MASTER'S THESIS 2020

Sim-To-Real: Domain Adaptation of Robot Trajectories with LSTM

Liam Neric

ISSN 1650-2884
LU-CS-EX 2020-63

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX 2020-63

Sim-To-Real: Domain Adaptation of Robot Trajectories with LSTM

Liam Neric

Sim-To-Real: Domain Adaptation of Robot Trajectories with LSTM

Liam Neric
tfy14lne@student.lu.se

September 29, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Alexander Dürr, alexander.durr@cs.lth.se
Elin Anna Topp, elin_anna.topp@cs.lth.se

Examiner: Volker Krueger, volker.krueger@cs.lth.se

Abstract

Training intelligent agents in autonomous robotics is a data-intensive process, but gathering data from robotic experiments can be a costly and inefficient process. Robot simulation on the other hand offers an efficient and consistent way to gather data instead, but can often be inaccurate and fail to capture real world complexities. We look into the problem of inadequate accuracy in robot simulators by investigating the discrepancies in trajectories between simulation and reality for a Franka Emika Panda robot. In our experiment we create an extensive dataset with free movements of our robot, repeat it in simulation, and subsequently use this data to develop a Long-Short Term Memory architecture that can transfer simulated sensor readings of position, velocity and torque into more realistic ones. Our architecture was able to compensate for the simulator's shortcoming: the estimation of torque, and reduced the root mean square error between simulated and real torque with at least 70%.

Keywords: Trajectory, Robotic arm, Dynamics, AI, LSTM, Sim-To-Real

Acknowledgements

First and foremost, I would like to thank my co-supervisor Alexander Dürr for the idea of this thesis, and for helping me through every step of the thesis process. I would also like to thank Elin Anna Topp for many helpful meetings and guidance, Volker Krueger for examining this work, and Mathias Haage for expert advice in the field of robotics. Additionally, I would like to thank my friend Marcus Grönvall for helpful insights in statistical algorithms. Finally I thank my friends and family for their constant help and support during the writing of this thesis.

Contents

1 Introduction . . . 7
1.1 Background . . . 7
1.2 Research Question . . . 8
1.3 Related work . . . 9
1.4 Contributions . . . 9
1.5 Thesis Overview . . . 10
2 Theory . . . 12
2.1 Franka Emika Robot specifications . . . 12
2.2 Controlling robot trajectories . . . 13
2.3 RRT-connect . . . 14
2.4 Artificial Neural Networks . . . 15
2.5 Recurrent Neural Networks . . . 16
2.6 Long Short-Term Memory (LSTM) . . . 17
2.7 Rolling-Window Method . . . 18
2.8 RMSProp . . . 18
2.9 Mean Square Error . . . 19
2.10 Mean shift detection . . . 19
2.11 System Identification . . . 20
3 Approach . . . 21
3.1 Sequence to Sequence learning . . . 21
3.2 Model implementation . . . 21
3.3 Residual loss function . . . 22
3.4 Model experiments . . . 22
4 Implementation . . . 25
4.1 Dataset . . . 25
4.2 Data Alignment . . . 25
5 Experimental Setup . . . 26
5.1 ROS Architecture . . . 26
5.2 MoveIt Motion Planning . . . 26
5.3 Gazebo . . . 26

5.4 Rosbag 26
5.5 Docker 26
5.6 Pickle 27
5.7 Experiment design 27
6 Results 32
6.1 Complete Trajectory Data 32
6.2 Predictions with Pekel Simulation parameters 36
6.3 Predictions on Linear Dataset with Pekel simulation parameters 39
6.4 Predictions with Gaz simulation parameters 42
7 Discussion 45
8 Conclusion and Future Work 47

1 Introduction

We investigate the use of Long-Short Term Memory networks to improve the accuracy of robot simulators.

1.1 Background

The development of intelligent autonomous robots is often dependent on large amounts of data. This data is used in reinforcement learning to train an agent given a policy, but gathering great amounts of data from robotic experiments is a costly and inefficient process. The alternative is to simulate a robot and generate data from virtual experiments. Using simulated data, an agent can be trained with the objective to transfer the learned policy from simulation to the real world. However, simulation can not produce a perfect model of a robot since the real world contains too many complexities and irregularities, and if the simulation is too inaccurate it can not be used at all.

One of the most common approaches to improving simulation is with system identification, where a mathematical model describing the robot dynamics is developed. This is done by analyzing the input and output of the robot system during executed tasks and identifying its dynamic parameters. The parameters are then used for the virtual model in the physics based robot simulation; creating a calibrated simulation of the robot.

However, the dynamic parameters are only estimated and are never perfect, and the mathematical models that are used are often unable to capture real world factors such as acquired joint friction and imperfect robot controllers. To combat this, we propose a state of the art approach called residual physics learning, that bridges the gap between the flawed simulation and reality. We use our approach to adapt between the domains of simulation and reality. It is a supervised learning approach where a model is deployed to estimate the difference between a prediction made by a physics engine and the measured data from a real experiment. As trajectories are not classified, but predicted, it is a regression task.

We investigate the potential of a residual physics model in a trajectory control task. We use a Franka Emika Panda robotic arm and execute free space movements, and then repeat the same movements in simulation. The Panda robot has 7 joints, all equipped with sensors for position, velocity and torque. The continuous joint-wise measurements of these dynamic variables results in a total of 21 time series for both robot and simulation. This data we refer to as the robot trajectory. The robot is moved by specifying a target configuration that the robot planner uses to create a plan for a trajectory that moves from its current configuration to the target one. The plan contains position and velocity waypoints that the robot controller has to follow when executing the plan. Usually the plan contains only a dozen of waypoints, not to overcomplicate the execution, which leaves a lot of room for interpretation by the robot controller. As the simulated and the real robot are controlled by two different controllers, the differences can be substantial.

Since our trajectory data is sequential, we propose a Long-Short Term memory (LSTM) architecture for our residual physics approach. The LSTM network has the capability to remember short and long temporal dependencies that can be useful for making inferences about the behavior of a whole trajectory. Our problem domain is within sequence to sequence learning, where we map a simulated sequence (trajectory) to a real one. We propose a deep, bilayered architecture to handle the high

dimensionality of this domain. We prove that our model architecture is capable of improving the simulation of trajectories in a myriad of different robot movements, especially in torque where our simulation was the most inaccurate.

1.2 Research Question

We try to answer the following question:

can a recurrent neural network, given simulated trajectory data, find an accurate transformation from a simulated to a real trajectory?

The figure below further illustrates the concept, simulation data is transferred to reality in our "sim-to-real transfer" model, and the output is an improved version of the simulation, ideally indistinguishable to the real robot data.

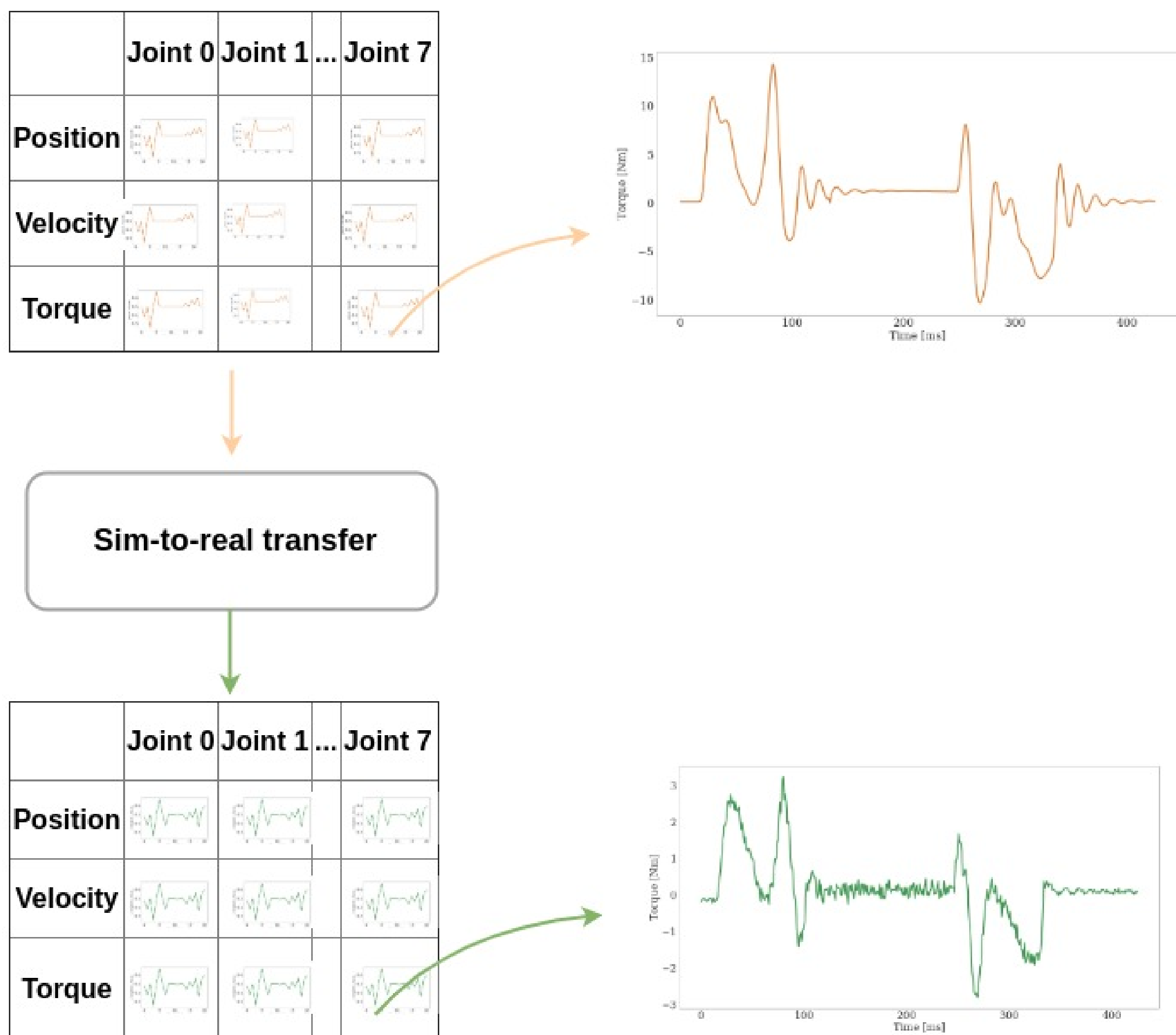

Figure 1: Simulation data (orange) is improved through the sim-to-real transfer approach to be more similar to real robot data (green). A torque trajectory from Joint 7 is magnified for clarification.

1.3 Related work

Zeng et al. [1] investigates if a robotic arm can learn to throw different objects into boxes. After a physics engine calculates the velocity to throw the object, a residual (extra) velocity is added on top of the velocity calculated by the physics engine. This residual velocity is predicted by a convolutional neural network. The residual model outperformed other approaches in terms of throwing accuracy. Ajay et al. [2] implement a recurrent neural network to predict the residual between simulated and real data in the case of planar pushing with focus on modelling uncertainty in the predictions. Using a hybrid model consisting of both a deterministic physics engine and a stochastic neural network was found to generalize better to different objects and required less data to train compared to purely data driven methods.
Kloss et al. [3] also look into the case of planar pushing, where they use a convolutional neural network to predict a residual velocity based on sensory input. This residual is added to the analytically calculated velocity. The model is only taking current sensor data into account, effectively supplementing the physics model. The residual model investigated was found to generalize better and could learn from less data. However, with enough training data, a purely data driven approach was in some cases found to perform better.

A very relevant related work is that of Gaz et al. [4], who retrieve dynamic parameters for the Franka Emika Panda robot using penalty based optimization. They visualize their improved estimation for torque in one trajectory using their developed dynamic model based on the Newton-Euler algorithm, which seems very promising.

1.4 Contributions

The main contributions from this thesis are:

- An approach to improving robot simulation that does not rely on system identification of the robot. We introduce a recurrent neural network architecture that is easily deployed in different robotic systems, and can compensate for particular weaknesses in simulation of trajectories.

- An evaluation of the ability of our architecture to work with inaccurate simulation, and a comparison in performance between two different types of loss functions.

- A developed Python pipeline for planning, executing, and recording trajectories on the Franka Emika Panda robot.

- The work from this thesis became a part of a paper currently in submission for the IROS 2020 conference. The paper elaborates on how our approach can be deployed in a real-time reinforcement learning application.

The LSTM architecture developed for the submitted IROS 2020 paper was developed jointly by the author of this thesis and Alexander Dürr. The same architecture used in the paper was used in this thesis. The rest of the work described was developed separately from the paper by the author.

1.5 Thesis Overview

We analyze the performance of a robot simulator and develop a model to increase its accuracy. In the theory section we explain relevant theory for both the robot and our supervised learning approach. In the sections of implementation and experiment setup we detail the data that was used, and how it was produced and preprocessed. In the approach section we account for our model design choices, and present the different models used in the results section. Finally a discussion regarding the results is presented along with conclusions from this thesis.

2 Theory

In this section, we begin with introducing the specification of our Panda robot, which is necessary to understand the results of this thesis. We then detail algorithms used for general robotic planning, and finally the algorithms used in our model architecture.

2.1 Franka Emika Robot specifications

The Franka Emika Panda robot has 7 joints, each one equipped with sensors to measure position, velocity and torque.

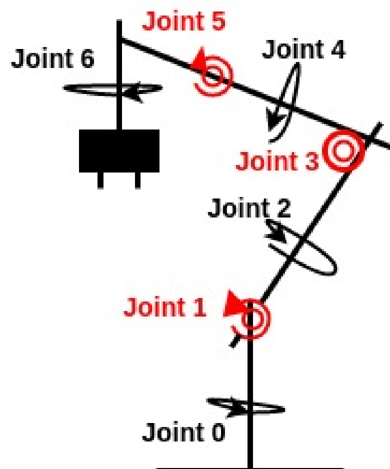

Figure 2: Franka Emika Panda robot, with joint names for every respective axis of rotation.

The seven joints of the robot guarantee seven degrees of freedom (dof) in movement. With the industry standard: 6-dof robot arm, there is free movement and rotation in the three dimensional space. For the panda robot, an extra joint is added for more flexibility, it allows for the robot to reach an end effector position from several different configurations, and improves its ability to perform tasks such as picking up and placing small objects, or efficiently switching between tasks such as grasping and screwing. With the added degree of freedom the robot has more options during planning, making it easier to avoid self collisions or violating the robot restrictions. The added possibilities also bring more complexity and uncertainty to the domain of robot control, introducing more data-driven approaches to problems that may traditionally be solved analytically.

The joint limits for the panda robot are shown in the table below.

Table 1: Shows the joint limits for position q, velocity $\dot{q}$ and torque τ

Name	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7
$q_{\max}$ (Rad)	2.8973	1.7628	2.8973	-0.0698	2.8973	3.7525	2.8973
$q_{\min}$ (Rad)	-2.8973	-1.7628	-2.8973	-3.0718	-2.8973	-0.0175	-2.8973
$\dot{q}_{\max}$ (Rad/s)	2.1750	2.1750	2.1750	2.1750	2.6100	2.6100	2.6100
$\tau_{j\max}$ (Nm)	87	87	87	87	12	12	12

2.2 Controlling robot trajectories

Robot dynamics

The dynamics of a n-dof robot manipulator can be described by the following equation

$$M(q)\ddot{q} + \dot{q}^T C(q)\dot{q} + g(q) = \tau \tag{1}$$

where q is a n x 1 long vector of joint values, M the n x n mass and inertia matrix, C the n x n x n Coriolis tensor, g the n x 1 vector of gravitational force and τ the n x 1 long vector of joint torque.

Finding good values to the parameters of this dynamic system for a robot ensures that the robot can be controlled effectively.

Time parameterization

By interpreting function q(s) in equation (1) as an underlying path of a trajectory, and by including the relationship to time t, we can differentiate $q(s(t))$ with respect to time and derive the following.

$$M(q)(q_s\ddot{s} + q_{ss}\dot{s}^2) + q_s^T C(q)q_s\dot{s}^2 = \tau(s) \tag{2}$$

This can be written in the following form:

$$a(s)\ddot{s} + b(s)\dot{s}^2 + c(s) + g(q) = \tau(s) \tag{3}$$

where:

$$\begin{aligned} a(s) &= M(q(s))q_s(s) \\ b(s) &= M(q(s))q_{ss}(s) + q_s(s)^\top C(q(s))q_s(s) \\ c(s) &= g(q(s)) \end{aligned} \tag{4}$$

By demanding that the torque for every joint i is kept within its limits, the following inequality is yielded.

$$\tau_i^{\min} \leq a_i(s)\ddot{s} + b_i(s)\dot{s}^2 + c_i(s) \leq \tau_i^{\max} \tag{5}$$

Finally, the bounds on $\ddot{s}$ can be written as:

$$\alpha(s,\dot{s}) \leq \ddot{s} \leq \beta(s,\dot{s}) \tag{6}$$

with

$$\begin{aligned} \alpha_i(s,\dot{s}) &= \left(\tau_i^{\alpha} - b_i(s)\dot{s}^2 - c_i(s)\right)/a_i(s) \\ \beta_i(s,\dot{s}) &= \left(\tau_i^{\beta} - b_i(s)s^2 - c_i(s)\right)/a_i(s) \end{aligned} \tag{7}$$

Now a minimal velocity profile can be picked from the vector field $\alpha(s,\dot{s})$, and a maximal velocity profile from vector field $\beta(s,\dot{s})$. By differentiating these profiles with respect to s, corresponding

acceleration profiles can be derived. The time optimal path parameterization (TOPP) (also called time parameterization) can be retrieved by integrating the maximal velocity profile β, which usually has several numerical obstacles that will not be covered here. Consequentially there are different algorithms for calculating the non-deterministic problem of TOPP. In robotics, limits on maximal acceleration and velocity can be set after planning a path, where a TOPP algorithm finds a time-effective path based on the new bounds. The algorithm is generally used to create a smoother trajectory.

2.3 RRT-connect

Rapidly-exploring Random Tree [5] is an algorithm where a high-dimensional space can be searched by incrementally building a space filling tree. The algorithm is biased towards exploring empty states of the search space, making it efficient with non-convex problems since the whole space is sampled more or less uniformly. This has made it a popular algorithm in robotic path planning, where the algorithm accounts for constraints like robot limits and self collision.
The algorithm is initialized with an initial state q_{init}, number of total vertices to add K, the edge length Δq, and a graph G initialized with the initial state q_{init}.

Algorithm 1: RRT Build Tree algorithm

```
1  Input: q_init
2  G = Initialize_graph(q_init)
3  for k = 1 to K do
4  |   q_rand ← rand_conf ()
5  |   q_near ← nearest_vertex(q_rand, G)
6  |   q_new ← new_config (q_near, q_rand, Δq)
7  |   Add vertex (q_new)
8  |   Add edge (q_near, q_new )
9  end
10 return G
```

A random target configuration q_{rand} is chosen in the function *rand_conf*. The nearest vertex in graph G is then calculated in *nearest_vertex*. A new vertex is then added Δq from the nearest vertex, in the direction of q_{rand}. Any time a straight line path exists between a new vertex and the target configuration, it is added to the graph G.

RRT-connect [6], is a modification of a normal RRT planner, where two graphs are created, G_{start} and G_{target}. One of the graphs is iteratively appended until it hits a constraint or a vertex from the other graph, when this happens, the other graph is appended iteratively instead. This version of the RRT algorithm simply tries to connect the initial and target configuration by building trees from both ends and connecting them. The idea is that this algorithm will still sample the configuration space well, while also adding a more greedy element to the algorithm.

In the case of robot path planning, robot constraints can be implemented as a subspace to the search space, and any vertex that would have an edge in the constraint space is rejected.

2.4 Artificial Neural Networks

Artificial neural networks (ANN) are algorithm-based computing networks, inspired by the biological neural networks that exist in human brains. These networks can learn to spot certain patterns or perform tasks without being explicitly programmed to do so. They can also be used as function approximators, where a neural network defines a mapping $y = f(x; \theta)$, where y is the target values, f is the approximated function, x the input, and θ the parameters that the network learns.

The most straightforward type of artificial neural network is a feedforward neural network. It consists of several neurons acting as processing units that form layers. A neuron of a layer is connected to all the neurons of the previous layer. These connections have weights, signifying the importance between two neurons. The value of a neuron Z is determined by the summation of all previous neurons x multiplied by their weights W, added to the neurons bias weight b and finally applied to an activation function a, according to the following equation:

$$Z = a\left(\sum_{i=1}^{n} W_i x_i + b\right) \tag{8}$$

The activation function usually acts as a threshold, normalizing the value of the neuron between [0,1] or [-1,1], either "activating" the neuron or not. If the weights are high enough, it is deemed important and activated. Most activation functions are non-linear, which can help the network learn complex connections.

The computational layers of a neural network are called hidden layers, where the last layer is called the output layer as it returns the output of the network. Figure 3 below shows a small feedforward network with two layers. With 3 inputs, a hidden layer with 3 units, and an output layer with 2 unis.

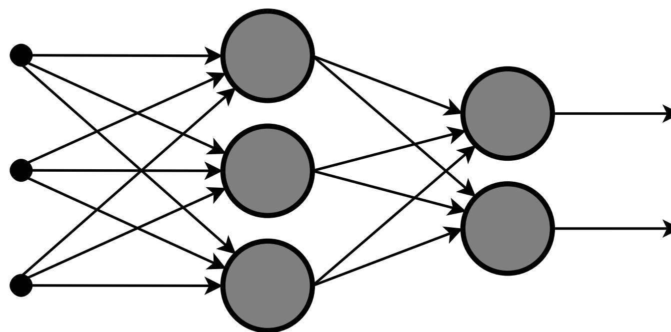

Figure 3: From left to right: inputs, hidden layer, and output layer in a two layer feedforward neural network. Source:Wikimedia Commons

Training a Neural Network

The training process is where the neural network learns to make connections. The process can be described with these steps:

1. Input-target pairs (x, y) are input to the network, and the input is used to compute an output from the network in a feedforward step.

2. The output of a neural network is compared to the target values by a loss function, and returns a loss based on the dissimilarity.
3. Gradients are computed in weight space with a method called backpropagation [7], using the loss from the loss function.
4. The gradients are input to an optimizing algorithm (see section 2.8 for our optimizer) that updates the weights.

These steps are done iteratively, with sets of input-target pairs gathered into "batches", where every iteration is performed on a whole batch to speed up training. All of the data is divided into batches, and the amount of times to repeat this algorithm for all the data is called the amount of "epochs". If the dataset that is used is relatively small compared to the complexity of the neural network, the network can learn to adjust to the particular dataset, and not the general connections that it represents. This is called "overfitting". To prevent this, a dataset is often divided into three parts, a training set for the training process, a validation set to evaluate how well the network can generalize to other data, and finally a completely independent test set for the final evaluation of the network performance.

2.5 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of neural network designed to handle sequential data $x_1, \ldots, x_N$. It is designed to keeps an internal state h that can act as a memory and be used to model the temporal relationships in time series.

The hidden state s_t of a timestep t is updated with a matrix multiplication between recurrent weights U and the hidden state of the previous neuron s_{t-1} summed with the input to the timestep x_t multiplied with weight matrix W:

$$s_t = a\left(Wx_t + Us_{t-1}\right) \tag{9}$$

Figure 4 below shows the interactions in a typical RNN for three timesteps t. In this particular example the output is also sequential, with every neuron computing its own output.

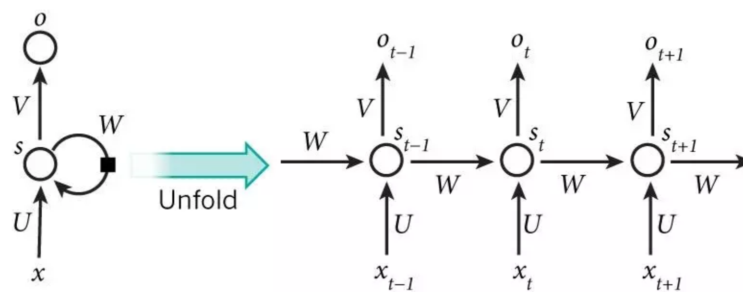

Figure 4: Illustration of interactions between three neurons in a basic fully RNN. Source: Wikimedia Commons

One flaw with basic RNNs is in the modeling of long term temporal dependencies, where neurons are several timesteps apart. This is mostly because of the vanishing gradient problem (see Hochre-

iter [8] for more information), that occurs because gradient values are becoming continually smaller as they are backpropagated through neurons in an RNN. To combat this, another type of recurrent neural network was invented: the Long Short-Term Memory.

2.6 Long Short-Term Memory (LSTM)

An LSTM [9] is a type of recurrent neural network that is able to handle long-term dependencies. This network consists of several LSTM units where every data point is input to a separate LSTM unit. To understand this network, the LSTM unit must first be explained.

An LSTM unit takes x_t as input, has hidden state s_t and cell state c_t. The input weights are noted W_f, W_i, W_o, recurrent weights U_f, U_i, U_o, and biases b_f, b_i, b_o of the forget gate (10), input gate (11), and output gate (12) respectively. It has activation functions σ_g and σ_h, that are normally chosen as the sigmoid and tanh function respectively. The gates and hidden states are updated according to the equations below, where $\circ$ denotes the element-wise product.

$$f_t = \sigma_g\left(W_f x_t + U_f s_{t-1} + b_f\right) \tag{10}$$
$$i_t = \sigma_g\left(W_i x_t + U_i s_{t-1} + b_i\right) \tag{11}$$
$$o_t = \sigma_g\left(W_o x_t + U_o s_{t-1} + b_o\right) \tag{12}$$
$$\tilde{c}_t = \sigma_h\left(W_c x_t + U_c s_{t-1} + b_c\right) \tag{13}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{14}$$
$$s_t = o_t \circ \sigma_h\left(c_t\right) \tag{15}$$

The key component of the unit is the cell state c_t, that regulates which information is saved and what is forgotten from previous LSTM units. The forget gate f_t in equation 10 regulates what parts of the cell state should be kept from previous cells. The input modulation gate $\tilde{c}_t$ (13) computes from the input x_t and hidden state s_{t-1} an interval between 1 and -1 (assuming a tanh activation function) where memory can be kept or forgotten. Input gate i_t (11) is multiplied with the input modulation gate to add new information to the cell state.
After the gates are updated, the new cell state is computed with equation 14, this step is usually referred to as the Constant Error Carousel (CEC), which solves the vanishing gradient problem, as the forget gate acts like both a recurrent weight and an activation function, creating an algorithm that has no restriction on how long to keep its cell state, therefore creating the "long-term memory". Finally the hidden state is then updated by a multiplication of the output with activated cell state as in equation 15.

These LSTM units form a network by stacking together in a structure like in figure 4, with one LSTM unit passing its hidden state to the next one. The output from an LSTM network can either be sequential; where every unit produces an output, or singular; where only the last LSTM unit produces an output.

2.7 Rolling-Window Method

Long sequences can be ineffective for training RNNs. This can be handled by dividing these long sequences into shorter ones. The rolling window method [10] selects a shorter sequence (a window), and incrementally moves (rolls) this window from start to the end of the longer sequence throughout training iterations. This method can be used with an RNN, where every batch of trajectories is chosen a few times every epoch with a different window position, making the model "see" slightly different trajectories every epoch. This augments the training data and prevents overfitting. The method is illustrated by figure 5 below.

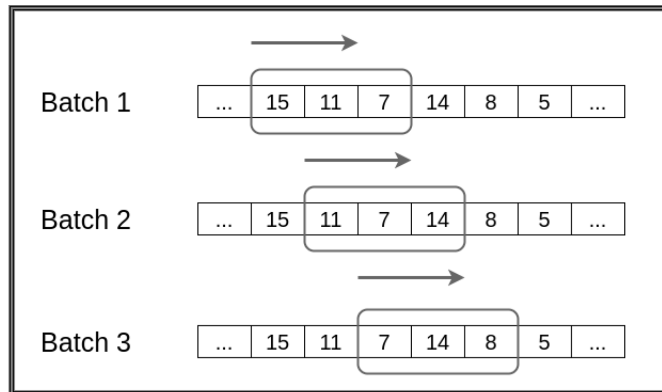

Figure 5: For every repeat of a batch, a window is "rolled" across the trajectory, resulting in several slightly different sub-trajectories that is seen in every epoch.

2.8 RMSProp

In stochastic gradient descent, only a few random samples from the batch (or training set) are chosen for updating the weights in optimization for every iteration. In contrast, normal gradient descent calculates the gradient for the whole batch in every iteration, a much slower but more precise method. Optimization with stochastic gradient descent usually converges faster than normal gradient descent, with the minor setback being that the loss is less minimized after convergence.

RMSProp (Root Mean Square Propagation) is an algorithm using stochastic gradient descent [11]. It is designed to avoid large fluctuations of the gradient by dividing it by the root mean square of the moving average for every batch. The mean square of the moving average v is updated in equation 16 with w being the weights, t time, γ a forgetting factor, and $Q(w)$ the loss function estimating the difference between the predicted robot readings and real ones.

$$v(w,t) = \gamma v(w,t-1) + (1-\gamma)\left(\nabla Q(w)\right)^2 \tag{16}$$

The weights are then updated according to equation 17 where η is the step size. In this equation, the new weights w are updated based on their previous value, this property is referred to as momentum

and increases convergence rate by guiding shifting gradients to the right direction [12].

$$w = w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q(w) \tag{17}$$

RMSProp has generally been found to effectively adapt the learning rate and converge quickly.

2.9 Mean Square Error

The mean square error (MSE) is a measure for the accuracy of an estimator [13]. The MSE is defined as

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i \right)^2 \tag{18}$$

where Y is a vector of true values, and $\hat{Y}$ a vector of estimated values. MSE is often used as a metric for evaluating the closeness of predictions. One pitfall with MSE is that it heavily weighs outliers; if there are a lot of disproportionately poor estimations, the MSE can be a poor metric. To have the error in the same dimension as the input, the square root of the MSE can be calculated instead, then the measure is called the Root Mean Squared Error (RMSE).

2.10 Mean shift detection

When aligning two time series, maximizing the cross-correlation between them is the most common method [14]. However, if the start of the two time series is constant and then starts changing at a certain point — like the position sensors of a robot before a trajectory is executed — they can be aligned by finding this start point in both cases. The start point from which a time series starts to change in a statistically significant way can be found by locating the mean shift of the time series. This is possible with the Mean Shift Detection algorithm.

The expected value μ of a discrete stochastic variable is:

$$\text{Mean(x)} = \mu = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{19}$$

where x are observations and N the total amount of observations.

Using the expected value μ, the standard deviation σ of a discrete stochastic variable can be calculated with

$$\text{Std(x)} = \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}; \tag{20}$$

With these two measures we can explain the mean shift detection algorithm. As input, an initial interval length l and a window length for succeeding values S are specified. Also a factor n can be chosen to adjust the confidence interval for deviations.

Algorithm 2: Mean Shift Detection algorithm

```
1  for k = 0 to N − l do
2      μ_k = Mean(X[0, l+k])
3      σ_k = Std(X[0, l+k])
4      for i = 0 to S do
5          if |(X(i + k) - μ_k| < σ ∗ n then
6              break
7          if i == S then
8              return k
9      end
10 end
```

The central part step of the algorithm is to update the mean μ and standard deviation σ in every iteration. To initialize, an interval length l needs to be defined, since the mean and standard deviation should be calculated from a sequence and not a single point (which would be the case when $k = 0$). A window of length S is traversed along the sequence, checking if all the values inside the window deviate from the mean value μ. If they do, a significant and consistent change has occurred. The algorithm is then stopped and the start point of the window is returned.
This method relies on the sequence not being too noisy. It also requires some tuning of the parameters S and n, controlling the window length and standard deviation factor.

2.11 System Identification

System Identification is a method for building mathematical models for dynamical systems, by using an input and output signal. In our case of robot dynamics, the dynamic parameters have a linear relationship with values for position, velocity, acceleration and torque according to equation 1. By sampling these values for different trajectories that excite the dynamics of the robot, the dynamic parameters can be estimated by different methods such as penalty based optimization [4] or linear least squares [15].

After retrieval, the robot-specific dynamic parameters can be used for calibrating the virtual model in simulation to improve the accuracy of simulated trajectories.

3 Approach

Our objective is to find a model that can improve simulated trajectories by minimizing the residual between simulated and real sequential trajectory data. In this section we describe our LSTM architecture that is designed for handling the complexities in sequential trajectory data, and meticulously account for our design choices to increase the reproducibility of our approach. In addition, we evaluate the results from two different loss functions, and with two different sets of dynamic parameters for the robot, resulting in three models in total.

3.1 Sequence to Sequence learning

In time series forecasting a model is typically deployed to predict N future outcomes Y_{t+N} based on past outcomes Y_t. In contrast, in sequence to sequence learning, an entire sequence is mapped to another sequence. Our proposed LSTM is an instance of sequence to sequence learning, with the task of regression. Based on a sequence of x_t with $t \in \{0, \ldots, T\}$ an entire sequence y is predicted. To be clear, this is different from a "vanilla" LSTM that forecasts a single prediction per feed forward step.

3.2 Model implementation

As the trajectory data consists of several time series describing position, velocity and torque of all seven joints, the natural choice for a model was a type of RNN. Recording in 100 Hz, with second-long trajectories, these time series consists of a few hundred datapoints. For capturing various time dependencies of these longer trajectories either a Gated Recurrent Unit (see Chung et al. [16] for theory) or LSTM based model seems reasonable. Since we want our model to be able to remember information from a whole trajectory, we opted for the more complex LSTM layer, since it has three gates (input gate, update gate, output gate) instead of the two gates of the Gated Recurrent Unit (reset gate, update gate). The model was implemented in Tensorflow [17].

Because of the high dimensionality and complexity in our sequence to sequence regression problem two LSTM layers were concatenated. The first layer returns the hidden state of every LSTM cell to the same-indexed cell of the layer above. This can be interpreted as the first layer creating a feature space optimal for the second LSTM layer to output predictions. To capture the high dimensionality of the data, the first layer had a hidden size of 100, and the second layer a size of 21.

In a normal LSTM that propagates forward in time the hidden state of the first cells (receiving the input from the initial values) does not receive information from future cell states and can therefore only produce a suboptimal hidden state because of the lack of information. As we are training on a whole trajectory at once, we used bidirectional LSTM layers — consisting of two separate layers propagating in opposite directions, with their final output aggregated — so that information is propagated in both directions in time, making inference possible in earlier timesteps.

The default activation functions of the LSTM were used, where the recurrent activation σ_g is a sigmoid function, and activation σ_h is a tanh function. Since the data was normalized in the [0,

1] interval, the sigmoid function is a good choice of recurrent activation since it maps the output state to that same interval (see equation 12). The tanh is used for its [-1, 1] interval allowing parts of the cell state to be both forgotten or remembered (see equation 14).

The rolling window technique was used to shorten the sequences and augment the dataset. The window length was chosen as half the size of the shortest trajectory in the batch, and the window was "rolled" by uniformly randomizing its placement within the trajectories. The batch size was chosen to be 16, with 50 epochs, with a total of 1372 epoch steps. Since the training set had 1372 trajectories, this translates to every epoch going through every trajectory 16 times, with a different window of the trajectory every time. This causes a slight variation of the training data between batches, which augments the training set and reduces the amount of overfitting. In addition, drawing a shorter fixed length window from the variable length trajectories enables batch learning, — since all trajectories in the batch are cut to the same length and subsequently have known dimensions — and creates shorter gradients during training, which improves the efficiency of the training process.

RMSProp algorithm was chosen as the optimizer for its momentum and efficiency. The learning rate was set to 10^{-4} and a callback function that reduced the learning rate whenever the models performance plateaued was also implemented since this has been shown to improve learning.

3.3 Residual loss function

Having neural network based models predict a residual between input data (X) and ground truth (Y) has been found to make models generalize better and require less training data than predicting the ground truth directly [2] [3]. This is implemented by creating a custom loss function that calculates residual loss.
The custom loss function that was used was a modifed MSE function:

$$\mathrm{MSE}_{Residual}(X) = \mathrm{MSE}(X + \mathrm{Model}(X)) \tag{21}$$

$$= \frac{1}{T}\sum_{i=1}^{T}(Y_t - (X_t + \mathrm{Model}(X_t)))^2 . \tag{22}$$

3.4 Model experiments

We trained three models in total, all with the specifications from section 3.2:

1. Model with simulation parameters from Erdal Pekel's blog [18], with loss function from 3.3, referred to as "Residual".
2. Model with simulation parameters from Erdal Pekel's blog, with a normal MSE loss function from 2.9, referred to as "Normal"
3. Model with simulation parameters from Gaz et al. [4] with loss function from 3.3, referred to as "Residual-2

Since the simulation parameters of Erdal Pekel's blog were much better for our Panda robot individual, these parameters were used in all experiments, except the experiment in results section 6.4 where Gaz et al. [4] parameters were used. To test model robustness, the performance on the linear

dataset was compared between model 1 and 3. The performance of these two models with different amounts of training data was also tested.

4 Implementation

Our objective was to create a dataset that is extensive enough to accurately represent different dynamics within the working space of the robot. Additionally, this data needs to be processed before it can be used in our approach with supervised learning.

4.1 Dataset

The data consisted of pairs of simulation and real-robot trajectories; with measured position, velocity and effort. Only trajectories that executed successfully in both simulation and reality were included. Since the dynamics of a certain joint is heavily dependant on the orientation of the robot, there exists complex temporal dependencies between the joints. To reduce these complexities in the data, the amount of joints allowed to move in a trajectory was varied; in data where only a few joints move, the friction of the joints should be easier to learn for a neural network. To test the robustness of the model, a smaller set of linear movements was also collected. In a linear movement the end effector of the robot is moved linearly from one position to another, requiring high precision control from several joints. In linear movements there is also a much higher amount of waypoints that the robotic arm has to follow, making the trajectory harder to execute. If the model can capture these trajectories successfully as well, it indicates that the difference between the robot controllers in simulation and reality has been learned.

4.2 Data Alignment

To create a dataset for supervised learning, the trajectories recorded from simulation and the trajectories from the physical robot had to be aligned. The experiment was set up so that the robot rests before and after an executed trajectory, allowing the recording software to start in time. Since the simulation was made in real-time, the trajectory pairs had the same execution speeds. Therefore it was sufficient to find the starting point of a trajectory and cut out the interval where the robot was resting. This was done by using the mean shift detection method (section 2.10), where the mean of the joint positions was used as an indicator. The position data was used because it is less noisy than velocity or effort data. The robot was recorded with a frequency of 100 Hz, so the window length S and initial interval l were both chosen to be 10 timesteps (0.1s), and the confidence interval factor n was chosen to 1.6. The use of time series cross correlation to find the lag between a simulated and real trajectory was also attempted for alignment, but this method did not perform as well as mean shift detection, and was therefore not used.

Trajectories that did not execute properly were found by analyzing the position data: if the total mean square error between simulation and reality was too large it was disregarded. Since the simulation was very accurate for position, the threshold for removing faulty trajectories could be set so that it did not disregard any valid (but poorly simulated) trajectories.

5 Experimental Setup

We chose the software in our setup based on what was supported and documented for the Franka Emika Panda robot. To increase the reproducibility of our experiment, we explain the programs and libraries we used, which were all open source. We also present the main algorithm of our experiment in pseudocode. The purpose of the experiment in this section was to create the dataset described in section 4.1.

5.1 ROS Architecture

The Robot Operating System (ROS) [19] was used to control the robot. ROS is a framework designed to simplify the process of building applications for robots. One of the main advantages is the communication between software and hardware components in the ROS architecture. It consists of a large set of libraries and tools covering several domains in robotics such as robot geometry, simulation and planning.

5.2 MoveIt Motion Planning

MoveIt Motion Planning [20] is a motion planning library that runs on top of ROS. The Moveit interface was used to plan and execute robot trajectories. The plans consist of waypoints that specify joint position, velocity, and acceleration at certain timepoints. The user can choose to either plan to a joint configuration in joint space, or Cartesian space. Linear moves can be planned in Cartesian space, where the planner creates closely spaced waypoints in joint space that ensure linear movement. MoveIt was also used to:

- Model the workspace environment and avoid collisions with it in the planning stage
- Time parameterize planned trajectories
- Execute plans on the robot

5.3 Gazebo

Gazebo is an open sourced robot simulator capable of running on top of ROS. In our experiment we used Gazebo with the physics engine ODE, which is also the default engine.

5.4 Rosbag

Rosbag is ROS library that can record ROS processes such as trajectories. Our recorded trajectories were recorded in rosbags and converted to .csv (comma separated values) files when the execution was finished.

5.5 Docker

Docker is a service that allows users to work from a container, which contains an isolated software environment. A user can upload software such as libraries and configuration files, and finally share

this container with other users. The advantages with containers are that they are shareable, have all the imported libraries installed, and eliminate any interference or bugs originating from the local computer. We used a docker container for our ROS environment for these aforementioned advantages.

5.6 Pickle

Pickle is a Python library that is used to serialize and deserialize Python objects so they can be saved to a local disc. It was used in the experiment to save plans made in MoveIt so they could be loaded again to be executed in simulation.

5.7 Experiment design

A ROS environment for controlling the robot (both physical and simulated robot), was created in a docker container. MoveIt Motion Planning was used to plan and execute trajectories. The plans were modified with a version of the path parameterization algorithm, called "Iterative Parabolic Time Parameterization" in MoveIt documentation. The parameterization was made with varying upper bounds on velocity, ranging from 20% to 100% of maximal velocity, creating a dataset with varied execution speeds.

The data was gathered with an algorithm containing the following functions:

Initialize_joint_limits: Creates two vectors with the lower and upper position limits of the joints from table 1.
Robot.get_Current_Values: Robot function that retrieves the current joint positions
Robot.set_joint_value_target: Sets a target configuration to plan towards
Plan Plans a trajectory with the RRT-connect algorithm (see section 2.3).
Planchecker: Confirms that the plan does not cheat and moves more joints than it is supposed to
Load_Workspace: Creates the robot workspace to avoid collisions with walls and objects that may exist in the robot's reach.
Robot_Retime_trajectory: Modifies a plan with Time Optimal Path Parameterization.
Record: Starts a rosbag recording of the robot joint values
Stop: Stops the recording and saves it to csv
Error recovery: Resets the robot

These functions were used in the following algorithm.

Algorithm 3: Plan and execute algorithm

```
1  Input Robot, joints_to_move, velocity_bound, acceleration_bound, max_attempts
2  Robot.Set_robot_max_velocity = velocity_bound
3  Robot.Set_robot_max_acceleration = acceleration_bound
4  joints_upper_limits [], joints_lower_limits [] = Initialize_joint_limits()
5  planning_attempts = 0
6  while planning_attempts < max_attempts do
7      Joint_goal [] = Robot.Get_Current_Values()
8      for joint in joints_to_move do
9          Joint_goal[joint] = Random.Uniform(Joints_lower_limits[joint],
             Joints_upper_limits[joint])
10     end
11     Robot.set_joint_value_target(Joint_goal)
12     Plan = Robot.plan()
13     if Plan ≠ None and Planchecker(Plan, joints_to_move) == True then
14         Successful_Plan = True
15         break
16     end
17     else
18         Planning_attempts += 1
19     end
20
21 end
22 if Successful_Plan == True then
23     Load_Workspace()
24     Plan= Robot.Retime_Trajectory(Plan, Velocity_Bound)
25     Record()
26     Robot.Execute(Plan)
27     Stop()
28     if Robot.Get_Current_Values() ≠ Joint_goal then
29         Error_Recovery()
30         Load_workspace()
31     end
32     Save_plan()
33 end
```

Given a set of joints, the algorithm finds a configuration for those joints to move towards where no other joint is allowed to move. Line 6-15 is the core part of the algorithm, where a target configuration is randomized in the for loop at lines 8-10, and a plan is made towards this target configuration in line 12. Sometimes the robot is in an awkward configuration, where moving a certain joint will lead to a collision or be impossible; therefore the algorithm only attempts to find a plan for a particular set of joints a certain number of times (max attempts). When a plan is successful, the while loop is interrupted, and the plan is executed and recorded in lines 22-26. Finally the plan is saved in a Python Pickle file.

Once the plan is saved, executing the plan in simulation is very simple and is done with the following functions:

Algorithm 4: Execute saved plan in simulation

```
1 Input Robot, Plan
2 Record()
3 Robot.Execute(Plan)
4 Stop()
```

The following figure summarizes the two data collecting algorithms:

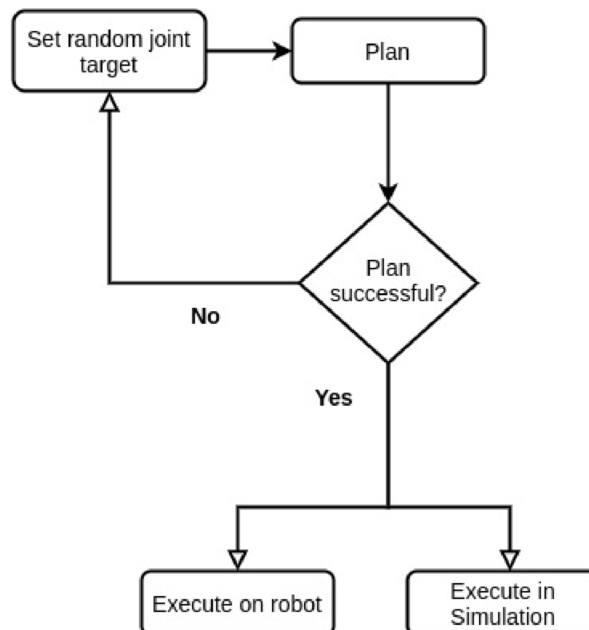

Figure 6: Summarizing schematic over the data collection process

The datasets (see section 4.1) were collected by recording the robot with rosbag. The updating frequency of Gazebo and real robot controllers were set to 100 Hz. The collected data consisted of one dataset with 850 multi-joint moves, one dataset with 870 single joint moves, and one with 50 different linear Cartesian moves in xy, xz, and yz direction. The joint moves were sampled randomly in the workspace.

The dataset was split in the following fractions: 80% for training, 15% for validating, and 5% for testing. The linear dataset was not split, as it was only used for testing.

For the two joint datasets, the simulation was done twice with different hyperparameters describing inertia and mass properties of the Panda robot:

1. Parameters from Erdal Pekel's blog [18]
2. Parameters from Gaz et al. [4]

The simulation of the linear dataset was only done with the parameters from 1.

6 Results

Since the dynamic parameters retrieved from Pekel [18] described our robot dynamics much better than the parameters from Gaz et al. [4], we used the better Pekel parameter simulation for most of the results section.

6.1 Complete Trajectory Data

To illustrate how the data from an executed trajectory looks, the seven joints are plotted in position, velocity and torque for an arbitrary trajectory. Both robot data and simulated data is plotted, using the more accurate Pekel simulation parameters.

Figures 7, 8 and 9 below show how accurate the simulation is for position and velocity. In contrast, the torque simulation is very inaccurate, where the simulation usually follows the trend of the real robot torque, but oscillates substantially. Since torque is the only unreliable part of the simulation, we will continue to study mostly the torque results.

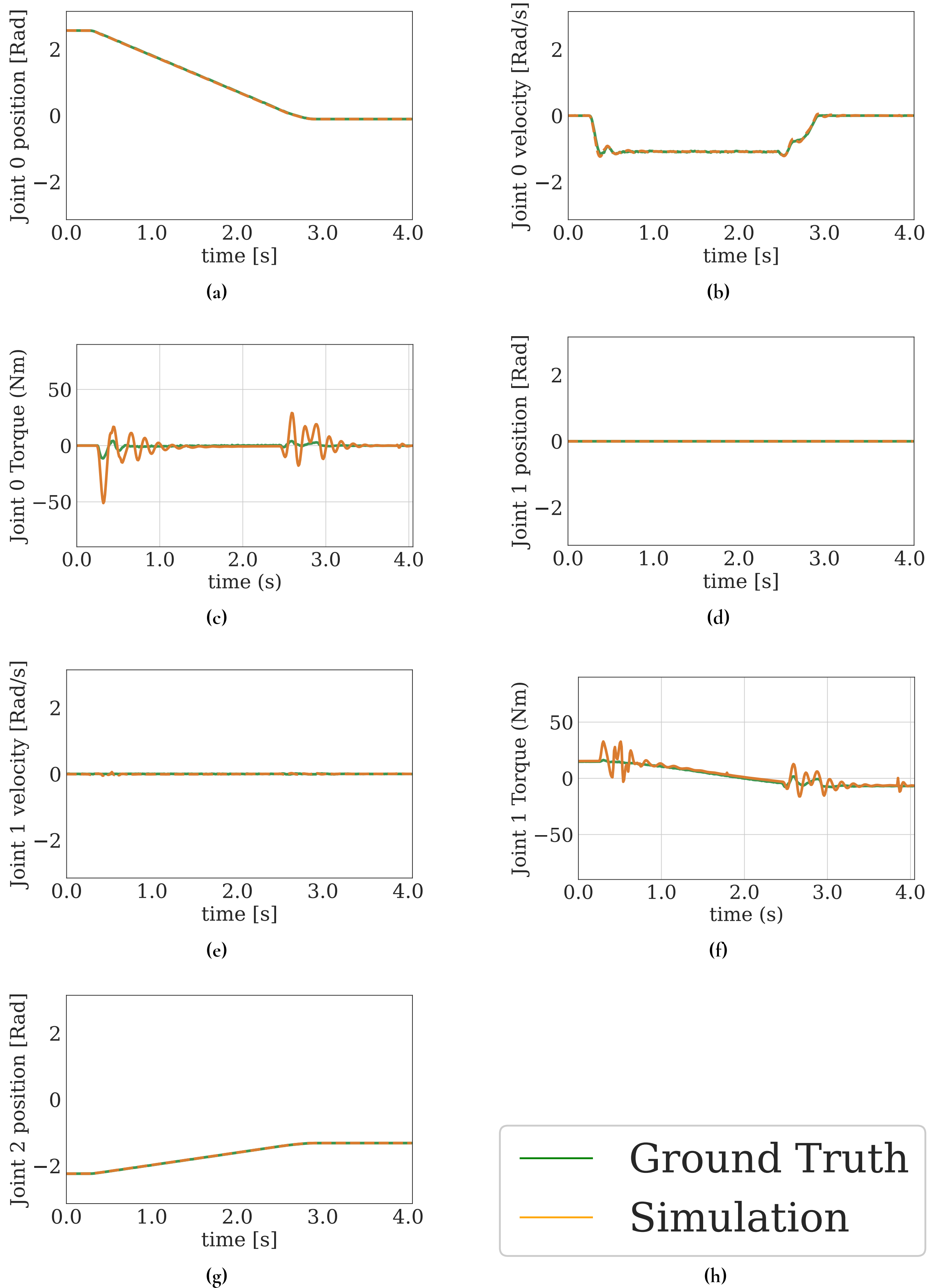

Figure 7: Position, velocity, and effort data for robot (green) and simulation (orange) for joints 0, 1, 2

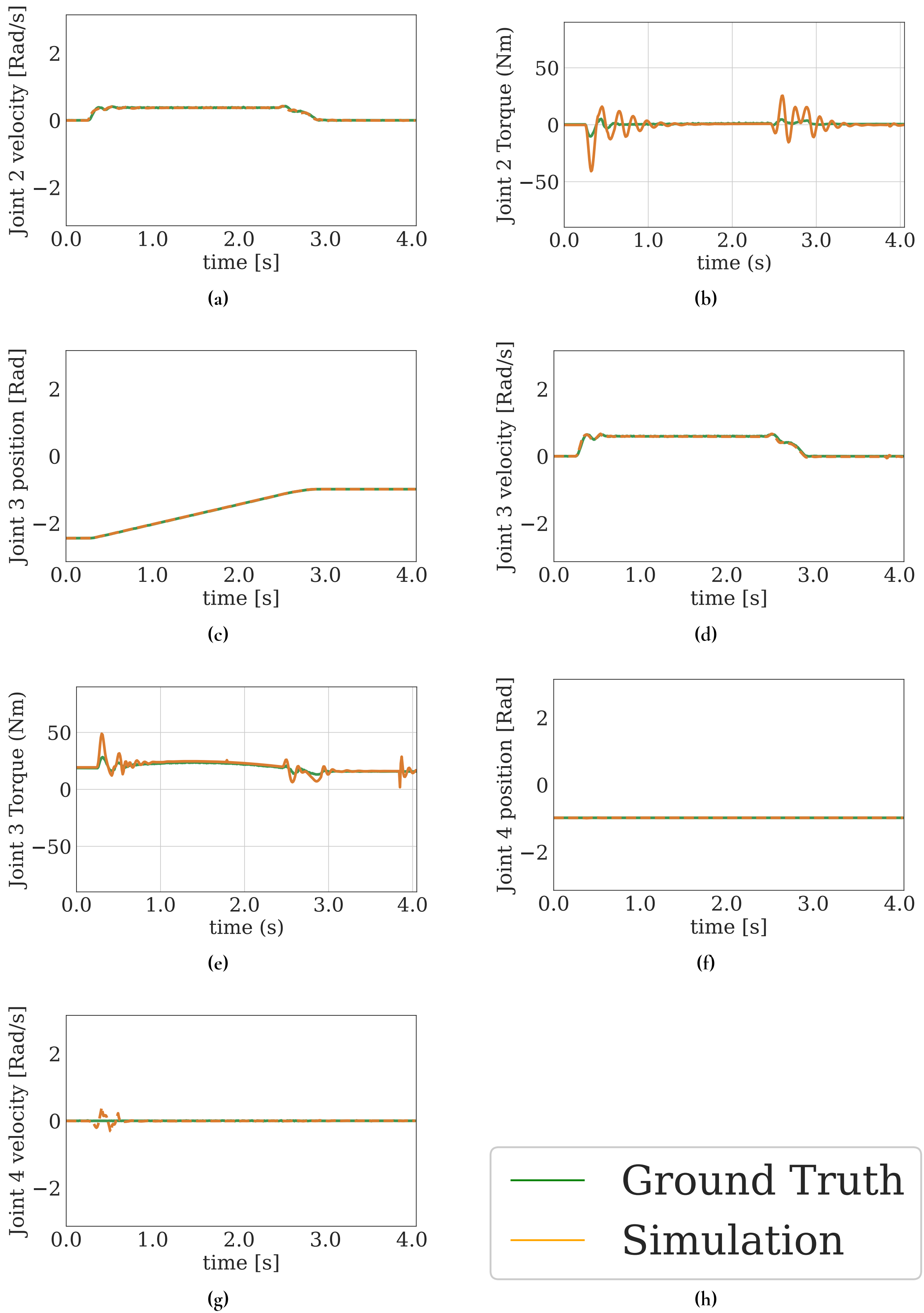

Figure 8: Position, velocity, and effort data for robot (green) and simulation (orange) for joints 2, 3, 4

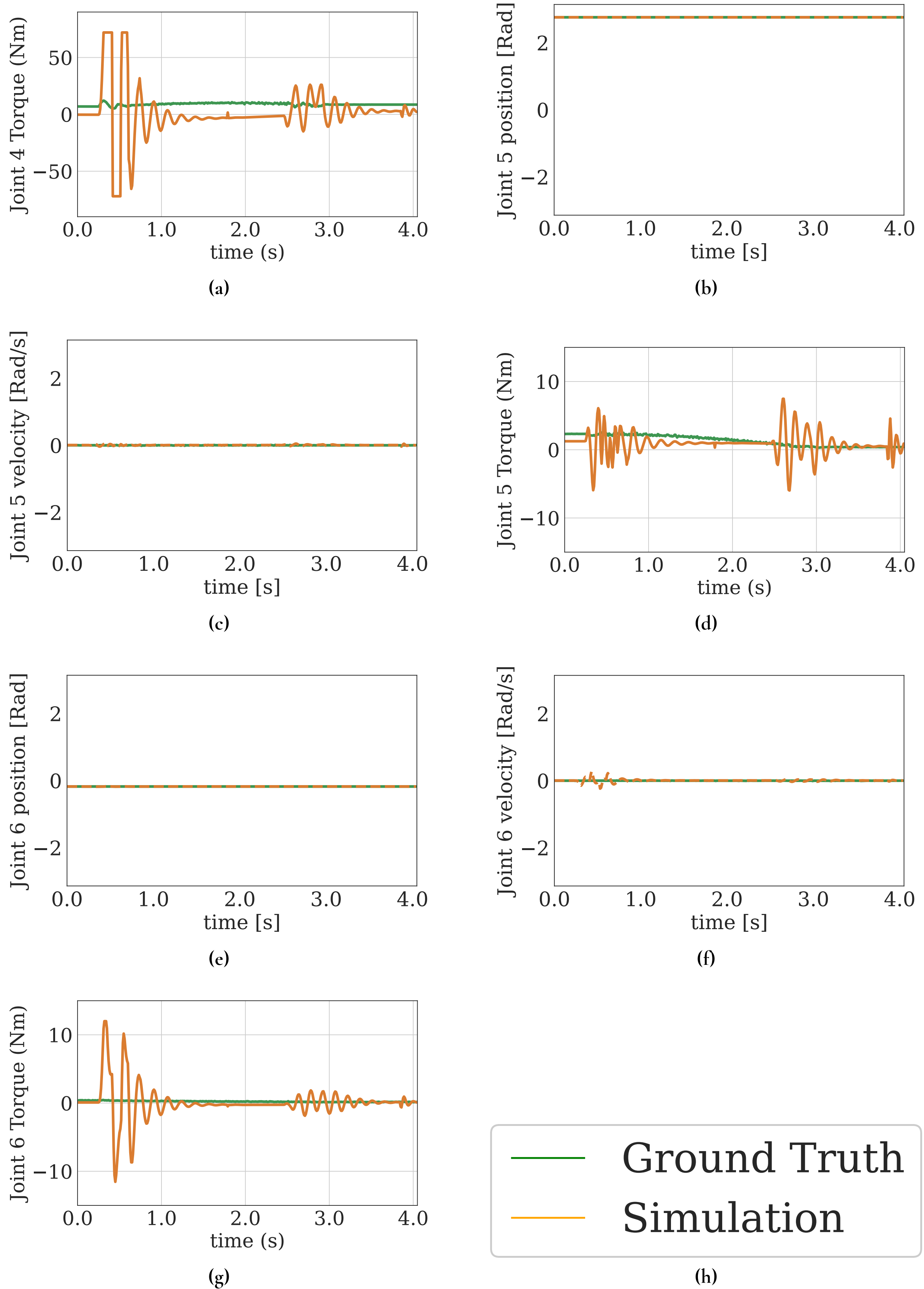

Figure 9: Position, velocity, and effort data for robot (green) and simulation (orange) for joints 4, 5, 6

Since the simulation of torque is oscillating more than the real robot, the trend of the simulation was analyzed by using the moving average over 0.5 seconds (50 timesteps) of the simulation:

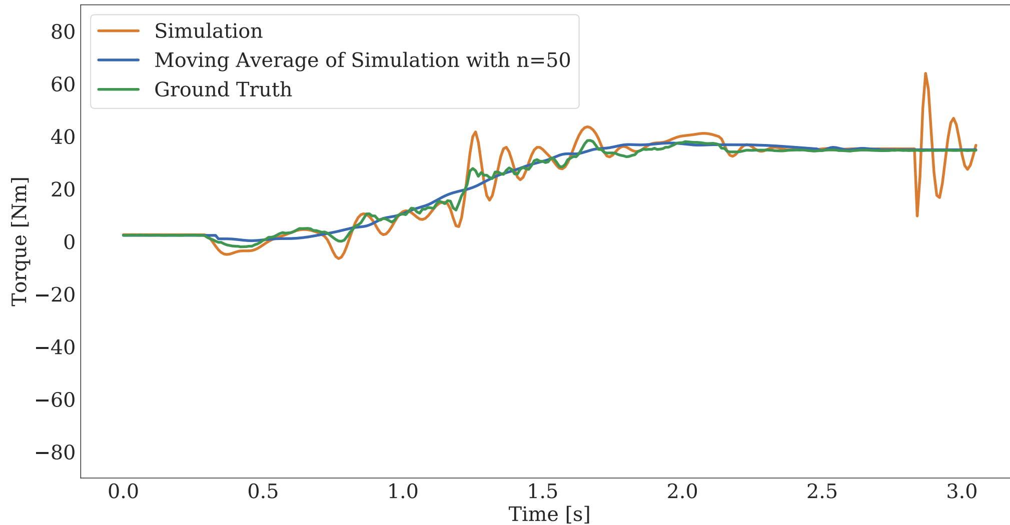

Figure 10: Moving average (blue) of simulated torque, for an arbitrary trajectory of joint 0

Applying this moving average filter to the simulation of all trajectories decreased the RMSE of torque between simulation and reality with **50.5%**. This filter was never used in the experiment.

6.2 Predictions with Pekel Simulation parameters

We plot the results from our trained model, in comparison of simulation and the robot data (ground truth) from an exemplary trajectory.

Figure 11 below shows how well the model can improve the torque simulation for some trajectories. It is shown how, for Joint 0, 1, 2 and 5, the model does not only improve the simulation accuracy, but also replicates the torque behavior of the robot to near perfect accuracy.

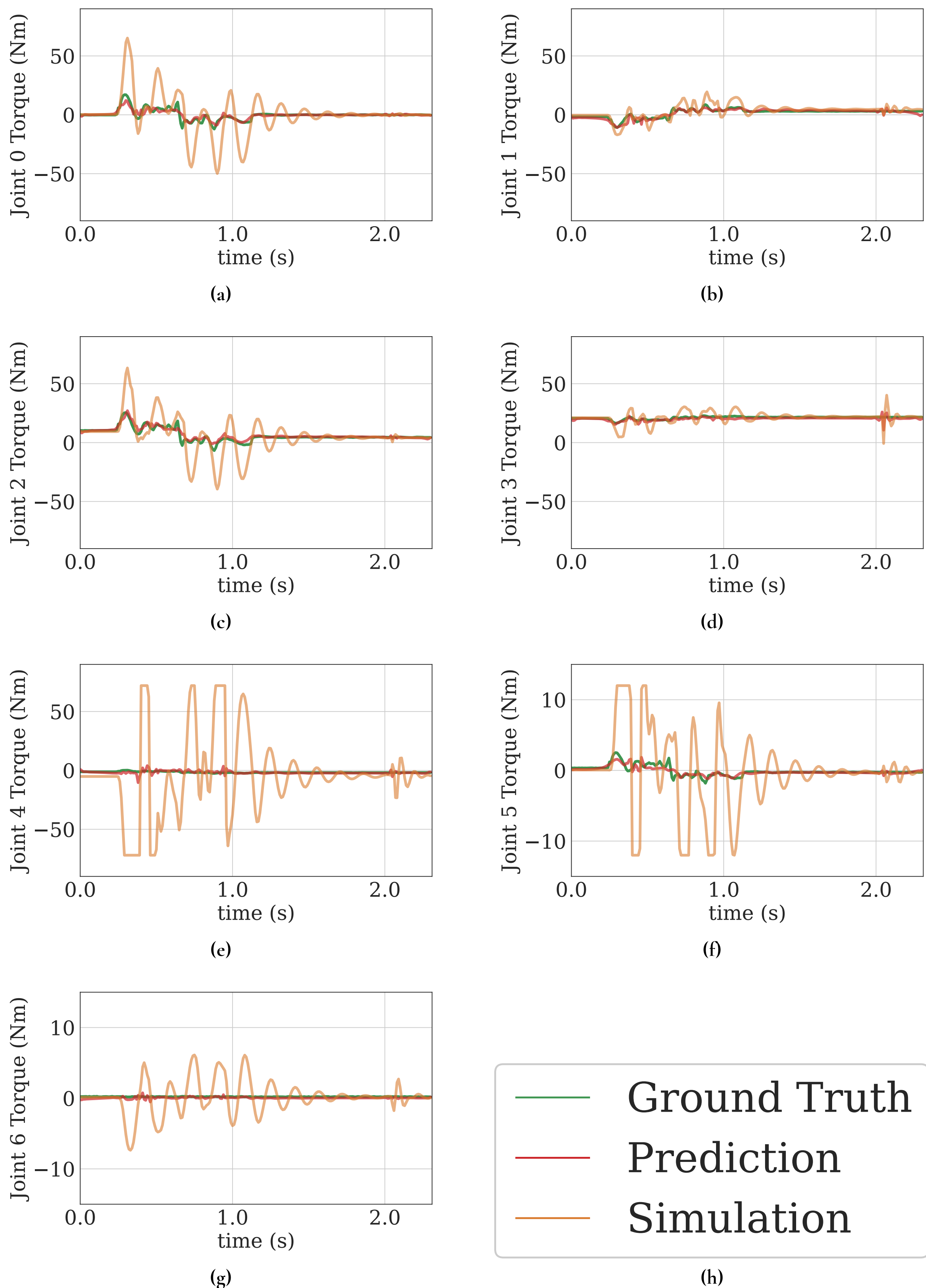

Figure 11: Torque data for robot (green) and simulation (orange) and model prediction (red) for a multi-joint trajectory

The complete performance of the model is shown in table 2 below, where the "Residual" model is the model trained with the residual loss function from section 3.3. It is clear from the table that the residual between simulation and reality is very small for position and velocity data confirming the findings of section 6.1. There is also no consistent improvement in position or velocity. For torque, in joint 0-3, the RMSE is reduced with a factor between 3 and 4, and for joint 4-6 it reduced with a factor of 10.

Table 2: Mean squared residual error in position, velocity and torque for the test data set with Pekel simulation parameters.

Joint		0	1	2	3	4	5	6
Position - Simulation	(rad)	4.63e-02	4.19e-02	2.20e-02	4.10e-02	6.80e-02	1.85e-02	5.23e-02
Position - Residual	(rad)	5.47e-02	3.50e-02	2.36e-02	3.16e-02	5.89e-02	2.85e-02	5.35e-02
Velocity - Simulation	(rad/s)	5.00e-02	6.04e-02	3.61e-02	6.93e-02	9.27e-02	4.81e-02	7.54e-02
Velocity - Residual	(rad/s)	6.15e-02	4.77e-02	3.63e-02	5.61e-02	8.12e-02	6.55e-02	7.39e-02
Torque - Simulation	(Nm)	5.65e+00	5.41e+00	4.61e+00	4.33e+00	3.20e+00	2.75e+00	2.26e+00
Torque - Residual	(Nm)	**1.17e+00**	**1.78e+00**	**1.42e+00**	**1.42e+00**	**3.17e-01**	**3.45e-01**	**2.41e-01**

To evaluate how the custom residual loss function performs in comparison to a normal mean square error one, the mean prediction error across all joints is plotted for both loss functions versus the amount of training data.

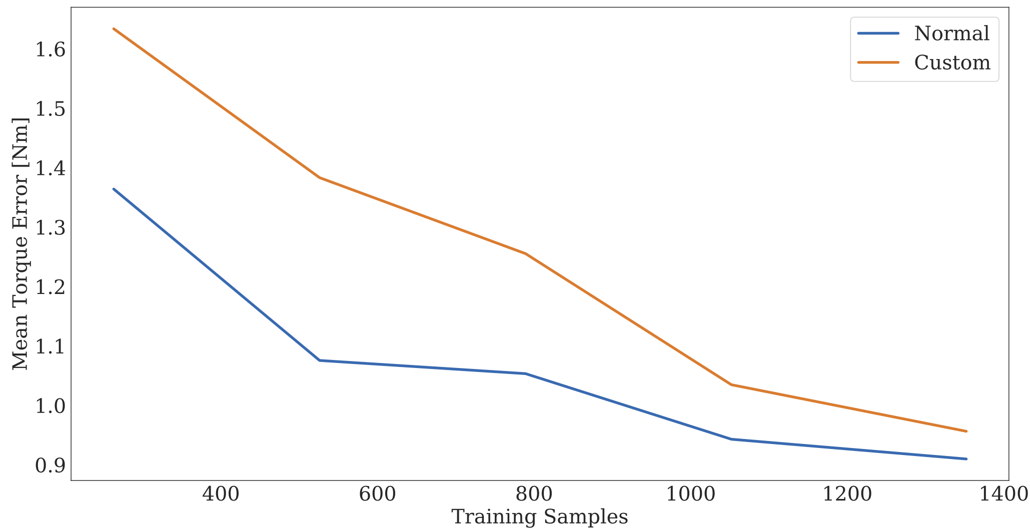

Figure 12: Test set prediction error for torque, mean across all joints: residual prediction (orange) with custom loss funcion, versus prediction with normal MSE loss function (blue)

It is evident that the normal loss function outperforms the custom residual loss function for all training sizes, with the largest difference in small training sets.

6.3 Predictions on Linear Dataset with Pekel simulation parameters

Figure 13 below shows the results of the model — that is trained on non-linear movements — applied to a linear trajectory. To satisfy the waypoints, the simulation is changing the torque substantially between small timesteps. Even here the model can be deployed effectively, showing a very accurate prediction of the robot data.

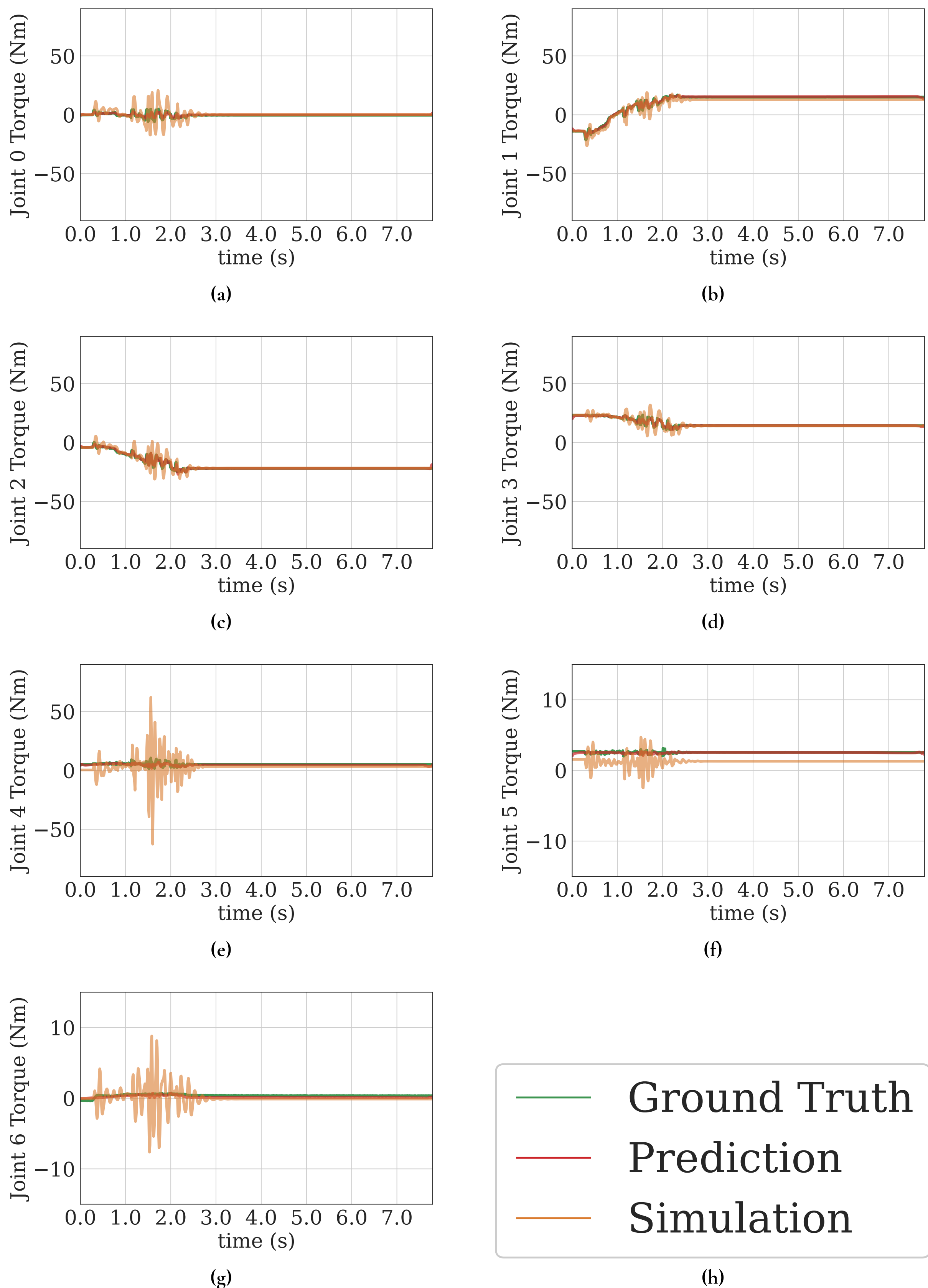

Figure 13: Torque data for robot (green) and simulation (orange) and model prediction (red) for a linear movement trajectory

The complete performance of the model on the linear dataset is shown in table 3 below. The results of the model with the mean square error loss function (Normal) is compared to the model with a custom residual loss function (Residual). Overall, there is no discernible difference between the results of the two models. The results show a reduction in RMSE of torque between a factor of 2-3 for joint 0-3, and at least a factor of 4 for joint 4-6 for both models.

Table 3: Root mean squared residual error in position, velocity and torque for linear data set

Joint		0	1	2	3	4	5	6
Position - Simulation	(rad)	3.46e-03	3.29e-03	3.63e-03	5.65e-03	4.57e-03	4.16e-03	4.05e-03
Position - Residual	(rad)	9.29e-03	8.40e-03	7.61e-03	1.32e-02	1.16e-02	1.56e-02	9.62e-03
Position - Normal	(rad)	1.73e-02	1.52e-02	2.19e-02	1.93e-02	1.73e-02	1.43e-02	1.51e-02
Velocity - Simulation	(rad/s)	2.10e-02	1.63e-02	2.19e-02	2.63e-02	2.15e-02	1.75e-02	3.25e-02
Velocity - Residual	(rad/s)	2.19e-02	1.64e-02	2.48e-02	2.98e-02	2.26e-02	1.88e-02	2.13e-02
Velocity - Normal	(rad/s)	2.31e-02	2.35e-02	2.89e-02	4.23e-02	3.25e-02	2.52e-02	2.80e-02
Torque - Simulation	(Nm)	3.52e+00	2.58e+00	2.74e+00	2.31e+00	1.46e+00	1.61e+00	1.29e+00
Torque - Residual	(Nm)	**8.51e-01**	**1.20e+00**	**9.77e-01**	**9.24e-01**	**2.22e-01**	**2.35e-01**	**3.30e-01**
Torque - Normal	(Nm)	**9.11e-01**	**1.14e+00**	**1.08e+00**	**1.07e+00**	**2.67e-01**	**2.54e-01**	**3.44e-01**

6.4 Predictions with Gaz simulation parameters

Figure 14 below shows a trajectory with the model trained on the suboptimal simulation with Gaz et al. [4] simulation parameters. From Joint 4 and 6 it is visible how the simulation is completely inaccurate, where the simulation is not oscillating around the correct value, but instead fails to predict the torque completely. It is also possible to see the model making very bold and flawed predictions, like at the 2.6 second mark of Joint 2.

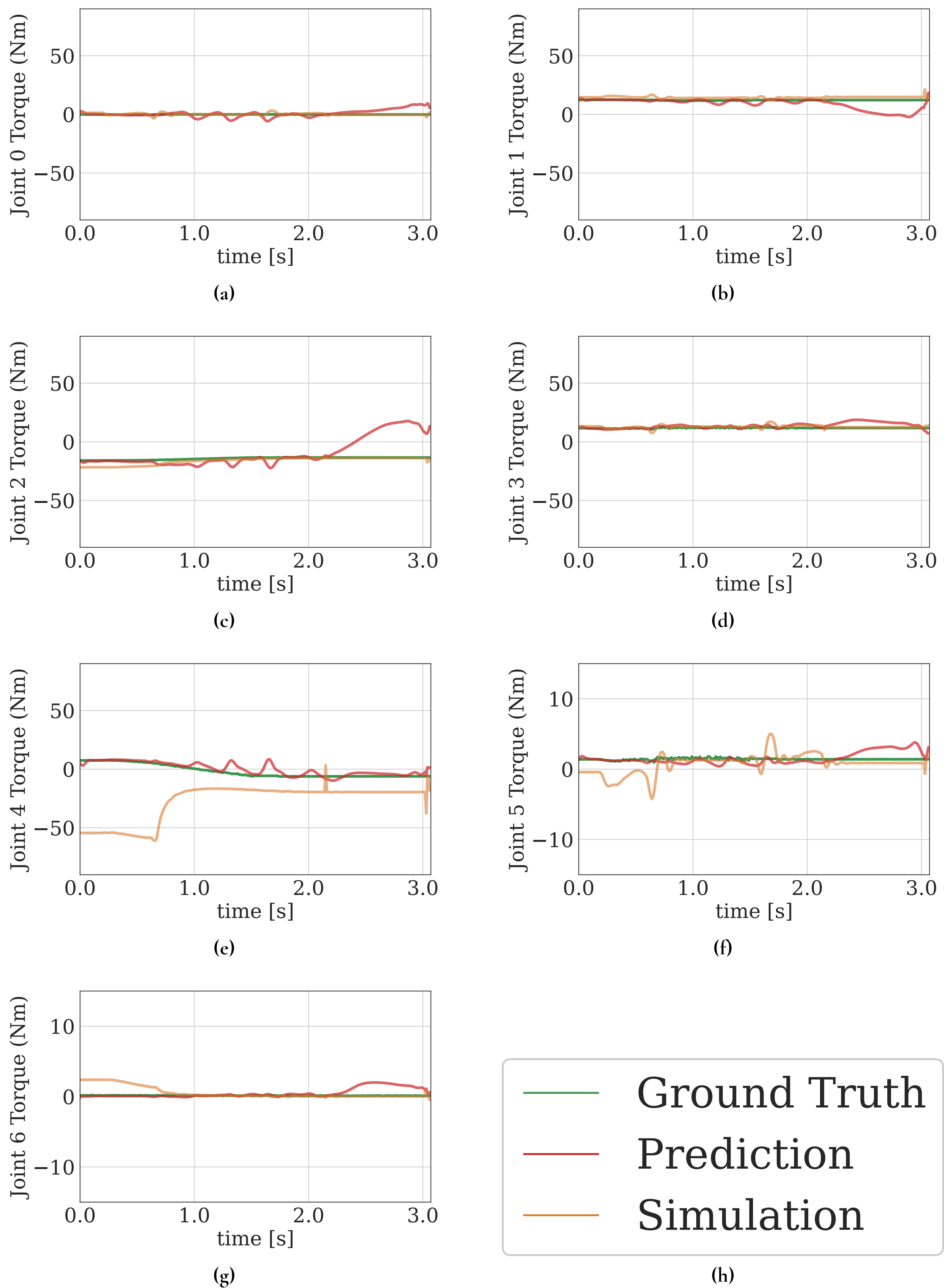

Figure 14: Torque data for robot (green) and simulation (orange) and model prediction (red) for a multi-joint trajectory

The complete performance of the model is shown in table 4 below, where the "Residual-2" model is the model trained with the residual loss function from section 3.3, with the Gaz et al. [4] simulation parameters. The residual between simulation and reality is still very small for position, but the velocity simulation is considerable worse, with an increase of RMSE with at least a factor of 10 across all joins compared to the Erdal simulation. The simulation of torque is also much worse, with a substantial increase in RMSE across all joints. The improvements from the model are also record breaking, where the final torque predictions are only slightly worse, even though the simulation is much worse, and the velocity RMSE is significantly reduced.

Table 4: Mean squared residual error in position, velocity and effort for the test data set with Gaz simulation parameters

Joint		0	1	2	3	4	5	6
Position - Simulation	(rad)	4.64e-02	4.16e-02	2.21e-02	4.06e-02	6.81e-02	1.89e-02	5.23e-02
Position - Residual-2	(rad)	4.97e-02	4.26e-02	2.45e-02	3.87e-02	5.25e-02	2.66e-02	5.14e-02
Velocity - Simulation	(rad/s)	1.05e+00	1.76e-01	1.16e+00	1.62e-01	4.45e-01	9.94e-02	2.25e-01
Velocity - Residual-2	(rad/s)	**7.66e-02**	**6.22e-02**	**8.73e-02**	**7.35e-02**	**1.03e-01**	**5.39e-02**	**7.54e-02**
Torque - Simulation	(Nm)	4.40e+01	1.93e+01	4.46e+01	1.17e+01	7.91e+00	4.37e+00	4.62e+00
Torque - Residual-2	(Nm)	**1.44e+01**	**2.33e+00**	**1.69e+00**	**1.84e+00**	**3.54e-01**	**3.64e-01**	**2.61e-01**

7 Discussion

In this thesis, our focus was not on improving the execution of a specific robotic task, instead the objective was the general improvement of simulation for arbitrary trajectories in the robot's workspace. Since there was no task in mind, we introduced planning limitations such as velocity and acceleration bounds and amount of joints allowed, to keep the complexities of the dataset low and be able to evaluate the potential of our approach. For the same reason we chose not to include trajectories where the Panda robot is grasping an object. Errors during the execution of trajectories were not uncommon, and we chose to identify and delete those trajectories during the preprocessing steps. A better solution might have been to delete the recording every time an error is published in the ROS framework.

With the Erdal simulation parameters, the simulation is almost identical to the real robot for position and velocity. Sometimes discrepancies occur like in the velocity data for joint 4 in figure 8 around the 0.5s mark. This usually lasts for a fraction of a second before the simulation returns to being accurate. These disturbances seem to occur quite sporadically and would be hard for a neural network to predict. Hence, it is not surprising that none of the models could improve the results. For torque, the simulation generally seems to oscillate more, even when the real torque is kept constant (like for Joint 4, 5 and 6 of figure 9). The moving average of every 50 timesteps of simulation helps to filter sensor noise and disregards the oscillations, only focusing on the trend. Figure 10 illustrates how the trend can be a much better prediction. Since just the moving average of simulation reduced the simulated RMSE in torque by 50%, this should at minimum be used as an intuitive benchmark. A model that can not outperform this reduction would not be useful. However, it is clear from table 2 that the model outperforms the moving average of the simulation. For joint 0-3 the RMSE is reduced with a factor between 3 and 4. For joint 4-6 it was reduced with a factor of 10.

One reason for the difference in improvement between the joints can be found in the configuration of the robot. Figure 2 shows how the joints are numbered from bottom to top. Joint 6 and 5 only control the "head" of the robot, therefore they are not very dependant on the robot configuration. The torque of joint 1, 2, 3 and 4 on the other hand will depend more on the robot configuration since it relates to the amount of torque needed to counter the force that is exerted on the robot arm by gravity. Joint 0 rotates the whole robot, where the robot inertia plays a key role, which is determined by the configuration. The results from the table confirm that the robot configuration plays a vital part in predicting torque, and adding more executed trajectories to the dataset in various robot configurations could improve the model performance further.

As earlier stated in the introduction, we applied a version of residual learning where our hybrid model made predictions on top of the simulation results. Earlier work [3] [1] has had a similar approach, where a residual velocity was calculated. Contrary to earlier work, we did not observe that our residual model generalized better or trained with less data than our normal model. In figure 12 it is shown how the model with the residual loss function performs much worse with little training data than the normal model. The generalization of the two models on a linear dataset shown in table 3 does not indicate any significant difference in performance between them. The difference in findings could be explained by the difference of approaches. We do not use our simulation

to calculate the minimum torque required for an action, and then use our LSTM architecture to predict how much residual torque to add to counter factors such as friction. Instead our hybrid model learnt all the shortcomings of the simulation and improved upon it. Considering the high dimensionality of the data which consists of 7 joints in 21 dimensions in hundreds of timesteps, it is possible that the Residual model is punished for making predictions that deviate too much from the simulation, therefore getting stuck in local minima close to the simulation. The normal model on the other hand, is forced to learn the mapping between simulation and reality from scratch, and since the model has deep hidden layers and a relatively small dataset it is able to do so efficiently.

The Gaz et al. [4] simulation parameters performed much worse than Pekel's [18]. Comparing the good simulation in table 2 with the bad one in table 4, the velocity simulation RMSE is higher by a magnitude between one and two for all joints. The torque RMSE is also higher with a factor between 2 and 10, where joints 0-3 have the highest increase in RMSE. In figure 14, the plot of the torque in joint 4 shows how the simulation is predicting completely wrong torque during a slow trajectory. The predictions of the model in this figure also seem to converge to different values, indicating that the simulation parameters are simply wrong: forcing the model to make awkward adjustments. Nevertheless, the Residual-2 is making accurate predictions, even with inadequate simulation. The RMSE of the predictions of torque in joint 1-6 in table 4 are just 10% to 30% worse than the predictions with the Pekel simulation. In joint 0 the Gaz simulation is extremely inaccurate, which could explain why the Residual-2 model prediction of that joint is also inaccurate.

Because the Gaz simulation of velocity is inaccurate, there is more room for improvement. The Residual-2 model made much better predictions, proving that it is also capable of improving simulation of velocity. With the model using the Pekel simulation, the model loss was dominated by torque input, making the model prioritize improvements in torque instead of improving position and velocity data.

The more popular alternative to our approach of domain adaptation with machine learning is to identify dynamic parameters and create a mathematical model describing the robot dynamics, which was done by Gaz et al. [4], also for a Franka Emika Panda robot. In their paper, they did not estimate the RMSE between their model prediction and their measured data for a dataset like we did, therefore it is hard to evaluate the difference in results between the two methods. However, the results they do show for torque seems very promising, although their approach is more focused on a single trajectory. Visually, our model seems to show equal performance for some trajectories like in figure 11.

There is nothing limiting the use of a joint approach of both system identification and our approach of domain adaptation. System identification can first be used to retrieve a better mathematical model for the simulation, and then domain adaptation can applied to compensate for the flaws of the mathematical model as well as model complex real world factors that are not modeled. This approach could bring robustness from the analytical system identification approach and flexibility from the data driven domain adaptation, bringing the best of both worlds.

8 Conclusion and Future Work

We have developed a pipeline for executing trajectories, recording them, and processing the recorded data, for both the physical and the simulated Franka Emika Panda robot; with the objective to develop a robust and easy approach to bridge the gap between simulation and reality, and prove that it works for simple free movements of the panda robot. Our approach relies on several factors that may not be existent in every robot application. Sensors that accurately measure position, velocity and torque does not exist in every robot. Different robots may have more nonlinear behavior originating from factors such as heat development in joints, cable stretch or the controller system.

We found a set of simulation parameters that produced adequate results for position and velocity data, and hence focused on improving the suboptimal simulation of torque. We proposed an LSTM architecture suitable for the task, and have shown how our model, when deployed after simulation, can produce an improved simulation where the RMSE is on average reduced by a minimum of 70% across all joints. We demonstrate robustness of our model by deploying it on a dataset of linear movements and observing a consistent decrease of torque RMSE across all joints. Relating this to the research question in section 1.2, our model does manage to find a sim-to-real transformation that is generally more reliable and in some cases substantially better. To really grasp the viability of our approach, a more quantitative comparison to system identification is in order, where both methods are applied to the same robot individual, and evaluated for the same trajectories.

Moreover, we evaluate a residual loss function for our model. Contrary to the findings of earlier work, we found that the model needed more data and did not perform any better with the residual loss function.

By repeating our approach with more ill-fitting simulation parameters we show that our approach does not depend on an accurate simulation. Our model was also shown to be able to improve not only simulation of torque but also that of velocity.

In our approach, we did not attempt to retrieve dynamic parameters of our own Panda robot individual, but used parameters found from other robot researchers [18] [4]. By retrieving the specific parameters, the simulation of torque could potentially be improved drastically, which would also improve the predictions made by our approach. It would be valuable to evaluate how well our approach works in combination with system identification, to understand the potential and eventual limitations.

It would also be useful to research how the behavior of the robot changes with a grasped object, and how well our approach works for the resulting trajectories. Gathering more data — 10.000 trajectories or more — can help assess the potential in our proposed LSTM architecture. Controller errors were not uncommon during the execution of plans, and developing a framework for detecting plans that cause errors could be helpful for future work. Finally, trying our proposed approach on different types of robots would help evaluate the viability of our method.

References

[1] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *arXiv preprint arXiv:1903.11239*, 2019.

[2] Anurag Ajay, Jiajun Wu, Nima Fazeli, Maria Bauza, Leslie P Kaelbling, Joshua B Tenenbaum, and Alberto Rodriguez. Augmenting physical simulators with stochastic neural networks: Case study of planar pushing and bouncing. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3066–3073. IEEE, 2018.

[3] Alina Kloss, Stefan Schaal, and Jeannette Bohg. Combining learned and analytical models for predicting action effects from sensory data, 2018.

[4] Claudio Gaz, Marco Cognetti, Alexander Oliva, Paolo Robuffo Giordano, and Alessandro De Luca. Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, 4(4):4147–4154, 2019.

[5] S LAVALLE. Rapidly-exploring random trees: a new tool for path planning. *Research Report 9811*, 1998.

[6] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001. IEEE, 2000.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*.

[8] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[10] S Aparna. Long short term memory and rolling window technique for modeling power demand prediction. In *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 1675–1678. IEEE, 2018.

[11] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[12] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[13] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[14] Rabiner LR and RW Schafer. Digital processing of speech signals, 1978.

[15] Maxime Gautier and Ph Poignet. Extended kalman filtering and weighted least squares dynamic identification of robot. *Control Engineering Practice*, 9(12):1361–1372, 2001.

[16] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[17] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

[18] Munich Technical University Erdal Pekel, PhD candidate in Robotic Imaging. Integrating franka emika panda robot into gazebo. `https://erdalpekel.de/?p=55`, 2020.

[19] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. URL `https://www.ros.org`. Accessed on 18/05-2020.

[20] Sachin Chitta, Ioan Sucan, and Steve Cousins. Moveit![ros topics]. *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT*, 19:18–19, 03 2012. doi: 10.1109/MRA.2011.2181749.

EXAMENSARBETE Sim-To-Real: Domain Adaptation of Robot Trajectories with LSTM
STUDENT Liam Neric
HANDLEDARE Elin Anna Topp, Alexander Durr (LTH)
EXAMINATOR Volker Krueger (LTH)

Från simulering till verklighet – en robotarms resa

POPULÄRVETENSKAPLIG SAMMANFATTNING **Liam Neric**

Datadriven modellering av en robotarms rörelse med syftet att öka effektiviteten i simulerade experiment. Arbetet kan komma att effektivisera utvecklingen av autonoma robotar.

Vem skulle inte vilja äga en intelligent robotarm som kan hjälpa till med matlagning, eller kanske byta kanal på tv:n? Innan detta är möjligt behöver robotens mjukvara tränas i tusentals, om inte miljontals experiment för att lära sig analysera robotarmens misstag och utföra kommandon utifrån olika omständigheter. Dessa experiment kan påskyndas genom att utföra dem på en simulerad version av roboten, där ett kommando som kanske tar tjugo sekunder i verkligheten kan ta en halv sekund i simuleringen.

Simuleringen består av en fysikmotor med en virtuell matematisk modell av robotarmen. Dessa modeller har svårt för att fånga komplexiteter som finns i verkligheten, som bristfällig robotstyrning eller värmeutveckling i robotens leder. Om simulering ger alltför annorlunda resultat blir det svårt att relatera lärdomar från simuleringen till verkligheten. Därför är det viktigt att hitta metoder som kan minska gapet mellan simulering och verklighet och därmed öka användbarheten hos virtuella experiment. Det finns främst två olika sätt att utföra detta, antingen kan metoder utvecklas för att direkt förbättra simuleringen, eller så kan metoder hittas som kan uppskatta verkligheten utifrån simuleringsdatan; där en uppgraderad ”syntetisk” version av simuleringen skapas vid en separat plattform.

I detta arbete har fokus lagts på det senare alternativet, där syftet varit att hitta en allmän metod som kan förbättra simuleringsdata av rörelsebanor för en robotarm. Detta medförde en analys av robotarmens alla leder i position, hastighet och vridmoment; för både simulering och verklighet i olika fria rörelser. Målet var också att bevisa konceptet av att en datadriven modell kan lära sig olika mönster i en robotarms rörelser, vilket kan ha stor betydelse i framtida applikationer.

Experimentet gick ut på att samla in data från cirka 2000 olika rörelsebanor som både den verkliga och virtuella robotarmen utförde. Utifrån dessa rörelsebanor utvecklades en maskininlärningsmodell som använde simuleringsdata och producerade syntetisk data. Resultaten visade att den syntetiska datan var betydligt bättre, med en genomsnittlig reduktion av 80% av simuleringens fel. Detta reflekterar hög potential hos den utvecklade metoden då det påvisar att maskininlärningsmodellen är kapabel till att lära sig verkliga rörelsebanor utifrån simulering.

Den utvecklade modellen kräver ingen finjustering av parametrar eller kalibrering till en specifik robot. Framtida arbete kan bestå av att testa variationer av vår metod i olika robottillämpningar för att få en bättre bild av dess potential.