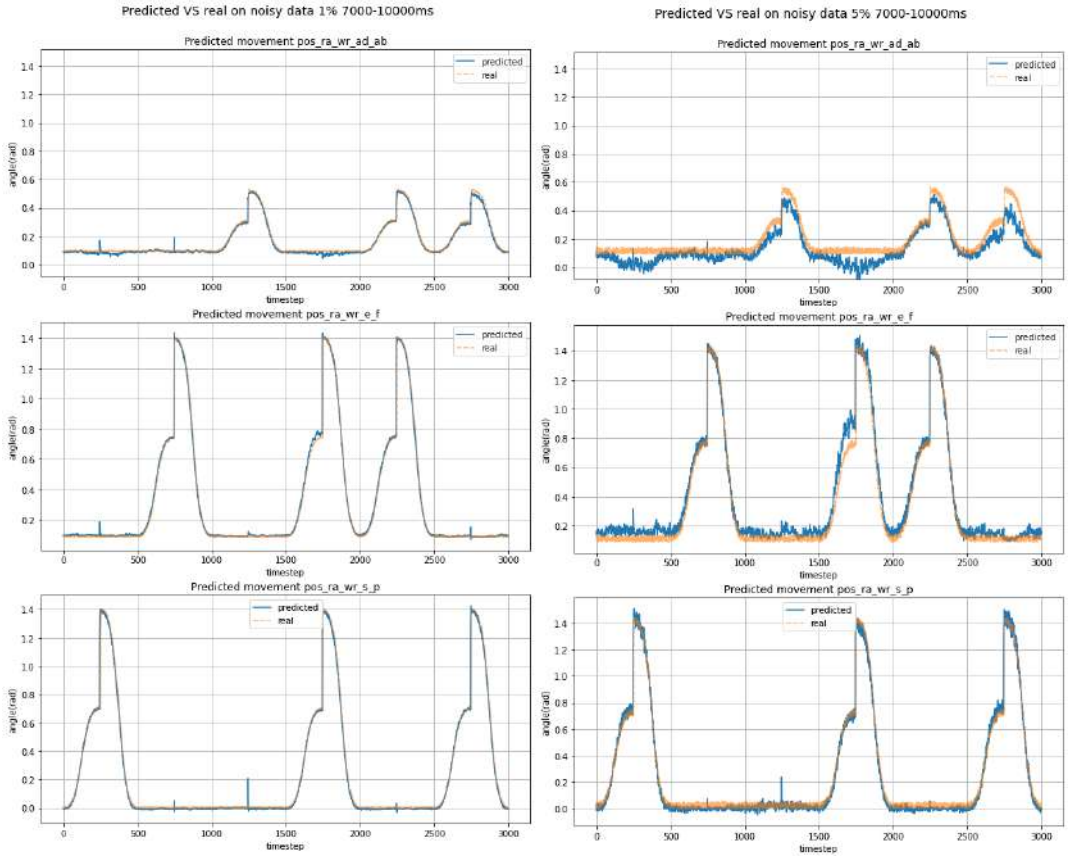

FIGURE 4.6: Prediction of the model trained on markers on noise of 1% and 5% in the sequence from 7000 ms to 10000ms

I also run the tests to check how the number of layers impacts the performance on noise, especially 5%, because I noticed that it worsens the model performance too much. I also did some hyperparameter tuning in those experiments (I tuned a number of neurons in the layers and the learning rate) and measured MSE on noisy data.

I did not receive the increase in accuracy on noise, or it was very small and did not differ too much from the one I had on the simpler model.

So I conclude that increasing the number of layers cannot improve the model stability (performance on noisy inputs).

			train MSE	val MSE	Noise 1%	Noise 5%	Speed
Experiment 4	hidden	240	3.12E-05	3.21E-05	0.0008	0.0066	0.012 ms
	hidden2	240					
	learning rate	0.001					
	dataset type	basis					
	optimizer	Adam					
	epochs	40000					
Experiment 5	hidden	300	2.91E-05	2.60E-05	0.0007	0.0037	0.015 ms
	hidden2	300					
	learning rate	0.0002					
	dataset type	markers					
	optimizer	Adam					
	epochs	15000					

TABLE 4.2: Results for model train on preprocessed data VS model trained on marker data

We measured the computation speed for this experiment as well. It is 0.015 ms for one prediction.

Table 4.2 shows the difference in results for 4.3.4 and 4.3.5 experiments.

We can conclude that the model trained on markers shows better results, both on noise and during the training, but it is slightly slower.

4.3.6 CNN architecture on markers data

Because I face a decrease in performance when the model has to predict the noisy data, I tried another architecture. This approach required the use of convolution layers in the model. I perform a 1-D convolution before passing data to the subsequent layers. I have two 1-D convolution layers, first with a kernel size of 5 and the second with a kernel size of 3. After convolution, I do ReLU and pass the data to the output layer. The model outputs three torques, one in each degree of freedom.

I try different layer sizes and different learning rates, eventually selecting those that yield the best loss.

I use batches for this training with size 64.

In this experiment, I focus the most on the stability of the model. I check it with a series of plots with different resolutions.

Fig. 4.7 shows bad performance on the noisy data. Then I select the first 2000 timesteps from the data and plot it. The plot shows that the range of predictions exceeds the range of errors.Fig. 4.8 I cannot say that convolution somehow improved the solutions' stability to the noise.

I ran many experiments to determine the best possible solution for the inverse kinematics and gathered enough information to select the most suited model.

4.4 Experiments on ID

Our next task was to see if I can apply the architecture I had in the IK solution to the inverse dynamics solution.

I selected the simple model that showed good results in experiment 4 and the experiment with markers.

It is a model with one hidden layer.

I prepared the inverse dynamics data in the same way - I merged the data into the sequences of 5 and performed the train/test split. I did not have the validation split because the amount of data was too low. I also needed to scale that data to fit into the range (-1, 1).

When I run the experiment on the selected model, I received poor performance. The model could not achieve good results with this setup. To fix this, I applied a more complex model that consists of 3 hidden layers. I did the hyperparameter tuning in the same manner as in the previous experiments and determined the following number of neurons: the first hidden layer had 330 neurons, second - 60, third - 165. the learning rate is 0.0001.

The test loss was 0.000137.

I also plotted the prediction on the full dataset to see how the model predicts torques in general. Fig. 4.9

4.5 Inverse kinematics to inverse dynamics

I need to know how the error in inverse kinematics impacts the error on inverse dynamics. To check it, I selected the best model to predict angles. Next, I run the prediction on clean data and form the dataset of angles. Then I incorporate this dataset into a prediction of inverse dynamics. After this, I compare the MSE on ID prediction on clean data and ID prediction on our data.

The MSE on angles prediction was 0.00054.

MSE on ID prediction was 0.1727.

The results are not accurate and require additional investigations. I plot the data in a way that I can see where it goes wrong. Fig. 4.10 It seems that the model estimates the torques slightly higher than they should be. The trajectory seems similar, but its position is off.

It is a common thing that error in inverse kinematics disrupts the inverse dynamics calculations a lot. I will address this issue in our future work.

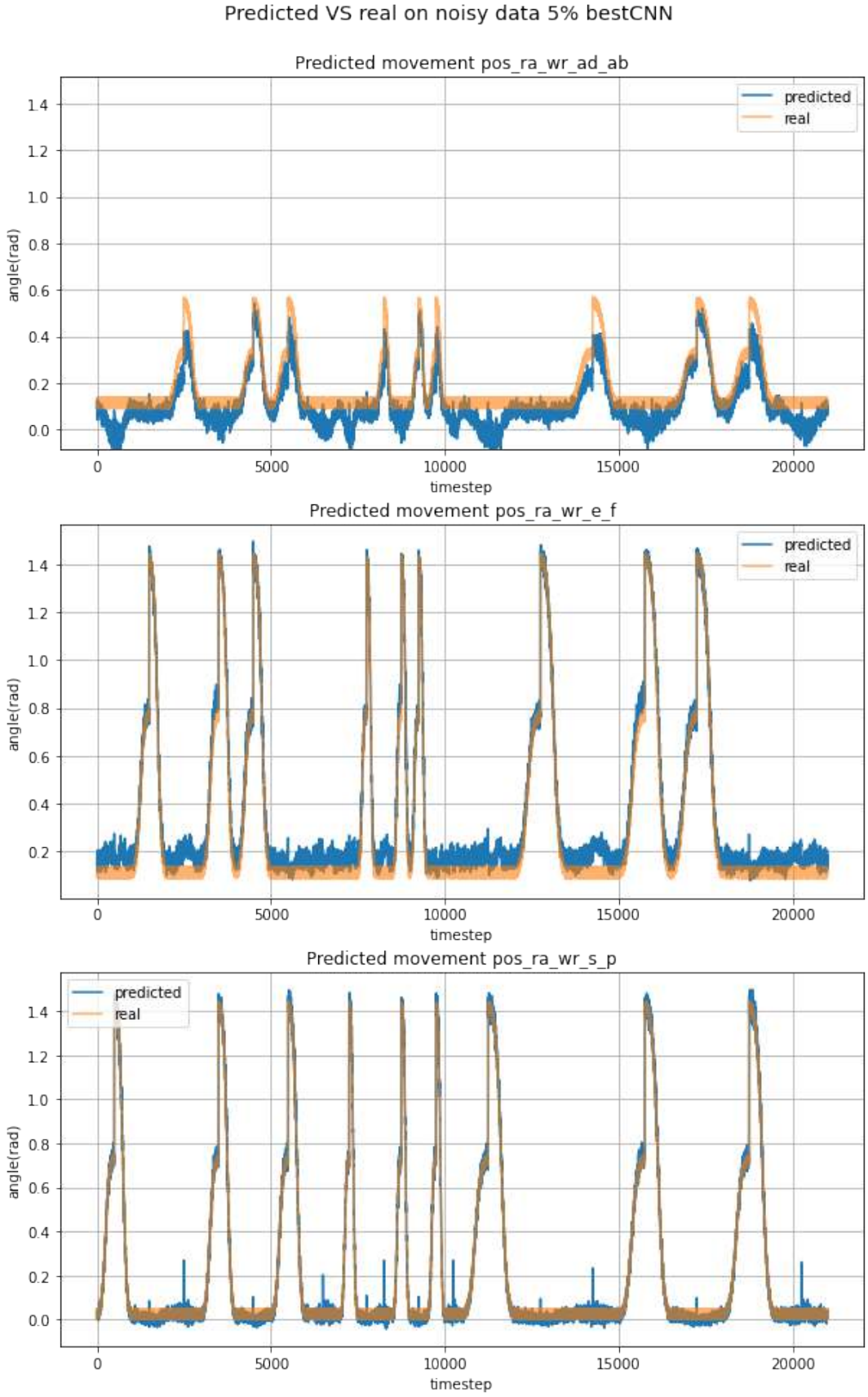

FIGURE 4.7: Predictions for noisy data using convolution layers in the model

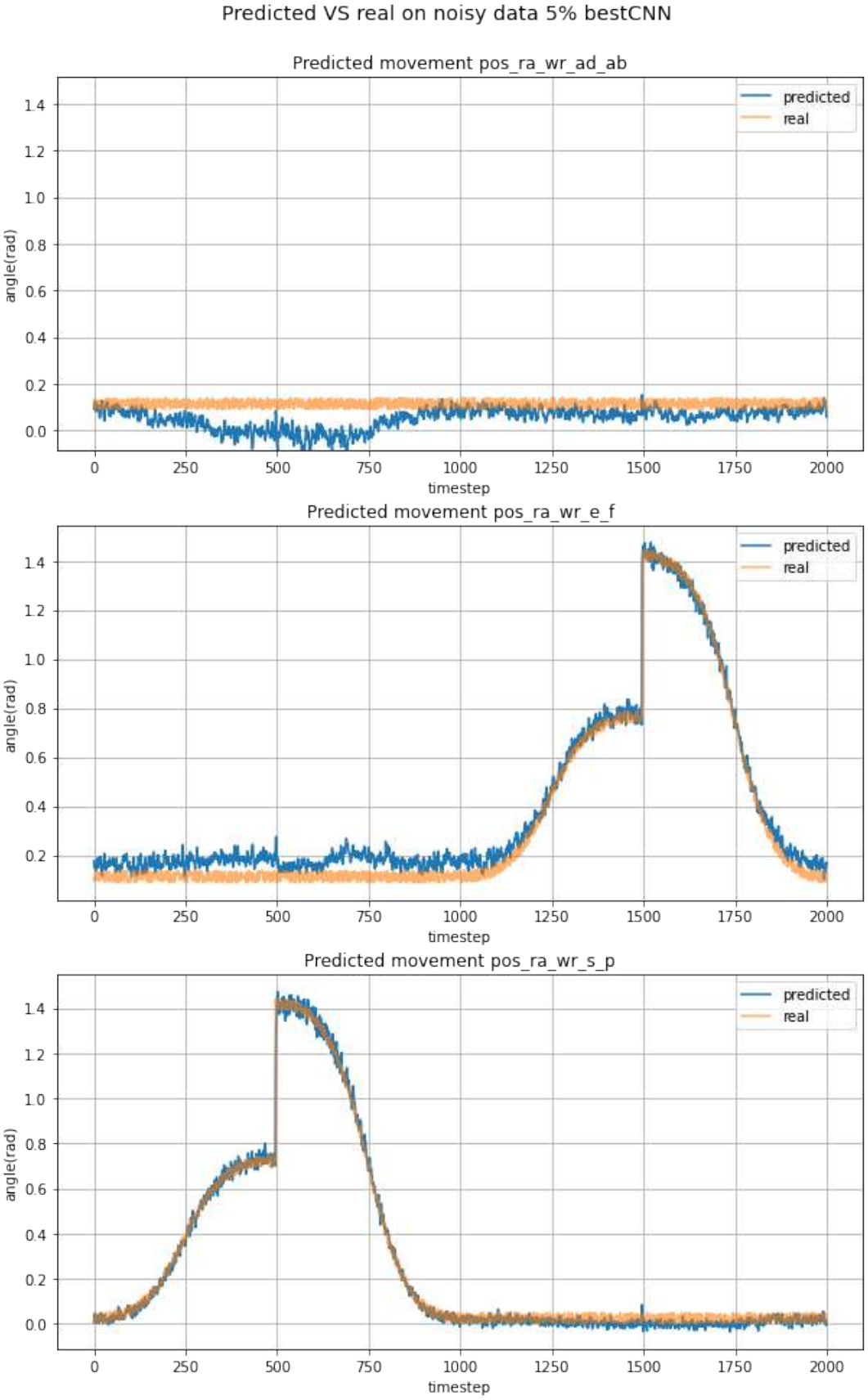

FIGURE 4.8: Predictions for first 2000 timesteps of noisy data using convolution layers in the model

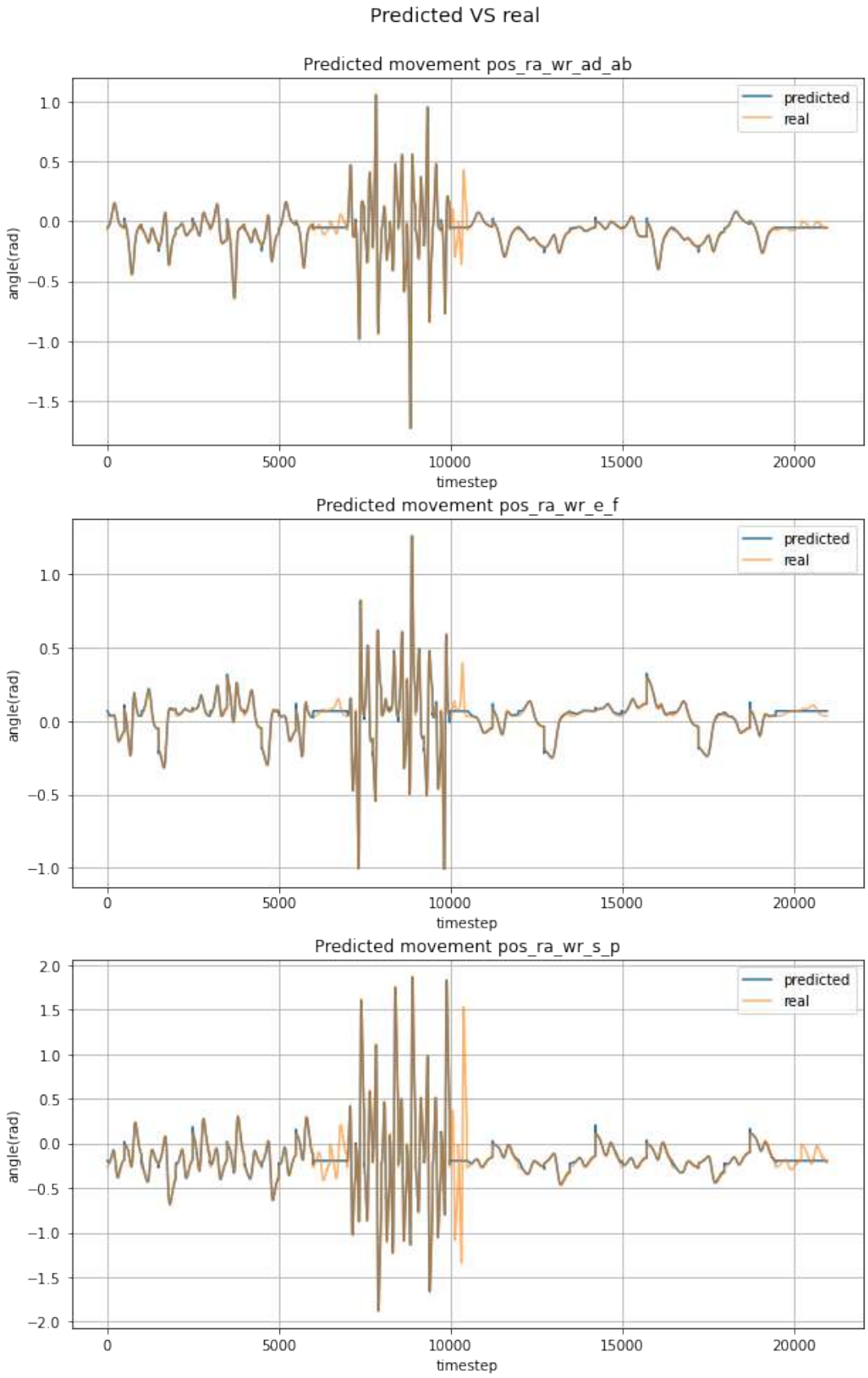

FIGURE 4.9: Prediction of torques

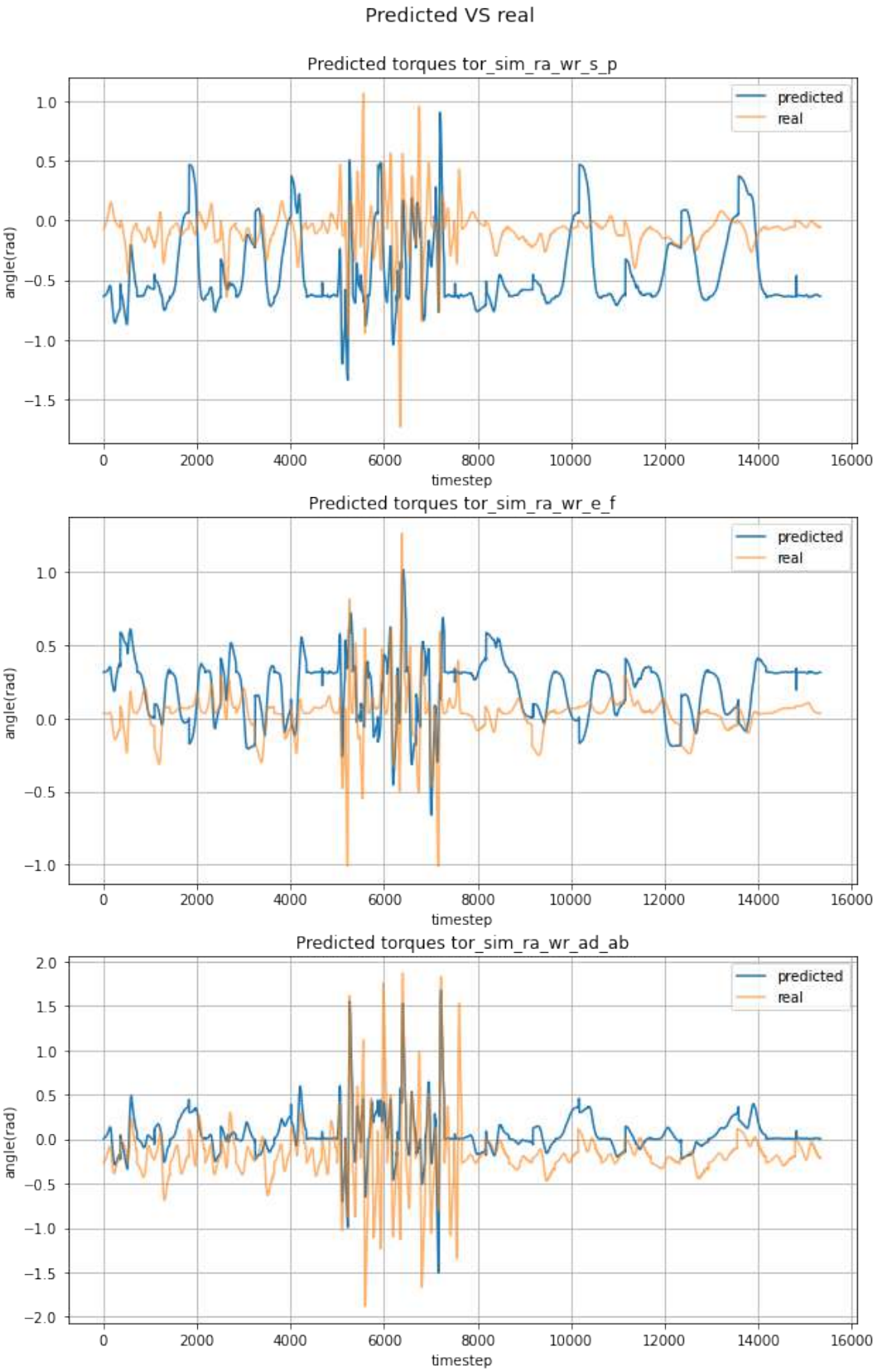

FIGURE 4.10: Prediction of torques, based on angles predictions

Chapter 5

Results and conclusion

We had four goals for this project: create the model that can solve the inverse kinematics problem, test how this model reacts to near-real-world data, find the model that can solve the inverse dynamics task, check how the error in inverse kinematics estimation impacts the error in inverse dynamics estimation.

We completed our project goals. Our experiments allowed us to make several conclusions.

The first conclusion is that on the data we have available, we can train an effective model both on raw data and preprocessed data. Both approaches have advantages and disadvantages. In defense of the usage of marker data, we can mention that this method requires no preparation of the data and might potentially open the way to a faster pipeline on the end device. The disadvantages of using marker data are that people have the length of the different segments, and we might face issues with that. In our data, the segment had a constant length. This issue is not present when using preprocessed data (data with basis and unit vectors projections). However, preprocessed data requires additional work.

We measured the speed of prediction of a single entry. It is less than 2 ms, and the model trained on markers is slightly slower than the model trained on preprocessed data.

The second conclusion was that we could achieve model stability in the noise. As we are going to use this solution with real people, we will face the issue of marker coordinates changing a lot. People are not machines and cannot move the hand in a perfectly straight manner. We run many tests on noisy data that was available for us. We can conclude that model that has been trained on markers shows an error on noise twice as smaller as the same model trained on preprocessed data. Another conclusion on this is that the increased complexity of the model (additional hidden layers or convolution layers) could not improve the stability of the solution.

The third conclusion applies to the inverse dynamics solution. The task of inverse dynamics requires a more complex model than the inverse kinematics task.

The fourth conclusion is that the error in inverse kinematics, although small by itself, dramatically influences the error in the inverse dynamics solution.

Overall, this work presents the solution to the inverse kinematics and inverse dynamics problem. We can use those solutions in future works. The future works will be focused mainly on the minimization of error and increasing the stability of the model.

Chapter 6

Future work

This solution can be used in the scientific researches of the NERL. The next logical step is to apply the IK model and ID model to the real motion capture experiment. One of the unofficial goals of this work was to check if it is possible to develop a model that will work in this domain with the given data. This work was a preparation for the more complex experiments. We propose to use the motion capture data for further training of the model. We can retrain the model on the real-world coordinates and see how it works with them. There is a possibility for several experiments:

1. test the existing model in the mocap experiment;
2. include the mocap data into the training process;
3. apply the solution not only to the wrist joint but to some other rotational joints, and see how it generalizes on the new body part.

The solution has the issue of noise sensitivity. One of the techniques to solve this is some kind of denoising filter, like the Kalman filter. We could include the denoise step into the estimation process. This will ensure that the model receives nearly clean data.

We tried to keep the solution very simple in case we have to implement it on the end device. However, there is still a possibility to increase the model complexity and try different architectures. A good candidate for it is recurrent neural networks. We had some version of history in the model, but recurrent neural networks deal with timesteps better.

To conclude, there are several ways to improve the solution: we can diversify the training data, so the model sees unique movement during the training; we can also include some filters in the model to increase its stability, and finally, we can increase the complexity of the model itself.

Bibliography

Duka, Adrian-Vasile (2014). "Neural Network based Inverse Kinematics Solution for Trajectory Tracking of a Robotic Arm". In: *Procedia Technology* 12. The 7th International Conference Interdisciplinarity in Engineering, INTER-ENG 2013, 10-11 October 2013, Petru Maior University of Tirgu Mures, Romania, pp. 20–27. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2013.12.451. URL: https://www.sciencedirect.com/science/article/pii/S2212017313006361.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. Adaptive computation and machine learning. MIT Press, p. 195. URL: https://books.google.co.in/books?id=Np9SDQAAQBAJ.

Kim, Ji-Hwan et al. (2009). "Forearm Motion Tracking with Estimating Joint Angles from Inertial Sensor Signals". In: *BioMedical Engineering and Informatics*, pp. 1–4. DOI: 10.1109/BMEI.2009.5305761. URL: https://doi.org/10.1109/BMEI.2009.5305761.

OpenSim. *Getting Started with Inverse Kinematics*. URL: https://simtk-confluence.stanford.edu/display/OpenSim/Getting+Started+with+Inverse+Kinematics (visited on 12/25/2020).

Pizzolato, C. et al. (2017). "Real-time inverse kinematics and inverse dynamics for lower limb applications using OpenSim". In: *Computer Methods in Biomechanics and Biomedical Engineering* 20 (4), pp. 436–445. DOI: 10.1080/10255842.2016.1240789. URL: https://doi.org/10.1080/10255842.2016.1240789.

Polydoros, Athanasios S., Lazaros Nalpantidis, and Volker Kruger (2015). "Real-time deep learning of robotic manipulator inverse dynamics". In: *Intelligent Robots and Systems*, pp. 3442–3448. DOI: 10.1109/IROS.2015.7353857. URL: https://doi.org/10.1109/IROS.2015.7353857.

Rapetti, Lorenzo et al. (2019). "Model-Based Real-Time Motion Tracking using Dynamical Inverse Kinematics on SO(3)." In: DOI: 10.3390/a13100266. URL: https://doi.org/10.3390/a13100266.

Rapetti, Lorenzo et al. (2020). "Model-Based Real-Time Motion Tracking Using Dynamical Inverse Kinematics". In: *Algorithms* 13.10. ISSN: 1999-4893. DOI: 10.3390/a13100266. URL: https://www.mdpi.com/1999-4893/13/10/266.

Seth, Ajay et al. (2010). "OpenSim: a musculoskeletal modeling and simulation framework for in silico investigations and exchange". In: *Procedia IUTAM* 2 (1), pp. 212–232. DOI: 10.1016/j.piutam.2011.04.021. URL: https://doi.org/10.1016/j.piutam.2011.04.021.

Stanev, Dimitar and Konstantinos Moustakas (2019). "Modeling musculoskeletal kinematic and dynamic redundancy using null space projection." In: *PLOS ONE* 14 (1). DOI: 10.1371/JOURNAL.PONE.0209171. URL: https://doi.org/10.1371/JOURNAL.PONE.0209171.

Tolani, Deepak, Ambarish Goswami, and Norman I. Badler (2000). "Real-time inverse kinematics techniques for anthropomorphic limbs". In: *Graphical Models graphical Models and Image Processing computer Vision, Graphics, and Image Processing* 62.5, pp. 353–388.

Tompson, Jonathan et al. (Sept. 2014). "Real-Time Continuous Pose Recovery of Human Hands Using Convolutional Networks". In: *ACM Trans. Graph.* 33.5. ISSN: 0730-0301. DOI: 10.1145/2629500. URL: https://doi.org/10.1145/2629500.

Zhang, Zhengyou (2012). "Microsoft Kinect Sensor and Its Effect". In: *IEEE MultiMedia* 19.2, pp. 4–10. DOI: 10.1109/MMUL.2012.24.