Master Thesis

Modelling and Simulation of Spacecraft Hardware Based on Machine Learning Techniques

Ayush Mani Nepal

Matrikelnummer: 4817578

27.03.2019

1. Prüfer:	Prof. Dr. -Ing. habil. Manfred Krafczyk
2. Prüfer:	Prof. Dr. rer. nat. Martin Geier
	Institut für rechnergestützte Modellierung im Bauingenieurwesen (iRMB)
	Technische Universität Braunschweig
	Pockelsstr. 3, 38106 Braunschweig, Germany
Betreuer:	Dr. rer. nat. Andreas Gerndt
	Deutsches Zentrum für Luft- und Raumfahrt (DLR)
	Institut für Simulations- und Softwaretechnik
	Lilienthalplatz 7, 38108 Braunschweig, Germany

Declaration of the Academic Honesty

I hereby confirm that the present Master Thesis with the title

„Modelling and Simulation of Spacecraft Hardware Based on Machine Learning Techniques"

was composed by myself and that the work contained herein is my own. I also confirm that I only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practise and indicated by the exact references to the original source.

The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version.

I understand that the provision of incorrect information may have legal consequences.

Braunschweig, 27.03.2019

Ayush Mani Nepal

Abstract

Research and development projects involving complex systems are often executed in a collaboration between engineers and scientists of various institutions. In such a collaboration, different parts of the system are developed by different companies located in different physical locations. Whenever this happens, testing the system and its components is difficult, since the adjacent hardware required to test the communication channels may not be available on site. This is further problematic in the space industry, because replicating a spacecraft hardware is expensive and arises other technical difficulties as every hardware is a prototype. A solution to this problem is to use a hardware emulator application in the development cycle of the Onboard Software (OBSW) which can reflect the true hardware behaviour and can be shared easily between the distributed development teams. However, precise low-level hardware emulators are very tedious to build and require deep domain knowledge.

A noble alternative solution to this problem is to learn a black-box model from the temporal data-space of the candidate hardware using a Machine Learning (ML) algorithm which can reflect the realistic hardware behaviour. Recurrent Neural Network (RNN) based models with the Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) memory units are excellent in learning sequences and are able to capture long-range dependencies in the temporal dataset. In this study, different LSTM and GRU models are applied to the task of modelling a physical hardware system. Input and output signals from the real hardware are used to train the RNN models following a supervised learning method. The obtained result show that both types of RNN models are capable of simulating the realistic hardware behaviour with a considerable accuracy. The performance of the well configured GRU model is seen to be slightly better than that of the equivalent LSTM model. Nevertheless, the signals predicted by both RNN models showed more than 90% resemblance to the actual signals obtained from the hardware in their frequency spectrum.

Contents

List of Tables

List of Figures

Listings

Nomenclature

.pcap	packet capture
AI	Artificial Intelligence
BEC	Bose-Einstein Condensate
BN	Batch Normalization
BPTT	Backpropagation Through Time
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DSL	Domain Specific Language
EHR	Electronic Health Record
EMF	Eclipse Modelling Framework
FFT	Fast-Fourier Transform
FNN	Feedforward Neural Network
FPGA	Field Programmable Gate Array
GRU	Gated Recurrent Units
IP	Internet Protocol
LSTM	Long Short-Term Memory

MAIUS	Matter-Wave Interferometry in Weightlessness
MBSD	Model-Based Software Development
ML	Machine Learning
MSE	Mean Squared Error
NIPS	Neural Information Processing Systems
OBC	Onboard Computer
OBSW	Onboard Software
QUANTUS	Quantum Gases in Weightlessness
ReLu	Rectified Linear Unit
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
SC	Simulation and Software Technology
SSP	Surface Similarity Parameter
SST	Sea Surface Temperature
SVM	Support Vector Machines
T-Stack	Test-Stack
Tanh	Tangent Hyperbolic
TCP	Transmission Control Protocol
UML	Unified Modelling Language

CHAPTER 1

Introduction

This chapter revels the motivation and objective behind this thesis work. In the second section, the goal of this study is set and the scope is defined. Then, the requirements of the project are listed. The chapter ends by summarizing the structure of this document.

1.1 Motivation

In the development process of complex systems, interdisciplinary teams of engineers and scientists are involved. Often times, such teams are formed by a cooperation between various institutions. The research and development report of Deutsches Zentrum für Luft- und Raumfahrt (DLR) in year 2014/2015 [1] describes projects in different domains that the space agency carried out in cooperation with other industry, research institutions, and universities. In such a cooperation, resources, human or otherwise, are distributed under various administrations which are located in different physical locations. In such cases, testing the system is troublesome. For example: it is difficult to test the hardware drivers of the system when every hardware components required to test the communication channels are not available on site. This becomes further problematic in the space industry, because the spacecraft hardware is expensive and not easily producible, therefore, it is not practical to have them replicated in every development sites. The Onboard Software (OBSW) developers have to deal with this kind of shortcomings.

When the peripheral hardware is missing, software developers use mock-up methods to simulate the channel values, which should otherwise be coming from the peripheral component. Mock-up methods do not reflect the real hardware behaviour and may result in *false-positive* or *false-negative* test results. A solution to this problem is to use hardware emulator applications in the continuous integration system, which simulate the realistic hardware behaviour. Unlike the real hardware device, hardware models are easily portable and replicable. Models are basically pieces of code which can easily be shared between development teams distributed in various physical locations. Simulating a physical system for testing has many other advantages such as increased insight and flexibility, efficient in resource utilization, execution speed, and timing fidelity in comparison to the real machine [2]. Furthermore, the hardware simulations are helpful in analysing the dynamic behaviour of the system. In addition to that, one can perform experiments in the model of the system, which may be highly risky to execute in the actual hardware.

However, highly precise low-level hardware models are expensive and require deep domain expertise to build. For example: a license of Simulink, a commercial simulation tool developed by Mathworks, costs 1200€ per annum [3] (price from March 2019). Nevertheless, hardware models which are used in the continuous integration system to assist the rapid OBSW development may not be highly precise as long as they reflect the realistic hardware behaviour. Such models can be developed from the data-space of the actual system following a black-box modelling technique. The requirement of any black-box modelling technique is that the data-space of the candidate system is known and available. Unlike white-box models, high-level black-box models are cheap to build and do not require detail domain knowledge and yet they can still be fairly precise.

Machine Learning (ML) algorithms as a black-box modelling tool are getting more and more popular in the past few decades, especially, when the large datasets are available. The process of creating a black-box model using a ML method involves in learning a generalized model from the training set. Thus, the training set has to be large enough so that a generalized model can be constructed from it, which performs well with the unseen data in the test phase. Training of such models can be computationally expensive. But, once they are trained, they work fast in the application phase. Nevertheless, due to availability of fast CPUs and GPUs, the computation cost is no longer an issue. Different research papers show concrete evidence of the effectiveness of ML algorithms [4–8] in pattern recognition, time-series prediction, data clustering, and classification.

In this study, the problem of a physical system modelling is formulated as a time-series prediction problem (temporal sequence modelling), such that a ML algorithm can be applied. In such a formulation, all actuators in the system act as causes and therefore collectively define the state of the system and sensor values are the corresponding effects. Since, experiments are executed over time, time is included as an important factor. Hence, the problem-space of this study lies in investigating state-of-the-art ML algorithms for the temporal sequence modelling.

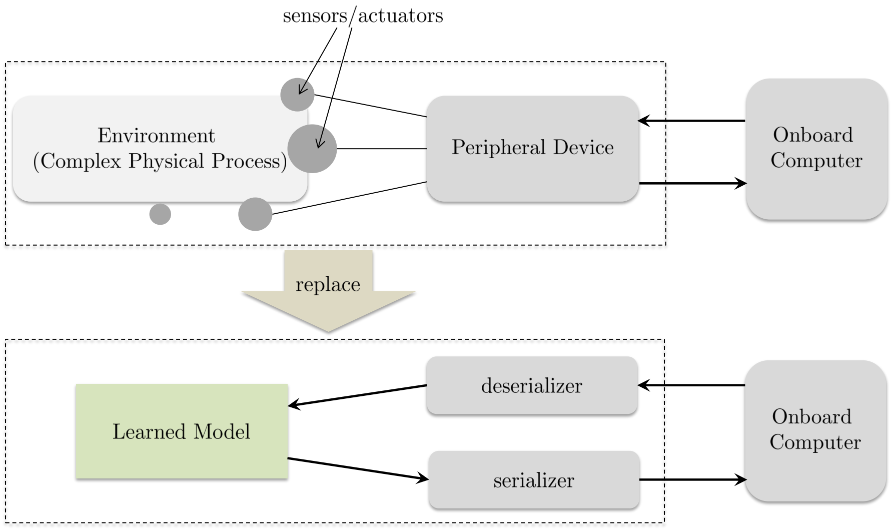


Figure 1.1: Schematic of the idea where a physical system is replaced by the learned model.

1.2 Goal and Scope

Keeping the use-case of the physical system modelling in mind, investigating state-of-the-art ML algorithms in temporal sequence modelling is the primary goal of the thesis work. Building up strategies and applying them to create a hardware model from the temporal data collected from actuators and sensors is the task. Further important milestones are to make strategies for preparing the datasets for training, and to design and build the model which generalizes the problem well, such that the prediction error is as small as possible. After been sufficiently trained, the hardware model is expected to predict the sensor values for the given set of actuator states. Nevertheless, the prediction error is defined and may only increase linearly when the setup is enlarged; the same is true for the training time as well as the memory requirement of the model.

Due to the limited time-frame of the thesis work, detail investigation of each available sequence modelling algorithms is not feasible. Therefore, relying on the literature, scope of the thesis is limited to the investigation of the state-of-the-art Recurrent Neural Network (RNN) models. RNNs are the most recognized neural networks for the time-series prediction problem [9]. RNN models have internal feedback loops, therefore, they can make use of past outputs to predict the next time-step [9].

Particularly, detail study of the Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) networks is performed. LSTM and GRU are special kind of RNNs which perform better even in high variance situations [8, 10, 11]. Furthermore, RNN models provide better generalization options via weights/activity regularization, Dropout layers [12], etc., therefore they work better with the unseen data, thus perform well in the application phase. Models such as Markov models, conditional random fields, etc., work with the sequential data, but are ill-equipped to capture the long-range dependencies in the dataset [7]. Other modelling techniques require strong domain knowledge and specialized feature engineering which makes it harder to discover hidden patterns. In contrast, neural network models learn representations and can discover unforeseen structures on their own [13].

1.3 Requirements

Fulfilment of the requirements determines the success of the thesis work. Therefore, it is important to list them beforehand. The Table 1.1 lists the requirements which the thesis project shall fulfil. Each requirements is assigned with an identifier (ID) which is given in the first column of the table. This ID is used to refer to the requirement in upcoming chapters. The statement of the requirement is given in the second and elaborated in the third column.

ID	**Statement**	**Explanation**
req_01	*The model shall be as abstract as possible and as detailed as necessary.*	The model design should be simple to understand. Unnecessary complications must be avoided.
req_02	*The hardware system and its components shall be treated as a black-box.*	Modelling should be based only on the input and output data-space of the candidate hardware, and should be implemented without the need of domain expertise.
req_03	*The model data-space shall also be highly abstract and outside of the system.*	The model must be operable without any dependencies to the actual system.
req_04	*Training data shall be collected without having to interfere with the normal Onboard Computer (OBC) operation.*	The data collection procedure must not change or disturb the communication between OBC and the candidate hardware.
req_05	*Training data shall be collected without having to change the OBSW.*	It is not allowed to change the OBSW in order to collect the telemetries and the telecommands for training the network.

req_06	*The model shall be able to deal with various types of input and output signals, such as, binary, continuous, step-like, constant, etc.*	Modelling a physical system requires to deal with data from the actuators and sensors. It is expected that the actuators and sensors operate with variety of signals. Thus, the model should be able to deal with them.
req_07	*The model shall be able to replicate the system in the test phase, when the unseen input is passed to the system.*	The model must not over-fit to the training-data, moreover, it should be able to work well with the new inputs in the application phase.
req_08	*The model shall be scalable in space.*	The model must be designed in a such a way that it is expendable in space, so that, the same modelling process could be followed in the future to create hardware models with many actuators and sensors.
req_09	*The model shall have a defined error.*	A threshold value should be determined for the model up-to which the prediction error can be expected.

Table 1.1: Requirements of the study.

1.4 Structure of the Document

The second chapter gives the background information regarding the candidate hardware system, the state-of-the-art learning algorithm, and the similar work done by fellow researchers. A short introduction to the software tools and the specification of hardware used during this study is given in the last section of the second chapter. The third chapter describes the methodology of the thesis work, furthermore, different phases of the project's lifeline are explained. Different project phases include the training data preparation procedure, model design, training, and the testing procedures. The fourth chapter describes the implementation procedure for each project stages, which includes, how the experiment is designed, how the data is collected, and how the model is written. In the fifth chapter, results of various model instances are presented and compared with each other. Based on the results from chapter five, conclusion of the thesis work is drawn and presented in the sixth chapter, moreover, limitation of the modelling technique is mentioned along with the potential improvement in the future.

CHAPTER 2

Background

This chapter gives background information required to understand the concept and the idea behind the thesis work. The chapter starts with the theoretical background and develops towards the state-of-the-art technologies in detail. The chapter highlights few interesting work done by other researchers and ends by listing the tools and softwares used during the project.

2.1 MAIUS Experiment and Role of DLR

The Matter-Wave Interferometry in Weightlessness (MAIUS) mission is a part of the Quantum Gases in Weightlessness (QUANTUS) project, where the feasibility of experimenting with the ultra-cold atoms in space is investigated [14]. In the experiment, temperature of a gas cloud is reduced to absolute zero, thus producing Bose-Einstein Condensate (BEC). BEC is the state of matter where the quantum effects can be observed [14]. In 23 January 2017, the MAIUS payload was lunched on a sounding rocket from Kiruna, Sweden, and in its six minutes flight, BEC was successfully produced in space for the very first time [14]. The Simulation and Software Technology (SC) group of DLR is responsible for the development of the software used to control and monitor the MAIUS experiments. The payload hardware of the MAIUS experiments is designed by the Leibniz University Hanover.

2.1.1 MAIUS Hardware

The hardware used to control and monitor MAIUS experiments is custom-built, which consist of different circuit board modules, referred as *Cards*, which can be connected by stacking them together into a *Stack*. Each card posses a Field Programmable Gate Array (FPGA) controller.

Communication between the MAIUS stack and the OBC is done through Ethernet via Transmission Control Protocol (TCP) protocol. The TCP communication is handled in the stack by a special Ethernet card which has a separate IP address. The special power-supply card regulates power within a stack. Each stacks, therefore, have one Ethernet and one power-supply card. Cards themselves may not necessarily have sensors and actuators on them, however, they provide interfaces for actuators to control and sensors to monitor the MAIUS experiment and are be attached as required. Each cards has input- and output-channels. Data flowing direction in MAIUS is seen from OBC perspective, so sensor values shall be read via input-channels (input for the OBC) and write command shall be sent to actuators via output-channels (output for the OBC).

Test-Stack

In a typical MAIUS experiment, numerous number of cards and stacks may be used. However, MAIUS team in SC only use few cards in Test-Stack (T-Stack) for testing and debugging the OBSW and communication procedures.

Cards currently present in T-Stack are:

- **Ethernet-Interface** is the special Ethernet card which handles the communication between the T-Stack and the OBC.
- **Power2** is the special power-supply card which regulates power within the T-Stack.
- **Led1** contains seven-segment display and provides one output-channel to write one byte value to the display. It also contains a rotary-encoder and provides an input-channel to read the rotary-encoder value.
- **NTCIn2** provides 24 input-channels to read temperature value from 24 different temperature sensors. The card itself does not contain any sensor rather provides ports to connect them externally. There are no output-channels in the NTCIn2 card.
- **Fan1** provides 8 output-channels with boolean values. Internal current driver controls current in these ports, when set to "true" the connected device is supplied with current and it is turned ON. Analogous if it is the "false" command. Therefore, any device which meet the power requirement shall be connected to these ports, it does not necessarily have to be a fan.
- **PhotodiodeIn2** provides 24 input-channels to read photodiode values. Analogous to the other cards, this card also do not contain any photodiode in it, but provides interfaces to connect them.
- **AnalogOut1** provides 16 output-channels for handling the actuators. Each of these 16 ports shall be written with values ranging from -10 to +10, these value represent voltages in volts.

- **TriggerOutput** provides 13 trigger output-channels.

2.1.2 MAIUS Software

The custom-built MAIUS hardware requires specialized drivers. These drivers are automatically generated from the data-model written in a Domain Specific Language (DSL). DSL are programming language developed to define small class of problems. They are generally used to define data-models in the Model-Based Software Development (MBSD) process, such that executable code are generated from these data-models.

In MAIUS, such data-models describe interfaces of the MAIUS stacks and cards. Stack DSL instantiates all cards in the stack with a unique name and address. It also instantiates input- and output-channels of different cards that are used in the experiment. Moreover, the stack DSL also describes the Internet Protocol (IP) address and port of the stack.

A card DSL describes the input- and output-channels of the card. Channels are the interfaces which allow access to different memory addresses of the card and are used to read-/write card-parameters. In order to write/read value to/from these memory addresses, series of commands is sent to the card by the OBSW. In MAIUS, one command only reads or writes one byte of the value. So, if the card-parameter is more than one byte long, multiple commands are sent consecutively. Moreover, individual commands is dispatched as a separate TCP package. Each of these packets are three bytes long, which consists of one byte address, one byte of data and a read/write flag. Since, each packets of the data begins with a unique address, each access to the card's channels makes a unique pattern in the byte-stream. This makes it possible to easily read channel values directly from the TCP byte-stream when the card DSL is given, without having to modify the OBSW.

2.2 Modelling Theory

Modelling, in general, can only cover a certain aspects of the subject and may cast many shades of meaning. A system model can be interpreted as a mathematical abstraction in terms of a set of differential equations, or on the other extreme, an engineer may prefer to take a model as a scaled replica of the system [15].

In terms of the necessity of the domain knowledge of the candidate system, models can be classified as:

- **White-box models** require detail domain knowledge about the system. Access to the system internal states and parameters are needed for the white-box modelling approach.
- **Black-box models**, in opposition to the white-box approach, do not need access to the internal states of the system, rather they work with its peripheries, by only looking at the inputs and outputs of the candidate system.

- **Grey-box models** are the hybrid version of the white-box and black-box modelling approaches. Here, the system is transparent for certain extend, however, major part of the system is still unknown or unused in the modelling process.

In any case, a model is desired to represent the system partially, if not completely. In this study, using a learning method and following a black-box modelling approach, a scaled simulation model of an experiment with the T-Stack is desired.

2.3 Machine Learning

ML is the state-of-the-art in the applied Artificial Intelligence (AI) field. Arthur Samuel, an American pioneer in the field of computer gaming and artificial intelligence, coined the term "machine learning" in year 1959. The major breakthrough which led to the emergence of ML is the idea developed by Arthur to teach machines to learn on their own instead of teaching them everything they need to know in order for them to carry out certain tasks [16].

In [17], the author suggests definition of ML that he claims to be more suitable for computer scientists and statisticians, which states: *ML is the process (algorithm) of estimating a model that's true to the real-world problem with a certain probability from a dataset (or sample) generated by finite observations in a noisy environment.*

The core objective of a ML algorithm is to transform a real world problem into a parameter optimization problem and learn a generalized form of these parameters from the experience that comes from the training data. Parameter refereed here are the weights/kernel values and not the hyper-parameters of the neural network. Generalization is very important in ML, it determine the ability of the learning algorithm to perform accurately on the unseen data (test set) after having experience with the learning data (training set) [18]. For the good generalization, balance between the complexity of the model and function underlying the data has to be maintained. Simple model for a complex or large dataset, and a complex model for a simple or a small dataset, both result in poor learning. It is the job of the engineer to find the trade-off between these two.

ML algorithms can be divided into different categories, three main categories are:

- **Supervised learning** is basically a function approximation and can only be applied when the training data is labelled i.e. for each input feature there is a known target output. During the supervised training, the algorithm learns to model relationships and dependencies between input features and target outputs such that it can be generalized, and based on that, prediction can be made for the new input. Neural Networks, Support Vector Machines (SVM), Nearest Neighbour, etc. are few commonly used supervised learning algorithms.
- **Un-supervised learning** is applied when the data-set is unlabelled. They are mainly used in pattern detection, and to group the data points, which help in de-

riving meaningful insights in order to describe the data in a better way. K-Means clustering algorithm is one of the un-supervised learning methods.

- **Reinforcement learning** is an iterative learning algorithm which learns from its experiences with the environment to take action which would maximize the reward and minimize the risk. Q-Learning, Deep Adversarial Networks are the common reinforcement learning algorithms.

Difficulties with ML algorithm

ML algorithm must be able to deal with various difficulties [17]. In theory, the ML algorithm can learn arbitrarily many features from a dataset, but there can only be finite number of observations. It is not possible to generalize so many features by only making finite observations.

The other usual problem is that the learning algorithm compute their output by a linear combination of the input and the weighting factor. For example: an output of a feed-forward layer of a neural network model is given by $\mathbf{s} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{x}$, $\mathbf{W}$, and $\mathbf{b}$ are the input vector, weighting matrix, and the bias vector, respectively. However, the real world problem is usually highly non-linear. The algorithm designer has to take care of these problem when using ML algorithms as a modelling tool.

2.4 Neural Networks

Neural network is a ML algorithm inspired by the working of neurons in biological brain for learning. It is one of the most popular ML tool in regression and classification. The neural networks have neurons referred as *units*. These units appear in a hierarchical order; these hierarchies are referred as the *layers* in the network. Depending on the network topology, units are connected with the other units of different layers via links. Each of these links hold a weighting value which controls the information flow through them. Dataset for a neural network is a high dimensional vector. For the simplest case the dataset has a form of a matrix, where each column vectors represents one sample.

A neural network topology consists of the following basic components:

- **Input layer** is the first layer of the network. Number of units and dimension of the input layer is equal to that of features in the input data. In the multiple input network topology, there may exist more than one input layer connected to the same hidden layers.
- Arbitrary number of **hidden layers** can have arbitrary number of units in them. They determine the depth and breadth of the network.
- **Output layer** is the last layer of the network. Number of units in this layer and its dimension is equal to the target outputs in regression or the target classes in case of

the classification problem. In the multiple output network topology, there may exist multiple number of output layers.

- **Set of weights and biases** are the key components of the neural network. The whole idea of training is to find the optimum set of weights and biases. Depending on the topology and the type of the layer, weights can be trainable or non-trainable. Trainable weights are adjusted during the training process to minimize the output error, whereas non-trainable weights are constant during training as well as testing.

  One unit in each layer is a special one, which is connected to all units of the next layer. The link between them posses a special weighting factor known as bias. Similar to normal weights biases can also be either trainable or non-trainable. Biases can be use to favour one feature over the another in regression and to make one class be classified more often than the other in classification.

- **Activation layer** transforms each unit's output from the previous layer by a known function. Thus, it has units equal to that of the preceding layer. The units of activation layers are connected only to the adjacent unit of the preceding layer. For input and hidden layers this function is usually non-linear and depending on the problem the output layer may or may not have an non-linear activation. Common practice in regression is not to alter the output of the last layer, therefore the output layer of the network usually only has a linear activation.

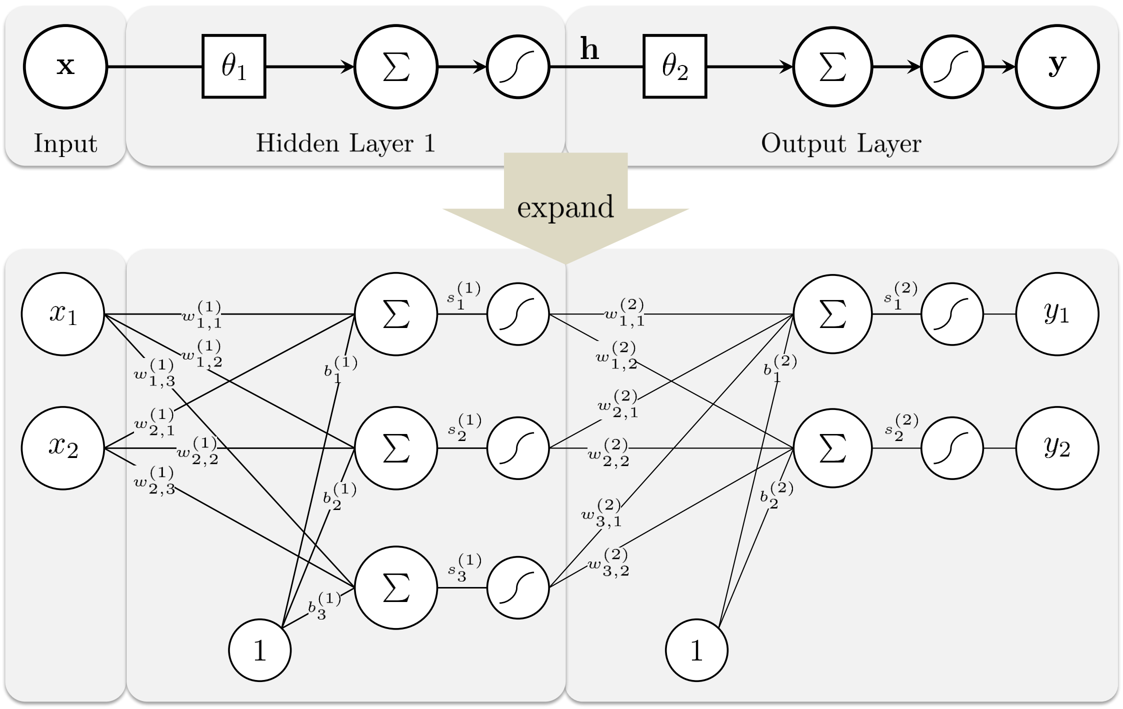


Figure 2.1: Standard Feedforward Neural Network (FNN) topology with one hidden layer.

In FNN, the activation of $\jmath$th unit ($h_\jmath$) of an arbitrary layer l is computed as shown in Equation 2.1 [19]. The Figure 2.1 shows a typical FNN topology with one hidden layer. In FNN, the input sample is fed from the input layer and moves only forward through the network.

$$s_\jmath^{(l)} = \sum_\imath w_{\imath,\jmath}^{(l)} \, x_\imath + b_\jmath^{(l)} \tag{2.1a}$$

$$h_\jmath^{(l)} = \mathcal{F}(s_\jmath^{(l)}) \tag{2.1b}$$

Here:

$s_\jmath^{(l)}$ is the state of $\jmath$th unit in layer l,
$h_\jmath^{(l)}$ is the output of $\jmath$th unit in layer l,
$\mathcal{F}(.)$ is the activation function of layer l,
$w_{\imath,\jmath}^{(l)}$ is the weight between $\imath$th unit of $(l-1)$th layer and $\jmath$th unit of lth layer,
$b_\jmath^{(l)}$ is the bias of $\jmath$th unit in layer l.

Equations 2.1a and 2.1b are combined and rewritten in the vector notation in Equation 2.2.

$$\mathbf{h} = \mathcal{F}(\mathbf{W}\ \mathbf{x} + \mathbf{b}) \tag{2.2}$$

While designing the neural network topology, there are methods which could be followed to overcome the difficulties of ML introduced in the previous section. For instance: as a remedy to the problem of having an arbitrarily many features to be learned from a finite set of samples, one can use features extraction layers in the beginning of the network. Feature extraction layers do not learn all features in the dataset but select the most important ones. Furthermore, activation layers mentioned earlier are used to introduce non-linearity into the neural network model.

Backpropagation

Supervised learning of weights and biases in neural network is performed using the backpropagation algorithm, which uses chain-rule to compute gradients in the gradient-descent learning method. Although already been discovered in th 70s, popularity of backpropagation in neural network training begin after the publication of the famous paper [20] by David Rumelhart in year 1986.

In order to apply backpropagation algorithm, first of all the cost function $J(\Theta)$ is chosen, where Θ is the set of weights and biases. Cost function determines the loss $\mathcal{L}$ of the prediction made by the network. $\mathcal{L}$ is the measure of deviation of network's output from the target. Measures such as Mean Squared Error (MSE), Mean Absolute Error (MAE), Correntropy Criterion (CEC), are few popular cost functions used in regression. At next, gradients of the loss with respect to all weights and biases are computed starting from the last layer up to the first one. The partial derivative gives the measure on how quickly the cost changes when the respective weights or biases are changed. During the process, weights and biases are adjusted in the direction which minimizes the final loss, using some scaling factor μ known as the *learning rate*. The weights adjustment process starts at the last layer and is done back-wards towards the first.

2.4.1 Recurrent Neural Networks (RNN)

In contrast to FNNs, RNNs have forward as well as recurrent links. The forward links are same as of the conventional FNN, however, the recurrent links connect adjacent time-steps. There are also self connecting links, which connect same units across the time dimension, thus giving the notion of time to the model [21]. Figure 2.2 shows a schematic of a RNN model with one hidden layer; the model is unrolled in the temporal dimension.

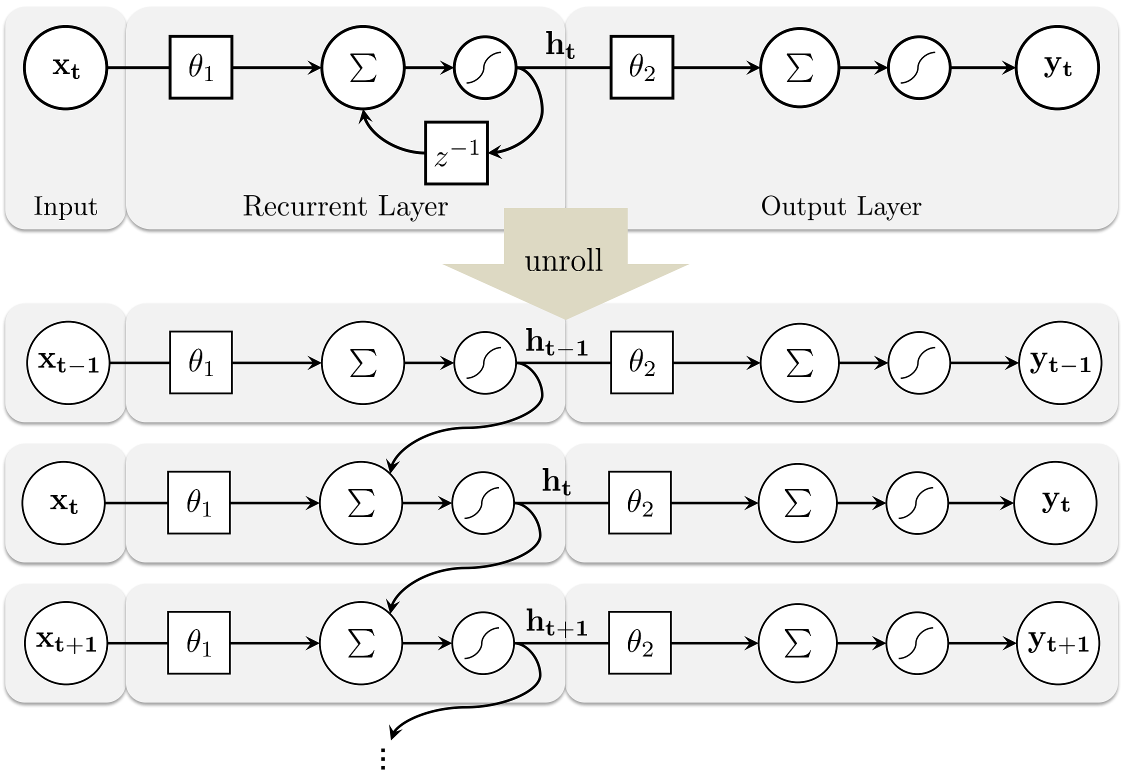


Figure 2.2: Standard RNN topology with one recurring hidden layer.

In analogy to the formulation of FNNs in Equation 2.2, the Equation 2.3 gives the output of an arbitrary recurrent layer at time-step t [21]:

$$\mathbf{h}_t = \mathcal{F}(\mathbf{W_{hh}}\ \mathbf{h}_{t-1} + \mathbf{W_{hx}}\ \mathbf{x}_t + \mathbf{b}) \tag{2.3}$$

where, $\mathbf{x}_t$ is the current input vector. $\mathbf{h}_t$ and $\mathbf{h}_{t-1}$ are the hidden node values at time-steps t and $t-1$, respectively. $\mathcal{F}$ is a logistic sigmoid function. $\mathbf{W_{hx}}$ is the matrix containing weights between the input and the hidden layer and the matrix $\mathbf{W_{hh}}$ contains recurrent weights between the hidden layer and itself in the adjacent time-step. Vector $\mathbf{b}$ contains the bias parameters. The recurrent layers therefore form a temporal dimension T where the history of input samples is saved. This ability of RNNs serves as a memory and allows the network to remember previously processed data and enables it to discover temporal correlations between samples that were observed over time [22].

Cost of the recurrent layer is computed as $J = \sum_{t=1}^{T} J_t$, where $J_t = \mathcal{L}(\hat{\mathbf{y}}_t, \mathbf{y}_t)$ is the loss at time-step t, and T is the total number of sample's history available to the network. Now, for the gradient based learning, gradients of the cost function is computed as [22, 23]:

$$\frac{\partial J}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial J_t}{\partial \theta}, \tag{2.4a}$$

$$\frac{\partial J_t}{\partial \theta} = \sum_{\nu=1}^{t} \Big(\frac{\partial J_t}{\partial s_t} \frac{\partial s_t}{\partial s_\nu} \frac{\partial^+ s_\nu}{\partial \theta} \Big), \tag{2.4b}$$

where, ${}^{\partial^+ s_\nu}/_{\partial \theta}$ refers to the immediate partial derivative of the state s_ν with respect to θ, ${}^{\partial J_t}/_{\partial \theta}$ is a gradient component which gives the measure of how θ at step ν affects the cost at time-steps $t > \nu$.

Exploding and Vanishing Gradients

Each recurrent layers in RNNs can be unfolded to be represented as a FNN, where each time samples act as an extra embedded layer. This is why a RNNs structure is very deep, therefore, learning to store information over extended period of time is slow [23]. Furthermore, in gradient based learning methods such as Backpropagation Through Time (BPTT) [24], the magnitude of the error signal propagating backwards in time depends exponentially on the magnitude of the weights [25], therefore, the back-propagating error either decays too fast and *vanishes* or grows exponentially large and *explodes*. Hence, due to this vanishing/exploding gradient problem, the conventional recurrent networks have difficulty in learning long-range dependencies in the training samples [21, 23, 26].

Many techniques have been introduced and practised to overcome this problem. Whether the gradient vanishes or explodes depends on the weighting factor of the link [21]. Training with the regularization term can suppress the weights in the recurrent links to values with which the vanishing or exploding of gradient do not occur [22]. Truncated Backpropagation Through Time (TBPTT) is another solution to the exploding gradient problem [21, 27]. In TBPTT, the error flow is controlled to a maximum time-step, such that, the gradient do not explode. However, by using TBPTT for training one cannot learn arbitrarily long-ranged dependencies. There are special model architectures with gate units such as LSTM, GRU, Phased LSTM, which are designed to handle this kind of shortcomings of the conventional RNN [13], and to learn long-range dependencies in the training samples.

2.4.2 Long Short-term Memory (LSTM)

LSTM networks are seen to be very good in variety of sequence processing tasks such as natural language processing, speech recognition, handwriting detection [28–30]. One of the main task within this thesis work is to investigate the capabilities of LSTM in temporal sensor data prediction. Figure 2.3 shows the schematic of the LSTM unit [28].

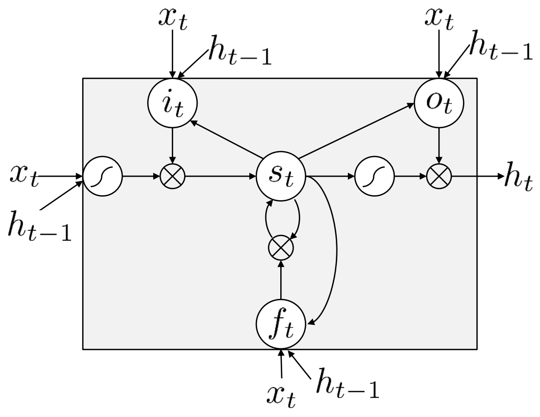


Figure 2.3: Schematic of a LSTM unit.

LSTM was introduced by Hochreiter and Schmidhuber in 1997 to overcome the problem of conventional RNNs to store information over long period of time due to insufficient error backflow i.e. vanishing gradient [31]. This is achieved in the LSTM networks by the use of three multiplicative gate units, namely, *input*, *output*, and *forget* gate. Each of these gates learn to open and close access to ensure that the error does not vanish. In addition, unlike RNNs which overwrites its content at each time-step, the update of the LSTM cell is regulated by these gates [32]. With the help of the forget gate, LSTM networks learn to *forget* irrelevant details in the training sample and at the meantime, they learn to *remember* useful feature informations. Open and close gates control access to the cell state vector. Furthermore, LSTMs are local in time and space; its computational complexity per time-step and weight is $\mathcal{O}(1)$ [31].

In Equation 2.5a-2.5e formulas for updating the state of a LSTM cell at time-step t are given. These formulas are analogous to the ones listed in [28].

$$\mathbf{i}_t = \mathcal{F}\big(\mathbf{W_{xi}}\ \mathbf{x}_t + \mathbf{W_{hi}}\ \mathbf{h}_{t-1} + \mathbf{W_{si}}\ \mathbf{s}_{t-1} + \mathbf{b_i}\big) \tag{2.5a}$$
$$\mathbf{f}_t = \mathcal{F}\big(\mathbf{W_{xf}}\ \mathbf{x}_t + \mathbf{W_{hf}}\ \mathbf{h}_{t-1} + \mathbf{W_{sf}}\ \mathbf{s}_{t-1} + \mathbf{b_f}\big) \tag{2.5b}$$
$$\mathbf{s}_t = \mathbf{f}_t \odot \mathbf{s}_{t-1} + \mathbf{i}_t \odot \tanh\big(\mathbf{W_{xs}}\ \mathbf{x}_t + \mathbf{W_{hs}}\ \mathbf{h}_{t-1} + \mathbf{b_s}\big) \tag{2.5c}$$
$$\mathbf{o}_t = \mathcal{F}\big(\mathbf{W_{xo}}\ \mathbf{x}_t + \mathbf{W_{ho}}\ \mathbf{h}_{t-1} + \mathbf{W_{so}}\ \mathbf{s}_t + \mathbf{b_o}\big) \tag{2.5d}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh\big(\mathbf{s}_t\big) \tag{2.5e}$$

Here, $\mathbf{i}$, $\mathbf{f}$, $\mathbf{o}$, and $\mathbf{s}$ are the state vectors for the input gate, forget gate, output gate, and the LSTM cell, respectively. All of these vectors have same length as of the hidden output vector $\mathbf{h}$. In analogy to the cell update equation of the conventional RNN cell given in Equation 2.3, $\mathcal{F}$ is also a logistic sigmoid function. The weighting matrix subscripts have the obvious meaning, for example $\mathbf{W_{hi}}$ is the matrix containing weights between the hidden and input gate, the matrix $\mathbf{W_{xo}}$ holds weights between the input and output gate, and so on. The weighting matrix between the LSTM cell and the gates are diagonal, such that an element p in each gate vectors ($\mathbf{i}$, $\mathbf{f}$, and $\mathbf{o}$) only gets input from element p of the $\mathbf{s}$ vector [28]. Similarly, vector $\mathbf{b}$ holds the respective biases. The current cell state $\mathbf{s}_t$ is updated with a fraction of the previous cell state controlled by the forget gate state $\mathbf{f}_t$ and the fraction of new state determined by the input gate state $\mathbf{i}_t$ (Equation 2.5c). Hidden output value is controlled by the open gate state $\mathbf{i}_t$ (Equation 2.5e). All vector multiplications are done element-wise and is denoted by the $\odot$ operator.

2.4.3 Gated Recurrent Unit (GRU)

GRU is a simpler derivative of the LSTM unit, proposed by K. Cho in 2014 [33]. Similar to the LSTM unit, the GRU unit also have gates to control the update of cell states, thus, they have the same goal of learning long-range dependences in the sequential data by reducing the vanishing/exploding gradient problem of the conventional RNN. Unlike LSTM unit, a GRU unit does not have separate input and forget gates but it has a *reset* gate and a *update* gate. When the reset gate is close to 0, the previous hidden state is ignored, thus allowing the cell to forget information which are seen to be irrelevant in the future [33]. The update gate controls how much information from the previous hidden state is used while computing the current hidden state. The schematic of a GRU unit is visualized in Figure 2.4.

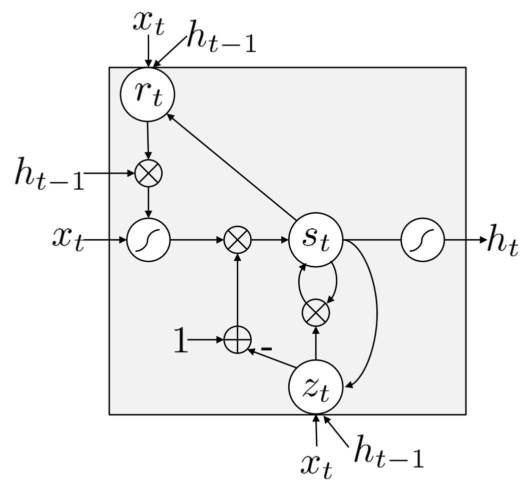


Figure 2.4: Schematic of a GRU unit.

In analogy to that of the LSTM cell, update equations for GRU are given below, which are analogous to the ones given in [33]:

$$\mathbf{r}_t = \mathcal{F}(\mathbf{x}_t\ \mathbf{W}_{\mathbf{xr}} + \mathbf{h}_{t-1}\ \mathbf{W}_{\mathbf{hr}} + \mathbf{b}_{\mathbf{r}}), \tag{2.6a}$$

$$\mathbf{z}_t = \mathcal{F}(\mathbf{x}_t\ \mathbf{W}_{\mathbf{xz}} + \mathbf{h}_{t-1}\ \mathbf{W}_{\mathbf{hz}} + \mathbf{b}_{\mathbf{z}}), \tag{2.6b}$$

$$\mathbf{s}_t = \mathbf{z}_t \odot \mathbf{s}_{t-1} + (1 - \mathbf{z}_t) \odot \tanh\left(\mathbf{W}_{\mathbf{xs}}\ \mathbf{x}_t + \mathbf{W}_{\mathbf{hs}}\ (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_{\mathbf{s}}\right), \tag{2.6c}$$

$$\mathbf{h}_t = \mathcal{F}(\mathbf{s}_t), \tag{2.6d}$$

here, $\mathbf{r}$ and $\mathbf{z}$ are the state vectors for the reset and the update gate, respectively and they have same length as of the hidden output vector $\mathbf{h}$. All other notations as well as the weighting matrix subscript convention in Equation 2.6 are analogous to the ones for the LSTM cell listed in Equation 2.5.

2.5 Related Work

LSTMs have been highly studied for sequence learning ever since Hochreiter and Schmidhuber introduced it in 1997 [31]. Major earlier work include the introduction of BPTT by Rumelhart [20] in 1985 and successful training of RNNs to perform supervised learning with sequential inputs and outputs by Elman in 1990 [34].

In [35], the conventional RNN is compared with the LSTM and GRU models in a sequence modelling task with music and raw speech signal data. The result of the paper show improved performance with the gated units, however, no concrete evidence is given on which of the gate units is better.

In [4], the author put LSTM network to the financial market prediction task and also show the backtest overfitting problem in time-series forecasting task. In [5], LSTM model is used to predict the sensor temperature for axle bearing of a railway. They train LSTM model using the history of temperature sensor readings along with other physical properties such as pressure, velocity of train, etc., and predict the temperature of axle at six different points. In [6], LSTM is used to model Sea Surface Temperature (SST), and predict future temperature given history to previously recorded SSTs. In [7], LSTM is used to predict diagnoses given Electronic Health Record (EHR) of patients. EHR data have varying length and are irregularly sampled, nevertheless, the paper show good result of LSTM model together with dropout [12] in classifying diagnoses of critical care patients given clinical time-series data. In [28], LSTM network is employed in text prediction, handwriting prediction and synthesis task. In [29], LSTM model is used for speech recognition. In [8], convolutional LSTM network is used for weather forecasting.

2.6 Tools and Softwares

In this section, tools and softwares used within this thesis work are mentioned in brief.

2.6.1 Wireshark

Wireshark is the network protocol analyser tool [36]. It is used to record the TCP/IP communication between MAIUS T-Stack and OBC. The recorded file is saved in packet capture (.pcap) format, which is later loaded in Python for constructing the dataset for neural network.

2.6.2 Python, Keras with Tensorflow Backend

Python is used for preparing dataset for the neural network training. The network itself is built in Python using Keras [37] library. Keras is a high level neural network API written in Python and is capable of running on top of Tensorflow [38] or Theano. Tensorflow is an open-source software library developed by Google for high performance numerical computation. It is one of the widely used in machine learning and deep learning tasks. Tensorflow API supports computations on CPU and GPU platforms.

In this thesis work, Keras v2.2.4 with Tensorflow v1.11 backend is used to build and compile all neural network models.

2.6.3 GPyOpt

GPyOpt [39] is a Python open-source library for Bayesian Optimization. It is based on GPy [40], which is a Gaussian processes framework in Python. GPyOpt performs global optimization of black-box functions using Gaussian processes. The python package is used to find optimum hyper-parameters of the neural network.

2.6.4 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) facilitates MBSD process by allowing source code as well as other artefacts generation directly from the user-defined data model using tools like Xtend. Data model in MAIUS is developed using EMF and is based on *Ecore* meta-model. Ecore is a subset of Unified Modelling Language (UML).

2.6.5 Xtext

Xtext is a open-source framework used to develop DSLs. It allows user to define grammar of the DSLs as an instance of the Ecore language. Xtext is build on the top of the EMF.

2.6.6 Xtend

Xtend is a high-level programming language and a dialect of Java. In the Xtext framework, Xtend is used for the code generation and model transformation. Artefacts generated from

MAIUS stack and card DSLs are used in the data preparation. The generator application which generates these artefacts are written in Xtend language.

2.6.7 High Performance Data Analytics (HPDA) Cluster

The training of all the neural network models are executed in the Graphics Processing Unit (GPU) node of the HPDA cluster of SC, DLR. The cluster is a Linux machine. The specification of the GPU node are as follows [41]:

- 2x Intel Xeon Scalable Processor "Skylake" Silver 4112, 2.60 GHz, 4-Core Socket 3647
- 192 Gigabyte Double Data Rate 4 (DDR4) Random Access Memory (RAM)
- 2x 256Gigabyte (GB) storage
- 2x 10GBase-T Ethernet ports
- Infiniband network card with Endpoint Detection and Response (EDR)
- 4x nVidia Tesla V100 graphics card

CHAPTER 3

Methodology

This chapter explains the methodology of the thesis work. The chapter introduces and elaborates different steps on which the thesis project is executed. It sheds light on the training-data preparation procedure, illustrates the neural network architecture, and other technical details.

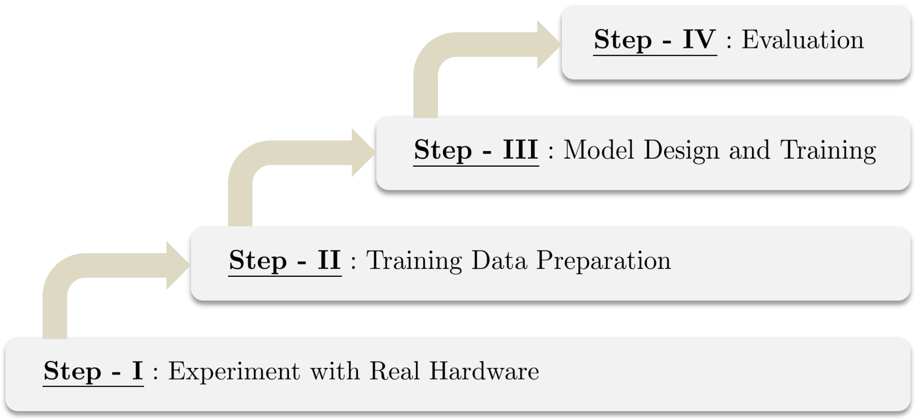

Step - IV : Evaluation
Step - III : Model Design and Training
Step - II : Training Data Preparation
Step - I : Experiment with Real Hardware

Figure 3.1: Different steps of the project phase.

The methodology of the thesis can be divided into four phases. These phases are illustrated as *steps* of the ladder structure in Figure 3.1. In Step-I, an experiment with the candidate hardware is performed and input-output data is collected. The acquired data is analysed in Step-II, from which the datasets to train the RNN are prepared. In Step-III, the model architecture is designed and the hyper-parameters are optimized. Different model instances are trained using the optimum configuration obtained from the hyper-parameter search. Step-IV is the last step where all model instances are evaluated. In the following section, these project phases are explained in detail.

3.1 Experiment with Real Hardware

The learning process and the network architecture depend on the amount, type, and dimension of the available dataset, therefore data collection and preparation is done before the learning begins.

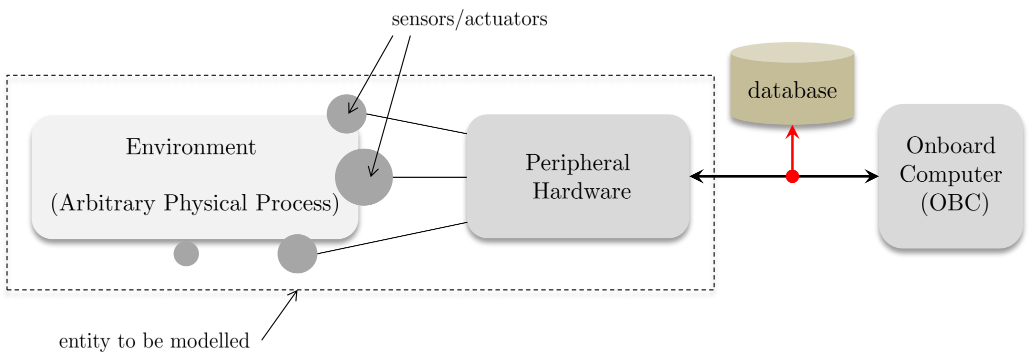


Figure 3.2: Setup of the experiment with the real hardware.

In order to collect data for training, an experiment is performed with the real hardware as shown in Figure 3.2. The entities in the dotted region are the candidate to be modelled by the neural network. Arbitrary sensors and actuators are denoted by dark-grey circles. The environment is where the physical process happens. The peripheral hardware provides interfaces for the external components such as actuators and sensors. Actuators are used to make changes in the environment and manipulate the ongoing physical process. Sensors take measurements and give the measure of the effect caused by the manipulation. The hardware accepts commands from the central processing unit hereby allowing access to its channel values. The central processing unit can be a OBC or any kind of a computing unit. The hardware component along with all the actuators and sensors, as well as the ongoing physical processes in the environment are taken as a whole to be modelled together by the neural network.

The experiment as seen from the data-space perspective is basically random read/write commands sent to the peripheral device via OBC. A write command sets the state of an actuator, whereas, a read command gets the reading of a sensor. The process of randomly dispatching commands is automated, thus it can be executed for arbitrary period of time. During the execution of the experiment, the communication between the peripheral device and the OBC is recorded into a database from the communication channels, which is done once the message has been dispatched by the peripheral device and the OBC. So, the data-collection procedure do not interfere with the communication between the peripheral device and OBC. Thus, the data collection procedure does not demand changes on the OBSW, therefore, it is in compliance to requirement *req_04* and *req_05*.

The experiment is configured in such a way that dependencies between the channels of the peripheral hardware are created. Actuators states are changed from outside of the system (by the user via OBC), thus, they are the independent variables of the system. Sensors readings depend on the state of the system, hence, they are the dependent variables. The RNN as a supervised learning method learns the mapping between these dependent and independent variables in time (covers requirement *req_02*).

The physical process is not defined yet; the experiment is generic and can be replaced by any other in the future. Moreover, the dimension of the experiment is generic as well. At this point, it is assumed that an arbitrary number of sensors and actuators can be involved in the experiment. Details on the sensors and actuators used to test this hypothesis are given in Section 4.2.

3.2 Datasets

The learning process of the neural network depends on the dataset. In order to ensure efficient learning, dataset should only contain features which are relevant to the underlying problem, otherwise the learning process may deviate heavily from the solution resulting in poor or no convergence. In theory, dataset can contain arbitrarily large number of features. Carefully selecting and refining features helps to optimize the learning process. The bound and scale of individual features also affect the learning process. If the features-scaling is not done correctly, the network will end up emphasizing some feature more than others [42], thus the effect of some features can be completely lost. In addition to this, poorly scaled features demand very low learning rate, thus resulting in slow convergence. Lower learning rate also increases the risk of convergence to a local optimum.

In the supervised learning method, the model learns to map an input to a known target, therefore, the input data and target data has to be aligned perfectly otherwise the whole learning process breaks down. In addition, the structure and dimension of the input and target data should also be carefully engineered, as they affect many aspects of the learning process. The shape and dimension of the input data determine the shape and dimension of the input layer of the neural network, and therefore, also affect that of the hidden layers. Similarly, the shape and dimension of the output layer should be same as of the target data. Furthermore, the number of features in the input affects the number of units in the input layer, thus affecting the breadth of the network and thereby the computational cost. Moreover, values of the hyper-parameters also depend on the dataset. Important hyper-parameters are explain in section 3.4.

The byte-stream collected in the previous step is filtered to find the actuators and sensor values from which the input/target matrices as illustrated in Equation 3.1 and 3.2 are constructed. Each column vectors of these matrices represent a sample. Since, supervised learning requires to have a target for each input, both matrices have same N number of columns. Altogether, there are P number of features in an input sample which map with Q number of targets. Since, actuator states are independent variables they are the input features, similarly, sensor reading are the dependent variables which are to be predicted, thus they are the targets. Since, time is crucial in modelling a physical process, in addition to actuator's states, the time-interval between two samples is collected from the communication protocol and is included as a extra feature and takes the first position in the input column vector. The time-interval value act as an excitation signal for the neural network model when all actuators are in the same state for a long period of time.

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} & \cdots & x_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{p,1} & \cdots & x_{p,n} & \cdots & x_{p,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{P,1} & \cdots & x_{P,n} & \cdots & x_{P,N} \end{pmatrix} \tag{3.1}$$

$$\mathbf{Y} = \begin{pmatrix} y_{1,1} & \cdots & y_{1,n} & \cdots & y_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ y_{q,1} & \cdots & y_{q,n} & \cdots & y_{q,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ y_{Q,1} & \cdots & y_{Q,n} & \cdots & y_{Q,N} \end{pmatrix} \tag{3.2}$$

The process of finding the actuators and sensors values in the byte-stream is *target-driven*, meaning, each input and target column vectors are constructed when values of all sensors for one time step are found. Readings of all sensors are taken in a fixed interval, thus, this interval resembles a time step. The values of actuators are updated as they are found, otherwise their last state is maintained.

3.2.1 Normalization

Next step of the data preparation for the neural network training is normalization or scaling of feature and target values or *data standardization*. When features are not properly scaled, back-propagation emphasizes some features over the other, which result in some features contributing to the weight update more than the other, hence, resulting in a higher error rate [42].

Different data normalization techniques can be used, for examples, data could be normalized between 0 and 1, or -1 and 1. These normalization techniques are known as *linear transformation* [42]. Another popular technique is that *statistical standardization*, which transforms the data to have a zero mean and unit variance [42]. This is achieved by applying Equation 3.3 to all features and target values.

$$x_{p,n} = \frac{x_{p,n} - \bar{x}_p}{\sigma_p} \tag{3.3}$$

Here, $\bar{x}_p$ and σ_p are the mean and standard deviation of the feature p, respectively. Figure 3.3 shows the example histogram of the transformed data using the statistical standardization method. The red curve represent the probability distribution function computed from the histogram i.e. the integral of this curve is equal to one. This technique provides statistical bounds to the feature and target values, in return, it results in smaller weights in the neural network layers which ultimately improves the learning process by stabilizing the gradient descent process [42].

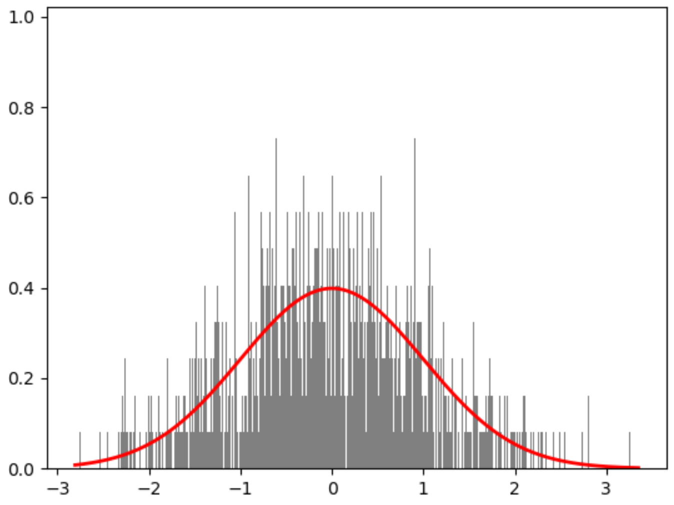


Figure 3.3: Histogram of a example data transformed using statistical standardization method. The red curve shows the probability distribution function of the underlying data.

In this study, input as well as target datasets are normalized to have a zero mean and a variance of one in the sample axis i.e. each features and target values are normalized independently using Equation 3.3. All further steps are preformed in the normalized dataset. Since, the neural network is trained with the normalized data, the output produced by it is also normalized one. Therefore, target values predicted by the network should be denormalized back to their proper representation using the same normalization parameters. For this reason, the normalization parameters: mean and standard deviation of each actuator values and sensor readings are saved.

3.2.2 Input Data

The input data is fed into the network. It must have same shape as of the first layer in the network. So, unless it is intended that the input is one dimensional, the input matrix constructed in the earlier step must be reshaped according to the desired input dimension of the first layer.

The RNN networks take a temporal sequence as an input. Therefore, the input matrix created from the channel values is reshaped to facilitate the time-dimension of length T. Moreover, RNN layers such as LSTMs, GRUs in Keras framework requires the temporal dimension to be in the second axis of the input, such that the required input shape is $[N \times T \times P]$.

The temporal dimension is created by implementing *look-back* in the input samples, meaning, the previous $T-1$ input vectors are included in a input sample. For the first $T-1$ input vectors, the oldest vector is repeated $(T-n)$ times, where n is the sample index. Equation 3.4 shows the input data as a 3-dimensional array, each elements in the first axis is a sample of shape $[T \times P]$ – there are N number of elements in the first axis. In order to maintain similarity with input matrix representation from Equation 3.1, each samples in Equation 3.4 are shown as the *transpose* of $[P \times T]$ matrix. Hence, the required shape of the input data i.e. $[N \times T \times P]$ is achieved.

$$\left[\underbrace{\begin{pmatrix} x_{1,1} & \cdots & x_{1,1} \\ \vdots & \ddots & \vdots \\ x_{p,1} & \cdots & x_{p,1} \\ \vdots & \ddots & \vdots \\ x_{P,1} & \cdots & x_{P,1} \end{pmatrix}^{\top}}_{\text{1st sample}}, \cdots, \underbrace{\begin{pmatrix} x_{1,N-T+1} & \cdots & x_{1,N} \\ \vdots & \ddots & \vdots \\ x_{p,N-T+1} & \cdots & x_{p,N} \\ \vdots & \ddots & \vdots \\ x_{P,N-T+1} & \cdots & x_{P,N} \end{pmatrix}^{\top}}_{N\text{th sample}}\right] \tag{3.4}$$

Input data is divided into three datasets from the first axis (N dimension) to get the training, validation, and the test set. Training set consists of the first 70% of the samples, validation set consists of the next 20%, and the remaining 10% of the samples are separated for the testing purpose. The validation set is used during training to compute the loss such that the weights are adjusted to minimize the error on the validation data and not on the training data. This reduces the over-fitting of the network as the validation set is not used to train the network, thus simulates the test scenario. Over-fitting is the condition, when the network learns irrelevant details from the training set. The test set is unused during training and is only used to evaluate the model after the training is over.

3.2.3 Target Data

Each $[T \times P]$ shaped input sample produces an output of shape $[Q \times 1]$, therefore, there is no need to extend the dimensions of the target matrix from Equation 3.2, however, it needs to be reshaped by taking a transpose, so that the length of the first dimension is equal to batch size N. Hence, the target data has shape $[N \times Q]$. In analogy to the input data, target data is also divided into three datasets keeping the alignment with the respective input set.

3.3 Model Design and Training

Within this study, two RNN topologies each with the LSTM and GRU memory units are trained and evaluated separately. All network topology follow a simple design pattern which basically has four *block* of layers. The first and last *neural-blocks* consist of one feed-forward layer each, which are the input and output layers, respectively.

The second one is the *memory-block*, which is composed of recurrent layers. The LSTM and GRU as recurrent layers in the memory block make two instances of a model architecture. Two network architectures are developed within this study, which are described in Sections 3.3.1 and 3.3.2. Each recurrent layer is followed by a Batch Normalization (BN) layer. While training deep neural networks, the distribution of each layer's inputs changes, this phenomenon is referred as *internal covariate shift* [43]. This effect slows down the training significantly, requires lower learning rate, and carefully engineered weight initialization techniques [43]. BN layer address this problem by allowing normalization to be a part of the model architecture and performs normalization for each mini-batch during training. BN of an arbitrary input x_i over a mini-batch is given by y_i and is computed as [43]:

$$\bar{x}_{\mathcal{B}} = \frac{1}{b} \sum_{i=1}^{b} x_i \tag{3.5a}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{b} \sum_{i=1}^{b} (x_i - \bar{x}_{\mathcal{B}})^2 \tag{3.5b}$$

$$\hat{x}_i = \frac{x_i - \bar{x}_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon_{BN}}} \tag{3.5c}$$

$$y_i = \gamma_{BN}\ \hat{x}_i + \beta_{BN} \tag{3.5d}$$

Where, $\mathcal{B} = \{x_1, \ldots, x_b\}$ is the mini-batch for training. $\bar{x}_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ are the mini-batch mean and variance, respectively. $\hat{x}_i$ is the normalized value and y_i is the same after scaling and shifting it by γ_{BN} and β_{BN}, respectively. Parameters γ_{BN} and β_{BN} are learned during training. ϵ_{BN} is a fixed small value used to avoid singularities.

The third block of the network topology is composed of time-distributed dense layers. Dense layers are composed of feed-forward units and are fully connected in sequence axis T. Each of the dense layers are followed by a Dropout layer and a BN layer. Dropout layers address the over-fitting problem by randomly removing units from the layer along with their weights. The randomness is controlled by a probability factor referred as the *dropout* *rate*. The dropout rate is a hyper-parameter which is further discussed in Section 3.4.9. There is a Flatten layer between the third and last block. Flatten layer is a transition layer whose sole function is to reduce input dimension to one and has no trainable parameters. In RNN, the input has two dimensions (time and feature) but the target only has one dimension, thus flattening is needed.

3.3.1 One-to-One Architecture

One-to-one model architecture has one input layer and one output layer, the shape of these layers are $T \times P$ and $Q \times 1$, respectively. The Figure 3.4 illustrates the one-to-one network topology in its general form. The structure of the memory and time-distributed dense block are shown on the right-hand side (sub-block). A variable number of these sub-blocks are concatenated to form the respective blocks. The number of sub-blocks and units in these layers are determined by the hyper-parameter search.

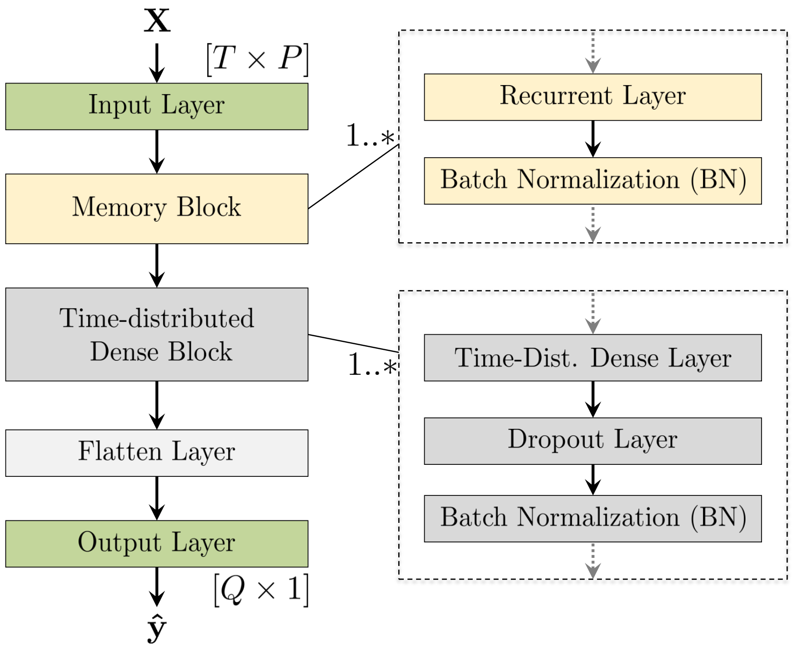


Figure 3.4: One-to-one network architecture – network topology with one input and one output layer.

3.3.2 One-to-Many Architecture

The one-to-many architecture has only one input layer but has multiple output layers. The model branches in the middle to facilitates these multiple output tails. Number of such *neural branches* are equal to the number of targets. The Figure 3.5 shows the one-to-many network architecture. The neural blocks are analogous to the ones from the one-to-one architecture, however their internal structure as well as the hyper-parameter values may differ.

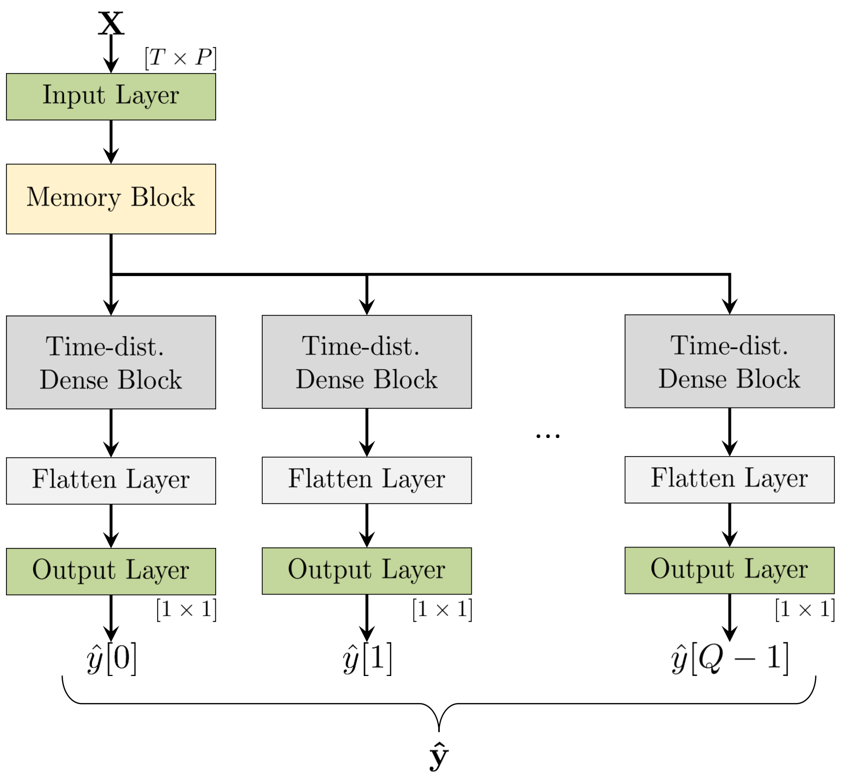


Figure 3.5: One-to-many network architecture consisting of one input layer and multiple output layers.

Generally, neural networks are used with similar type of output data, such as labels, images, speech-signals, etc. Since, the goal of this study is to model a physical system consisting of multiple sensors and actuators with various signal types. It is a unique neural network application. Empirical method show that producing different ranges of values through a single output layer is difficult, mainly because, the output behaviour differ so much that the model optimized to produce continuous output signal fails to simultaneously produce the step-function and vice-versa.

The one-to-many neural architecture developed within this study allows the customization of the tail of the network for each target outputs separately, hence, predicting variety of signals at once, without having to compromise on the prediction accuracy, becomes possible. The neural branches are independent of each other in their structure and hyper-parameter values. Further benefit of the branching structure of the model is that each branch could be trained separately. It further opens the door of concurrent training where each branches are trained in parallel. However, within this study, all neural branches are trained in a single process. Parallel training shall facilitate efficient training of a significantly large network in the future. Moreover, the branching is done after the memory-block which allows the model to share the RNN layers between different neural branches, hence the weights in the memory block are shared.

3.3.3 Training the Network

Two instances of each model architecture described earlier are trained separately, once with the LSTM as the memory unit, and the other time with the GRU memory unit. Optimum hyper-parameter values are used to design these models and the training sets defined in Section 3.2 are used to train them. The models are trained in the GPU node of the HPDA cluster.

During the training, $\mathcal{L}\{CEC\}$ in the validation set is minimized. The formulation of $\mathcal{L}\{CEC\}$ is given in Equation 3.11. All models are trained for about 10 epochs. Empirical method show all models already converge sufficiently at the 10th epoch, training longer causes the model to over-fit to the training set and result in high error on the test set. In all training sessions, the mini-batch size of 256 is used. The value is also chosen such that the model do not over-fit to the training set. Furthermore, the initial learning rate of all training sessions is fixed to 0.002. This value show good result in all types of target signals and model types. The learning rate decays by 0.02 factor after each weight update. An update happens when a mini-batch of sample is passed through the network.

Callback functions from Keras are used to assist the training process of all model instances. These functions are called at the end of each epoch of training. The following callback methods are used:

Early Stopping

Although the training is said to be iterated *epoch* times, it can be terminated earlier in case the performance stops improving [44]. For this purpose, a built-in callback function from Keras *Early Stopping* is used. Validation loss is monitored, if it does not decrease for 4 consecutive epochs the training session is terminated. Early stopping helps to prevent over-fitting of network by not letting it to learn irrelevant details from the training set.

Reduce Learning Rate on Plateau

This callback method also monitors validation loss, if no improvement is seen for 2 consecutive epoch learning rate is reduced to 20% of the original learning rate. 2 more epochs of training is conducted using the reduced learning rate, if the performance does not improve, the training session is terminated.

Model Checkpoint

This callback method provide an option to save optimum weights during training, such that, not the last set of weights but the optimum one among all epochs is used in the evaluation phase. Validation loss is monitored, set of weights saved in a file only when the validation loss is smaller than the previously saved epoch.

3.4 Hyper-Parameters

Deep learning with neural networks consists of the optimization of a complex function which takes an input and give a predicted output. There are many parameters involved in this process such as, number of epochs to train, mini-batch size, learning rate, etc., which are not selected by the network itself. In order to get full advantage of the network architecture and the used dataset, optimum values for these parameters must be found and fixed before the network could make meaningful predictions [45]. These parameters are generally referred as *hyper-parameters* of the neural network. Different hyper-parameters affect different aspect of the neural network model. The Table 3.1 summarize them in a list, where different hyper-parameters are given in the row and in the column, different aspects of the model which are affected by tuning these parameters are marked.

#	Hyper-parameter	Stability	Convergence Speed	Over-fitting	Under-fitting	Computation Cost
1	Cost Function	x	x			
2	No. of Layers		x	x	x	x
3	No. of Units		x	x	x	x
4	No. of Epochs			x	x	x
5	Mini-batch Size			x		x
6	Activation Function	x	x			
7	Learning Rate	x	x			x
8	Kernel Initializers	x	x			
9	Kernel Regularizer			x		
10	Dropout rate			x		
11	BN Parameters	x	x	x		

Table 3.1: Different hyper-parameter affect different aspects of the neural network model.

Hyper-parameter search is mostly driven by the requirement *req_07* (see all requirements in Table 1.1), which ensures the model to work well with the unseen data in the test phase. Poor performance of the model in the test phase is mainly due to over-/under-fitting of the neural network model. Hyper-parameters such as number of layers, number of units, training epochs, mini-batch size, regularization factor, dropout rate, early-stopping criterion, etc. address these problems.

There are several hyper-parameters is a neural network, adjusting all of them to their optimum value for the given problem is one of the most difficult task in deep learning with neural networks. Thus, hand-crafting them takes lot of time and is inefficient, therefore the search process needs to be automated. In this study, the Bayesian optimization algorithm is used to find the optimum hyper-parameters of the neural network model. The algorithm is much faster in comparison to grid-search and random-search algorithms [46, 47]. For the automatic optimization process, CEC is maximized. Not all hyper-parameters are optimized using the Bayesian optimization process; due to time constraints and logical reasons, some of them are done empirically. Detail on the usages of the algorithm is explained in Section 4.6. The optimization algorithm itself is not within the scope of the thesis work.

Nevertheless, due to large number of tunable hyper-parameters of the RNN models, only course-grain search is made which give fairly good result. Fine-grained search of these parameters is the work to be done in the future. The hyper-parameter value which showed good result in similarity metrics on the test set are selected. In the following sub-sections, the most important hyper-parameters are explained and their values are given.

3.4.1 Cost Function

During the training, the model predicts outputs for each input samples in the training set. Then, the cost function is computed, which measures the error of the predictions. Then, the learning method modifies the weights in the network to minimize the error. In order to modify the weight values properly, gradients of the cost/error function are computed with respect to each weights in the network. These gradients indicate by what amount the error would increase or decrease if the weight is increased by a small amount. Then, the weights are adjusted to the negative direction to the gradient vector, such that the error is minimized. The Figure 3.6 shows the gradient descent to minimize error $J(\Theta)$; $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{\mathcal{H}}\}$ is the set of weights and biases in the network, where θ_1 is the matrix containing the weights and biases between input layer and the first hidden layer, similarly, matrix $\theta_{\mathcal{H}}$ contains weights and biases between $(\mathcal{H}-1)$th and $(\mathcal{H})$th layer.

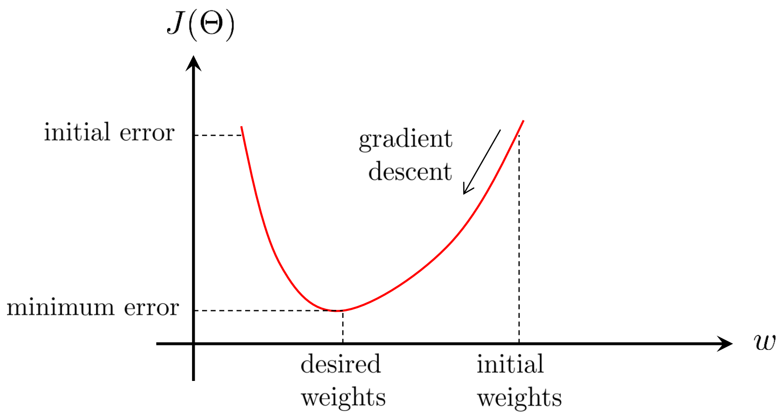


Figure 3.6: Gradient descent to minimize error [48].

The cost function when averaged over all samples in the training set have kind of a hilly surface in the high-dimensional space of weights [13]. Thus, a negative gradient direction indicates the direction of the steepest descent in the hilly surface moving towards the bottom, where the average prediction error is lowest [13]. The weight update for an arbitrary weight w is computed as:

$$\Delta w = -\mu \frac{\partial J(\Theta)}{\partial w}, \tag{3.6}$$

where, μ is the learning rate. Then, the weight update at iteration ξ is given by: $w(\xi+1) = w(\xi) + \Delta w(\xi)$. The partial derivative $\frac{\partial J(\Theta)}{\partial w}$ is computed for all weights in the network.

Three different cost functions are compared, formulation of these measures are given below:

- **Mean Squared Error (MSE)** is one of the popular measure for quantifying regression models [49]. It is easy to understand and also easy to compute. MSE is computed by squaring the difference between each samples in $\mathbf{y}$ and $\hat{\mathbf{y}}$ and finally averaging them such that the measure is a scalar.

$$MSE = \frac{1}{N}\sum_{n}^{N} (y_n - \hat{y}_n)^2 \tag{3.7}$$

  MSE penalize large errors more (squaring the difference) make it very sensitive to outliers, which is one of the drawback of this measure. Moreover, MSE is not within the scale of the data, therefore, common practice is to square the result to compute another metric, namely, Root Mean Squared Error (RMSE).

- **Mean Absolute Error (MAE)** is not quadratic as MSE. It is computed by averaging the absolute difference between each samples in $\mathbf{y}$ and $\hat{\mathbf{y}}$ as following:

$$MAE = \frac{1}{N}\sum_{n}^{N} \text{abs}(y_n - \hat{y}_n) \tag{3.8}$$

  Thus, MAE treat large and small error the same way and do not increase the penalty when the error is large.

- **Correntropy Criterion (CEC)** is a generalized correlation function which is getting popular in signal processing and machine learning [49, 50]; maximization of correntropy has increasingly been used to replace the minimization of MSE in training regression models. Correntropy is a non-linear and local similarity measure between two signals [50]. The locality of CEC is given by the bandwidth of the kernel function [50]. Unlike MSE, correntropy are insusceptible to outlying predictions. It is computed as:

$$CEC = \frac{1}{N}\sum_{n}^{N} \kappa_\lambda(y_n,\ \hat{y}_n), \tag{3.9}$$

  where, $\kappa_\lambda(\cdot,\cdot)$ is the Gaussian kernel function, which is defined as:

$$\kappa_\lambda(y_n,\ \hat{y}_n) = \exp\Big(-\frac{(y_n - \hat{y}_n)^2}{2\lambda^2}\Big), \tag{3.10}$$

  where, λ is the bandwidth of the Gaussian kernel. $CEC = 1$ represent perfect match of two sequences, on the other hand, $CEC < 1$ represent the mismatch. When used as a loss function, Equation 3.9 is modified as following:

$$\mathcal{L}\{CEC\} = 1 - CEC \tag{3.11}$$

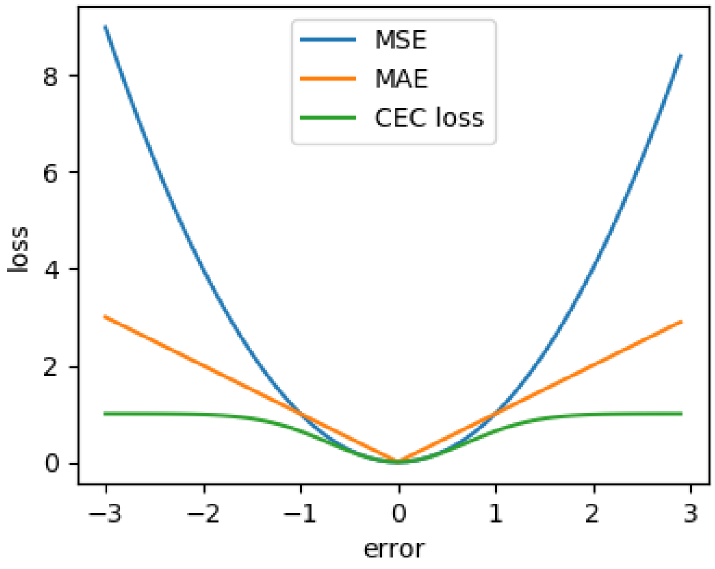


Figure 3.7: Candidate loss functions.

These three candidate loss functions are shown in Figure 3.7. Separate training sessions are conducted using each of them, and their predictions on the test-set are observed. Worst performance is seen when the model is trained using MAE as the cost function. As also illustrated in Figure 3.7, MSE increases rapidly when the error is large, thus it is very sensitive to occasional outlying predictions. The CEC loss on the other hand, performs as MSE when the error is small, and saturates about 1 when the error is large, thus its performance is much stable. The saturation rate of the CEC loss is determined by the kernel bandwidth, denoted by λ in Equation 3.10. Large λ value result in slow saturation. In all calculations, $\lambda = {}^{1}\!/\!\sqrt{2}$ is used; this value of is λ obtained empirically. Hence, $\mathcal{L}\{CEC\}$ is minimized to train all types of models.

3.4.2 Hidden Layers and Hidden Units

Conventionally, every layer between input and output of a neural network are referred as hidden layers. The number of hidden layers and the number of unit in each of those layers determine the depth and breadth of the network, respectively, therefore they directly correspond to the complexity and the computation cost of the model. A large number of hidden layers with many units in them result in very deep and wide network. Efficient training of such large networks is very difficult and requires enormous amount of training samples. Large networks are able to learn irrelevant details from the training samples, and hence, result in overly fitted model. Also, back-propagation is slow in large networks, as there are large number of weights to be adjusted. Furthermore, when gradients flow through very deep network, they tend to either grow very large and explode causing instability in the model. Or, decay excessively and vanish, which causes loss of valuable information and no learning becomes possible. Large networks are also computationally expensive to train.

On the other hand, less number of hidden layers with inadequate number of units result in shallow network and result in under-fitting of the model. Under-fitting is the opposite of over-fitting of neural networks where the network is too simple and cannot properly model the underlying problem. Thus, adequate number of hidden layers and number of units in them are hyper-parameters which need to be tuned. The number of layers and the number of units in each of the one-to-one and one-to-many models are given in Table 3.2. These values are obtained from the hyper-parameter search.

Architecture	**Output Type**	**No. of Layers × No. of Units**	
		LSTM/GRU	**Time-distributed Dense**
One-to-one	All	1×50	5×100
One-to-many	Step Function	1×50	2×60
	Constant		2×50
	Continuous		3×60

Table 3.2: Number of layers and units in different neural blocks of two different network architectures.

3.4.3 Training Epochs

One epoch of training is said to be finished when the whole batch of samples (N_{train} number of samples) from the training set are passed through the network. Usually, these samples of data are passed through the network several time during training. The number of epoch to train depends on the complexity of the network, complexity of the underlying problem, and the type and amount of samples available for training.

Number of epochs to train also addresses the under-/over-fitting problems of the model. Training very less number of epoch may not be sufficient to update all the weights to their optimum values – under-fitting of the network. On the other extreme, training for large number of epochs results the model to fit the training data so well that it performs poorly in unseen data – over-fitting of the network.

Number of epochs for training is obtained empirically by monitoring the validation loss. Saturation point of the validation loss is seen around ≈ 10 epochs for all models.

3.4.4 Mini-batch Size

In the stochastic learning method, in each update step, weights are adjusted by averaging the gradients of small number of samples from the training set [13]. This small batch of samples are referred as the *mini-batch*. One epoch of training consist of many number of mini-batches. The size of the mini-batch also addresses the over-fitting problem. When the mini-batch size is large, weights are updated only after looking into so many samples from the training set that it fits them better – over-fitting of the model. Too small mini-batch size result in a large number of weight update per epoch, which is obviously slow. Moreover, updating weights too frequently without sufficient examples results in convergence to a local minima resulting in high error. Models are trained with different mini-batch size from 32 to 512, each time doubling the size. Maximum CEC on the test set is achieved when the mini-batch size is 256.

3.4.5 Activation Function

In each layer of the neural network, the output is computed by a linear combination of its input and weights. In most cases, the underlying problem is very complex and can not always be modelled by a linear system, therefore, non-linearity has to be introduced in each layer's output, this is done by evaluating the layer's output by a non-linear function, commonly refereed as an activation function.

There are many kinds of activation functions. Following are the formulation of the ones used in the models:

- Sigmoid: $\mathcal{F}(h_\jmath) = \frac{1}{1+e^{-h_\jmath}} \in [0, 1]$
- Tangent Hyperbolic (Tanh): $\mathcal{F}(h_\jmath) = \tanh(h_\jmath) = 1 - \frac{2}{1+e^{-2h_\jmath}} \in [-1, 1]$
- Rectified Linear Unit (ReLu): $\mathcal{F}(h_\jmath) = \max(0, h_\jmath)$
- Linear: $\mathcal{F}(h_\jmath) = h_\jmath$

where, $h_\jmath$ is the output of $\jmath$th neuron of an arbitrary layer.

Activation function of a layer is chosen according to the range of its input as well as the desired output. The ReLu function facilitates smooth gradient flow [13], therefore it is used in all time-distributed dense layers. However, the ReLu function do not bound positive values, therefore, when used in a recurrent layer, the recurring activation grows exponentially which cause instability in the network; this behaviour is observed by empirical methods. Therefore, tanh function is used to activate forward signals, and Sigmoid function is used in the recurring signals of all recurrent layers in both model architectures.

In regression, the basic idea is to disfigure the input by the hidden layers in a non-linear way such that linear discrimination is possible in the output layer [13], therefore, a linear activation function $\mathcal{F}(h_\jmath) = h_\jmath$ is used in the output layer.

3.4.6 Optimizer

In this study, Adam, a first-order gradient-based stochastic optimization algorithm introduced by Kingma in 2014, is used as the optimization function. The algorithm calculates an exponential moving averages of the gradient and the squared gradient [51]. The moving averages are estimates of the first moment (the mean) and the second raw moment (the un-centred variance) of the gradient [51]. The algorithm then uses these to compute individual adaptive learning rates for different weights. Parameters β_1 and β_2 control the decay rate of these moving averages [51].

Instead of looking for other learning methods, more focus is made on optimizing hyper-parameters for Adam. Hyper-parameters for Adam optimizer include initial learning rate μ, learning rate decay μ_{decay}, moment decay rates (β_1 and β_2), and epsilon ϵ_{adam}.

In all training sessions, $\mu = 0.002$ with $\mu_{decay} = 0.02$ is used. These values are obtained by the Bayesian optimization process. The moment decay rates, $\beta_1 = 0.9$ and $\beta_2 = 0.999$, are left to their default values as given in the Keras package.

3.4.7 Kernel Initializer

Neural network trained using stochastic learning algorithm start with randomly initialized weights on all layers, optimum values for weights are searched through back-propagation during training. The performance of the learning algorithm based on the stochastic gradient descent (Adam), therefore, depends on the initial weights [52].

Moreover, it is important to initialize them with a random distribution, when initialized with a same value (zeros/ones) all weights update would be same therefore no learning is possible. Also, when initialized with a random distribution any particular feature is not favoured at the beginning and therefore learning variety of features becomes possible.

Different types of initializers *glorot normal* [53], *glorot uniform* [53], *orthogonal* are compared by monitoring the convergence rate. In order to ensure reproducibility of the result, a constant seed value 1337 is used for all weight initializers. This seed value is obtained empirically, and also by monitoring the convergence rate.

Hyper-parameter search result show fastest convergence when the weights of forward signals in the recurrent layers are initialized with a uniform distribution with variance scaled by a factor of 2, *glorot uniform* [53]. For the weights of the recurring signals in the memory blocks and for that of the time-distributed dense blocks, orthogonal initializer showed faster convergence.

3.4.8 Kernel Regularizer

Kernel regularization is one of the techniques to fight the over-fitting problem in neural networks. The idea here is to penalize large weights so that they do not dominate others. This is done by modifying the cost function as:

$$J(\Theta) = J(\Theta) + \epsilon_2 \|\mathbf{w}\|_2, \tag{3.12}$$

where $\|\mathbf{w}\|_2$ is the L_2 norm of the all-weight vector $\mathbf{w}$. Vector $\mathbf{w}$ contains all weights in Θ (see Section 3.4.1). ϵ_2 is the regularization factor for the L_2 norm.

Weight update Equation 3.6 is then modified as:

$$\Delta w = \Delta w - \mu \epsilon_2 w \tag{3.13}$$

A large ϵ_2 result in harsh penalty causing loss of important feature information, on the other hand, too lower values might be ineffective or insufficient penalty allowing larger weights to be still dominating. Thus, the regularization factor ϵ_2 is an important hyper-parameter which needs to be optimized. The Table 3.3 show the values for the kernel regularization in each layers. The two comma-separated values of the LSTM and GRU layer are the regularization values of the forward and recurring signals, respectively. The regularization of the recurring signal is 0.99 which is very harsh, this is required to prevent the recurring activations from growing exponentially large. High recurrent regularization also prevents over-fitting of the RNN model.

Architecture	**Output Type**	**Kernel Regularization**	
		LSTM/GRU	**Time-distributed Dense**
One-to-one	All	0.4, 0.99	0.6
One-to-many	Step Function	0.4, 0.99	0.4
	Constant		0.4
	Continuous		0.9

Table 3.3: Kernel regularization parameters in different neural blocks of two different network architectures.

3.4.9 Dropout

Dropout is another technique to address the over-fitting problem of the neural networks. The key idea of dropout is to randomly drop units along with their connections from the network during training [12]. The dropping of weights is done randomly, controlled by a probability parameter p. This is an effective way to significantly reduce over-fitting and to improve regularization [12], however, dropout causes random loss of information, therefore large dropout are avoided. Hyper-parameter search is performed to find the appropriate value of p for all layers in the model. The dropout rate of 0.3 is used in all time-distributed dense layers and no dropout is used in the recurrent layers.

3.5 Evaluation Measures

Qualitative as well as quantitative analysis are performed to evaluate all models. Evaluation metrics defined in this section are used to make evaluation of the models in Chapter 5.

3.5.1 Qualitative Analysis

Qualitative analysis of neural network models already begins in the training phase. During training, the loss function as well as other evaluation metrics are computed at the end of each epoch. This is done using the training set as well as the validation set. When the training session is over, losses and metric values are plotted against epochs of training. By observing these graphs, especially the loss on training and validation set, one can make decisions about the efficiency and sufficiency of the training session as well as the quality of the trained model. Following observation are made via empirical methods:

1. **Training loss $J(\Theta)_{train}$ and validation loss $J(\Theta)_{vali}$ are undefined or very large.** This could either mean that the dataset is not correctly normalized; the learning algorithm is not appropriate for the dataset; or the learning rate is too high. In summary, the network is not learning at all. Different techniques such as normalizing the dataset differently, using smaller learning rate, and different optimization criteria, are tried to address this issue.
2. **Even the training loss does not decrease sufficiently, $J(\Theta)_{train} \gg 0$.** This is the indication of under-fitting of the neural network, which means that the network is too simple to model the problem [52]. In such cases, the number of parameter to train is very less in comparison to the data samples available for training. To address this problem, deeper and wider network is trained for increasing number of epochs till the training loss starts to decrease.
3. **Training loss decreases continuously, however validation loss starts to increase after some point.** This is usually the case when the amount of dataset and the network structure are fairly good to model the problem, however, the network starts to learn un-necessary detail from the training set, which led to the poor performance in the validation set after some time of training, indicating an over-fitting [52]. There are different ways to address this problem. One is to decrease the parameter space of the model [44], meaning use less number of layers, decrease the number of units in them, etc. Another way is to decease the size of weights through regularization [44]. BN also have some regularization effect [43] and help to reduce over-fitting. Another technique is to terminate the training session when the model stops improving through *Early Stopping* [44] mentioned in Section 3.3.3.
4. **Training and validation losses decrease constantly and saturate about the same value, which is ideally very close to zero.** This is the desired spot between under-fitting and over-fitting of the neural network. Once this stage is reached, networks performance can further be improved by fine tuning other hyper-parameters.

3.5.2 Quantitative Analysis

Predictions are made with the optimally trained models in an iterative fashion – by taking one input sample from the test set at a time to make the prediction. This is different approach than in training stage, where the predictions were made in mini-batch of samples. The reason to make iterative prediction on the test data is to simulate the real-world test environment where only one input is available at a time. Sensor values predicted in the iterative way $\hat{\mathbf{y}}$ and the true sensor values obtained from the hardware $\mathbf{y}$ are used to compute different evaluation metrics, which give the quantitative measure for the quality of the models. Since it is difficult to judge one metric over the other as they capture different properties of the prediction performance, more than one performance metrics are computed. Distance measures and similarity measures are described in the following subsections.

Distance Measures

Distance measures are computed directly from raw data without making use of any other signal information such as mean and variance. All the distance measures when zero represent perfect agreement. Prediction can be infinitely worst, so they can have arbitrarily larger value. For the quantitative evaluation purpose, the euclidean distance measure MSE and Manhattan distance measure MAE are computed. Since, the MSE is quadratic and not within the scale of the data, square-root is taken to get an another metric, namely, RMSE. Finally, the RMSE and MAE are normalized to range from 0 and 1 using the minimum and maximum of the respective signal.

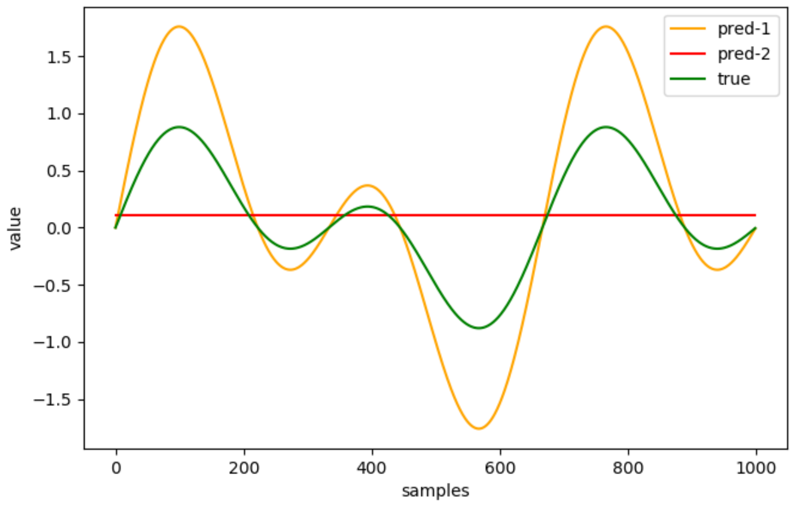


Figure 3.8: Example graph showing original signal in green and arbitrary predictions in orange and red.

Similarity Measures

Distance based measurements are highly sensitive to scaling and shifting of candidate signals. Similarity measures make use of other information about the signals, such as, frequencies and phases, apart from the raw temporal data. Figure 3.8 illustrates an example prediction scenario, where green signal represent the original signal, the red signal is just the mean of original signal and orange signal is the original signal with amplitude scaled by 2. The distance measures show that the red signal is closest to the original signal, however, in-terms of the behaviour resemblance, the orange signal is more appropriate. Therefore, there is the need of similarity measures which can make better comparisons. The CEC parameter given in Equation 3.9 is a similarity measure computed in the time-domain. The Surface Similarity Parameter (SSP) score described below is computed in the frequency domain using the frequency spectrograms of the candidate signals.

SSP is a quantitative comparison of two sequences, spatial or temporal, introduced in [54]. Time domain comparison methods are too general and do not make use of fundamental properties of signals such as frequencies and phase of the candidate signals [54]. SSP makes use of phase as well as amplitude information in frequency domain and is derived as:

$$SSP = \frac{\left(\sum_\ell \sum_k \mid Y_{\ell,k} - \hat{Y}_{\ell,k} \mid^2\right)^{1/2}}{\left(\sum_\ell \sum_k \mid Y_{\ell,k} \mid^2\right)^{1/2} + \left(\sum_\ell \sum_k \mid \hat{Y}_{\ell,k} \mid^2\right)^{1/2}} \tag{3.14}$$

where, $Y_{\ell,k}$ and $\hat{Y}_{\ell,k}$ are the Fourier transform of true and predicted signals, respectively. Subscript terms ℓ and k are the frame index and frequency-bin index, respectively. $\mid \cdot \mid$ denotes the modulus operation of the complex term. SSP is normalized and dimensionless, such that $SSP \in [0, 1]$, with zero representing perfect agreement and one representing perfect disagreement between two signals [54].

Computation of the SSP score requires to compute the Fourier transform of $\mathbf{y}$ and $\hat{\mathbf{y}}$, which is done using Fast-Fourier Transform (FFT) algorithm. For the framing purpose, *Hanning* window of length 256 is used, with 50% overlap-shift. In order to choose the window length, the frequency spectrum of the each signal is recovered back in time-domain performing Inverse Fast-Fourier Transform (IFFT), then the MSE against the un-transformed time-signal is computed. Longer frame length result in better FFT resolution, however, while recovering signal in time-domain, one frame of the signal is lost (half frame length from front and other half from the back of the signal) due to the over-lap add operation of IFFT algorithm. Thus, the window length of 256 is chosen which balances the trade-off between recovery precision and the signal loss.

CHAPTER 4

Implementation

In this chapter, the experimental setup used to evaluate the hypothesis of the study is described. Sensors and actuators involved in the experiment are explained and their dependencies with each other are defined. Furthermore, the chapter explains the software modules developed within this thesis work. The chapter realizes the methodology in order to fulfil the requirements.

4.1 The System

Figure 4.1 gives an overview of the entire system, where, HP-Search represents the hyper-parameter search process which gives the set of optimum hyper-parameter values with which the model is designed and trained. The terms $\bar{\mathbf{y}}$ and $\sigma_{\mathbf{y}}$ are the vectors containing the mean and standard deviation of different targets, respectively. All other terms have their usual meaning.

The system is divided into four parts, which correspond to the four project phases shown as a ladder structure in Figure 3.1 at the beginning of Chapter 3. In Figure 4.1, these parts are grouped by solid rectangles. The top-left part corresponds to the Step-I of the project phase and shows the experiment with the real hardware, which in this case is the MAIUS T-Stack. In MAIUS, the communication between the OBC and the T-Stack is via TCP protocol. During the experiment, the messages exchanged between the OBC and the T-Stack are recorded as TCP packets. The top-right part of the system diagram corresponds to the Step-II and shows the filtering of these TCP packets using patterns generated by the pattern-generator. Similarly, the middle section visualizes Step-III of the project phase, where the designing and training of the RNN model are done. At the bottom, the testing and evaluation of the trained RNN model are shown. And finally, the evaluation metrics are computed. The system diagram is valid for all model instances i.e. for one-to-one as well as one-to-many model architecture with both, LSTM and GRU memory units.

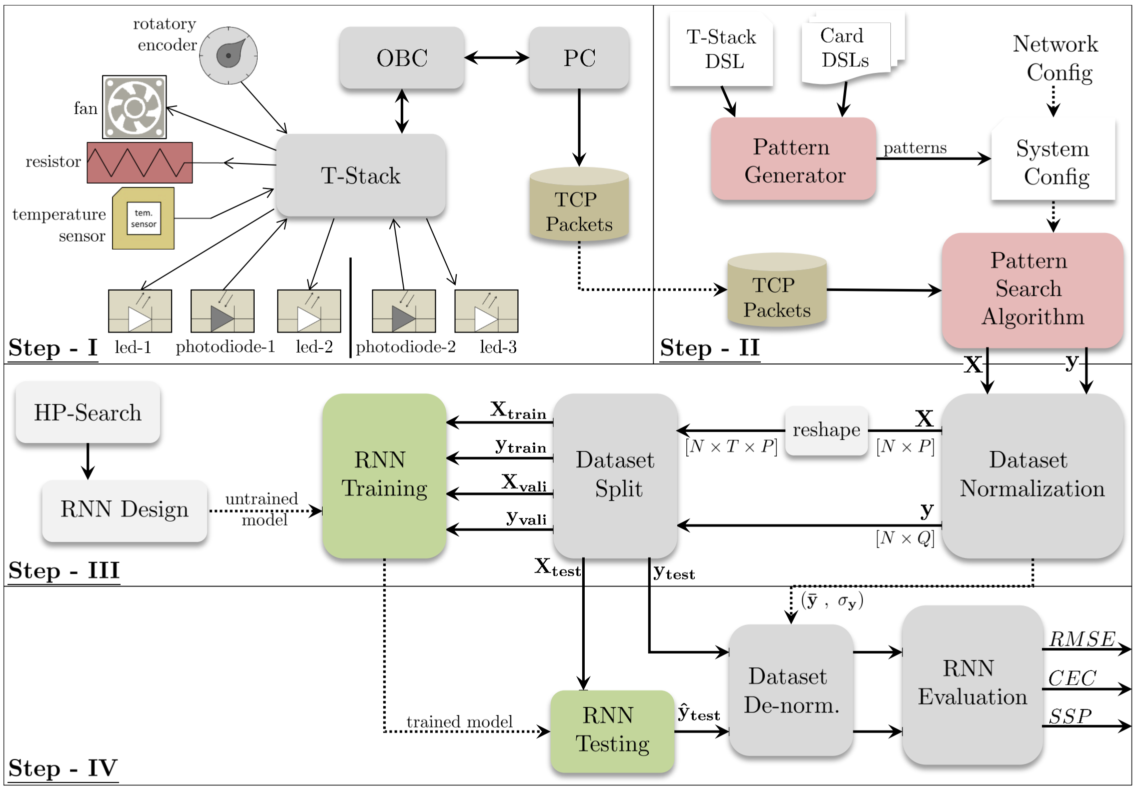


Figure 4.1: The system diagram.

4.2 Experimental Setup

MAIUS cards described in Section 2.1.1, page 7 of this report are used to design the experiment. The experiment is designed in a such a way that it covers the requirements of the study and is yet simple to understand. Components for the experiment are chosen accounting the requirement *req_06* which enforces the model to work with various type of signals such as, binary, continuous, step-like, and constants. Table 4.2 and 4.4 lists the sensors and actuators used in the experiment along with the associated card and their signal types.

Sensors and actuators listed in Table 4.2 and 4.4 are connected with the MAIUS T-Stack as shown in Figure 4.2. The figure also shows the OBC and the PC connected with the T-Stack. A fan and a resistor are connected to the output-channels of the Fan1 card and a temperature sensor is connected to the input-channel of the PhotodiodeIn2 card. The fan and resistor only have two states, i.e. they can be turned ON and OFF, thus they

Card	# Sensor	Signal Type
NTCIn2	1 temperature sensor	continuous
PhotodiodeIn2	2 photodiodes	step
Led1	1 rotatory encoder	constant

Table 4.2: Sensors used in the experiment.

Card	# Actuator	Signal Type
Fan1	1 resistor	binary
Fan1	1 fan	binary
AnalogOut1	3 LEDs	step

Table 4.4: Actuators used in the experiment.

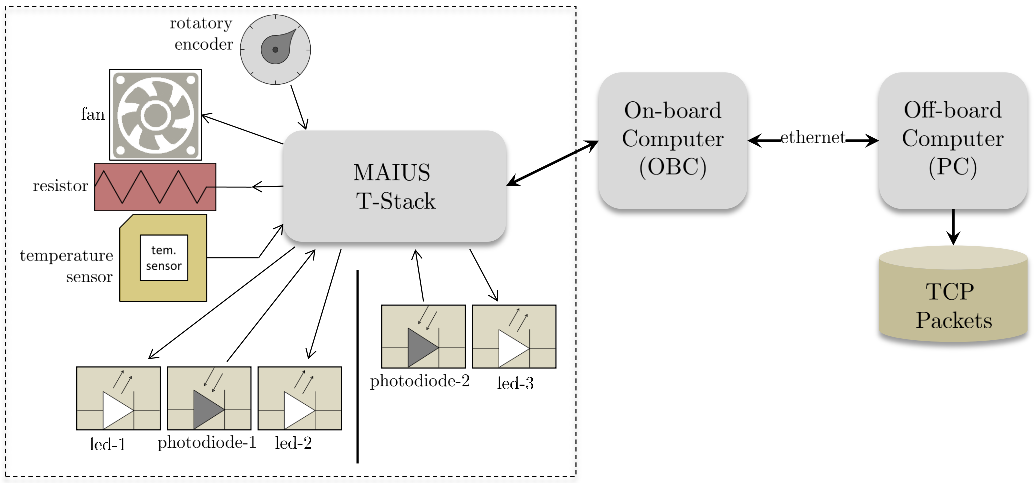


Figure 4.2: The experimental setup.

resemble two independent binary variables. The temperature sensor produces continuous value depending on the state of the fan and the resistor, therefore it resembles a dependent continuous variable. During the experiment, the fan and the resistor are turned ON and OFF in random interval between 2 to 20 seconds and the temperature is monitored every $\approx 500ms$.

Two LEDs, namely, led-1 and led-2 are connected together, both of them pointing towards the photodiode-1 allowing maximum light to the photodiode. Similarly, led-3 and photodiode-2 are connected together. There is a divider in between these photodiodes such that neither of the LEDs are affecting the other photodiode's reading. During the experiment, voltages in all LEDs are changed randomly to 0, 2.5 and 5 Volts. Depending on the voltage level on the LEDs, their brightness differ which is reflected by the photodiode reading. Thus, the LEDs and photodiodes together make-up the step-like independent and dependent variables, respectively. In analogue to the temperature-sensor, the photodiode readings are also taken in the same interval.

The experiment is automated and is conducted for approximately 150 minutes. The communication between the OBC and the T-Stack is via Ethernet. The network monitoring tool Wireshark introduced in Section 2.6, page 18 is used to capture the TCP packets during the experiment.

4.3 Dataset Preparation

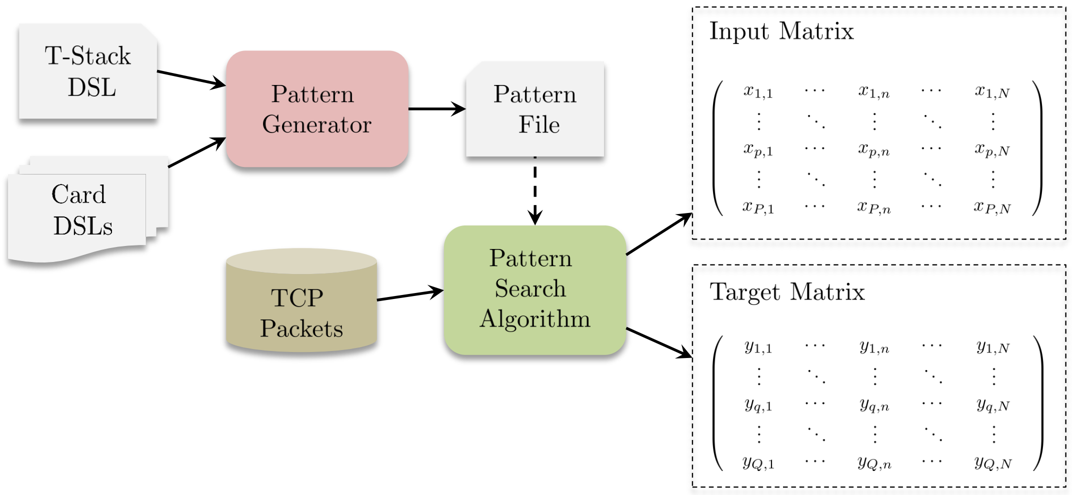

T-Stack DSL
Card DSLs
Pattern Generator
Pattern File
TCP Packets
Pattern Search Algorithm
Input Matrix
$\begin{pmatrix} x_{1,1} & \cdots & x_{1,n} & \cdots & x_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{p,1} & \cdots & x_{p,n} & \cdots & x_{p,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{P,1} & \cdots & x_{P,n} & \cdots & x_{P,N} \end{pmatrix}$
Target Matrix
$\begin{pmatrix} y_{1,1} & \cdots & y_{1,n} & \cdots & y_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ y_{q,1} & \cdots & y_{q,n} & \cdots & y_{q,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ y_{Q,1} & \cdots & y_{Q,n} & \cdots & y_{Q,N} \end{pmatrix}$

Figure 4.3: Block diagram shows the pattern search process taking the pattern file and TCP byte-stream as an input to compute input and target matrices defined in Section 3.2.

4.3.1 Pattern Generator

In the MBSD methodology, the OBSW source-code is generated from the DSL data-models. The data-models in MAIUS are written using Xtext framework, and can be used to generate executable code as well as other artefacts. The source-code generator for MAIUS is written in Xtend language.

In this study, a pattern-generator is developed using the Xtend language. It uses T-Stack data-model to generate the *pattern-file*, which is basically a Python dictionary and contains unique pattern for each channel of cards present in the stack. These patterns are used to find channel values from the TCP communication between OBC and T-Stack. This implementation is driven by requirements *req_04* and *req_05*. Requirement *req_04* forces the data-collection procedure to be independent of the normal OBC operation, and requirement *req_05* does not allow the OBSW to be changed for the data-collection purpose. Collecting the data from the TCP packets do not require any changes on the OBSW, moreover, the normal communication between the OBC and the T-Stack is also not disturbed. Requirements are listed in Table 1.1 on page 5 of this report. The block diagram in Figure 4.3 shows the pattern search process which takes the pattern file and the TCP byte-stream as an input and computes the input and target matrices as defined in Section 3.2.

A simple pattern-generation instance is explain below. Since, the MAIUS software is not in focus within this study, only the most relevant information is given. Listing 4.1 shows a typical input-channel definition from the PhotodiodeIn2 card (also see Section 2.1.1). In the first line, the channel parameter is defined with a data-type, range, and a default value. In the next line, the *channel* variable is set to 1, indicating an access to that channel. Then, a call to the *getADCResult* process is made. Finally, a linear-scaling of the *data* is done to the range defined in the parameter definition, and finally, the resultant scaled value is returned.

Listing 4.1: Definition of a typical input-channel in MAIUS card data-model.

```
parameter photodiode1_FullRange float 0.0 to 1e6 default = 1.0
inputchannel photodiode1 float read only
begin
  set channel = 1
  call getADCResult
  return extern linearScalingRange(data, 0, photodiode1_FullRange)
end inputchannel
```

Definition of the *getADCResult* process is shown in Listing 4.2. A *process* basically makes read operation to *out* variables and write operation from *in* variables. The read and write operations are done sequentially and are defined as a series of *patterns*. Each of these patterns put 3-bytes into the TCP byte-stream. The first byte indicates the *memory address*. The memory address is composed of a unique card address and an auxiliary address within the card. Second byte is the data. The read/write operation is done for one byte at a time. If the data is larger than one byte, the operation is carried out in several steps, each time reading and writing different bits of the value until all bits are read or written. For example, in the *getADCResult* process, first the bit-positions 15 to 8 of the *data* variable is read, then later, the remaining bits 0 to 7 are acquired. The third byte is the access flag, such that "1" indicates a write access and "0" indicates a read access.

Listing 4.2: Definition of the *getADCResult* process in the *PhotodiodeIn2* card data-model.

```
process getADCResult
  in channel
  out data
  pattern w 0x00 0 0 channel[4:0] 1
  pattern r 0x00 data[15:8]
  pattern w 0x00 0 0 channel[4:0] 0
  pattern r 0x00 data[7:0]
end process
```

Thus, each call to the process makes an unique mark in the byte-stream, which is used to identify the process been called and capture the underlying data. Hand-crafting these patterns is tedious and is prone to errors, thus they are automatically constructed using the T-Stack data-model and written in a file by the pattern-generator. The generated file is produced as a Python dictionary in order to facilitate easy usage as all further steps are written in Python. List 4.3 shows the pattern of the *getADCResult* process for the input-channel defined in Listing 4.1.

Listing 4.3: Pattern of *getADCResult* process for an input channel.

```
'name': 'getADCResult_photodiode1',
'pattern': [0x62 + 0x00, 0x5, 0x1,
            0x62 + 0x00, 0x0, 0x0,
            0x62 + 0x00, 0x4, 0x1,
            0x62 + 0x00, 0x0, 0x0, ],
'mask': [1, 1, 1,
         1, 0, 1,
         1, 1, 1,
         1, 0, 1, ],
'in_param': {
    'count': 1,
    0: {
        'name': 'channel_1',
        'byte_positions': [1, ],
        'bit_positions': [[2, 3, 4, 5, 6, ], ],
    },
},
'out_param': {
    'count': 1,
    0: {
        'name': 'data_1',
        'byte_positions': [4, 10, ],
        'bit_positions': [[0, 1, 2, 3, 4, 5, 6, 7, ],
                          [0, 1, 2, 3, 4, 5, 6, 7, ], ],
    },
},
```

The pattern is indicated by "pattern" key in the dictionary and is represented by a hexadecimal array. Starting from the index 0, elements in the pattern-array can be grouped together to form a set of three elements, each of these sets corresponds to the respective *pattern* defined in the process definition (see Listing 4.2) and occupy exactly 3 bytes in the byte-stream. In this way, the pattern-array always has length which is a multiple of 3.

In Listing 4.3, the address byte is (0x62 + 0x00) hex, which means, the card-address is 0x62 hex and the auxiliary address is 0x00. The second byte value 0x5 is computed from the first *pattern* in the process (see Listing 4.2), here, the data-byte is presented in its binary form as [0 0 0 0 0 0 1 1], where the bits printed in black are the five bits (index 0 to 4) of the *channel* parameter. In this example, value of the *channel* parameter is 1, as shown in Listing 4.1. The access byte is 0x1 representing the first pattern in the process is a write operation.

Analogously, rest of the byte values are computed form the pattern-array. Since, the example shown here is of an input-channel, the parameter-of-interest is the *out* variable which is *data* defined in the process. Value of variable *data* is unknown and needs to be acquired from the byte-stream, therefore, its position is masked with 0x0.

Moreover, there is a *mask-array* in the dictionary consisting of binary values. In analogy to the pattern-array, positions where the value of parameter-of-interest lies are masked with 0 and rest of the positions have value 1. Furthermore, there are relative byte and bit position arrays for each parameters involved in the process. Byte positions are integer arrays; each element of which represents the corresponding relative byte position of the parameter. For each byte position, there is a corresponding array of bit positions, hence, the bit positions are defined in a matrix. In the above example, the value of the *channel* parameter can be found on bit positions 2 to 6 of the second byte (start index is 0). The parameter *data* has 16 bits; these bits are collected from two locations, precisely, 8 bits each are acquired from 4th and 10th byte positions.

4.3.2 Pattern Search Algorithm

Pattern search algorithm uses patterns generated by the pattern-generator to find channel values in the TCP byte-stream. The algorithm is written in Python and starts by loading all .pcap files from the *work-space* directory recursively using *scapy* version 2.4.0. Scapy is a Python program which can be used to analyse network packets [55]. Iterating over all packets, raw-data is collected in an array ignoring un-necessary TCP header informations. Direction of the packets are identified by the *source* and *destination* IP addresses. In the meantime, relative time between two consecutive packets is collected starting at 0 seconds for the first packet, such that, the raw-data array (the byte-stream) and the time-stamps are of the same length. In the next step, pattern search is performed in the byte-stream to find the channel values.

In the pattern search process, starting at index 0, a chunk of bytes with the same length as of the pattern-array is taken from the byte-stream. Every elements of this sub-array is multiplied with the respective byte of the mask-array, such that, the byte positions of the parameter-of-interest are masked. Then, the resultant array is compared element-wise with the pattern-array. The comparison is done for all patterns i.e. for all processes of each channels of every cards in the T-Stack, until all bytes in both arrays match. If there is no match at all, the start position is incremented by 1, and the search process is repeated, otherwise, the start position is incremented by the length of the pattern that has the match. For the comparison purpose, pattern-arrays of different processes are arranged in the descending order of their length i.e. longer patterns are matched first. This is important because short patterns of a certain process could be part of a longer pattern. If there is a match, all bits of the parameter-of-interest are collected from their respective positions. Bits are arranged accordingly and are converted to a decimal value. Optionally, the decimal value is scaled according to the given parameter range. The scaling is done only when it is defined in the channel definition (see Listing 4.1).

Finally, the parameter is identified to be either an input or a target. Parameters declared as *out* in the process definition (see Listing 4.2) are inputs, similarly, parameters declared as *in* are the targets. If the parameter is identified as an input, the input vector is updated at the respective parameter position defined in the config file (see Listing 4.4). Similarly, when the parameter is a target, the target vector is updated. The data-collection procedure is target-driven as already mentioned in Section 3.2, so input and target vectors are appended in their respective matrices only when values of all targets listed in the config file are obtained. The time interval is also updated at this point. All sensor values are read sequentially in a fixed interval, therefore all of them are expected to be found together in the byte-stream. Latency caused by the communication channels is assumed to be negligible and act as noise for the neural network. Perhaps the performance can be improved by sophisticated data-sampling techniques in the future.

Hence, the pattern search process outputs two matrices, namely, the input and target matrices. These matrices have the structure as defined in Section 3.2. Furthermore, the input matrix is reshaped according to Section 3.2.2.

4.4 System Configuration

Configurations for the system are defined as a Python dictionary in the *system config* file. Patterns generated by the pattern-generator are also a part of the config file. Beside the pattern information, the config file contains informations regarding the inputs and targets for the neural network as shown in Listing 4.4. In "nn_config" section of the file, indices of different features and targets in their respective vectors are given. In order to identify a feature and a target, a simple naming convention *<card name>_<parameter name>_<postfix>* is followed. The *postfix* is optional. It is added only when multiple channels make call to the same process, in such case, the *parameter name* would be the same, therefore the postfix is needed to deal with the ambiguity.

The config file is identifiable using the integer ID, such that, easy switching between different configurations is possible. The config file can further be extended by defining other neural network settings, such as the model type, data-normalization technique, hyper-parameters value, etc. However, such extension is left for the future improvement.

Listing 4.4: System configuration file showing the network configuration which is used for preparing the datasets.

```
id = 55
nn_config = {
    'input_data': {
        'no_of_samples': None,
        'no_of_features': 6,
        'features': {
            'delta_t': 0,
            'fan_output1': 1,
            'fan_output2': 2,
            'AnalogOutDAC_data_0': 3,
            'AnalogOutDAC_data_1': 4,
            'AnalogOutDAC_data_2': 5,
        },
    },
    'target_data': {
        'no_of_samples': None,
        'no_of_targets': 4,
        'targets': {
            'photodiode_data_0': 0,
            'photodiode_data_1': 1,
            'led_rotEnc': 2,
            'ntc_data': 3,
        },
    },
}
```

4.5 Keras Implementation

All RNN models in this study are built in Keras using the functional API. Keras is a high-level Python library that runs on top of the Tensorflow framework. Tensorflow is a opensource machine learning framework developed by Google. In the core of the Tensorflow, every mathematical expression is defined, optimized, and calculated in a multi-dimensional array called tensor. Thus, tensor is the basic data-structure in the Tensorflow language. Building a neural network model in Tensorflow involves in defining each layer in the network as a tensor, including the activation functions, cell states, weights, and biases. These tensors are then arranged and stacked in same order in which they are executed, such that, a graph structure of the tensors is formed. This tensor-graph basically resembles the neural network model. For training the network, a Tensorflow session is initialized and data is passed through the tensor-graph while tuning the internal weights and biases according to the learning algorithm.

The following example shows a simple implementation of a LSTM model with one input, one output, and one hidden layer using the functional API in Keras.

Listing 4.5: An example one-to-one LSTM model implementation using the Keras functional API.

```
from keras.models import Model
from keras.layers import Input, Dense, LSTM

input_layer = Input(shape=(100, 6), name='in0')
hidden_layer = LSTM(50, name='lstm0')(input_layer)
output_layer = Dense(4, name='out0')(hidden_layer)

model = Model(inputs=input_layer, outputs=output_layer)
```

In the above example, the input layer is defined with shape given by the tuple, where the first element of the tuple gives the sequence length and the second element gives the feature count. The LSTM layer has 50 units and the output layer has 4 units. It is important to note that only the shape of the input layer is defined, because for all other layers, the shape is same as the output shape of the preceding layer in the model. For this purpose, the tensor instances are called by passing the previous layer as an argument; this call basically connects these tensors together. In order to make the successful connection, the output shape of the preceding layer must match the input shape of the succeeding layer. Optionally, each layer can be given a unique name. This name can be used to query different layers.

Similarly, the model with multiple input and output layers can be defined. In such case, a dictionary containing the inputs and output layers are passed to the model as shown in the Listing 4.6. The multiple input/output model have neural branches, which can split/merge inside of the model. In the Tensorflow framework, it is possible to train the individual layers separately. In order to do so, the layers which aren't to be trained should frozen by setting their *trainable* flag to *false*. The name of the layers are helpful for this kind of query operations. The weights and biases of the frozen layer are not updated during training.

Listing 4.6: Instantiating Keras model with multiple inputs and outputs.

```
model = Model(inputs={'in0':input_layer0,
                      'in1':input_layer1},
              outputs={'out0':output_layer0,
                       'out1':output_layer1})
```

Once the model is instantiated, it is compiled by calling the *compile* method on the model. During compilation the cost function, evaluation metrics, and the optimization technique to use are defined. For the model with only one output layer, an scalar loss function is given, whereas, for the one with multiple outputs a dictionary of loss functions to compute at the end of each output layers is given. The output layer dictionary and the loss dictionary have same number of entries and the same set of keys, so that, the loss functions and the output layers are mapped.

Listing 4.7: Compilation of the Keras model.

```
from keras.optimizers import Adam
from keras import backend as K

def correntropy_loss(y_true, y_pred, sigma=1.):
    return 1 - K.mean(K.exp(-K.square(y_true - y_pred)/sigma))

model.compile(loss=correntropy_loss,
              metrics=['mae', 'mse'],
              optimizer=Adam(lr=0.0005, decay=0.01))
```

The Listing 4.7 shows the compilation steps of the model. Here, the model is compiled with the CEC loss; the method to compute the cost is defined as *correntropy_loss*. Additionally, MAE and MSE metrics are computed during training. The Adam optimizer is used with an initial learning rate of 0.002 and the learning rate decay of 0.02.

After the model has been compiled, it is ready to be trained. There are different ways to train the model in Keras framework. In this study, all models are trained by simply calling the *fit* method with the references to the training and validation datasets, along with the number of epochs and the mini-batch size. In addition, a list of callback methods to call at the end of each epoch, a shuffle flag, and a verbose parameter is passed to the method. The shuffle flag when *true*, shuffles the training samples after each epoch, so that, in each epoch the model receives samples in different order. This helps to reduce over-fitting of the model, as the model always sees the data in an unique order. The verbose parameter controls the verbosity of the training process. The verbosity of 0 means silent, 1 means detailed, and 2 means only one line per epoch is printed in the console during the training process.

The Listing 4.8 shows a call made to the fit method. Here, the model is trained for 10 epochs with the mini-batch size of 256. The *callbacks* argument contains the instances of the Keras callback methods explained in Section 3.3.3. The model is trained by shuffling the training samples in the N axis after each epoch. The verbosity parameter is 2. The fit process returns the history object, which contains informations about the training session, such as lists of training, validation loss values after each epoch and other metric values which were given during the compilation of the model.

Listing 4.8: Fitting the Keras model to the training samples.

```
history = model.fit(x=X_train,
                    y=y_train,
                    validation_data=(X_vali, y_vali),
                    epochs=10,
                    batch_size=256,
                    callbacks=[early_stopping,
                               reduce_lr_on_plateau,
                               model_checkpoint],
                    shuffle=True,
                    verbose=2)
```

Testing of the model is done by calling the *predict* method on the trained-model as shown in Listing 4.9. In order to simulate the real-world scenario, the predictions are made for each samples, therefore the batch size is 1.

Listing 4.9: Testing the Keras model with the test samples.

```
y_hat = model.predict(x=X_test, batch_size=1)
```

Thus obtained *y_hat* and the *y_test* from the actual hardware are used to compute the evaluation metrics. The methods to compute the evaluation metrics are also written in Python. This is illustrated in the bottom part of the system diagram in Figure 4.1.

4.6 Bayesian Optimization Process

Optimum values for the hyper-parameters explained in Section 3.4 are found using the Bayesian optimization process. GPyOpt Python library is used to implement and configure Bayesian optimization process. The library can run Bayesian process with user-specific configurations. The Bayesian process take many argument such as model type, the type of the acquisition function to use, evaluation interval, evaluation type etc. In this study, only the required parameters are given; For rest of the parameters, their default values given in the GPyOpt package are used. Perhaps, the optimization process can achieve better result with different configuration; this is left out for the future work. The standard Gaussian process is used as the model type and the Expected Improvement (EI) as the acquisition type; the model is updated every iteration and is evaluated sequentially.

A Bayesian optimization object from the GPyOpt package is instantiated with the objective function and a search domain. The setup is illustrated in Figure 4.4. Search domain is a Python dictionary containing hyper-parameter's bounds, for which the optimum values are to be search. In the objective function, neural network model is created, trained and then tested. The average test result is returned to the Bayesian process. The optimization process is started by calling *run_optimization* method. The process runs for given number of iterations. At the end, the process returns the best set of hyper-parameter values. The optimization process only takes numerical values, therefore, non-numerical hyper-parameters are mapped to an integer value in the search domain. Several iteration of the search is carried out with different range of hyper-parameter values. If the optimum value lie on the border of the range, iteration is repeated with values beyond the border. Finally, a hyper-parameter set with the best values of all hyper-parameters is selected and used to train the network.

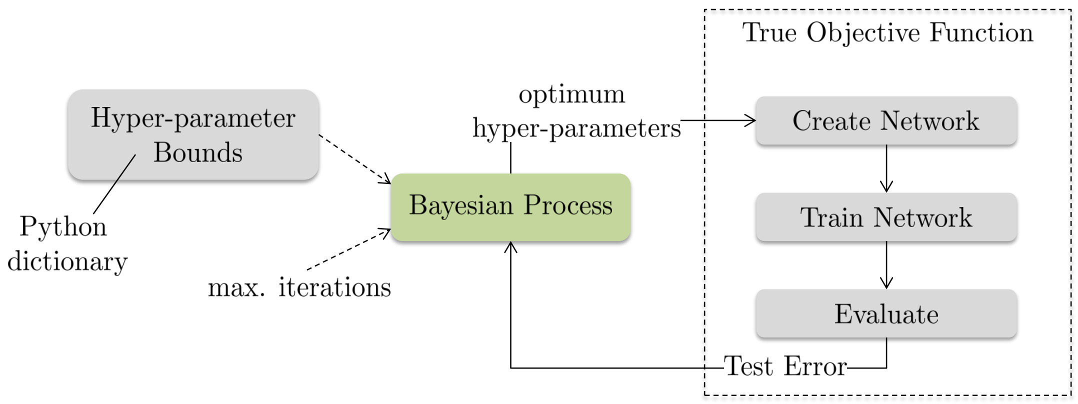


Figure 4.4: Setup for hyper-parameter search using Bayesian optimization process.

CHAPTER 5

Results

This chapter begins by presenting the results of the qualitative and quantitative analysis on all the model instances. In the last section, the coverage of the requirements as identified in the first chapter are presented.

In this chapter, the evaluation results of altogether four RNN model instances, which are developed within this study, are presented. Two of them have the one-to-one architecture, out of which one with the LSTM and other with the GRU memory unit. Similarly, the other two model instances implement the one-to-many architecture. These model architectures are defined in Section 3.3.

All four models are evaluated qualitatively as well as quantitatively. These evaluation approaches are described in Section 3.5.1 and 3.5.2. For the quantitative analysis on the prediction results, time and frequency domain evaluation metrics are computed. For all evaluation purposes, the test-set is used. The test-set is unseen by the network during training. Moreover, in later part of this chapter, the requirement coverage based on the methodology and results are presented.

Altogether $N = 24067$ samples are collected from the experiment. These N samples are divided into training, validation, and the test sets, as explained in Section 3.2. The performance of the training as well as the accuracy of the predicted results can be improved by using more training samples. However, in this study, the limited number of samples are taken as constraint and is solved in the model.

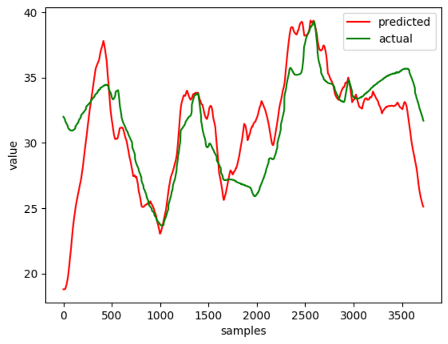
Furthermore, extendibility of the models in space is tested. For the space-scalability tests, the scaled dataset is constructed by replicating the original, rolling the replica in N axis by 2000 samples, and concatenating them back together in the feature/target axis. In this way, extended datasets are produced very fast, which are used to train the scaled networks. The number of samples to roll is incremented by additional 2000 samples every-time the scaling factor is increased. Hence, each samples have different set of actuator/sensor values, thus resembling new signals. In order to maintain the correlation between the input and target datasets, the scaling is done for both datasets in the exact same way.

The models are scaled up-to 8 times, each time with a step of 2. Hyper-parameters are optimized for the unscaled condition and the same configuration is used when the models are scaled. An unscaled model consists of 6 inputs (5 actuators + 1 time duration between two consecutive samples in seconds), and 4 targets (sensors). The experimental setup with these hardware components are explained in Section 4.2. Scaling the model by 2 in the above mentioned fashion mimics the scenario of modelling 10 actuators and 8 sensors together. Similarly, the model when scaled by 4 incorporates 20 actuators and 16 sensors, and so on.

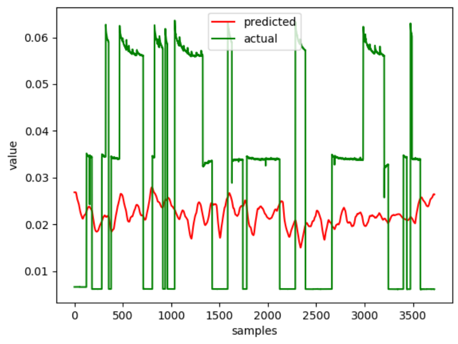
Whenever an up-scaling happens, the input as well as the output-space of the model increases, this in return increases the number of irrelevant inputs (noise) per output that the network must discard in order to maintain the accuracy of the prediction. The scalability test gives the measure of the model's capability to deal with the increasing irrelevant input information. Moreover, it checks the feasibility of the model in terms of the error increase rate, increment of the training time, and the increment in the number of weights in the model at different scaling conditions.

5.1 Qualitative Approach

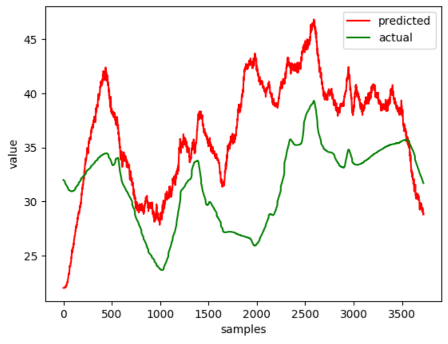
The one-to-one model performs poorly and cannot predict different types of sensor values simultaneous. The red signal in Figure 5.1a and 5.1b are the predictions made by the one-to-one LSTM model, and the ones in Figure 5.1c and 5.1d are from the same model but configured differently. From these graphs, it is seen that when the one-to-one model predicts the photodiode values properly, it fails to simultaneously maintain the accuracy of the temperature sensor prediction, and vice-versa. The photodiode outputs different states depending on the voltage level of the corresponding LEDs, thus has step-function as an output signal. The temperature sensor produces a continuous temperature readings depending on the state of the resistor and the fan. The problem of the one-to-one model is because of vastly different output-space, which models like neural networks have difficultly producing through the same output layer. Having a single body and a output layer would mean that the weights, cell states as well as the activation functions are shared between different targets. But, the output of the network is bounded by the activations of each hidden layers, thus producing different ranges of value through the same layer is difficult. This is a unique problem, which does not occur in normal neural network application, as they usually only deal with similar kind of data, such as images, speech-signals, or categories of classification.



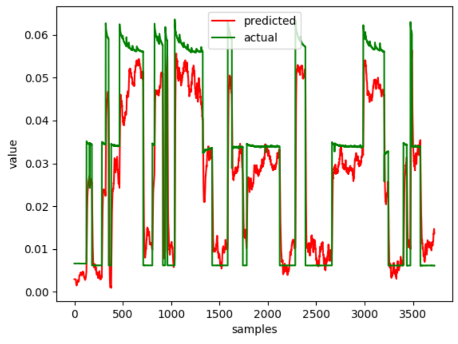
(a) A temperature prediction.



(b) A photodiode-2 output prediction.



(c) A temperature prediction.



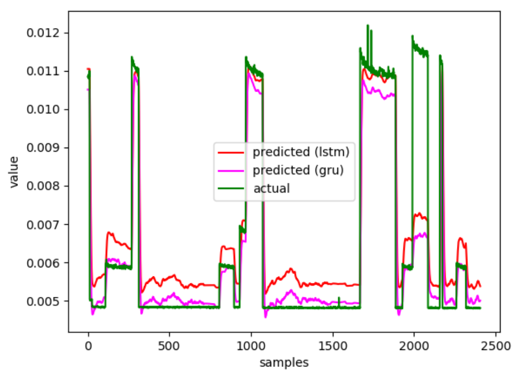
(d) A photodiode-2 output prediction.

Figure 5.1: Predictions made by the one-to-one LSTM model are shown in red colour. The green signal is the one obtained from the respective hardware.

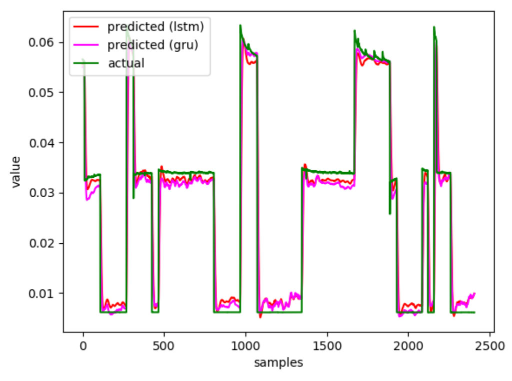
Perhaps a deeper and wider network trained with sufficiently large number of samples can discriminate between variety of signals through common layers, such that the problem of one-to-one model is solved. However, in this study, an alternative solution is proposed, which addresses the problem by changing the network architecture to have neural-branches, so that each of such branches can be optimized independently to produce a specific type of output signal. The proposed model thus have one-to-many architecture, meaning, a common input layer feed into multiple neural branches, and thus, posses many output layers, one for each target. The branching network is not only able to maintain but is capable of significantly improving the prediction accuracy of all output signals without needing more training examples than that of the one-to-one model.

Figure 5.2 shows the sensor value predictions made by the unscaled one-to-many model, which in comparison to the one made by the unscaled one-to-one model is significantly better. The red signals are predicted by the LSTM model whereas the ones in the magenta colour are from the GRU model with the one-to-many architecture. The sub-figures (a) and (b) show the output of photodiode-1 and photodiode-2, respectively. The photodiode-1 is the one with 1 LED and the photodiode-2 is with the 2 LEDs (detailed in Section 4.2). The sub-figure (c) shows the output of the rotatory encoder, and finally, in the sub-figure (d) the output of the temperature signal is shown.

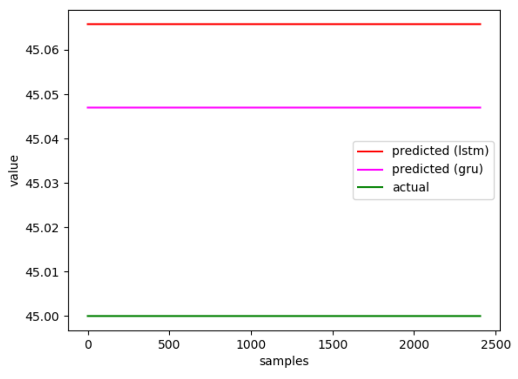
Predictions made by the GRU model are better than the LSTM model in all types of signals. Due to the inadequate performance of the one-to-one model already in the unscaled condition, the model architecture is discarded and further evaluations are performed only on the one-to-many architecture.



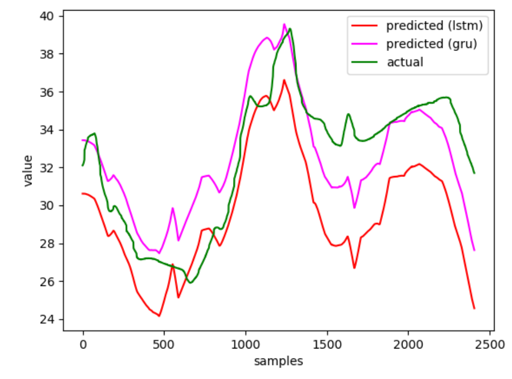
(a) Photodiode-1 output.



(b) Photodiode-2 output.



(c) Rotatory encoder output.



(d) Temperature sensor output.

Figure 5.2: Predictions made by an instance of the one-to-many LSTM model are shown in red colour and the ones from the GRU model are shown in magenta.

Further analysis on the one-to-many architecture is made based on the loss values computed during the training sessions in different scaling conditions. The outcome of this analysis acts as a baseline that the networks are sufficiently and correctly trained. The sufficiency and efficiency of the training, as well as, the accuracy of the predictions made by the model are dependent on the amount and the quality of the training data. Using more training data, the efficiency of the training and hence the accuracy of the results can further be improved. The training statistics of the LSTM and the GRU models are similar.

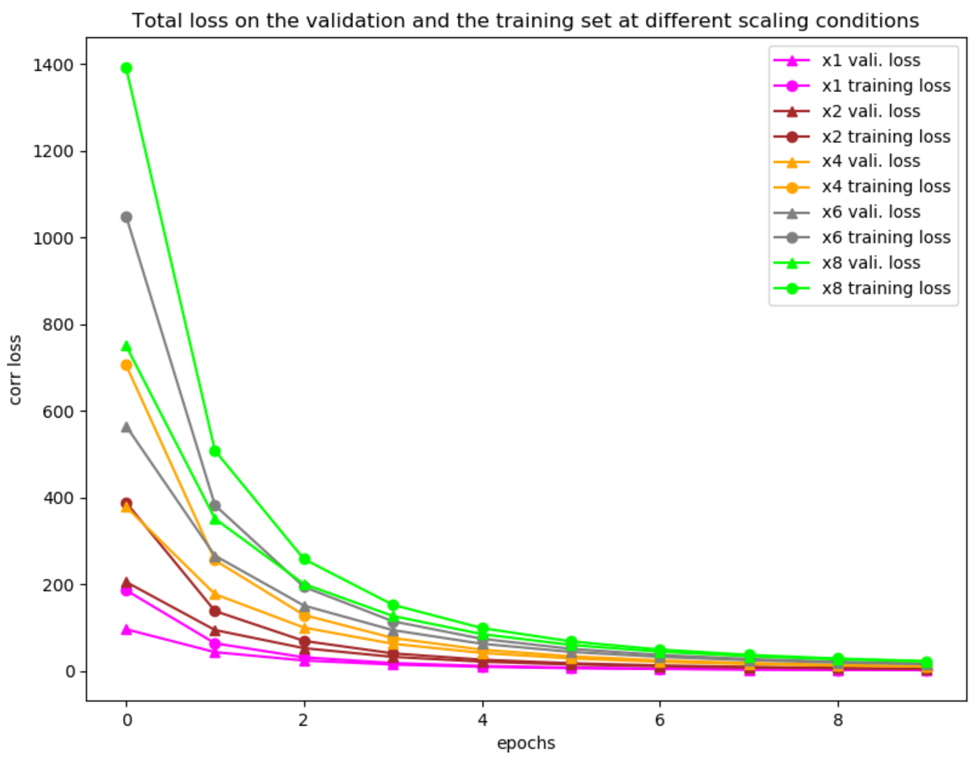


Figure 5.3: Total loss on the training and the validation set computed at the end of each epoch while training the one-to-many LSTM model at different scaling conditions.

Figure 5.3 shows the loss on the training and the validation set computed at the end of each epoch while training one-to-many LSTM models at different scaling conditions. The colour-code is per the scaling conditions, the scaling factor is shown with a preceding ’x’. The circular marker represents the loss on the validation set, whereas, the triangular marker represents the loss on the training set. From the loss graph, it is clearly seen that the model already converges sufficiently at the 10th epoch. The validation loss is less than the training loss in all cases, which shows that the model is not over-fitting to the training set. Furthermore, it is seen that, when the model is scaled, the initial losses is significantly high, however, only after few epochs of training the network is able to minimize the loss such that the value of all scaling conditions are already close at the 10th epoch.

The bar-graph in Figure 5.4 visualizes the execution time and the memory usage, while training the LSTM and GRU models at different scaling conditions. These values are obtained by training these models for 10 epochs in the GPU node of the HPDA cluster. Specification of the GPU node is given in Section 2.6.7, page 20. The statistic show that training time as well as the memory usage of the GRU model is lower than that of the

LSTM model in all cases. The GRU model has one *gate* less in each of its units (see Section 2.4.3, page 17), thus, has less number of weights to train in comparison to the LSTM model. Therefore, the GRU model is lighter than the equivalent LSTM model in-terms of the number of trainable parameters, and faster in-terms of the required training time.

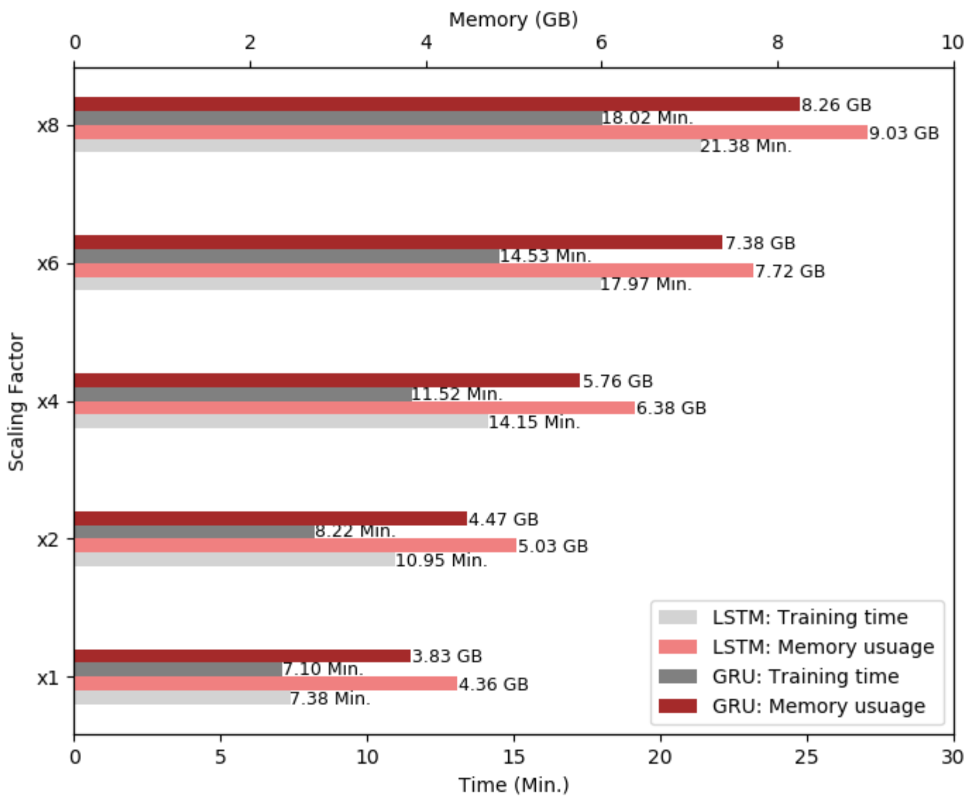


Figure 5.4: The graph shows the execution time and the memory usage while training the LSTM and GRU model at different scaling conditions.

The largest form of the model in this study (the one scaled by factor 8) simulates 40 actuators and 32 sensors instances. The training time of this model with the LSTM memory unit is about 21 minutes and when the GRU unit is used, the training time is reduced to about 18 minutes. There are altogether approximately 1 million weights in the 'x8' models, training of which costs about 9GB of memory in the Linux machine when the model uses LSTM memory unit, and about 8.2GB when it has the GRU memory unit. Hence, in terms of computation cost and the memory usage, the modelling process is affordable, as in the current market a normal PC already meets this memory requirement.

5.2 Quantitative Approach

A detailed evaluation of the LSTM and GRU models with one-to-many architecture is done using quantitative measures explained in Section 3.5.2.

5.2.1 Time-domain

Time-domain sensor signals predicted by the model $\hat{\mathbf{y}}$ and the true sensor values obtained from the hardware $\mathbf{y}$ are used to compute distance measures for each sensors separately. For the unbounded distance measures, such as, RMSE, after the computation, the resultant is normalized to range from 0 to 1. The normalization is done using the minimum and maximum value of the original signal $\mathbf{y}$ of the corresponding sensor from the collected dataset.

In the following section, the comparison between the predictions made by the LSTM and GRU models are done for different scaling conditions. In order to compute the metric of a particular sensor, the same signal predicted by variably scaled models is used. Since, it is not within the scope of the thesis work to optimize models in all scaled instances, only the unscaled model is optimized by the hyper-parameter search, and the same configuration is used in all scaled conditions. Therefore, the result of the unscaled model act as a baseline and the results from the scaled conditions are compared against it.

Sensor	**Min.**	**Max.**
Photodiode-1	0.0061	0.0642
Photodiode-2	0.0047	0.0121
Rotatory Encoder	45.0	45.0
Temperature Sensor	17.91°C	52.08°C

Table 5.1: Minimum and maximum values of the sensor outputs.

The input data contains the states of the actuators and the time-interval between samples. The actuators states are basically, 0 and 1 for the states of the fan and the resistor, voltage levels 0, 2.5 and 5 Volts for the states of LEDs, and time interval between consecutive samples. Since the data-acquisition rate is constant, the time interval between two samples are more-or-less on the same level. Thus, the input space consists similar values that only occur in different order, which is quite simple for neural network models. Therefore, a fairly simple model already over-fits the dataset. However, the mapping between the input- and output-space is still complicated and the underlying physics is difficult to model by a simple network. Therefore, the difficult part of the modelling process is to find the trade-off, such that, the model is simple enough that it does not over-fits the dataset and does not learn irrelevant details, but at the same time, it is complex enough to be able to model the underlying physics and does not under-fit the problem. This condition is reflected by the results presented below.

The Figure 5.5 shows the Normalized RMSE (NRMSE) of the sensors at different scaling conditions, the normalization is done using the ranges of sensor values listed in the Table 5.1. The NRMSE gives the measure of +/-offset of the prediction error in percentage for the given range. The bar-graph is the average NRMSE among different sensor outputs at the same scaling condition. In average, the result of the GRU model is slightly better than that of the LSTM model in all scaling conditions.

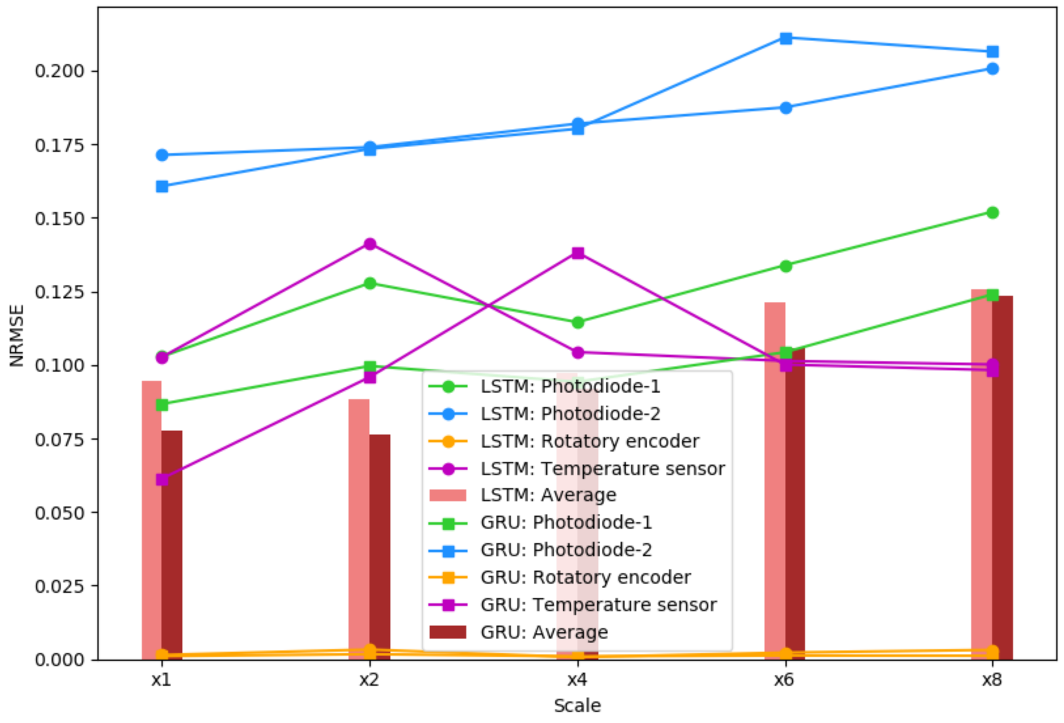


Figure 5.5: The NRMSE on sensors value predicted by the LSTM and GRU models at different scaling conditions.

In Figure 5.5, one can observe that the error on all predictions initially increases when scaling is increased to 2, and some of them decrease back when the scaling factor is further enlarged. In some sensors the increase and decrease rates are higher than the other. This behaviour illustrates the sea-saw between the over-fitting and under-fitting conditions between the dataset and the model. When the scaling is done by the factor of 2, the dataset doubles, so does the model's topology. Doubling of the dataset increases the irrelevant information per sensor output (noise) and doubling of the model increases its width (complexity). When the model is twice as wide, it becomes more complex than necessary for the added noise, thus the model is capable of capturing irrelevant information during training despite of the harsh regularization and the dropouts. This causes it to perform poorly in the test-set, hence, the test error increases. However, when the scaling is further increased, the effect is reversed. Now, the noise in the input is dominant than the model's complexity, thus, the model cannot over-fit to noise and discards it – the test error starts to decrease. This behaviour shall further be seen, if the models are not configured for the given condition. In order to address this problem, the model structure and hyper-parameters should be optimized for each scaling conditions.

From the above mentioned behaviour of the different model instances, it is clear that finding the proper network configuration for the given dataset is one of the most important job while designing neural network based models. Therefore, the number of units, number of layers, as well as the regularization parameters in the models are important hyper-parameters, and need to be tuned properly.

In the LSTM model, the error decreases once the scaling factor is four, in contrast, this happens in GRU model only when the scaling is six times over. This shows that the GRU model can fit the dataset already with smaller topology and less number of units. For example: in the scaling condition 4, the GRU model is able to learn noise, so it over-fitted the temperature signal, but the equivalent LSTM model is unable to do so. This feature of GRU model make it lighter and faster than the equivalent LSTM model, which is also reflected by the bar-graph showing the training time and memory usage of these models in Figure 5.4.

It is very easy for the RNN models to predict the constant value, so the NRMSE of the rotatory encoder output is very close to zero. The NRMSE graphs in Figure 5.5 show the highest percent error for the photodiode predictions. The reason behind this is mainly because of the time-lag in the predicted signal. For example: whenever the photodiode output rises from low to high, and falls from high to low, the time-lag in the prediction only by one sample results in nearly 100% error in the time-domain distance measure, thus increases overall percentage error. The SSP parameter presented in the next section makes use of the amplitude as well as th phase of the signals, thus it makes better judgement on the similarity between two signals.

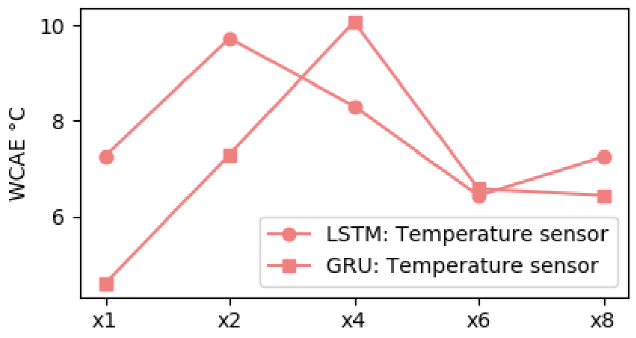


Figure 5.6: The Worst-case Absolute Error (WCAE) on sensors value predicted by the LSTM and GRU models at different scaling conditions.

In Figure 5.6, the WCAE on the temperature prediction made by the LSTM and GRU models at different scaling conditions is shown. The WCAE graph also reflects the trade-off between the over-/under-fitting of models at different scaling conditions explained earlier. The WCAE gives the measure of maximum deviation of the prediction from the actual value in °C. In the unscaled condition, the maximum absolute error is about 7.2°C in the LSTM model and about 5.5°C in the GRU model. Thus, the well-configured GRU model can predict the temperature value with the worst-case accuracy of about 84% of the temperature range.

5.2.2 Frequency-domain

The SSP parameter introduced in Section 3.5.2 is computed using the predicted and actual sensor output signals in frequency domain. The SSP, unlike the time-domain measure, makes use of the amplitude as well as the phase information, thus can make better judgement on the predicted signals. The SSP is already normalized to range from zero and one, with 0 representing perfect agreement and 1 representing perfect disagreement between two signals.

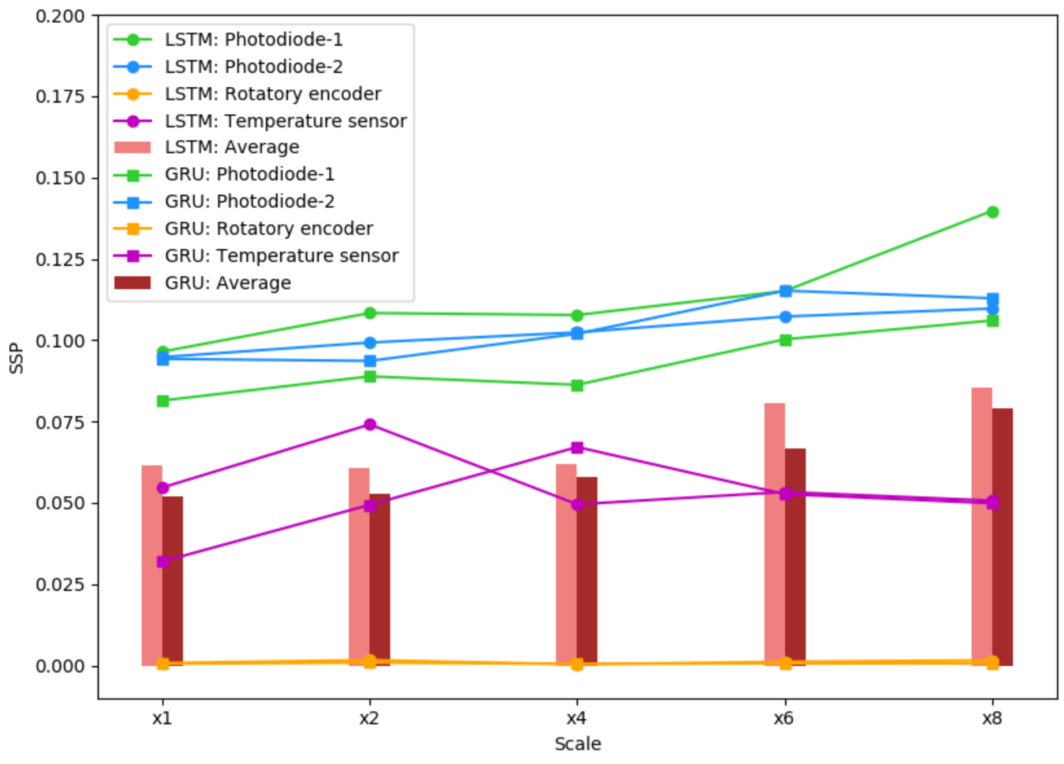


Figure 5.7: The SSP on sensors value predicted by the LSTM and GRU models at different scaling conditions.

Computation of the SSP requires to compute the Fourier transform of $\mathbf{y}$ and $\hat{\mathbf{y}}$, which is done as explained in Section 3.5.2. The Figure 5.7 shows the SSP parameter of sensor outputs by the LSTM and GRU models at different scaling conditions. The SSP result reflects the prediction behaviour seen in the NRMSE graph, which is due to the same over-/under-fitting problems discussed earlier. The SSP result also suggest that the GRU network fit the dataset better than the equivalent LSTM model in most of the conditions. The SSP score of all type of signals show lower percentage error in comparison to the time-domain distance measure, which show that the SSP score are not sensitive to small shifting of amplitude and phases of the candidate signals. For the well optimized models (the unscaled condition), the prediction error in terms of the SSP score is less than 10%; the predicted signal show more than 90% resemblance to the actual signal in the frequency domain.

On the basis of results presented above, it is clear that the GRU model with the one-to-many architecture fits the dataset collected from the candidate hardware better than the equivalent LSTM model. The GRU model under proper configuration is capable of predicting sensor values for the given actuator states which reflect the realistic hardware behaviour for more than 90% of the time.

5.3 Requirement Coverage

In this section, the requirements identified in the first chapter (Section 1.3, page 4) are restated along with their coverage by the model implementation in Table 5.2.

ID	**Statement**	**Explanation**
req_01	*The model shall be as abstract as possible and as detailed as necessary.*	The model design addresses the over-fitting and under-fitting problems. The well-configured model is only complex enough to not under-fit the underlying physics. Meantime, the model is simple enough that it does not over-fit the relatively simple dataset. Furthermore, the model design with the neural-branches facilitates easy extension and integration of new set of inputs and outputs by simply attaching new branches of the desired structure and configuration, such that, completely different physical process can be modelled. However, the model shall be trained and the hyper-parameter shall be tuned for that particular dataset.

req_02	*The hardware system and its components shall be treated as a black-box.*	The RNN based supervised learning method learns to map the input and output data-space overtime without explicitly knowing the underlying physics. The LSTM and GRU models developed within this study operate on the inputs that otherwise be going into the hardware system.
req_03	*The model data-space shall also be highly abstract and outside of the system.*	Once trained, the RNN models operate standalone without needing any infrastructure from the actual system.
req_04	*Training data shall be collected without having to interfere with the normal OBC operation.*	The datasets for training all networks are extracted from the TCP packets which are the recording of messages between the OBC and the MAIUS T-Stack during the experiment. For this purposes, the network analyser tool Wireshark (Section 2.6.1) is used, which records the network communication without causing any disturbance in the operation.
req_05	*Training data shall be collected without having to change the OBSW.*	The training data are collected once they are dispatched from the OBC into the network, thus for this purpose the OBSW does not need to be changed.
req_06	*The model shall be able to deal with various types of input and output signals, such as, binary, continuous, step-like, constant signals, etc.*	The experimental setup introduced in Section 4.2 is designed with sensors and actuators which covers verity of the signal types. The fan and the resistor states are the binary signals. Temperature sensor output continuous signal. The photodiodes output step-like signal, depending on the voltage levels on the LEDs. The rotatory encoder is set before the experiment began to value 45 and remains constant. The qualitative and quantitative analysis show the RNN based models with the multiple output layers are capable of modelling variety of the signal types at once.

req_07	*The model shall be able to replicate the system in the test phase, when the unseen input is passed to the system.*	Over-fitted model do not perform well in the test-phase. The modelling process mainly focuses on addressing this problem. Regularization and dropout are used to make the model robust. The model is trained only for 10 epochs and is kept simple enough so that it is capable of learning underlying relationship between the input and output but not the irrelevant detail on the dataset. The prediction results on the test-set explained in Section 5.1 and 5.2 show that the LSTM and GRU model is capable predicting all types of signals depending on inputs which are not seen during training. The predicted signals showed more than 90% resemblance to the actual signal in the frequency domain.
req_08	*The model shall be scalable in space.*	The space-scalability of the model is tested by expanding the dataset and the model topology up-to 8 times, simulating the scenario with 40 actuators and 32 sensors. The results explained in section 5.1 and 5.2 show that prediction error only increases linearly when the model is enlarged.
req_09	*The model shall have a defined accuracy of operation.*	For the optimally configured one-to-many RNN models, the maximum prediction error is determined to be approximately 10% in terms of the SSP score. The SSP is computed in the frequency domain and it takes also the phase-lags of predicted signals into account. Thus, the well-configured one-to-many RNN models are expected to produce all kinds of output signals, which are in phase with the actual ones, with an accuracy of about 90%.

Table 5.2: Requirement coverage by the model implementation.

CHAPTER 6

Conclusion

Hereby, it is concluded that the RNN based models are capable of generalizing different physical processes and can predict the realistic hardware behaviour with the considerable accuracy. Through the design, implementation, and the results of the RNN model, the requirements of the thesis work defined in the first chapter of this report are successfully achieved.

The study focused on the relatively unique RNN application of the physical system modelling. RNNs have the reputation of being an excellent sequence modelling tool and have been widely used in the temporal sequence modelling tasks involving speech/music data, images, videos, etc. Through the results of the thesis work, it is seen that when configured properly the RNN models with the LSTM and GRU memory units can faithfully predict different types of sensor values for the given set of actuator states with more than 90% resemblance to the actual signal in terms of the SSP score. The SSP score is computed using the frequency spectrograms of the actual and predicted sensor outputs. Thus, the predicted signals reflect the realistic hardware behaviour most of the time.

6.1 Summary

Within this thesis work, the problem of simulating a physical hardware system is formulated as a time-series prediction problem and altogether four RNN based models are developed and evaluated to address it. Physical processes are defined by connecting a fan, a resistor, a temperature sensor, three LEDs, and two photodiodes to five different MAIUS cards in the T-Stack. The choices on the sensors and actuators are made so that variety of signals such as, binary, continuous, step, and constant signals are covered by the experiment. The actuator states are identified as the independent variables and the sensor readings are identified as the dependent variables of the system. Two RNN model architectures, namely, one-to-one and one-to-many, each of them with the LSTM and GRU memory units are designed and trained using the supervised learning method to learn the underlying physics between the independent and dependent variables. In order to ensure that all models are sufficiently and effectively trained, a qualitative analysis on the training sessions of all models are made by monitoring the loss values on the training and validation datasets. Moreover, the quantitative analysis on the sensor values predicted by all model instances are made with the help of evaluation metrics computed in time-domain and frequency domain. In addition to that, the scalability tests on the one-to-many models are performed.

Regardless of the memory unit used, the one-to-one model architecture is unable to produce all types of sensor outputs with considerable accuracy at once. The problem is identified and addressed by the development of the one-to-many model architecture, which splits the RNN topology in the middle to facilitate one neural-branch for each target output. Multiple neural-branches of the one-to-many allow the configuration of individual branches differently, which makes it possible to predict various types of signals at once without compromising the accuracy. Furthermore, one-to-many design also facilitates the extendibility of the model.

Results from different quantitative evaluation measures suggest that the GRU model with one-to-many architecture perform better than the equivalent LSTM based model in most of the test scenarios. The Euclidean distance based percent error (NRMSE) when averaged over all sensor outputs showed that the prediction error of GRU model is about 1.7% lower than that of the LSTM model in the unscaled condition. When the model is scaled by 8, the average NRMSE on both models is $\approx$ 12%. In the worst-case absolute temperature prediction error of the GRU model is $\approx$ 2.5°C less than that of the LSTM model in the unscaled case, and on the maximum scaled condition, it is still about 1°C less than that of the LSTM model. The SSP parameter computed in the frequency domain suggests that the GRU model has $\approx$ 1% better result than that of the LSTM model in average. The SSP is computed using the amplitude and phase spectrograms of the two signals, therefore, it provides better insight on their similarity with each other. The GRU based one-to-many model also showed better performance than the one with the LSTM unit in terms of the memory usages and the training time in all scaling conditions. This observation is in coherence with the GRU architecture, as its cell has one gate-unit less than that of the LSTM cell, which results in at-least 7 less trainable parameters per memory-cell. The lowest accuracy of the unscaled GRU model in terms of the SSP parameter is for the photodiode-1, which is about 92%.

Through the thesis work, it is seen that the learning methods, such as RNNs, are able to generalize a physical process through the experiences gained from the training examples. This is relatively new RNN application, as they are mainly used in other sequence modelling tasks, such as, natural language processing, video processing, etc. Thus presented modelling technique allows fast creation and deployment of system-models from the input-output data-space of a complex system, without having to acquire any domain specific knowledge. The only time-consuming task is identified to be finding the proper configuration of the model which properly fits the given dataset and underlying problem, nevertheless, once this is achieved, the RNN models are able to simulate the hardware behaviour and produce realistic outputs.

The use-case targeted in the study intends to use the learned hardware-model to assist the OBSW development process by integrating it into the continuous integration system of the OBSW. However, application of such modelling technique have much broader scope. Such models can be used in any other domain for any kind of applications. Thus defined modelling technique shall be used whenever there is a need of a black-box system model and the system's input-output data-space is defined and the training data can be collected. Nevertheless, the model shall be sufficiently and correctly trained before the desired accuracy is reached.

6.2 Future Outlook

The data collection procedure can be improved in the future. Currently, the input and target data vectors are constructed once all target values are found in the TCP byte-stream, thus the data collection rate is same as the data acquisition rate of the experiment (neglecting the latency of the communication channels). This set-up works up-to certain extend when the number of targets are less and the noise cause by the time-lags in the network are considerably lower and handleable by the model, however, once the threshold is breached, this set-up may cause problems. One solution to this problem could be to define a data-sampling rate outside of the communication channel and produce input and target samples at an adjustable frequency and update values once they are received from the hardware and maintain the previous value when there is a delay in the channel.

When dealing with signals from a hardware system for modelling the underlying physical process, it is imaginable that, at some point in time, one must deal with variably sampled data which are produced by different sensors in an asynchronous manner. The current modelling technique is only been tested with the regularly sampled dataset; the sampling rate is defined by the data acquisition rate of the experiment. There has already been publication released introducing new LSTM based memory cell known as the Phased-LSTM [56]. The author in [56] claims that the Phased-LSTM model naturally integrates irregularly sampled inputs through its resonating memory unit. Thus, designing the RNN topology to deal with such irregularly sampled sensor data can be a potential improvement of the thesis work.

Bibliography

[1] German Aerospace Center (DLR). *Research Report and Economic Development.* 2015. URL: `https://www.dlr.de/dlr/en/Portaldata/1/Resources/documents/2017/FUW_2014-2015_GB_180ppi.pdf`.

[2] J. Engblom, G. Girard, and B. Werner. "Testing Embedded Software using Simulated Hardware". In: *Proc. of ERTS Embedded Real Time Software and Systems.* Stockholm, Sweden, Jan. 2006, pages 1–9.

[3] Mathworks. *Pricing and Licensing.* 2019. URL: `https://www.mathworks.com/pricing-licensing.html?prodcode=SL&intendeduse=undefined`.

[4] D. Bailey, J. M. Borwein, A. Salehipour, et al. "Backtest Overfitting in Financial Markets". In: *Journal of Automated Trader Magazine, Issue 39* 2 (May 2016).

[5] C. Luo, D. Yang, J. Huang, and Y.-D. Deng. "LSTM-Based Temperature Prediction for Hot-Axles of Locomotives". In: *Proc. of International Conference on Information Technology and Applications (ITA).* Beijing, China, June 2017, pages 1–6.

[6] Q. Zhang, H. Wang, J. Dong, et al. "Prediction of Sea Surface Temperature Using Long Short-Term Memory". In: *Journal of IEEE Geoscience and Remote Sensing Letters* 14.10 (Oct. 2017), pages 1745–1749.

[7] Z. C. Lipton, D. C. Kale, C. Elkan, and R. C. Wetzel. "Learning to Diagnose with LSTM Recurrent Neural Networks". In: *Proc. of International Conference on Learning Representations (ICLR).* San Juan, Puerto Rico, May 2016, pages 1–5.

[8] X. Shi, Z. Chen and H. Wang and D.-Y. Yeung, W.-k. Wong, and W.-c. Woo. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *Proc. of Neural Information Processing Systems.* Montreal, Canada, Dec. 2015, pages 802–810.

[9] G. White, A. Palade, and S. Clarke. "Forecasting QoS Attributes Using LSTM Networks". In: *Proc. of International Joint Conference on Neural Networks (IJCNN).* Rio, Brazil, July 2018, pages 1–9.

[10] H. Labbaci, B. Medjahed, and Y. Aklouf. "A Deep Learning Approach for Quality-Aware Long-term Service Composition". In: *Proc. of International Conference on Service-Oriented Computing.* Malaga, Spain, Nov. 2017, pages 1–8.

[11] N. Laptev, J. Yosinski, L. E. Li, and S. Smyl. "Time-series Extreme Event Forecasting with Neural Networks at Uber". In: *Proc. of International Conference on Machine Learning.* Sydney, Australia, Aug. 2017, pages 1–5.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (June 2014), pages 1929–1958.

[13] Y. LeCun, Y. Bengio, and G. Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pages 436–444.

[14] D. Gonzalez, Dr. S. Seidel, Dr. T. Driebe, and Prof. E. Rasel. *MAIUS 1 – First Bose-Einstein condensate generated in space.* URL: `https://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10081/151_read-20337/#/gallery/25194`.

[15] P. E. Wellstead. In: *Introduction to Physical System Modelling.* ACADEMIC PRESS LTD, 24/28 Oval Road London NW1, 1979, pages 12–16.

[16] J. McCarthy and E. Feigenbaum. "Arthur Samuel: Pioneer in Machine Learning". In: *Journal of AI Magazine* 11.3 (Sept. 1990).

[17] J. Wang and Q. Tao. "Machine Learning: The State of the Art". In: *Journal of IEEE Intelligent Systems* 23.6 (Nov. 2008), pages 49–55.

[18] C. M. Bishop. "Pattern Recognition and Machine Learning (Information Science and Statistics)". In: Berlin: Springer Science+Business Media, LLC, 2006.

[19] E. A. Lawer, A. Luguterah, and S. Nasiru. "Comparison of Artificial Neural Network and Binary Logistic Regression for Classification of Chiropteran Dietary Specializations". In: *International Journal of Statistics and Application* 4.4 (Apr. 2014), pages 204–211.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Representations by Back-propagating Errors". In: *Neurocomputing: Foundations of Research.* MIT Press, 1988, pages 696–699.

[21] Z. C. Lipton, J. Berkowitz, and C. Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *CoRR* abs/1506.00019 (June 2015).

[22] R. Pascanu, T. Mikolov, and Y. Bengio. "On the difficulty of training recurrent neural networks". In: *Proc. of International Conference on Machine Learning (ICML).* Atlanta, USA, 2013, pages 1–7.

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

[24] R. J. Williams and D. Zipser. "Gradient-based Learning Algorithms for Recurrent Networks and Their Computational Complexity". In: *Backpropagation*. L. Erlbaum Associates Inc., 1995, pages 433–486.

[25] F. A. Gers, J. Schmidhuber, and F. A. Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Journal of Neural Computation* 12.10 (2000), pages 2451–2471.

[26] Sepp Hochreiter, Yoshua Bengio, and Paolo Frasconi. "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies". In: *Field Guide to Dynamical Recurrent Networks*. IEEE Press, 2001.

[27] R. J. Williams and D. Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In: *Neural Computation* 1 (Sept. 1989), pages 270–280.

[28] A. Graves. "Generating Sequences With Recurrent Neural Networks". In: *Journal of CoRR* 1308.850 (Aug. 2013), pages 157–166.

[29] A. Graves, A. Mohamed, and G. E. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing*. Vancouver, Canada, May 2013, pages 1–5.

[30] A. Graves and J. Schmidhuber. "Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks". In: *Proc. of Neural Information Processing Systems (NIPS)*. Vancouver, Canada, Dec. 2008, pages 1–8.

[31] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Journal of Neural Computation* 9.8 (Nov. 1997), pages 1735–1780.

[32] T. Cooijmans, N. Ballas and C. Laurent, and A. C. Courville. "Recurrent Batch Normalization". In: *Proc. of International Conference on Learning Representations (ICLR)*. Toulon, France, Apr. 2017, pages 1–13.

[33] K. Cho, B. v. Merrienboer, C. Gulcehre, et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proc. of Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar, Oct. 2014, pages 1724–1734.

[34] J. L. Elman. "Finding Structure in Time". In: *Journal of Cognitive Science* 14.2 (1990), pages 179–211.

[35] J. Chung, C. Gülçehre, K. Cho, and Y. Bengio. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014).

[36] Wireshark. *About Wireshark*. URL: `https://www.wireshark.org/`.

[37] Keras. *Keras: The Python Deep Learning library*. URL: `https://keras.io/`.

[38] TensorFlow. *About TensorFlow*. URL: `https://www.tensorflow.org/`.

[39] The GPyOpt authors. *GPyOpt: A Bayesian Optimization framework in python*. 2016. URL: `http://github.com/SheffieldML/GPyOpt`.

[40] GPy. *GPy: A Gaussian process framework in python.* 2016. URL: `http://github.com/SheffieldML/GPy`.

[41] HPDA Team. *HPDA-Cluster.* 2019. URL: `https://wiki.dlr.de/display/SC/HPDA-Cluster`.

[42] M. Shanker, M. Y. Hu, and M. S. Hung. "Effect of data standardization on neural network training". In: *Omega* 24.4 (Aug. 1996), pages 385–397.

[43] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proc. of International Conference on Machine Learning (ICML).* Lille, France, July 2015, pages 448–456.

[44] L. Prechelt. "Early Stopping — But When?" In: *Neural Networks: Tricks of the Trade: Second Edition.* Springer Berlin Heidelberg, 2012, pages 53–67.

[45] Y. Bengio. "Practical Recommendations for Gradient-based Training of Deep Architectures". In: *Neural Networks: Tricks of the Trade: Second Edition.* Springer Berlin Heidelberg, 2012, pages 437–478.

[46] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. "Algorithms for Hyper-parameter Optimization". In: *Proc. of the 24th International Conference on Neural Information Processing Systems.* Granada, Spain, Dec. 2011, pages 2546–2554.

[47] A. Anghel, N. Papandreou, T. P. Parnell, et al. "Benchmarking and Optimization of Gradient Boosted Decision Tree Algorithms". In: *CoRR* abs/1809.04559 (Oct. 2018).

[48] K. Gurney. "Gradient Descent On An Error". In: *An Introduction to Neural Networks.* Bristol, PA, USA: Taylor & Francis, Inc., 1997. Chapter 5.

[49] Y. Feng, X. Huang, L. Shi, et al. "Learning with the Maximum Correntropy Criterion Induced Losses for Regression". In: *Journal of Machine Learning Research* 16 (2015), pages 993–1034.

[50] C. Liangjun, P. Honeine, Q. Hua, et al. "Correntropy-based robust multilayer extreme learning machines". In: *Pattern Recognition* 84 (2018), pages 357–370.

[51] D. P. Kingma and J. L. Ba. "Adam: A Method for Stochastic Optimization". In: *Proc. of IEEE International Conference on Learning Representations.* San Diego, CA, USA, May 2015, pages 1–15.

[52] I. Goodfellow, Y. Bengio, and A. Courville. "Machine Learning Basics". In: *Deep Learning.* MIT Press, 2016. Chapter 5, pages 96–161.

[53] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proc. of International Conference on Artificial Intelligence and Statistics.* Sardinia, Italy, May 2010, pages 249–256.

[54] M. Perlin and M. D. Bustamante. "A robust quantitative comparison criterion of two signals based on the Sobolev norm of their difference". In: *Journal of Engineering Mathematics* 101.1 (Dec. 2016), pages 115–124.

[55] Scapy. *Welcome to Scapy's documentation!* 2019. URL: `https://scapy.readthedocs.io/en/latest/`.

[56] D. Neil, M. Pfeiffer, and S. C. Liu. "Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences". In: *Proc. of Neural Information Processing Systems (NIPS)*. Barcelona, Spain, Dec. 2016, pages 1–9.

APPENDIX A

Class Definitions

In this chapter, definitions of the Python classes developed within this study are given.

A.1 FilenameProvider

FilenameProvider is responsible for maintaining the file paths for the entire project. The class is initialized with an instance of the configuration object and a *work-space* directory. The class defines getter methods to construct and return respective filenames. Moreover, filepath existence are checked and directories are created whenever necessary.

A.2 DataProvider

DataProvider provides methods for data collection and preparation procedures. It defines methods for data normalization as explained in Section 3.2.1 as well as for the de-normalization process. Moreover, it defines methods for reshaping and splitting the input/target matrices as defined in Section 3.2. Pattern search algorithm explained in Section 4.3.2 is also implemented in this class. The class takes following constructor arguments:

1. *config*: Configuration object.
2. *fn_provider*: FilenameProvider object.
3. *normalization_type*: Enumeration to determine the data-normalization technique.
    - "std": Data normalization to have zero mean and unit variance.

- "-1and1": Data normalization between -1 and 1.
- "0and1": Data normalization between 0 and 1.

4. *time_frames*: Integer value to determine the sequence length for the RNN models.
5. *load_data*: Boolean value, when true the pattern search algorithm is triggered automatically.
6. *debug_mode*: Boolean value, when true prints debug informations, write a portion of input/target matrices in text file for manual inspection, plots and saves feature and target data distributions.

A.3 ModelWrapper

The ModelWrapper class extends CommonModel class, which is written on top of the Keras model. It facilitates easy parametrization and deployment of different network topologies, with better code re-usability. CommonModel holds the Keras model instance and other common functionalities which can be shared by other model-wrappers. Common functionality include, training and testing procedures, callback function definitions, metric computations, and the method to plot results. High-level ModelWrapper class is therefore only responsible for the network architecture. The ModelWrapper class takes following constructor arguments:

1. *config*: Configuration object.
2. *fn_provider*: FilenameProvider object.
3. *data_provider*: DataProvider object.
4. *name*: String value define the name of the model. All output files produced in the training, testing process is identifiable via model name.
5. *mini_batch_size*: Integer value as mini-batch size for the training.
6. *epochs*: Integer value give the number of epoch for training.
7. *loss_func*: String name or loss function method object.
8. *rnn_layers*: Number of RNN layers to use in the memory block (see figure 3.4).
9. *rnn_units*: Integer value or tuple determine number of RNN units to use in the memory block. When integer value is given, same number of units are used in all layers. Otherwise, elements in tuple determine the number of units, in this case, number of element in the tuple must be same as the number of layers.
10. *rnn_activation*: Activation function name or instance for forward signals in RNN layers.
11. *rnn_r_activation*: Activation function name or instance for recurring signals in RNN layers.

12. *rnn_dropout*: Dropout rate as float for forward signals in RNN layers.
13. *rnn_r_dropout*: Dropout rate as float for recurring signals in RNN layers.
14. *rnn_k_reg_l2*: L_2 regularization factor as float for forward signals in RNN layers.
15. *rnn_r_k_reg_l2*: L_2 regularization factor as float for recurring signals in RNN layers.
16. *rnn_k_init*: Kernel initializer name or instance for the weights of forward signals in RNN layers.
17. *rnn_r_k_init*: Kernel initializer name or instance for the weights of recurring signals in RNN layers.
18. *t_dist_layers*: Number of time-distributed dense layers.
19. *t_dist_units*: Integer value or tuple determine number of units in the time-distributed dense layers.
20. *t_dist_activation*: Activation function name or instance for the time-distributed dense layers.
21. *t_dist_dropout*: Dropout rate as float for the time-distributed dense layers.
22. *t_dist_k_reg_l2*: L_2 regularization factor as float for the time-distributed dense layers.
23. *t_dist_k_init*: Kernel initializer name or instance for the weights between time-distributed dense layers.
24. *adam_lr*: Learning rate in float for the Adam algorithm (see Section 3.4.6).
25. *adam_lr_decay*: Decay rate of the learning rate in float for Adam.
26. *adam_beta1*: Decay rate of the first moment in float for Adam.
27. *adam_beta2*: Decay rate of the second moment in float for Adam.
28. *bn_axis*: Normalization axis for all BN layers. *bn_axis* $\in [-1, 1, 2]$ such that:
    - -1: Normalize along sequence as well as feature axis.
    - 1: Normalize along sequence axis only.
    - 2: Normalize along feature axis only.
29. *bn_eps*: Value of ϵ_{BN} in float (see Equation 3.5)
30. *shuffle*: Boolean value, if true the inputs are snuffled in N axis after each epoch, such that the network always sees the samples in different order.
31. *save_weights*: Boolean value, if true optimum weight is saved at the end of each epoch (see Section 3.3.3).
32. *verbose*: Boolean value determines the verbosity.

Such implementation allows variety of network architectures to be trained and tested without code duplication, thus it is easy to maintain and extend the setup for different kinds of models. Furthermore, since all hyper-parameters are parametrized, applying automatic model optimization techniques such as Bayesian optimization is convenient.

APPENDIX B

Models

B.1 Model Settings

The hyper-parameter search determines the model's settings. In this section, the settings for different models are listed. The settings for the LSTM and GRU layers are similar, therefore are listed together and are commonly referred as *rnn* in table B.1.

B.1.1 Settings for the One-to-One Model

Values of the hyper-parameter for the one-to-one model are given in table B.1. Hyper-parameters from 1 to 9 are the general settings for all layers of the model, thus, they are shared by the memory and the time-distributed dense blocks. Hyper-parameters from 10 to 19 are specific for the memory block of the model and the ones from 20 to 25 are for the time-distributed dense blocks. The model architecture is described in Section 3.3.1. The name of the hyper-parameter in the table reflect the constructor arguments of the *ModelWrapper* class described in appendix A.3.

#	**Name**	**Value**	**Comment**
1	*mini_batch_size*	256	Mini-batch size for training, Section 3.4.4
2	*epochs*	10	Number of epochs to train, Section 3.4.3
3	*loss_func*	$\mathcal{L}\{CEC\}$	Cost function, Section 3.4.1; CEC loss, Section 3.5.2.
4	*adam_lr*	0.002	Learning rate for Adam, Section 3.4.6

5	*adam_lr_decay*	0.01	Decay rate of the learning rate for Adam, Section 3.4.6
6	*adam_beta1*	0.99	Decay rate of the first moment for Adam, Section 3.4.6
7	*adam_beta2*	0.999	Decay rate of the second moment for Adam, Section 3.4.6
8	*bn_axis*	-1	Feature as well as sequence axis is normalized by the BN layer, Section 3.3.
9	*bn_eps*	0.05	Value of ϵ_{BN} in Equation 3.5, Section 3.3.
10	*rnn_layers*	1	Number of RNN layers in the memory block, Section 3.3.1.
11	*rnn_units*	50	Number of units in each RNN layers of the memory block, Section 3.3.1.
12	*rnn_activation*	Tanh	Activation function (Section 3.4.5) for the forward signals in each RNN layers.
13	*rnn_r_activation*	Sigmoid	Activation function (Section 3.4.5) for the recurring signals in each RNN layers.
14	*rnn_dropout*	0.0	Dropout rate (Section 3.4.9) for the forward signals in each RNN layers.
15	*rnn_r_dropout*	0.0	Dropout rate (Section 3.4.9) for the recurring signals in each RNN layers.
16	*rnn_k_reg_l2*	0.4	L_2 regularization factor (Section 3.4.8) for the forward signals in each RNN layers.
17	*rnn_r_k_reg_l2*	0.99	L_2 regularization factor (Section 3.4.8) for the recurring signals in each RNN layers.
18	*rnn_k_init*	*glorot_uniform*	Initializer (Section 3.4.7) for the weights of forward signals in each RNN layers.
19	*rnn_r_k_init*	*orthogonal*	Initializer (Section 3.4.7) for the weights of recurring signals in each RNN layers.
20	*t_dist_layers*	5	Number of time-distributed dense layers, Section 3.3.
21	*t_dist_units*	100	Number of units in each time-distributed dense layers, Section 3.3.
22	*t_dist_activation*	ReLu	Activation function (Section 3.4.5) for each time-distributed dense layers.
23	*t_dist_dropout*	0.3	Dropout rate (Section 3.4.9) for each time-distributed dense layers.
24	*t_dist_k_reg_l2*	0.6	L_2 regularization factor (Section 3.4.8) for each time-distributed dense layers.

25	*t_dist_k_init*	*orthogonal*	Initializer (Section 3.4.7) for the weights between time-distributed dense layers.

Table B.1: Settings of the one-to-one model. The settings is valid for the GRU as well as the LSTM model instances.

B.1.2 Settings for the One-to-Many Model

The Table B.2 lists the settings of the one-to-many model found through the hyper-parameter search algorithm. However, the settings values are only course-searched.

The single values are valid for all neural branches of the one-to-many model. The three comma separated values are for the neural branch of the photodiode, rotatory encoder, and the temperature sensor, respectively.

#	**Name**	**Value**	**Comment**
1	*mini_batch_size*	256	Mini-batch size for training, Section 3.4.4
2	*epochs*	10	Number of epochs to train, Section 3.4.3
3	*loss_func*	$\mathcal{L}\{CEC\}$	Cost function, Section 3.4.1; CEC loss, Section 3.5.2.
4	*adam_lr*	0.002	Learning rate for Adam, Section 3.4.6
5	*adam_lr_decay*	0.01	Decay rate of the learning rate for Adam, Section 3.4.6
6	*adam_beta1*	0.99	Decay rate of the first moment for Adam, Section 3.4.6
7	*adam_beta2*	0.999	Decay rate of the second moment for Adam, Section 3.4.6
8	*rnn_layers*	1	Number of RNN layers in the memory block, Section 3.3.1.
9	*rnn_units*	50	Number of units in each RNN layers of the memory block, Section 3.3.1.
10	*rnn_activation*	Tanh	Activation function (Section 3.4.5) for the forward signals in each RNN layers.
11	*rnn_r_activation*	Sigmoid	Activation function (Section 3.4.5) for the recurring signals in each RNN layers.
12	*rnn_dropout*	0.0	Dropout rate (Section 3.4.9) for the forward signals in each RNN layers.
13	*rnn_r_dropout*	0.0	Dropout rate (Section 3.4.9) for the recurring signals in each RNN layers.
14	*rnn_k_reg_l2*	0.4	L_2 regularization factor (Section 3.4.8) for the forward signals in each RNN layers.

15	*rnn_r_k_reg_l2*	0.99	L_2 regularization factor (Section 3.4.8) for the recurring signals in each RNN layers.
16	*rnn_k_init*	*glorot_uniform*	Initializer (Section 3.4.7) for the weights of forward signals in each RNN layers.
17	*rnn_r_k_init*	*orthogonal*	Initializer (Section 3.4.7) for the weights of recurring signals in each RNN layers.
18	*rnn_bn_axis*	-1	Feature as well as sequence axis is normalized by the BN layer, Section 3.3.
19	*rnn_bn_eps*	0.1	Value of ϵ_{BN} in Equation 3.5, Section 3.3.
20	*t_dist_layers*	2, 2, 3	Number of time-distributed dense layers, Section 3.3.
21	*t_dist_units*	60, 50, 60	Number of units in each time-distributed dense layers, Section 3.3.
22	*t_dist_activation*	ReLu	Activation function (Section 3.4.5) for each time-distributed dense layers.
23	*t_dist_dropout*	0.3	Dropout rate (Section 3.4.9) for each time-distributed dense layers.
24	*t_dist_k_reg_l2*	0.4, 0.4, 0.9	L_2 regularization factor (Section 3.4.8) for each time-distributed dense layers.
25	*t_dist_k_init*	*orthogonal*	Initializer (Section 3.4.7) for the weights between time-distributed dense layers.
26	*t_dist_bn_axis*	-1	Feature as well as sequence axis is normalized by the BN layer, Section 3.3.
27	*t_dist_bn_eps*	0.1	Value of ϵ_{BN} in Equation 3.5, Section 3.3.

Table B.2: Settings of the one-to-many model. The settings is valid for the GRU as well as the LSTM model instances.