Projektpraktikum Tactile Internet LMT
Networked Control System of a 2-DoF stabilizer using an IMU sensor

David Gastager
Technical University of Munich
david.gastager@tum.de

Yiming Wei
Technical University of Munich
yiming.wei@tum.de

Abstract

In this project we aim to control a 2-DoF actuator using an IMU sensor. In a non-ideal case, there would be delays in the networked control system. Our goal is to predict the motion of the actuator to compensate for such possible delays. Since our processors are connected through ethernet, the setup can be considered without any delay initially. To investigate the performance of our delay compensation algorithms, we manually introduced two kinds of delays: a fixed delay and an intervalic delay. Recurrent neural network (RNN) are utilized for motion prediction. We compared three model structures using different combinations of GRU, LSTM, convolutional and dense layers. After training the models, we ported them to the used embedded devices using Tensorflow Lite to realize live prediction. The three models demonstrated good performance in the validation and test datasets. The predicted values can follow the general trend of the ground truth signal. In live prediction with a fixed delay, the models didnt perform well, as they were not stable enough. In the compensation of intervalic delays, we get reasonable results in live prediction, where the general trend is predicted correctly, but the details are off.

1 Introduction

1.1 Problem Statement

In this project, a 2-DoF stabilizer is controlled using an IMU sensor over a network connection. The setup acts as a game, where the player is supposed to balance a ball on a remote controlled platform, using the IMU sensor as her/his remote control. In a real world scenario however, there might be an introduction of inevitable delays in the networked control system. Our task was to predict the motion of the plate to keep the ball balancing on it and to compensate for the delay in the system.

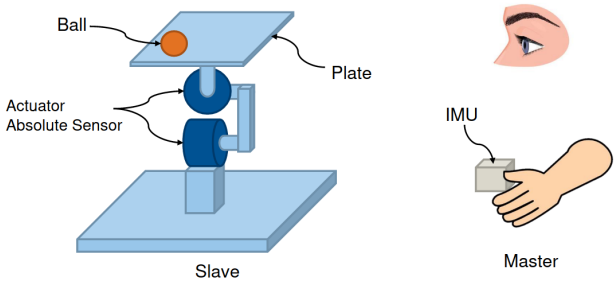

Figure 1: Studied telepresence system, where the user balances a ball by remote controlling the actuator with an IMU sensor.

A general telepresence system comprises a local side and a remote side, connected through a communication channel. In our system setup, such as depicted in Figure 1, the local side is equipped with an IMU sensor which collects the roll and pitch information from the user. The remote side is a 2-DoF stabilizer, consisting of two servo motors. A computer is used as a control station which is implemented with a simple roll, pitch controller of the joints based on sensor data. The three parts are connected with Ethernet and communicate over a ROS-based framework.

The idea of compensating for the delay is to predict the motion of the actuator. In this project, we utilized a RNN (Recurrent Neural Network) to achieve time series forecasting. However, normal RNNs have limitations through vanishing and exploding gradients. To overcome this problem, we utilize LSTM and GRUs. Both of these make use of gate units, so that they can deliberately remember and forget some historical information. The GRU provides more computational efficiency, as it has fewer gates and thus fewer parameters and states than LSTM units. We investigated models of three structures: GRUs with convolutional layers, interleaved LSTM with dense layers, and a simple LSTM model. To investigate the network delay compensation, we manually adjusted the sending time of IMU sensor data to introduce some delay in the network. In order to enable online predictions, we

integrated the trained models on the microprocessor with Tensorflow Lite.

1.2 Paper Outlook

The rest of the report is organized as follows: Section 2 gives an overview of our system setup. Section 3 presents the data preparation. In Section 4 we focus on different RNN models investigated and training results. In Section 5 we transplant the models to Nano Pis with Tensorflow Lite and introduce delay in the system. Section 6 illuminates the evaluation of different models and prediction performance and Section 7 summarizes our conclusions and discusses future work.

2 System Setup

2.1 Hard- and Software packages

The used hardware in this project consisted mainly of three computers, an IMU sensor and two servo motors. Two NanoPi Neo3s were used as a sensor controller and an actuator controller respectively. One desktop computer was used to run the ros master node and a simple controller. Everything was directly connected via ethernet through a network switch.

The main software package used for communication between the devices was ROS Melodic, based on Ubuntu Core 18.04. The controller code in the ROS scripts and the network models is written in Python. The machine learning models were all implemented using the tensorflow keras library and converted to tflite models using tensorflow lite.

2.2 ROS Setup

Figure 2 shows the hardware setup and data flow between Nano Pis and computer. Initially, there are three nodes in ROS as depicted in Figure 3. The ImuDriveNode runs in the sensor controller Nano Pi, publishing the IMU sensor message to ControlNode which runs in the computer. The simple pan/tilt controller in the ControlNode derives the corresponding roll and pitch angles and sends them to the actuator controller node dxlDriver.

As shown in Figure 4, to artificially introduce some delay in the network, we modified ImuDriveNode to put off the time of publishing the sensor data. We added a predictor node in the actuator controller Nano Pi. This node subscribes to the joint goal topic from ControlNode and publishes a received value or predicted value depending on the delay. The details will be described in section 5.

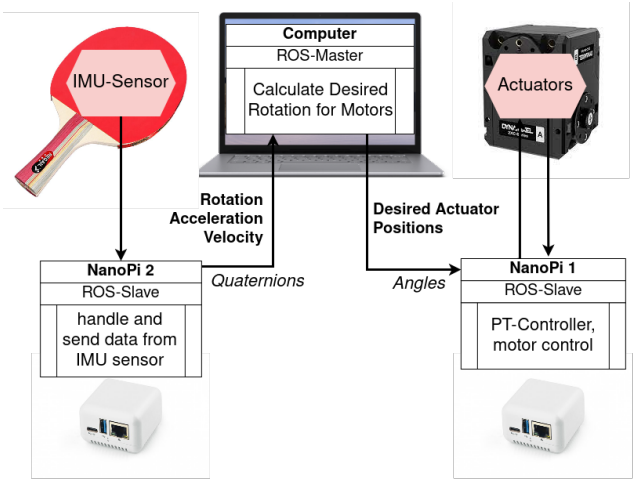

Figure 2: Hardware setup and data flow in ROS

Figure 3: Topic and node connection in ROS

3 Data

In order to analyze the system and build a machine-learning classifer, a dataset had to be recorded. The following sections provide an overview of how we collected, processed, and analyzed the data.

3.1 Data Collection

As our main objective was to counteract transmission and network delays, we needed to record the system in its best possible state with minimum delay as a baseline. We were specifically interested in the goal positions of the motors, as these were the values our models would need to predict. All processors were directly connected through a wired ethernet connection and the system was started with only the essential programs running on each device. We used the record functionality of the ROSBAG library and recorded all published messages in the available topics. A total of around 60 minutes of game-play were recorded. It was played by two different people and included a variety of playing styles, e.g. slow and careful balancing, fast movements, etc. This was done to ensure a larger variety of data and to cover more possible scenarios, from which the model could learn. The dataset was cleaned afterwards and only signals where the ball was actually balanced on the platform, were kept.

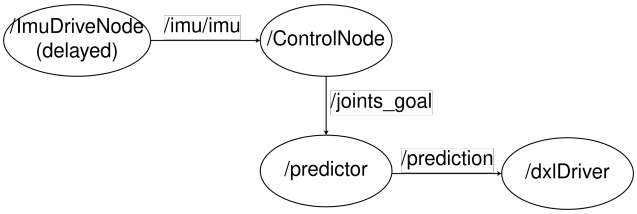

Figure 4: Topic and node connection in ROS, where delay and predictor are added.

3.2 Data Analysis and Processing

The timesteps between the signal values were not uniform and the publishing of the messages was not synchronized. Therefore we decided to interpolate the dataset to obtain uniform timesteps and synchronization between all signals. Because the IMU and the Pt controller published data at a frequency between $110Hz$ and $120Hz$, we decided to interpolate at a fixed frequency of $120Hz$. A variety of interpolation methods was tested and a simple linear interpolation was chosen, because it resulted in the smallest reconstruction error.

To get a better understanding of the dataset, its frequency components were analyzed. The power spectrum of the pitch-goal-positions over time is depicted in Figure 5. It indicates that almost all of the energy is encapsulated in the lower frequency bands. Figure 6 shows a plot of the average energy of $1Hz$ sub bands from $0Hz$ to $60Hz$ (Nyquist limit at $120Hz$), normalized by the total energy of the signal. As can be seen visually from the graph, the first frequency bands already carry the most significant amount of energy. At $5Hz$ already 98.24% of the average energy is included.

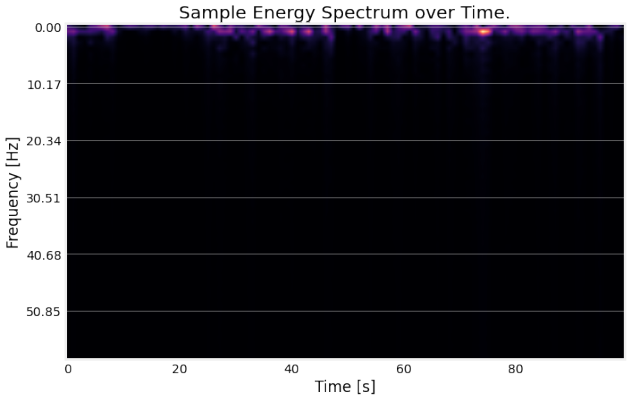

Figure 5: Energy Spectrum of pitch values over time. Bright corresponds to high energy, dark means low energy. Most energy is found in the low frequency bands.

In order to reduce computational complexity and to not feed the machine learning models with oversampled signals containing no valuable information in the higher frequency bands, all signals were filtered with a fifth order butterworth low-pass filter at a cutoff frequency of $5Hz$ and subsequently downsampled to a sample rate of $10Hz$.

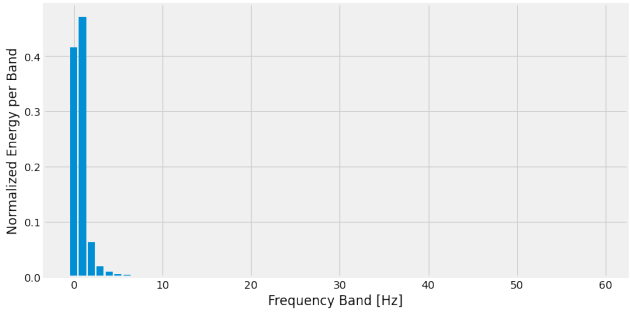

Figure 6: Normalized Average energy per frequency band.

As a last step, the amplitudes of the goal positions were investigated. While a light and careful playing style usually resulted in maximum amplitudes of ± 0.1 ($\approx \pm 6°$), a faster and more hectic one resulted in amplitudes of ± 0.2 ($\approx \pm 12°$). In order to make the model training more robust, all signals were scaled from the original range of $[-0.2, 0.2]$ to $[0, 1]$ using the following formula:

$$f(x[n]) = \frac{(x[n] - lowerbound)}{(upperbound - lowerbound)} = \frac{x[n] + 0.2}{0.4} \tag{1}$$

3.3 Data Preparation

In order to prepare the dataset for training a machine learning model, it was split into a training, validation, and a test set at a ratio of roughly 8:1:1. Because the utilized *tflite_runtime* library, which was used for inference on the nanopi, was not able to run models with dynamically sized input and output sequences, our model needed to have fixed input and output sizes. Inspired by [2] and [1], an input interval of three seconds (= 30 samples) and an output interval of one second (= 10 samples) was chosen.

The dataset splits were then windowed with a stride of one to create four second sub windows, consisting of three seconds of input signals and one second of target signals. The windowing was specifically done after the dataset was split to avoid dataleakage.

The final dataset splits consisted of $20,895$ samples in the training set, $2,338$ samples in the validation set and $2,298$ samples in the test set.

4 Signal Prediction

4.1 Machine Learning Models

Because our input is time series data, we decided to build Recurrent Neural Network models (RNNs). RNNs are a popular choice for time series prediction, as they are equipped with memory cells and can learn and use past information for their predictions. The actual model architectures were variations of models from three different papers published by the LMT. All models were implemented using the Tensorflow Keras library.

The architectures and sizes of all models were deliberately kept very different. First, to see how well different architectures can learn the underlying distribution. And second, to compare the models inference speed on the NanoPi 3. As we wanted the models predictions to be only dependent on the input window, all RNN layers were set to be stateless, meaning that the input state resets after every prediction.

4.1.1 Model 1: GRU, Conv

The first model was based on the proposed architecture in [2]. Figure 7 shows our variation of the model. The input signals are each fed through a 1D-Convolutional layer, followed by repeating structures of GRU and 1D-Convolutional layers. This is repeated five times, after which the output is pooled using a 1D-Max-Pooling layer. After an additional Dense Layer, the output is finally reshaped to obtain the (10 x 2) target dimensions. All GRU layers were followed by a Dropout layer to prevent overfitting during training. The model ended up with $89,780$ trainable parameters.

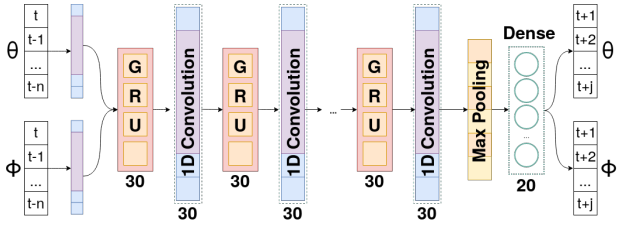

Figure 7: Model 1. A variation of the model proposed in [2].

4.1.2 Model 2: Interleaved Dense + LSTM

The second architecture was based on model *a)*, proposed in [1]. Just like in model 1, the inputs are each fed through 1D-Convolutional layers, which act as a kind of low pass filter to reduce noise [2]. This is followed by an interleaved structure of LSTM and Dense layers, which allows for an increase in the number of weights for an improved learning behaviour, while still maintaining the memorization characteristics of the LSTM layers [1]. All LSTM layers were followed by a Dropout layer to reduce overfitting. The final model had $20,315$ weights in total. Figure 8 shows the model architecture.

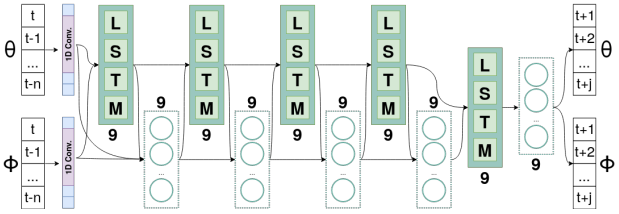

Figure 8: Model 2: A variation of the model proposed in [1].

4.1.3 Model 3: Simple LSTM

The third model was based on model *a)* from [3]. The inputs signals are fed through two LSTM layers, a Dense layer and are finally reshaped to obtain the desired output. After each LSTM layer, there is a dropout layer with drop rate 15% to avoid overfitting. With $11,212$ trainable parameters, the model was the smallest of the three. Its architecture can be seen in Figure 9.

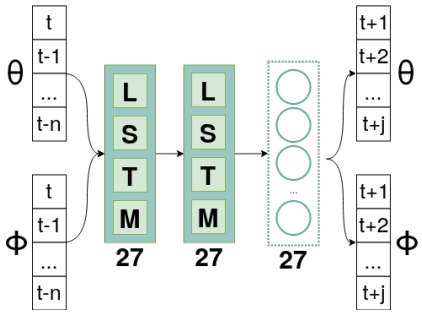

Figure 9: Model 3: A variation of model a) proposed in [3].

4.2 Model Training

After the models were built, a training pipeline was set up. The models were trained for 1000 epochs, using an earlystopper with patience 25 to prevent overfitting. The Adam optimizer was used with a batch size of 64, starting at a learning rate of 10^{-3}, decaying by 4% every 3000 batches. Mean absolute error (MAE) was chosen as the loss function. Figure 10 shows the training and validation losses of the three models over the training epochs. In the first 20 epochs all three models had significant drops in training and validation losses and all models ended up at MAEs of around 0.05. Even though the epochs were set to 1000,

none of the models trained this long, because of the used earlystopper.

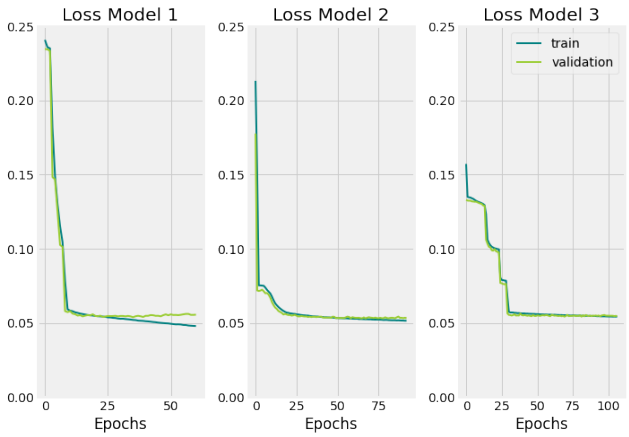

Figure 10: Training and Validation losses over the training epochs of all three models. All models were able to learn from the provided data and ended up at losses around 0.005,

4.3 TFLite Conversion

Inference using the trained tensorflow models is very computationally costly. In order to run them on the NanoPi devices, they needed to be transformed to a faster and lighter version. This was achieved by converting the model to a tensorflow light (TFlite) model, using the Tensorflow library. During the conversion, a series of quantizations and pruning is done to shrink the size of the model and to reduce the amount of necessary computations. This usually comes at the cost of flexibility, as tflite models can only run inference on fixed batch sizes. This was no problem in our case, as our system was making individual predictions anyhow. The loss of accuracy from the weight and input quantization was not marginal and not noticable in practice.

5 System Integration

After the models were trained and converted, their predictions had to be integrated into the live system. The goal was to counteract all networks delays and therefore the models were deployed on the receiving computer in the network, the motor controller.

A new *Predicter* node was created and put in between the *Control* node and the *Motor Driver* node, as shown in Figure 11. The *Predicter* was set up to receive all the positional data from the *Control* node with their respective timestamps and decide whether to forward the received values or the predicted values to the *Motor Driver*. Note that every measurement always came with its creation timestamp, meaning that even if a signals was sent with a delay, the receiver

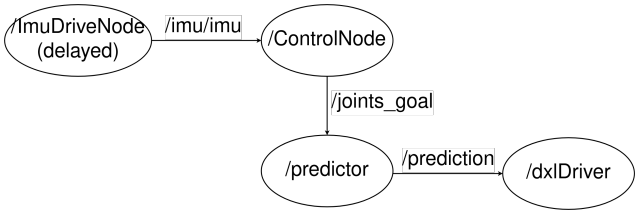

Figure 11: Node network in ROS. The IMU sensor node publishes its measurements, which are transformed by the control node and forwarded to the predicter node. The predicter decides wheter to directly forward the message or an inferred prediction to the motor controller.

knew when the signal should have arrived. Two different algorithms for embedding the models into real time use were implemented. Their main difference lies in the frequency of the calculation of the predictions. Algorithm one was designed to compensate permanent system delays, by calculating a new prediction every 25ms. Algorithm 2 always used a full prediction, once a delay was measured and was therefore designed to make up for spontaneous short delays.

5.1 Algorithm 1

The flow diagrm in Figure 12 shows the first implemented algorithm. There are three main loops running in parallel on the device. The *Subscriber* loop is responsible for receiving all incoming messages and saving them to a queue which holds the last three seconds of incoming messages. As it is receiving data from the *Control* node, its frequency varies between 100 and 120Hz. The *Prediction* loop constantly gets the values from the *Subscriber*s queue and uses them for a prediction. It runs the same processing pipeline used in the data preprocessing, explained in section 3.2. The signals are interpolated, filtered, downsampled and rescaled, before they are fed to the model for inference. The created prediction array contains the predictions starting time, as well as values for 10 timesteps, each being 100ms apart. The *Prediction* loop is set up to be executed every 25ms (=40Hz). This ensures that there is always a recent prediction used for every published value. The *Publisher* loop is responsible for deciding which values to send to the motor controller. If the last received message is less than 100ms old, it is published to the controller. If the message is older than 100ms, the publisher accesses the prediction array. It interpolates between the values of the predicted timesteps in order to obtain a value for the exact current time. The *Publisher* loop runs in fixed intervals of 25ms (=40Hz), which was empirically found to be a frequency where the game is still smoothly playable.

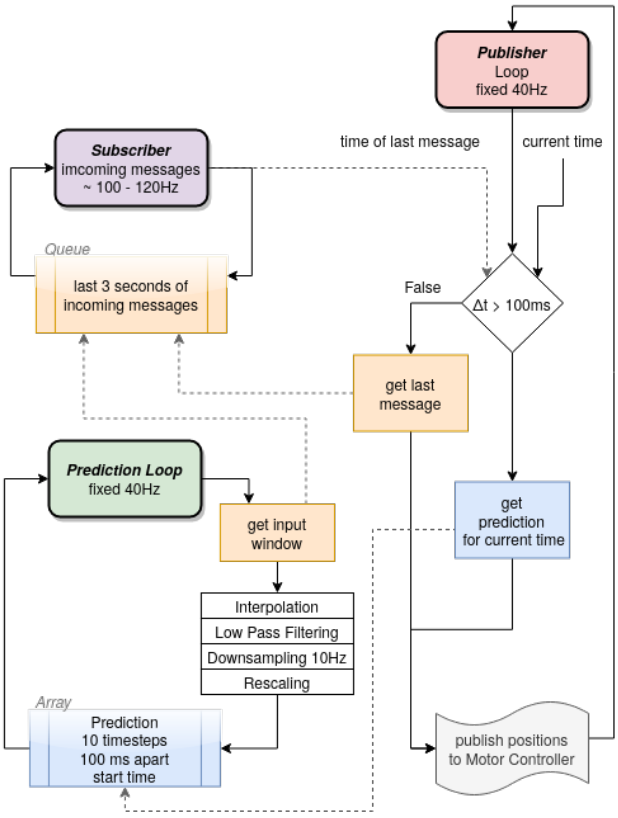

Figure 12: Flow-diagram of the first implemented algorithm running on the *Predicter* node. Three loops are running in parallel to permanently receive data, create predictions, and decide on the positional values to forward.

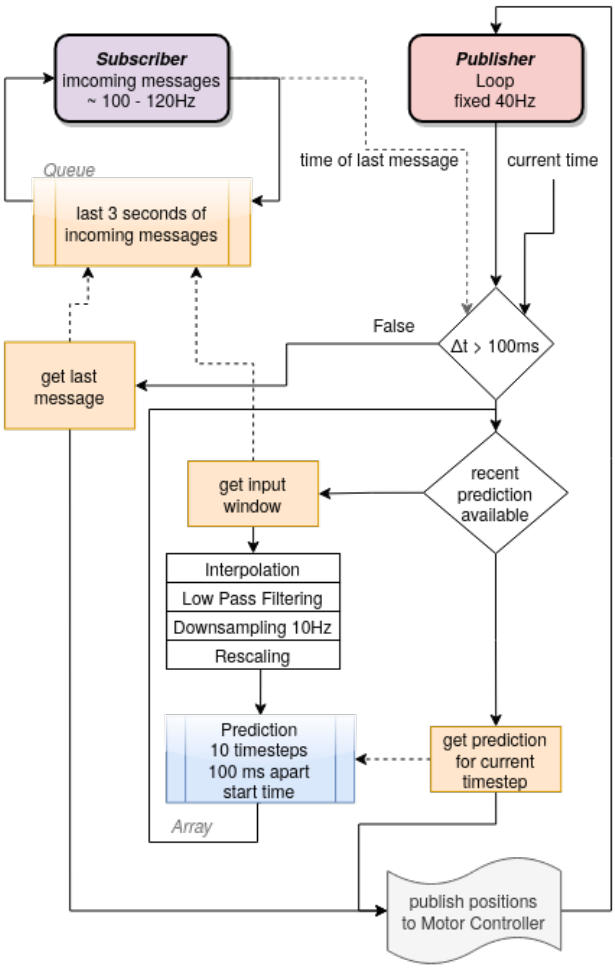

Figure 13: Flow-diagram of the second algorithm. A prediction is only being made when it is needed and is kept until it is out of date.

5.2 Algorithm 2

Figure 13 shows the second prediction algorithm. The *Subscriber* constantly receives the incoming messages and adds them to a queue. The *Publisher* loop runs at a fixed frequency of 40Hz and constantly checks if the the last received message is older than 100ms. If it is not, it is sent to the motor driver. If it is, a prediction is made and saved to the Prediction array. The prediction for the current timestep is chosen, sent to the motor driver and the loop starts again. If there is still no new message, the next value corresponding to the current timestep is chosen from the prediction array and being published. If the current timestep lies between two prediction values, it is interpolated. Once the current timestep exceeds the predicted timesteps, a new prediction is generated.

5.3 Delay Setup

In order to evaluate the quality of the system, network delays were simulated. The delays were implemented in the IMU node, where the IMU measurements were stored with their respective timestamps. A first-in-first-out (FIFO) queue was utilized to maintain the original message order. To ensure no packet is lost, the queue length was set to be infinite. Before publishing a sensor message to the control node, it should wait until the artificially introduced delay passes. For this purpose, we always compare the timestamp of the first element in queue with the current system time. When the current system time is later than the timestamp plus delay, the first message in queue is published. To test different system behaviours, two kinds of delays were implemented:

1. **Constant Delay**: The whole system is constantly delayed by a fixed Δt.

2. **Varying Delay**: The system delay changes in set intervals. In the first 80% of the interval, we set delay to

zero. In the rest 20% time, we set a delay of 200ms - 300ms.

6 Evaluation

The evaluation section is split into two categories. First, the performance of the models and their predictions will be compared, followed by an evaluation of the whole system.

6.1 Model Evaluation

6.1.1 Prediction Performance

The prediction performance of the models was compared on the used dataset splits. Table 1 shows the MAEs as well as the average angle error in degrees. Even though model one had the lowest training error, its errors on the validation and test set were the highest. This indicates that the model overfit to the training dataset, which can be explained by the comparatively high number of model parameters. We tried counteracting the overfitting by using weight and dropout regularization, but this was the best performance we were able to get. Model two had the lowest errors on the validation and test set, followed by model three. This indicates, that the architectural complexity of model two and the amount of available training samples was quite balanced. Model three also performed well, especially considering the low amount of trainable parameters and the comparatively simple architecture of the model.

Model	Set	MAE ↓	Avg. Angle Error [°] ↓
1	Train	**0.044**	**1.01**
	Validation	0.056	1.27
	Test	0.055	1.26
2	Train	0.05	1.15
	Validation	**0.053**	**1.21**
	Test	**0.05**	**1.16**
3	Train	0.053	1.22
	Validation	0.055	1.25
	Test	0.052	1.21

Table 1: MAE and average Angle error of the three models on the individual splits. Best results of each set are marked in bold font.

Figure 14 shows a comparison of the models predictions on unseen samples from the test set. They all follow the general trend of the ground truth signals, but an exact prediction of all values is not possible. This might stem from the fact that the general movement pattern in playing the game is quite repetitive, but the actual balancing is very delicate and comes down to lots of fine and spontaneous movements. The results indicate that the models are able to learn this basic structure, but are not complex enough to learn the fine details. Also, the rather short input window of three seconds might not be enough information to make such granular predictions.

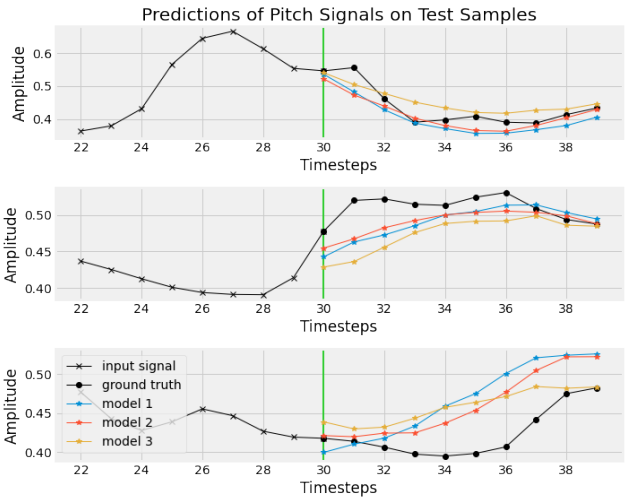

Figure 14: Comparisons of pitch signal predictions of random samples of the test set. All model predictions are compared to the ground truth signal. Data until timestep 30 shows a part of the input signal. The general trend of the ground truth signal is predicted correctly, but a detailed reconstruction is not possible.

6.1.2 Prediction Speed

To evaluate the prediction speed of the individual models, a benchmark was done directly on the NanoPi Neo3. All three converted tflite models were compared by letting them predict 1000 samples in a row and measuring the average execution time. Table 2 shows the models average predictions speeds next to the amount of parameters and their file sizes.

Model	Prediction Speed	Parameters	Filesize
1	17ms	89,780	412,7kB
2	1.7ms	20,315	94,7kB
3	1.6ms	11,212	50,1kB

Table 2: Comparison of the models prediction speed per sample, number of parameters, and filesize.

Model one had the lowest prediction speed out of the three models. This is probably caused by the larger number of parameters, which also contributes to its relatively large filesize. Even though the second model had almost twice as many parameters as model three, its execution time was only 0.1ms slower. The test showed that all models were able to run in real time on the used hardware, but model two and three should definitely be preferred, especially considering that the data collection and preprocessing for feeding the models will also take additional time.

6.2 System Evaluation

In order to evaluate the performance of the whole system, experiments were run utilizing the implemented delays explained in section 5.3. While playing the game, the delayed signals were sent to the prediction and motor controller, while the optimal positions without delay were recorded for reference.

6.2.1 Algorithm 1

As algorithm 1 was designed to deal with constant delays, all measurements were sent with an offset of 200ms. AS can be seen from table 3, all resulting errors were very high. None of the models was able to make good predictions under these conditions. Their results were usually were jittery and didn't follow the movement of the player well. We assume this is, because the models are all not very stable. Small changes in the input values can lead to drastically different outputs. Because the predictions are being done for almost every newly received value, the amount of varying predictions is very high and therefore also the introduced error.

Model	MAE ↓	Avg. Angle Error [°] ↓
1	0.1162	6.658
2	0.1307	7.489
3	0.1307	7.489

Table 3: MAE and average Angle error of the three models in live prediction with spontaneous delays of 200ms.

6.2.2 Algorithm 2

In order to evaluate the second algorithm, the varying delay was used. Every five seconds, all sent signals were delayed between 200ms - 300ms for the duration of one second. The predicted signals were recorded and compared to the actual non-delayed goal positions. Table 4 shows the resulting MAEs, as well as the average angle errors.

Model	MAE ↓	Avg. Angle Error [°] ↓
1	0.0477	2.733
2	0.0552	3.163
3	0.0364	2.086

Table 4: MAE and average Angle error of the three models in live prediction with spontaneous delays of 200ms.

All models performed significantly worse than on the recorded dataset. This indicates, that the dataset did not cover enough varieties of gameplay and needs to be extended to lower the models generalization errors.

Similar to the results in 6.1.1, the models were again able to predict the general trend of the signal, but had troubles with a detailed reconstruction of the amplitudes, as can be seen in Figure 15.

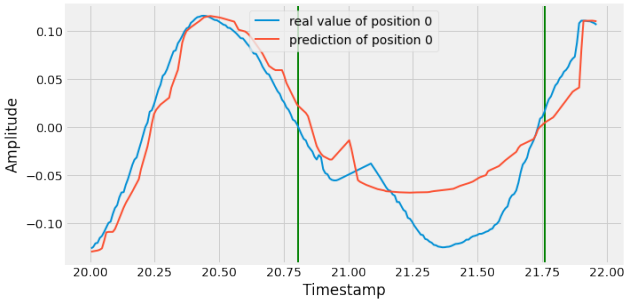

Figure 15: Live Prediction during a delayed interval, indicated by the signal between the green vertical lines. The predicted signals follows the general trend of the ground truth signal but a detailed reconstruction is not possible.

6.2.3 Playability

As section 6.2.1 already showed, the prediction error of all models in the live setup with a fixed delay was relatively high. This had a big impact on the actual playability of the game. In our subjective tests the game was not playable when a constant delay was present, because the instability of the predictions would make the platform jitter and the ball fall off. When the delay was set up to occur in intervals and the second algorithm was used, the prediction was able to save the ball from falling off every once in a while.

7 Conclusion and next steps

In this project, we recorded and analyzed data from the previously described telepresence game setup. We successfully built and trained three RNN models based on the processed data and implemented two algorithms for live usage in the system. One for dealing with a constant delay and one for varying delays. The models were able to learn and performed well on the recorded dataset with average prediction errors as low as 1.16 on the test set. Their predictions were able to follow the general trend of the ground truth signal, but were missing the details. The same behaviour could be measured with higher error in a live setup with varying delays. Their prediction results in the constantly delayed live system did not yield such good results, as the model predictions were not very stable over time and in general not accurate enough for controlling delicate system like this.

Our analysis and implementation lays a good foundation for further research on the topic and provides an infrastructure to experiment with different model architectures.

As possible next steps, we would suggest to record a larger amount of training data, which includes not only normal gameplay, but also other movements, in order to make the model more robust. In terms of model architectures, we would suggest to experiment with models that share weights over very long periods of time, like stateful RNNs. The models used in this project were all stateless. A custom loss function which emphasizes the earlier timesteps in the predictions could also help the model learn to compensate shorter delays better. In an ideal case the model would learn the underlying physics of the game, but as these are extremely complex, architectures with more parameters will be needed. To run such models on embedded devices in real time, they could be quantized and pruned even further or their calculations could be ported to external tensor processing units (TPUs).

We'd like to thank Mojtaba Leox Karimi and Martin Oelsch for supervising us on this exciting project, as well as Prof. Steinbach and the LMT for offering the tactile internet laboratory course.

References

[1] T. Aykut, M. Karimi, C. Burgmair, A. Finkenzeller, C. Bachhuber, and E. Steinbach. Delay compensation for a telepresence system with 3d 360 degree vision based on deep head motion prediction and dynamic fov adaptation. *IEEE Robotics and Automation Letters*, 3(4):4343–4359, August 2018.

[2] T. Aykut, J. Xu, and E. Steinbach. Realtime 3d 360-degree telepresence with deep-learning-based head-motion prediction. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(1):231–244, February 2019.

[3] M. Karimi, E. Babaians, M. Oelsch, and E. Steinbach. Deep fusion of a skewed redundant magnetic and inertial sensor for heading state estimation in a saturated indoor environment. *International Journal of Semantic Computing*, 15(3):313–335, September 2021.