# Recurrent and Convolutional approaches for System Identification

Paoli Leonardi Francesco
S297078

Bramucci Roberto
S303683

## Abstract

*The project focuses on system identification, which is the process of estimating the dynamical model of a physical system by input-output data collected experimentally.*

*The considered system is the gimbal's yaw motion of a two-wheeled self-balancing mobile robot, controllable through a brushless motor. The dataset has been collected experimentally by us, sending some given control signals to the yaw motor and sampling the corresponding yaw position and velocity.*

*We mainly focused on recurrent and convolutional approaches, employing classical layers (e.g., GRU, LSTM, CONV1D) as well as some relatively innovative approaches like TCNs (Temporal Convolutional Networks), which overall showed comparable (or even better) results with respect to the state-of-the-art system identification approach.*

*Several NN architectures have been explored, tested, compared and, after being properly trained, our models could eventually be used in real-case applications to forecast the future system's state over time.*

*We also discuss the details of what we have learned during the project, mainly about real-time data collection, NN architectures design, regularization and hyperparameters tuning.*

## 1. Introduction

A fundamental problem in the automatic control and estimation field is the one of finding a physical system's mathematical model that we can reliably trust to predict the system's state evolution over time, to properly design an automatic controller to control its dynamical behavior, and to simulate the system on software to speed up the controller testing process, while avoiding the need of expensive hardware for tests. Indeed, the absence of a proper mathematical description of our system would necessarily make it much harder for us to control the system and predict its future states.

In control theory, *system identification* is the set of techniques that face the above mentioned problem by building predefined state-of-the-art models [2] and estimating their coefficients by running optimization problems using the input-output data collected from the physical system.

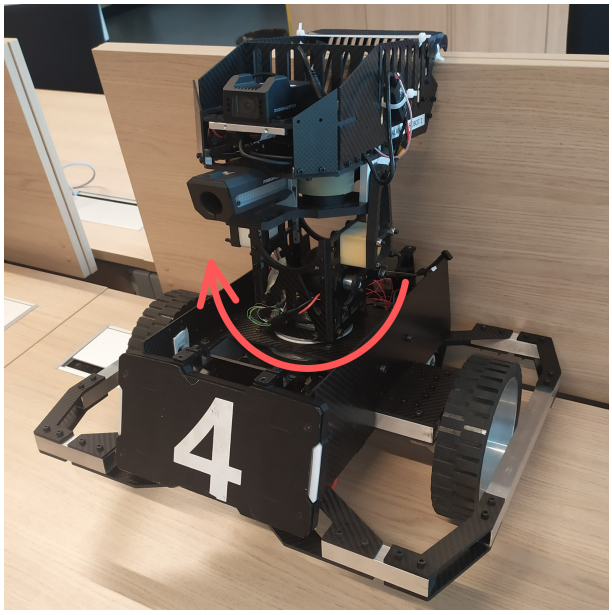



Figure 1: Two-wheeled self-balancing robot
(red arrow represents the gimbal's yaw motion).

In this project, our aim is to explore different NN architectures that could be used to perform system identification, achieving performance comparable to the state-of-the-art approach.

The physical system we considered for the project is the gimbal's yaw motion (i.e., gimbal motion around the $z$ vertical axis) of a two-wheeled self-balancing mobile robot (Figure 1). The gimbal is equipped with a paintball-like shooter, whose yaw position $\phi$ and velocity $\dot{\phi}$ can be directly controlled by means of a brushless motor, while its body can remain fixed on ground.

By sending voltage control signals to the yaw motor and sampling $\phi$ and $\dot{\phi}$ from the motor's encoder and from a microcontroller's IMU (Inertial Measurement Unit)

mounted on the robot, we built the raw dataset that we subsequently manipulated, normalized and used for the training, validation and testing of our NN models.

Our goal was to build NN models that could forecast the future robot's $\phi$ and $\dot{\phi}$ states, which could be used as mathematical models to *control* the system and to *predict* its future behavior over a medium-long period of time (around 20-40 seconds). To do so, we trained our models on two different datasets (both obtained from raw data), depending on their purpose:

- Models used for *prediction* purposes were trained on *contiguous* data, i.e., collected data properly converted into time-window samples of a certain length.
- Models used for *control* purposes were trained on *delta* data, i.e., collected data that, before being converted into time-window samples, were manipulated by subtracting each single measurement sample from the following one.

Further details on this are given in Section 2 and 3.

Trained models were compared by plotting their time response to some given test set inputs, evaluating their similarity to the experimentally-measured ground truths. Details are illustrated in Section 4.

The entire project's Python code to implement what is described in this paper has been written by us from scratch, and the dataset used to train, validate and test our models has been entirely collected by us.

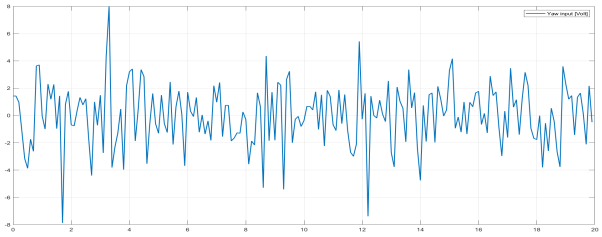
## 2. Data

### 2.1. Dataset structure and splitting

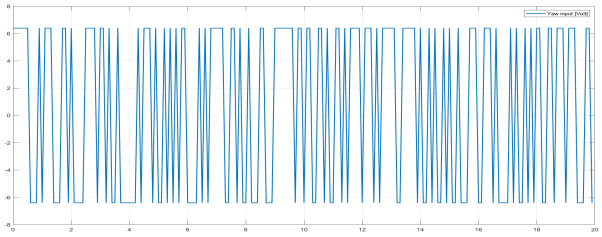
To build the dataset, we overall performed 181 data-collection experiments, collecting 151, 15 and 15 samples $(83.42\%, 8.29\%, 8.29\%)$ for the train, validation and test sets, respectively. We considered 181 time-series to be enough for our objectives because this is an amount of data quite larger with respect to the ones that are usually available for state-of-the-art system identification [1]. The percentages used to split the dataset in train, validation and test sets were chosen by assuming that overfitting and poor prediction performance would be very evident even with few validation and test samples, hence we decided to exploit most of our data to train the models.

### 2.2. Length of dataset samples

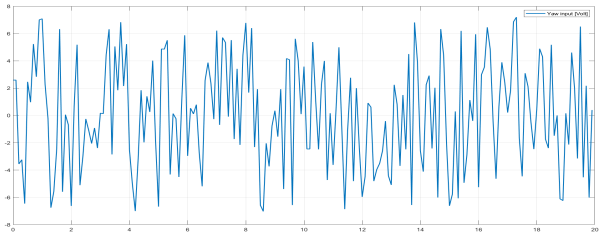
To bring more evidence of our NN models being able to learn the patterns of the physical system dynamics without overfitting the training data, we chose to collect train, validation and test samples of length 20, 40 and 40 seconds, respectively. This will show that our models are able to predict time-series longer than the ones used to train them.



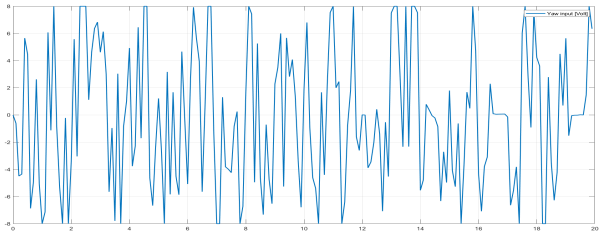
(a) White noise signal with Gaussian distribution



(b) Binary random signal



(c) Uniformly distributed random signal



(d) Remote controller signal

Figure 2: System inputs used to build the dataset

### 2.3. Inputs to the system

To build the dataset, we used four different types of input voltage signals, represented in Figure 2. Signals are saturated at $8$ Volt, which corresponds to the maximum motors torque capability.

- White noise signal with Gaussian distribution.
- Binary random signal.
- Uniformly distributed random signal.

- Pseudo-random signals generated through the joystick of a hand-held remote controller.

Gaussian distributed white noises are the standard in state-of-the-art system identification, while all other signals were generated with the goal of having the more heterogeneous and general dataset possible. All input sequences were generated on Python, saved in CSV files, loaded in a C code to be uploaded on the robot and used to power the yaw motor.

### 2.4. Challenges of experimental data collection

A challenge that we faced while collecting data was to make sure that our input signals could stimulate all system's normal modes [5], which is, to make sure that the collected dataset could summarize the entire system's dynamics. To ensure this, we mainly did two thing:

- We carefully selected the frequency of our input signals to the yaw motor by using multiple sinusoidal input signals of same amplitude and increasing frequency, stopping at the highest frequency that was not leading to an attenuation of the yaw motor response. From this procedure, we selected 10Hz.

- We performed a "whiteness" test: for each input signal, we used the MATLAB function `xcorr` to compute and plot the autocorrelation of the signal, i.e., the correlation between the signal and itself shifted of a certain amount of time instants. This guarantees that each time sample of the input signals is not correlated to the others, so that there's no hidden trend in the signals that we input to the system. As an example, the Figure 3 shows the autocorrelation plot of the signal in Figure 2a, where the two black horizontal lines are its 95% confidence intervals.

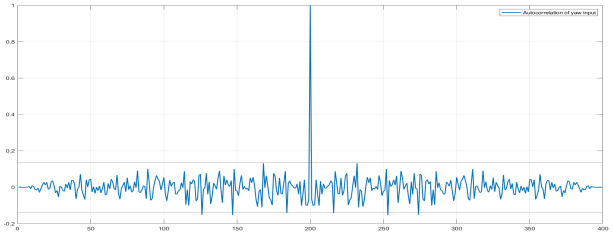


Figure 3: Autocorrelation of white noise signal (the peak in $x = 0$ shows the perfect correlation of the signal with itself).

### 2.5. Measurements sampling

Gimbal's yaw position $\phi$ and velocity $\dot{\phi}$ measurements were sampled at 200Hz to avoid any loss of info, then we downsampled those data by a factor of 20, leading to an actual sampling frequency of 10Hz to match the motor's input signals frequency. An example of sampled yaw positions and velocities is in Figure 4.

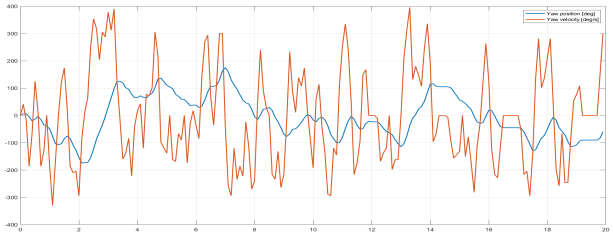


Figure 4: System response to input signal in Figure 2d (yaw position $\phi$ in blue; yaw velocity $\dot{\phi}$ in red).

All experimentally collected data are in the `data/raw` project folder, and were numbered by following a chronological order.

### 2.6. How much data quality matters

By performing the very first training experiments after the first data-collection session, the prediction performance of NN models were very poor, from which we learned how crucial is the quality (rather than just quantity) and "heterogeneity" of data in order to let the models learn the desired patterns. Indeed, we realized that we were using too many randomly generated input signals and not so many "smooth" hand-generated signals, hence the dataset lacked of some system's dynamics intrinsic information. We solved by doing two additional data-collection sessions (so, three in total) where we sampled some more input-output sequences with hand-generated inputs via remote controller, ensuring that each data-collection session was done under the same environmental conditions.

## 3. Methods

The functions to load, process and normalize data, train and validate NN models, perform predictions on never-seen-before data and visualize performance have been all implemented by us from scratch as Python modules, available in the `src` project's folder.

### 3.1. Features and ground truths for training

The measured $\phi$ and $\dot{\phi}$ were used as ground truths. As input features instead, together with the voltage control signals sent to the yaw motor, we used the time vector that, for each of the 181 dataset samples, starts from 0 and reaches 20 or 40 seconds (depending on the sample time duration). We added the time to the features by taking inspiration from [3] and, when performing our very first training exper-

iments, we actually saw that this helped the model's learning process to become a little more robust and consistent over time.

### 3.2. Data processing

To bring all dataset measurements to the same scale, both features and ground truths have been normalized, which improved the models training convergence.

Normalized data are then fed to a data generator (instance of the `SysIdentDataGenerator` class, implemented in `src/data_generator.py`), which creates the features `X` and ground truth `Y` tensors that are used to train the models. In the perspective of future system identification projects with larger datasets, the data generator have been implemented in such a way that it can generate the `X` and `Y` tensors both all at once (before training) and also at run time when needed (while training, which is slower, but it occupies just the strictly needed memory).

### 3.3. Models architecture

We designed 10 model architectures, training each of them on both the *delta* and *contiguous* set (details on this later in this section), hence 20 trained models in total. All trained models are available in the `models/tested_models` project's folder, and their architectures are showed in Figure 11.

- Models 1-4 are used to evaluate the state-of-the-art recurrent and convolutional blocks employed with time-series, which are SRNN (Simple Recurrent Neural Network), GRU (Gated Recurrent Unit), LSTM (Long-Short Term Memory), CONV1D (Convolutional 1-Dimension) and AveragePooling1D, highlighting their individual performance.

- Models 5-6 explore the possibility of using the aforementioned basic recurrent and convolutional blocks in a parallel way, followed by FC (Fully Connected) layers with a skip connection. As discussed in Section 4, this idea follows the "inception" intuition, and obtained very promising results. Model 6 also shows the benefits over model 5 that dropout and L2-regularization have during the training phase to avoid overfitting.

- Models 7-8 analyse the performance of, respectively, a decoder-like and encoder-like architecture, both based on LSTM.

- Model 9 explores the TCN (Temporal Convolutional Network), a valid alternative to classic LSTM/GRU that exploits both recurrence and convolution to guarantee good memory capacity with long time-series and stable gradients during the gradient descent convergence. The Keras TCN block was made available by the open source `keras-tcn` GitHub repository (`https://github.com/philipperemy/keras-tcn`).

- Models 10a-10b analyse the different effects of TCN hyperparameters tuning, together with different tunings of the dropout and L2-regularization hyperparameters.

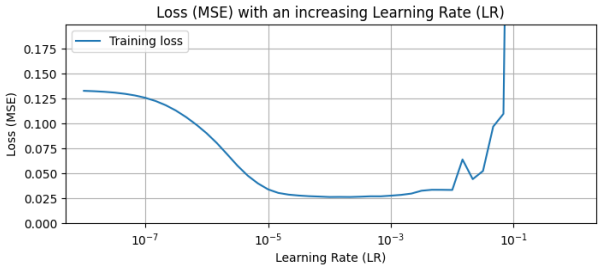



Figure 5: Loss of model 1 over an increasing LR

### 3.4. Challenges of hyperparameters tuning

For each NN architecture, before going through the actual training phase, the model's hyperparameters were carefully selected. The hyperparameters we found out to be the most challenging to tune are the learning rate (LR) of the gradient descent and the time-windows size used to build the train, validation and test time series. We tackled the LR and the window size tuning as follows:

- The LR was chosen by fixing the batch size (in our case, 32 for all architectures) and performing a preliminary training with the Keras `LearningRateScheduler` to explore a large range of possible LR values (approximately $[10^{-8}, 10^{0}]$), searching for the lowest (i.e., more "stable") value that was leading to the fastest convergence time (to the minimum loss plateau). We obtained very similar results for all model architectures; as an example, the LR search for model 1 (model in top-left corner of Figure 11) is in Figure 5. From this procedure, we set the LR to $10^{-4}$ for all models.

- The window size was chosen as a trade-off between the amount of available past information and the model complexity. Following this trial-and-error procedure, we set the windows size to $40$ for all models.

All other hyperparameters were chosen by trial-and-error, varying them individually while keeping all remaining hyperparameters fixed.

### 3.5. Training epochs, callbacks and optimizer

All models have been trained for (maximum) 30 epochs, and we defined four callbacks to be called at the end of each training epoch:

- `ReduceLROnPlateau` with `patience=1` and `factor=0.5`, to help the gradient descent convergence by reducing the LR near a train loss plateau.
- `EarlyStopping` with `patience=10`, to stop the training if the validation loss was not decreasing for 10 epochs in a row.
- `ModelCheckpoint` to backup the model at the current training stage.
- `TensorBoard` to save logs for post-training analysis of gradients (to spot possible vanishing/exploding gradients).

As optimizer, we chose to stick with ADAM for all model architectures, because of its momentum and because it generally reaches good results with all configurations, reducing the need of hyperparameters fine-tuning for each single architecture [4].

### 3.6. Training loss

As training and validation loss we used the Mean Squared Error (MSE), which we believe to be the most intuitive for our application, as it is also the mainly used one in state-of-the-art system identification.

### 3.7. Metrics for performance evaluation

For prediction performance analysis instead, we used the following metrics:

- *MAE* (Mean Absolute Error): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- *MSE* (Mean Squared Error): $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, which aims at penalizing more very large errors and penalizing less very small ones.
- $R^2$ (Coefficient of Determination), which is given by Eq. 1. It ranges from 1 to $-\infty$, where 1 represents perfect fit of data, while values under 0 indicate that the model's predictions are no better than simply using the mean value of the dependent variable as the prediction for all observations (minimum value is $-\infty$ because there's no limit to "bad" models).

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \quad \bar{y} \doteq \frac{1}{n}\sum_{i=1}^{n} y_i \qquad (1)$$

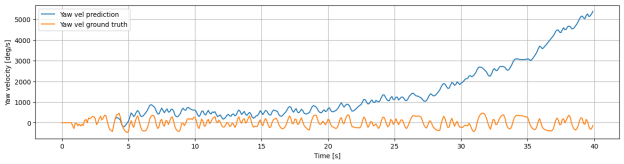



Figure 6: Model 1 prediction of $\dot{\phi}$ on *contiguous* set, after being trained on *delta* set (prediction in blue; ground truth in red).

### 3.8. *Delta* and *contiguous* datasets

As mentioned in previous sections, each model has been trained on two separate datasets that, from now on, we will call *delta* and *contiguous* sets:

- In the *contiguous* set, each single sample element represents the absolute position/velocity of the yaw motor at that time instant. Therefore, models trained on this set learn how to do multi-step predictions on the absolute $\phi$ and $\dot{\phi}$.
- In the *delta* set, each single sample element does *not* represent the absolute $\phi$ and $\dot{\phi}$, but represents the *difference* between the $\phi$ and $\dot{\phi}$ in the current instant of time and the ones in the previous time instant. Therefore, models trained on this set learn how to do multi-step predictions on the *variations* that $\phi$ and $\dot{\phi}$ have at each time instant with respect to the previous one.

Both sets are made up of experimentally-collected data, properly converted into time-window samples of a certain length for training our models.

### 3.9. Why we need both *delta* and *contiguous* sets

Originally, we aimed at using the *delta* set for predicting both $\phi$ and $\dot{\phi}$ variations and absolute values (by knowing how $\phi$ and $\dot{\phi}$ vary over time, we can cumulatively sum the *delta* values to directly compute the *contiguous* ones).
However, when training our first models on the *delta* set and looking at their *contiguous* predictions, we recognized the presence of drift in the $\phi$ and $\dot{\phi}$ measurements taken from the IMU sensor, i.e., a hidden "trend" of the measurements that let them constantly increase/decrease over time, even if the system was not subject to any input signal. As an example, Figure 6 shows how the model 1 trained on *delta* set have learned a dynamics that is affected by the unwanted drift overlapped to the $\phi$ and $\dot{\phi}$ measurements.

At first, to solve the IMU measurements drift problem, we tried to manually filter the raw data by computing the drift trend and subtracting it from each input-output sequence, but we realized that each dataset sample was affected from the IMU drift in a different way, so we could

risk to corrupt the true nature of the system's dynamics. Therefore, we decided to leave the data as they were, and try to let our NN models learn that the IMU drift does not belong to the real system's dynamics.

## 4. Experiments

For all models trained on the *delta* dataset, Table 1 and 2 show the training/validation metrics and the multi-step prediction performance on test sets, respectively. Table 3 and 4 show the same, but for models trained on the *contiguous* dataset.
Note that, due to the IMU measurements drift problem mentioned in Section 3.9, the models trained on the *delta* set have quite poor performance on the *contiguous* test set, while models trained directly on the *contiguous* set were able to successfully learn that the IMU drift is just a disturbance and is not part of the real system's dynamics.

### 4.1. "Inception" applied to system identification

From Table 2 and 4 we see that, on average, SRNN layers stop their learning process earlier (mainly due to vanishing gradients problems), while GRU, LSTM and CONV1D layers perform almost equally good on both *delta* and *contiguous* sets.

Inspired by the "inception" principle (commonly used in CNN architectures), we had the idea to build model 5 and 6 (Figure 11) by putting in parallel a GRU, a LSTM and a CONV1D layer; from Table 2 and 4, we see that this led to surprising results on multi-step predictions.

Indeed, on average, model 6 is the one that obtained better performance on both *delta* and *contiguous* set; its train and validation loss (MSE), MAE metric, $R^2$ metric and its prediction performance are showed in Figure 7, 8, 9, 10. The plots of all other models are reported in the GitHub repository of this project.

### 4.2. Regularization matters

Overall, we noticed the importance of dropout and L2-regularization to obtain models that are as "general" as possible. In particular, we learned that:

- Dropout and L2-regularization are crucial in medium-large models to avoid to overfit data, especially when trained on a small dataset. However, it is usually better to avoid dropout in the last 1 or 2 layers, because they are responsible for "correcting" errors induced by previous layers' dropout before the prediction happens.

- Dropout and L2-regularization in small models risk to be counterproductive: if a model already has few parameters, the presence of regularization could make its learning process stop earlier than expected.

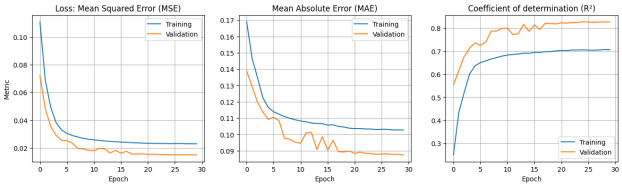



Figure 7: Train and validation loss (MSE), MAE and $R^2$ of model 6 on *delta* set (train curve in blue; validation curve in red).

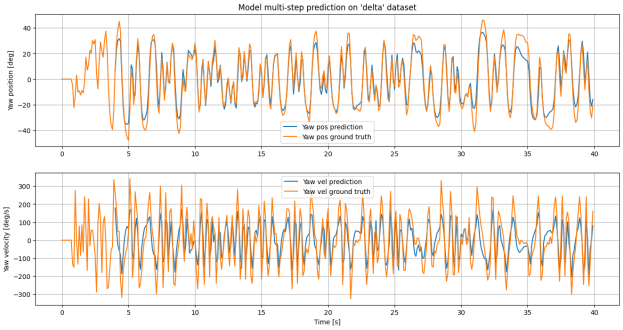



Figure 8: Prediction example of model 6 on *delta* set (prediction in blue; ground truth in red).

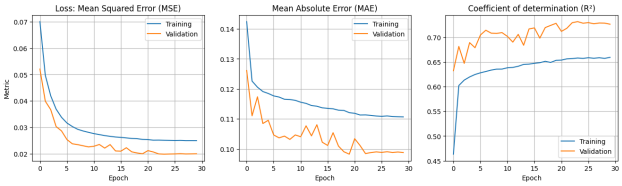



Figure 9: Train and validation loss (MSE), MAE and $R^2$ of model 6 on *contiguous* set (train curve in blue; validation curve in red).

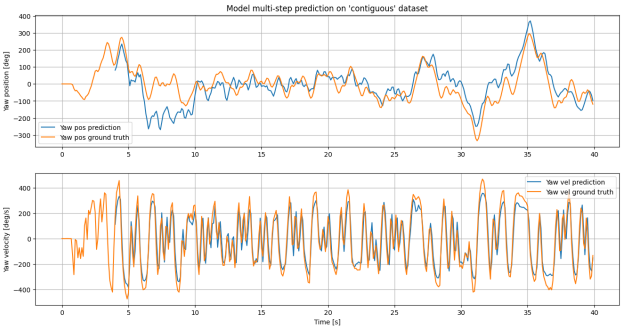



Figure 10: Prediction example of model 6 on *contiguous* set (prediction in blue; ground truth in red).

As an example, model 5 (that has no regularization) showed a quite remarkably "oscillating" validation loss curve, symptom of overfitting; in model 6, we were able to

smooth out the validation loss by introducing a 20% dropout rate and a kernel and bias L2-regularization with coefficient $\lambda = 0.02$ in the recurrent (i.e., GRU and LSTM) and convolutional (i.e., CONV1D) blocks, respectively.

### 4.3. Decoder and encoder architectures comparison

On average, the decoder and encoder NN architectures (respectively, model 7 and 8) showed very comparable prediction results. From Table 4, we see that the decoder architecture performed particularly well in the prediction of $\phi$ on *contiguous* test set.

### 4.4. Temporal Convolutional Networks (TCNs)

Table 1 and 3 show the promising ability of TCNs to convergence to very low train/validation loss values on both *delta* and *contiguous* set, thanks to their structure that exploits both 1D convolutions and recurrence (through the "dilation" of the 1D filters).
We believe that, with a much larger dataset, TCNs could have been the winning approach on both *delta* and *contiguous* set.

### 4.5. Comparison with state-of-the-art system identification

Just for the sake of comparison, we used the `arx` and `armax` functions of MATLAB to identify four models through state-of-the-art system identification approach:

- ARX321 and ARMAX2221 models for the *delta* set.
- ARX221 and ARMAX3331 models for the *contiguous* set.

As shown in Table 2 and 4:

- On *delta* set, the NN and state-of-the-art approaches reach quite comparable performance.
- On *contiguous* set, the NN models definitely outperform state-of-the-art system identification.

The supremacy of NN models over the state-of-the-art approach on the *contiguous* was quite surprising, but we could have expected it because state-of-the-art system identification models have very few tunable parameters (typically 8-15) with respect to any NN architecture that we employed, hence the latter managed to better fit the experimentally-collected data. The main drawback of NN models is that they need much more data and regularization to avoid overfitting.

## 5. Conclusion

We were thrilled to see that NN models can play their role in system identification, achieving comparable or even better results than state-of-the-art approach when it comes to future state predictions, paving the way for an extensive use of NNs in the automatic control field in general.

We explored NN architectures mainly based on SRNN, GRU, LSTM, CONV1D and TSN blocks. We learned how regularization techniques can help large models to reach training convergence without overfitting, but also that, sometimes, simpler models are better, faster, and more interpretable.

By collecting the dataset experimentally by ourselves, we also learned how crucial is the quality of data to properly train the models, perhaps much more important than the architecture of the models themselves.

For future extensions of this work, one could try other variants of the architectures we proposed, or try alternative approaches (like attention and transformers). Moreover, it is possible to use our trained models as plant to simulate the system on software, and train other NN models or Reinforcement Learning (RL) policies to learn a control law that could effectively control the yaw dynamics of the system.

## References


[1] Yue Dang and Carlo Novara. Identification of a servo-motor model from experimental data, 2019.

[2] Gerard Favier. An overview of system modeling and identification (invited paper). 12 2010.

[3] Kaicheng Niu, Mi Zhou, Chaouki T. Abdallah, and Mohammad Hayajneh. Deep transfer learning for system identification using long short-term memory neural networks, 2022.

[4] Antoine Richard, Antoine Mahé, Cedric Pradalier, Offer Rozenstein, and Matthieu Geist. A Comprehensive Benchmark of Neural Networks for System Identification. working paper or preprint, Sept. 2019.

[5] S.W. Shaw and C. Pierre. Normal modes for non-linear vibratory systems. *Journal of Sound and Vibration*, 164(1):85–124, 1993.

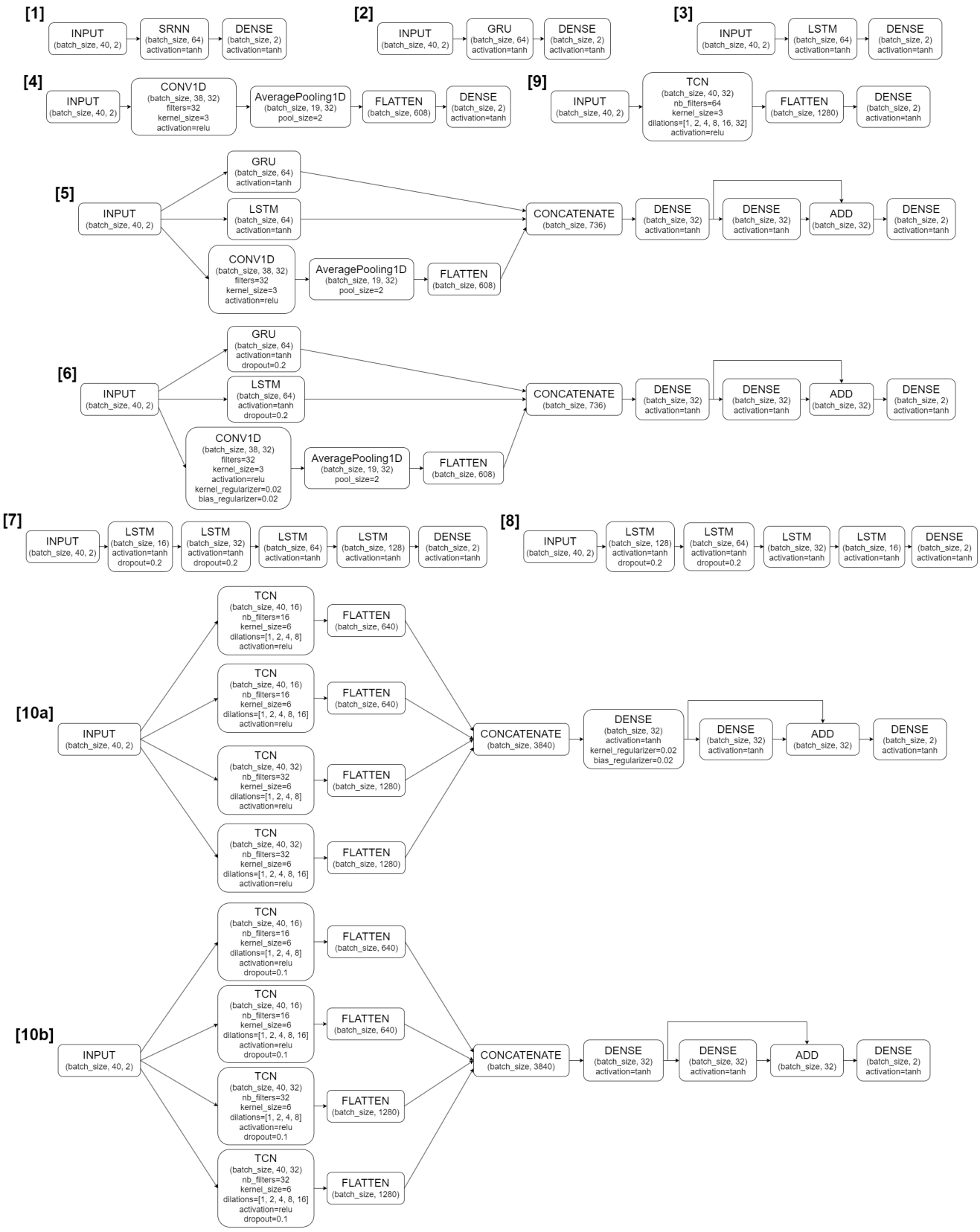



Figure 11: NN model architectures

| *Model ID* | *MSE (train/valid)* | *MAE (train/valid)* | $R^2$ *(train/valid)* | *N. of training epochs* | *Epoch of best model* |
|---|---|---|---|---|---|
| Model 1 (SRNN) | 0,0203/0,0147 | 0,1033/0,0946 | 0,7092/0,7972 | 30 | 30 |
| Model 2 (GRU) | 0,0205/0,0132 | 0,1038/0,0921 | 0,7051/0,8154 | 30 | 30 |
| Model 3 (LSTM) | 0,0203/0,0130 | 0,1031/0,0896 | 0,7097/0,8190 | 24 | 24 |
| Model 4 (CONV1D) | 0,0222/0,0144 | 0,1087/0,0926 | 0,6821/0,8008 | 30 | 30 |
| Model 5 (GRU-LSTM-CONV1D in parallel) | 0,0194/**0,0126** | 0,1003/**0,0875** | 0,7203/**0,8248** | 30 | 30 |
| Model 6 (GRU-LSTM-CONV1D in parallel, with dropout and L2-regularization) | 0,0205/**0,0124** | 0,1027/**0,0874** | 0,7067/**0,8271** | 30 | 30 |
| Model 7 (Decoder-like LSTMs) | 0,0539/0,0412 | 0,1686/0,1553 | 0,2415/0,4360 | 23 | 23 |
| Model 8 (Encoder-like LSTMs) | 0,0504/0,0324 | 0,1648/0,1393 | 0,2886/0,5553 | 23 | 23 |
| Model 9 (TCN) | **0,0161**/0,0140 | **0,0941**/0,0911 | **0,7686**/0,8035 | 30 | 30 |
| Model 10a (TCNs in parallel, with L2-regularization) | **0,0181**/0,0152 | **0,0931**/0,0918 | **0,7571**/0,8055 | 30 | 30 |

Table 1: Train and validation metrics for models trained on the *delta* dataset (best values in red; second-best values in blue).

| *Model ID* | *MSE of* $\phi$ *predictions on delta set* | *MSE of* $\dot{\phi}$ *predictions on delta set* | *MSE of* $\phi$ *predictions on contiguous set* | *MSE of* $\dot{\phi}$ *predictions on contiguous set* |
|---|---|---|---|---|
| Model 1 (SRNN) | 105,21 | 1339,59 | 2357,50 | 32794,89 |
| Model 2 (GRU) | 105,64 | **1203,79** | 2617,01 | **6197,28** |
| Model 3 (LSTM) | 101,08 | 1218,15 | **2318,55** | 8024,66 |
| Model 4 (CONV1D) | 107,04 | 1256,01 | 2615,15 | 28478,17 |
| Model 5 (GRU-LSTM-CONV1D in parallel) | **97,72** | 1227,26 | **2332,40** | 15621,97 |
| Model 6 (GRU-LSTM-CONV1D in parallel, with dropout and L2-regularization) | **98,54** | **1207,57** | 2547,90 | 7079,20 |
| Model 7 (Decoder-like LSTMs) | 257,16 | 1382,27 | 3047,22 | **3850,13** |
| Model 8 (Encoder-like LSTMs) | 223,47 | 1321,99 | 3006,87 | 6972,93 |
| Model 9 (TCN) | 105,40 | 1298,92 | 3685,23 | 57175,59 |
| Model 10a (TCNs in parallel, with L2-regularization) | 110,68 | 1280,42 | 3722,20 | 80025,29 |
| ARX321 | 105,60 | 3974,90 | / | / |
| ARMAX2221 | 54,81 | 3771,90 | / | / |

Table 2: Multi-step prediction performance on test sets, made by models trained on the *delta* dataset (considering only NN models: best values in red; second-best values in blue).

| *Model ID* | *MSE (train/valid)* | *MAE (train/valid)* | *$R^2$ (train/valid)* | *N. of training epochs* | *Epoch of best model* |
|---|---|---|---|---|---|
| Model 1 (SRNN) | 0,0246/0,0321 | 0,1128/0,1285 | 0,6517/0,5430 | 15 | 15 |
| Model 2 (GRU) | 0,0251/**0,0196** | 0,1144/0,1028 | 0,6461/**0,7198** | 16 | 16 |
| Model 3 (LSTM) | 0,0251/0,0198 | 0,1144/0,1033 | 0.6464/0,7189 | 18 | 18 |
| Model 4 (CONV1D) | 0,0256/0,0206 | 0,1160/0,1045 | 0,6371/0,7063 | 28 | 28 |
| Model 5 (GRU-LSTM-CONV1D in parallel) | 0,0242/0,0218 | 0,1115/0,1103 | 0,6586/0,6888 | 15 | 15 |
| Model 6 (GRU-LSTM-CONV1D in parallel, with dropout and L2-regularization) | 0,0241/**0,0192** | **0,1107**/**0,0988** | 0,6594/**0,7266** | 30 | 30 |
| Model 7 (Decoder-like LSTMs) | 0,0332/0,0225 | 0,1308/0,1101 | 0,5328/0,6815 | 20 | 20 |
| Model 8 (Encoder-like LSTMs) | 0,0319/0,0210 | 0,1279/0,1034 | 0,5524/0,7032 | 25 | 25 |
| Model 9 (TCN) | **0,0233**/0,0248 | 0,1117/0,1161 | **0,6717**/0,6476 | 30 | 30 |
| Model 10b (TCNs in parallel, with dropout) | **0,0207**/0,0206 | **0,1053**/**0,1004** | **0,7067**/0,7062 | 30 | 20 |

Table 3: Train and validation metrics for models trained on the *contiguous* dataset (best values in red; second-best values in blue).

| *Model ID* | *MSE of $\phi$ predictions on contiguous set* | *MSE of $\dot{\phi}$ predictions on contiguous set* |
|---|---|---|
| Model 1 (SRNN) | 3410,99 | 1221,49 |
| Model 2 (GRU) | 2014,50 | 1126,18 |
| Model 3 (LSTM) | 1971,27 | 1144,39 |
| Model 4 (CONV1D) | 1958,51 | 1203,74 |
| Model 5 (GRU-LSTM-CONV1D in parallel) | 2423,34 | 1142,09 |
| Model 6 (GRU-LSTM-CONV1D in parallel, with dropout and L2-regularization) | **1835,28** | **1078,32** |
| Model 7 (Decoder-like LSTMs) | **1793,35** | 1374,99 |
| Model 8 (Encoder-like LSTMs) | 2109,54 | 1132,15 |
| Model 9 (TCN) | 2444,11 | 1339,33 |
| Model 10b (TCNs in parallel, with dropout) | 2170,92 | **1090,60** |
| ARX221 | 9430,20 | 4954,90 |
| ARMAX3331 | 9530,10 | 4774,50 |

Table 4: Multi-step prediction performance on test sets, made by models trained on the *contiguous* dataset (considering only NN models: best values in red; second-best values in blue).