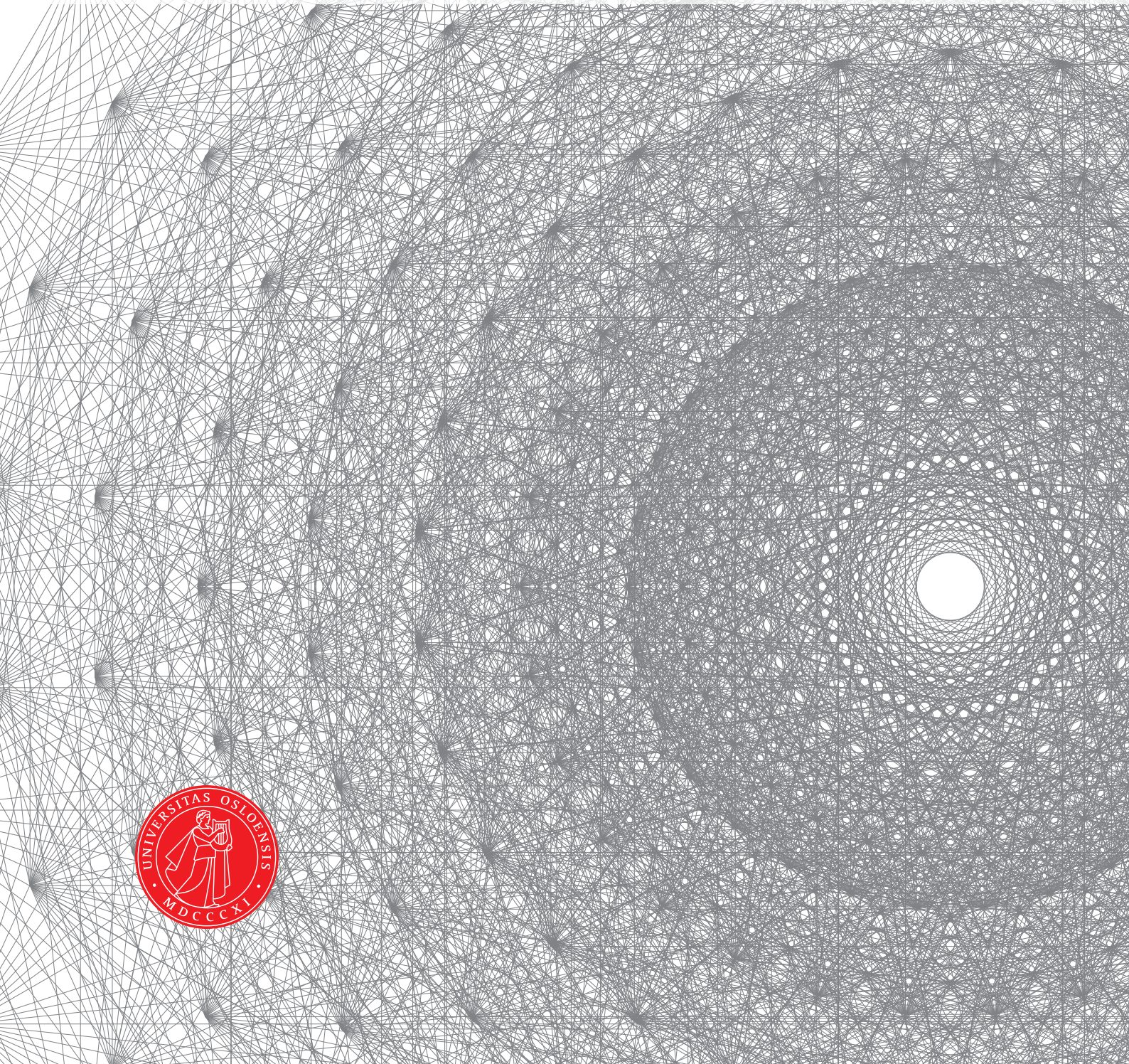
UiO : Department of Mathematics
University of Oslo

Recurrent Neural Networks for predicting ship motor temperatures

aiming to help prevent motor overheating

He Gu

Master's Thesis, Spring 2021



This master's thesis is submitted under the master's programme *Data Science*, with programme option *Data Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

For electric propulsion motors installed on marine vessels, the prevention of overheating is usually based on resistor temperature detector sensors. The monitoring system raises an alarm if the detected temperatures reach a predefined safety limit. Field experience reveals, however, that damage to the motors may have already occurred before the alarm is triggered due to the delay in the heat transfer process. This thesis aims at developing a data-driven approach to detect the real-time anomalies of motor temperatures, based on extensive field data provided by ABB, a Swedish–Swiss multinational technology corporation. This study consists of two parts. First, a novel algorithm is presented to enable the use of the enormous dataset with the available computing resources and time. Second, a data-driven approach is developed to predict the motor temperatures under a normal navigating state, based on convolutional neural network and long short-term memory models. The anomalies of motor temperature can then be identified by comparing the predicted normal temperatures and the ones detected by sensors. Compared to a baseline linear model, the developed approach provides an improvement of approximately 73.94% with respect to the mean squared error. The current study also investigates the performance of the developed approach in multiple simulated scenarios that can be of practical interest. The thorough evaluations have suggested a substantial potential of the approach with respect to its practicality, generalization, and precision.

Acknowledgements

Completion of this research work has only been made possible through the valuable contributions of a number of people over the past two years. It is a memorable experience to work with these respectable people, and I am grateful for the opportunity to write a master thesis on this topic provided by ABB and BigInsight.

I would like to express my sincere gratitude to my main supervisor Kristoffer H. Hellton, who has been a constant source of encouragement, guidance, support, and ideas throughout the whole duration of this work.

I would also like to thank my co-supervisors Morten Stakkeland and Prof. Ingrid K. Glad, for their precious insights and help.

A big thank you to all the unknown staff and associates at ABB, UiO, and BigInsight who have offered assistance.

Finally, I would like to thank all my family and friends for their endless encouragement and support.

Contents

CHAPTER 1

Introduction

1.1 Motivation and goal

For ships driven by electric propulsion motors, the prevention of overheating is one of the most crucial safety concerns. Any damage to the motors may not only cause a severe safety hazard, but also lead to substantial economic loss.

The standard practice of overheating prevention is usually based on a group of resistor temperature detection (RTD) sensors. The security system is designed such that it raises an alarm once the temperature reaches a warning limit, and ultimately shuts down the corresponding motors or systems if a higher trip limit is reached (typically at $155°C$ for the motors studied in this thesis).

These systems, however, have several limitations. The locations and number of installed sensors are limited, and the areas of overheating can differ from the monitoring spots. It takes a certain period for the heat to transfer from the actual overheating areas to the measurement locations such that RTD sensors can trigger the preset alarms. This duration can be short, but the consequent destruction to the motor may be catastrophic. In a fault situation, damage to the motors has been observed already by the time the trip limit was reached. (Hellton et al. 2021).

In the past, most efforts have been made in the development of physical models and predictive tools based on thermophysical knowledge and dedicated experiments. However, such approaches commonly demand a large amount of time, detailed knowledge, computing resources, and manpower. Furthermore, the derived approaches and models are usually customized to monitor a specific motor system. They can rarely form a generic tool that can be applied to other systems. The lack of a generic approach has therefore made this study meaningful, which revisits the problem by means of machine learning methods.

Through predicting the normal temperatures of a motor, the monitoring approach can then identify abnormal motor temperatures by comparing the detected temperatures measured by the sensors with the predicted normal-state temperatures. However, this thesis only aims at developing an adaptive data-driven approach based on machine learning methods, which can predict

the normal temperature of motors. Further study about the prevention of overheating is not discussed in this thesis, since it would involve some type of sequential analysis of residuals, which is beyond the scope of this work.

1.2 System overview

The exact setup of the electric propulsion system of ships differs between different ships, both with respect to the design of the motors and the configuration of the cooling system. In this thesis, we focus on the derivation of a generic data-driven approach with minimal emphasis on the physical models. Figure 1.1 shows the schematic of a generic electric motor system provided by ABB (Hellton et al. 2021). Despite their differences in mechanical design, the topologies of ship motors are however similar and consist of a propulsion control unit (PCU) and a multi-stage cooling system (air and water cooled).

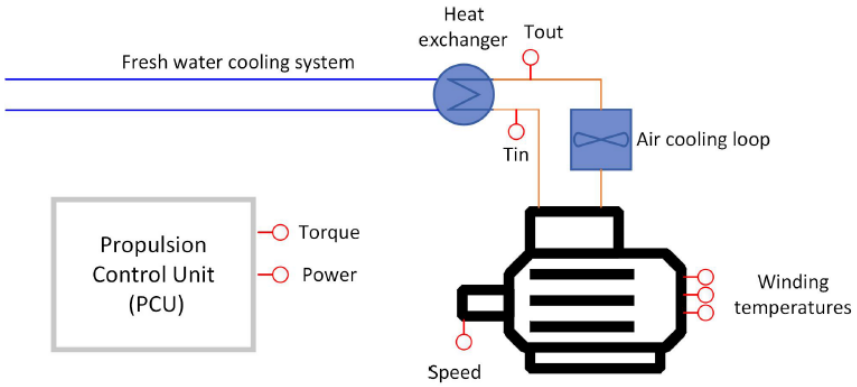


Figure 1.1: A schematic overview of the studied motor system. Tin and Tout correspond to features CI and CO in the given dataset. They are the temperatures of the inlet cooling air and the outlet cooling air, respectively. Torque, Power, and Speed correspond to features TO, PO, and SP, respectively. Six winding temperatures are recorded at two separate triplets, namely $U1$, $U2$, $V1$, $V2$, $W1$, and $W2$.

The motor itself is cooled by cold air circulated by multiple fans, illustrated as the air cooling loop in Figure 1.1. The heat in the hot air is removed by an on-board water-cooling system through a heat exchanger.

During the normal operation, the heat generated by the motor can be continuously removed by circulating cold water. However, under the circumstances of high torque demands or full load, the burst of high-power usage can result in abundant heat and rapid temperature increase, which leads to overheating.

1.3 Dataset

The dataset used in this thesis is generated by the motor systems of several vessels. Eleven features of the dataset were collected, which are Tin, Tout, Torque, Power, Speed, and 6 Winding temperatures, as shown in Figure 1.1. Tin and Tout represent the inlet and outlet temperatures of cooling air, which were provided by RTD temperature sensors.

The mechanical torque and electrical power were calculated and recorded by the Propulsion Control Unit (PCU), while the rotation speed of motors was directly collected from the motors. The six winding temperatures were recorded in two separate triplets and are highly similar.

In the development of the current data-driven model, an enormous amount of field data provided by ABB are utilized. In order to anonymise the dataset, the data were modified in a manner where the underlying dynamics of the system was preserved. The data were sampled at a temporal resolution of 1 Hz in the period from September 2010 to November 2011.

In total, 16 ships were included as the sampling sources, each of which has measurements from three different motors. This yields around 2.2 billion observations made available for the analysis in this thesis. Besides the timestamps of recordings, each observation also consists of 11 more entities, which are:

- Inlet and outlet temperatures of cooling air.
- Mechanical torque and electrical power, which were calculated and recorded by the PCU.
- Rotation speed of motors, which is taken directly from the motor output.
- Winding temperatures at two separate triplets, which are measured by 6 RTD sensors. When assessing the time series, the differences among them are shown insignificant.

Following the convention of a typical statistical or machine learning method, the data are assorted into 3 categories, i.e., timestamps, features, and targets. The following entities are utilized as features:

- CI (inlet cooling air temperature [$^{\circ}C$]).
- CO (outlet cooling air temperature [$^{\circ}C$]).
- PO (normalized motor power [%]).
- SP (normalized motor speed [%]).
- TO (normalized mechanical torque [%]).

which are used to predict the following targets (i.e., temperatures):

- $U1$, $V1$, and $W1$ (winding temperatures in triplet 1 [$^{\circ}C$]).
- $U2$, $V2$, and $W2$ (winding temperatures in triplet 2 [$^{\circ}C$]).

1.4 Previous related research

Following the recent prosperity of data mining and machine learning, some similar research was carried out at the Norwegian Computing Center and ABB. They developed a data-driven approach to resolve the same issues as described in this thesis (Hellton et al. 2021). Although the given datasets are different, the studied system in their research is the same as that of this thesis, as shown in Figure 1.1.

However, instead of employing machine learning as this thesis does, their research focused on a retrofitted linear model. Therefore, their developed approach relies to a larger extent on field experience and prior physical knowledge. As an extension of the previous study, the current work developed a more generic approach which may hopefully be employed with less prior knowledge.

1.5 Hardware specifications and software frameworks

The training, validation, and test of models are essential parts of the work in this thesis. Handling the vast amount of data demands a great deal of computing resources and efficient data analysis algorithms.

To reflect the computational expenses, we have discussed the computing time and resource as the work proceeds in this thesis. They are used as important measurements for manifesting the efficiency of an approach and also give an overview of the computational resources needed to perform the model training.

In this study, a high-performance computer cluster hosted at the Department of Mathematics, University of Oslo, was used. The corresponding hardware specifications are given in Table 1.1. It is worthwhile noticing that the hardware resources are not always fully employed during this study. Rather, they provide an upper limitation on the computing power which we can use in carrying out the present research.

Component	Model	Specifications
CPU	Intel(R) Xeon(R) CPU E5-2680 v3 ×48	Number of Cores: 12 Number of Threads: 24 Base Frequency: 2.5 GHz Max Frequency: 3.3 GHz
GPU	NVIDIA RTX 2080 Ti × 4	Core Clock: 1635MHz Memory Capacity: 11 GB
Memory	/	203 GB

Table 1.1: Hardware specifications of accessible computing resources, provided by University of Oslo (UiO).

For reference, the versions of several essential packages and drivers used in this study are tabulated in Table 1.2. It is expected that different versions can possibly lead to different computing time and memory demands.

Name	Type	Version
CUDA	toolkit	10.2.0
CUDA driver	driver	r440
cuDNN	library	8.0.1
Python	/	3.6.8
Keras	package	2.4.3
TensorFlow	package	2.3.1

Table 1.2: Software frameworks used in this thesis.

CHAPTER 2

Background theory

This chapter introduces the background theory employed in this thesis and describes in detail the preprocessing methods, basic statistical and machine learning models, and general training techniques.

2.1 Data preprocessing

Data preprocessing is a necessary preparation step for most statistical and machine learning methods. This is also the case for the principal machine learning models applied in this thesis, i.e., convolutional neural net (CNN) and long short-term memory method (LSTM).

Practically, data preprocessing needs to be customized for different dataset, model architectures, and the objectives of the task at hand. However, this section puts more focus on some universal preprocessing methods, which are applied in this thesis. The specific implementations in this thesis are given in Section 3.4.

2.1.1 Standardization

Standardization is one of the common prerequisites for most machine learning methods. Basically, standardization centers the mean and scales the input data into unit variance. Given inputs $(x_1, x_2, ..., x_n)$, the transformation of an input x_i after standardization is then given as

$$x_i' = \frac{x_i - \overline{x}}{s}. \tag{2.1}$$

Here, $\overline{x}$ and s are respectively the mean and the standard deviation of these inputs, and x_i' is the standardized input of x_i.

As found by Pedregosa et al. 2011, standardization may lead to an undesired performance if each individual feature of the given dataset does not follow a normal distribution. However, even if the given data in this thesis are not normally distributed, standardization still turns out to be the best-performing method, as discussed in Section 3.4.4.

2.1.2 Scaling

The scaling scales the given inputs into a certain predefined range. Among different scaling methods, min-max scaling is one of the most commonly used, and the min-max scaled values can be calculated by

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}(r_{max} - r_{min}) + r_{min}, \tag{2.2}$$

where x_{min} and x_{max} are the minimum and maximum values of the given inputs, respectively, while r_{min} and r_{max} delimit the scaling range. A practical range can be the unit interval $[0, 1]$, i.e., $r_{min} = 0$ and $r_{max} = 1$.

Another similar scaling method as the min-max scaling is the maximum absolute value scaling. As described by its name, this method scales each input x by the maximum absolute value of all inputs as

$$x_{scaled} = \frac{x}{|x|_{max}}. \tag{2.3}$$

After the transformation, the maximum absolute value of the inputs will be 1.

2.1.3 Normalization

Unlike scaling or standardization, the normalization normalizes each sample independently. It is commonly used in applications which are sensitive to distance metrics, e.g., clustering, since it normalizes each observation into a unit norm as

$$\boldsymbol{x}_{normalized} = \frac{\boldsymbol{x}}{||\boldsymbol{x}||}, \tag{2.4}$$

where the definition of norm $||\boldsymbol{x}||$ may vary from case to case, depending on the actual applications. $\boldsymbol{x}$ is an observation vector which consists of p variables (features) as

$$\boldsymbol{x} = (x_1, x_2, ..., x_p). \tag{2.5}$$

2.2 The baseline models

This thesis focuses on a supervised task and develops a supervised approach, i.e., utilizing multiple features to predict one target. Such an approach demands a set of criteria for evaluation, and one of the most common criteria is to introduce a baseline approach. By comparing the performance of our developed approach with that of the baseline, it is then possible to evaluate our approach. For example, a random approach is typically used as the baseline. If a given approach surpasses the random one, it can then be granted as effective.

However, we are here able to introduce a more appropriate baseline method. The previous related research, which was published by ABB and Norwegian Computing Center, indicated that linear regression models can be a superior baseline to the random approach (Hellton et al. 2021). Furthermore, we also propose the feedforward neural network model as an alternative baseline method, since it is worthwhile to illustrate that our model, which is based on recurrent neural network, will surpass a vanilla feedforward neural network.

2.2.1 Linear model

Given p independent variables $\{x_{i1}, x_{i2}, ..., x_{ip}\}$ and a corresponding dependent variable y_i, a linear model assumes a linear relationship between them, given as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \varepsilon_i. \tag{2.6}$$

Here, $\{\beta_0, \beta_1, ..., \beta_p\}$ are regression coefficients and ε is an error term under the assumptions of

$$\begin{cases} E[\varepsilon_i] = 0 \\ Var(\varepsilon_i) = \sigma^2 \\ Cor(\varepsilon_i, \varepsilon_j) = 0, \forall i \neq j \end{cases},$$

such that the error variance is considered constant for different values of y.

The matrix form of this model is given as

$$y_i = X_i \boldsymbol{\beta} + \varepsilon_i, \tag{2.7}$$

where X is a $(p+1)$-dimensional row vector, i.e., $(1, x_{i1}, x_{i2}, ..., x_{ip})$, and $\boldsymbol{\beta}$ is a $(p+1)$-dimensional column vector, i.e., $(\beta_0, \beta_1, ..., \beta_p)^T$.

In order to fit the model into n given observations $(X_1, y_1), (X_2, y_2), ..., (X_n, y_n)$ and search for an optimal $\boldsymbol{\beta}$, we aim at minimizing a loss function, which is usually the mean squared error (MSE)

$$Loss = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{2.8}$$

We see therefore that

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{n} (y_i - X_i \boldsymbol{\beta})^2 \right\}, \tag{2.9}$$

which can be given in matrix form as

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\{ (\boldsymbol{y} - X\boldsymbol{\beta})^T (\boldsymbol{y} - X\boldsymbol{\beta}) \right\}, \tag{2.10}$$

where $X = \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_n} \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}$ and $\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$.

We define that

$$D(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - X_i^T \boldsymbol{\beta})^2 = (\boldsymbol{y} - X\boldsymbol{\beta})^T (\boldsymbol{y} - X\boldsymbol{\beta}), \tag{2.11}$$

then, through solving the equation

$$\frac{\partial D(\hat{\boldsymbol{\beta}})}{\partial \hat{\boldsymbol{\beta}}} = 0, \tag{2.12}$$

we then have that

$$\hat{\boldsymbol{\beta}} \quad = \quad (X^T X)^{-1} X^T \boldsymbol{y}, \tag{2.13}$$

which is theoretically the optimally estimated parameter $\boldsymbol{\beta}$ that minimizes the loss function. The estimated prediction is then modeled by

$$\hat{y} = X\hat{\boldsymbol{\beta}}. \tag{2.14}$$

2.2.2 Neural network

Scientists have since the 1940s attempted to simulate the human brain and build a machine which is able to think like a human. Gradually, along with the discoveries in biology, it was identified that the functionality of human brains is based on neurons and axons. Electrical signals were sent by neurons through axons and eventually gather within certain parts of a brain. Our brain then analyses these signals and makes decisions (Cantile and Youssef 2015).

Inspired by such a mechanism, a (feedforward) neural network without hidden layers, which is called perceptron, was invented and implemented by Rosenblatt 1958, as illustrated in Figure 2.1.

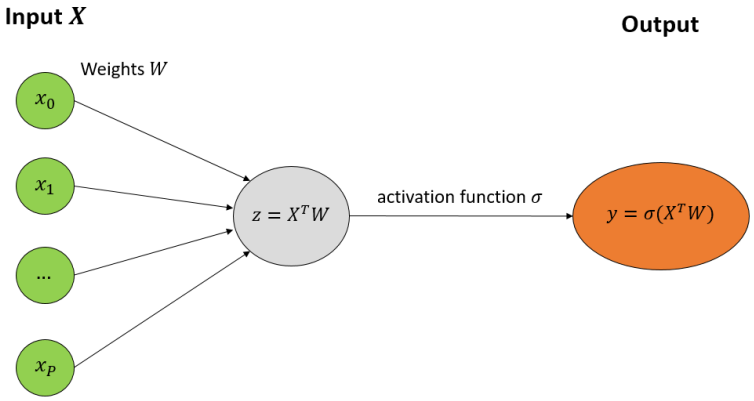


Figure 2.1: An overview of perceptron. z is an intermediate variable. The activation function is usually used to produce non-linearity for a perceptron model and varies for different situations.

It is proven that a single-layer perceptron is only capable of learning linearly separable patterns. However, it can be further improved by adding extra layers. The multi-layer perceptron was therefore introduced and is now also known as the (feedforward) neural networks (Minsky and Papert 2017). A typical structure of (feedforward) neural networks can be seen in Figure 2.2.

A neural network as in Figure 2.2 aims at relating p input variables $x_1, x_2, ..., x_p$ to q output variables $y_1, y_2, ..., y_q$ through m latent variables $z_1, z_2, ..., z_m$. The

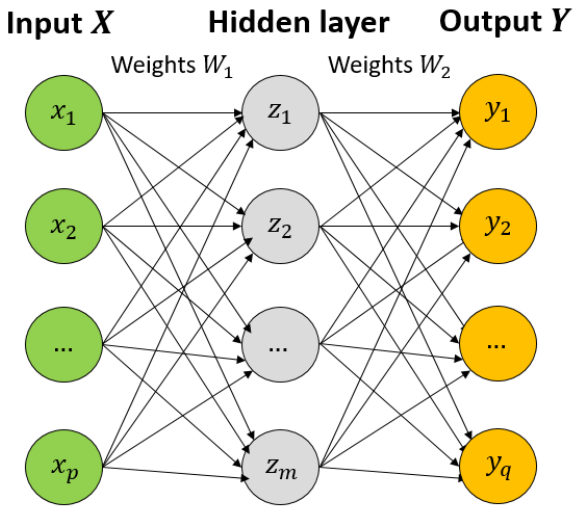


Figure 2.2: The overview of a 3-layer feedforward neural network. Each circle represents a node in the neural network. W_1 is a weight matrix used for calculating the value of nodes in the hidden layer, and W_2 is another weight matrix used for calculating the value of nodes in the output layer

relation between these variables is estimated through weights W and bias B, and is modelled by

$$\begin{cases} Z = \sigma(W_1 X + B_1) \\ Y = f(W_2 Z + B_2) \end{cases},$$

where X is a p-dimensional column vector $(x_1, x_2, ...x_p)^T$, Z is an m-dimensional column vector $(z_1, z_2, ...z_m)^T$, and Y is a q-dimensional column vector $(y_1, y_2, ...y_q)^T$. The function σ and f are so-called activation functions, which are designed to introduce non-linearity in the model.

In order to obtain parameter estimates, i.e., weights W and bias B, we select those which minimize a chosen loss function. Similar to a linear model, the squared loss can still be a natural choice as the loss function. However, rather than derive a formula which can directly calculate the optimal parameters, neural networks require a more practical approach, since they are far more complicated than linear models. It is often costly to derive formulas and even impossible for most neural network models.

Typically, we hence initialize the parameters into random values and update them progressively. The updates are advanced along the direction of the derivative of the loss function with respect to the updating parameter (Cauchy et al. 1847). If we denote parameters as θ_i, for $i = 1, 2, ...$, and denote the loss

function as $L(\theta)$, then the updates can be given as

$$\theta_{new} = \theta_{old} - \gamma \frac{\partial L(\theta)}{\partial \theta_i}\bigg|_{\theta=\theta_{old}}, \tag{2.15}$$

where γ is a so-called learning rate. It controls the step length for one update iteration and is determined by experience and experiments. The learning rate may also not be constant, but dynamic during the updating procedure. A typical practice is that we initialize the learning rate as a relatively large value and reduce it as we update the model, since larger learning rates can accelerate the speed of updating, while a smaller learning rate can ensure that the optimal parameters will not be passed by.

Furthermore, it may be beneficial to have more than 1 hidden layers for a neural network as in Figure 2.3. The corresponding calculation and optimization still follows the same procedure as in a 3-layer neural net.

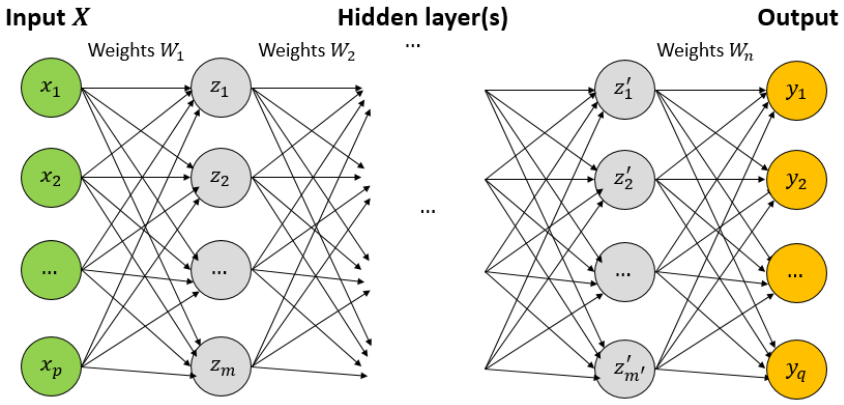


Figure 2.3: A multi-layer feedforward neural network. The input and output layers are typically counted for the number of layers. The (feedforword) neural network with n hidden layers are then named as an $(n+2)$-layer neural network. W_i is a weight matrix used for calculating the value of nodes in the $i+1$ layer.

2.3 Recurrent neural nets

There are many applications where the historical states influence the current state. A standard feedforward neural network or a linear model is not able to capture such characteristics, since these models focus only on the current state itself. The recurrent neural network (RNN) was therefore proposed as an extension by Rumelhart, G. E. Hinton and Williams 1986, which in addition takes the information of historical status into account.

2.3.1 Model

The recurrent neural nets assume that historical states have a hidden influence on the current state. This influence can be captured through a hidden state and will be transmitted into the future calculations. There will always be only 1 hidden state at 1 time point, although it keeps updating along the direction of the given sequence.

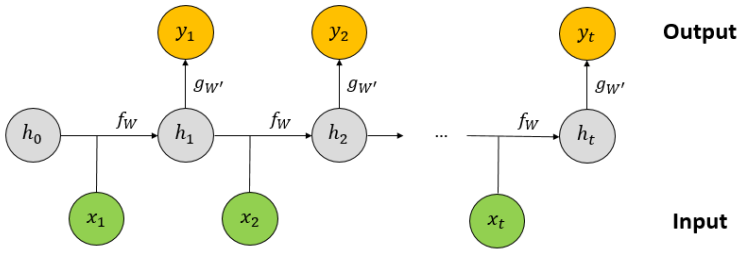


Figure 2.4: An overview of the recurrent neural network. Here, f_W and $g_{W'}$ are functions which are determined by matrices W and W', respectively, and h is the hidden state. f_W is used to calculate the hidden states while $g_{W'}$ is employed to produce the outputs.

The structure of recurrent neural nets can be illustrated in Figure 2.4, where

$$\begin{cases} h_t = f_W(h_{t-1}, x_t) \\ y_t = g_{W'}(h_t) \end{cases},$$

where f_W and $g_{W'}$ are functions which are determined by matrices W and W', respectively, and h is the hidden state.

Furthermore, the RNN model can also be given by

$$\begin{cases} h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \\ y_t = W_{hy}h_t + b_y \end{cases},$$

as in Figure 2.5, where σ represents an activation function. W_{hh}, W_{xh}, and W_{hy} are weight matrices, while b_h and b_y are bias matrices.

2.3.2 Optimization: backpropagation through time

In order to optimize the parameters of a given model, one has to define a metric beforehand, which can be a loss function, a utility function, etc. Here, the loss function is selected as an example to demonstrate the optimization. We have already illustrated the procedure of optimizing (updating) the parameters of a feedforward neural network in Equation (2.15) based on minimizing a loss function. In this section, we will further introduce an optimization

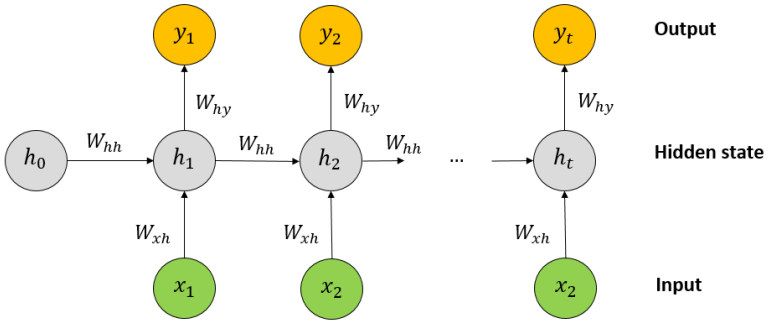


Figure 2.5: A detailed recurrent neural network. W_{hh} and W_{xh} are weight matrices used for calculating the current hidden states, based on the last hidden state and the current input, respectively. The outputs are then calculated by the current hidden state and the weight matrix W_{hy}.

approach for RNN, namely the backpropagation through time (BPTT), which was independently derived by numerous researchers. This approach will also achieve the optimization through minimizing the loss, as shown in Figure 2.6.

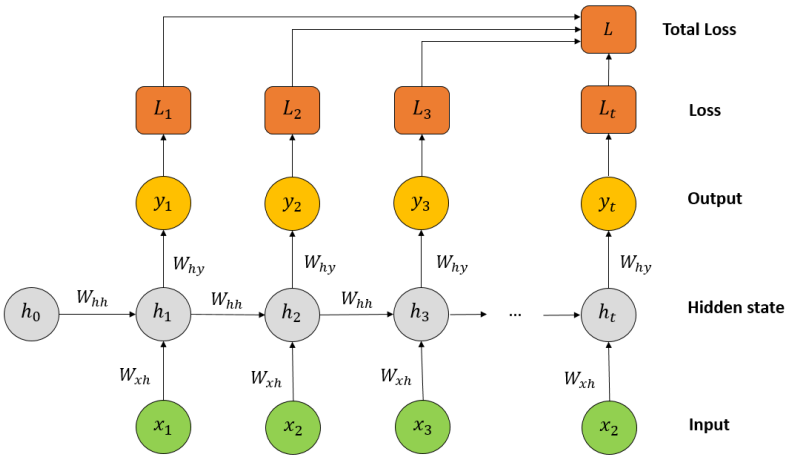


Figure 2.6: The overview of the loss of a recurrent neural network.

Based on the same idea of Equation (2.15), we aim at updating the parameters of an RNN along the direction of the derivatives of a loss function L with respect to each parameter, which is also known as gradient descent. According to the structure of RNN, the updating parameters of a recurrent neural net are the weight matrices W and bias matrices b, which indicates that the corresponding

updates should be respectively along the direction of $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$.

Recall that a (vanilla) recurrent neural network can be described as

$$\begin{cases} s_t = W_{hh}h_{t-1} + W_{xh}x_t + b_h \\ h_t = \sigma_1(s_t) \\ z_t = W_{hy}h_t + b_y \\ y_t = \sigma_2(z_t) \end{cases},$$

which indicates that

$$\begin{cases} \frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial z_t}\frac{\partial z_t}{\partial W_{hy}} \\ \frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial z_t}\frac{\partial z_t}{\partial W_{xh}} \\ \frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial z_t}\frac{\partial z_t}{\partial W_{hh}} \end{cases} \tag{2.16}$$

and

$$\begin{cases} \frac{\partial L}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial z_t}\frac{\partial z_t}{\partial b_y} \\ \frac{\partial L}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial z_t}\frac{\partial z_t}{\partial b_h} \end{cases}. \tag{2.17}$$

According to Equations (2.16) and (2.17), once given the loss function L and activation functions σ_1 and σ_2, $\frac{\partial L_t}{\partial z_t}$, $\frac{\partial z_t}{\partial W_{hy}}$, $\frac{\partial z_t}{\partial W_{xh}}$, $\frac{\partial z_t}{\partial W_{hh}}$, $\frac{\partial z_t}{\partial W_{by}}$, $\frac{\partial z_t}{\partial W_{bh}}$ will be also known. One may notice that apart from h_t, $\frac{\partial z_t}{\partial W_{xh}}$ and $\frac{\partial z_t}{\partial W_{hh}}$ is also related to h_{t-1}, h_{t-2},..., h_0. However, since h_0 is a pre-defined initial value, the derivatives $\frac{\partial z_t}{\partial W_{xh}}$ and $\frac{\partial z_t}{\partial W_{hh}}$ are still calculable. Therefore, the optimization of an RNN model can be progressed through the same updating procedure as in Equation (2.15).

2.3.3 Gradients explode and gradients vanish

The hidden state of an RNN follows the recursion

$$h_t = \sigma_1(W_{hh}h_{t-1} + W_{hx}x_t + b_h).$$

Therefore, both the derivatives $\frac{\partial z_t}{\partial W_{xh}}$ and $\frac{\partial z_t}{\partial W_{hh}}$ include the terms $\frac{\partial h_t}{\partial h_i}$, which can also be given as

$$\frac{\partial h_t}{\partial h_i} = (\sigma_1' W_{hh})^{t-i} \quad for \quad t = 0, 1, ..., t-1. \tag{2.18}$$

Since W_{hh} is a square matrix, we are able to factorize it using the eigenvalue decomposition

$$W_{hh} = V \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_r \end{pmatrix} V^{-1}, \tag{2.19}$$

where λ are eigenvalues of W_{hh}. We hence have that

$$(\sigma_1' W_{hh})^{t-i} = V \begin{pmatrix} \sigma_1'\lambda_1^{k-i} & & \\ & \ddots & \\ & & \sigma_1'\lambda_r^{k-i} \end{pmatrix} V^{-1}. \tag{2.20}$$

Along with an increasing time step t, the corresponding part inside this term may exponentially explode if $|\sigma_1' \lambda_i| > 1$, or may exponentially vanish if $|\sigma_1' \lambda_i| < 1$, and the derivatives $\frac{\partial L}{\partial W_{xh}}$ and $\frac{\partial L}{\partial W_{hh}}$ will therefore be influenced. Furthermore, since the weight matrices W_{xh} and W_{hh} will gain updates respectively along the direction of $\frac{\partial L}{\partial W_{xh}}$ and $\frac{\partial L}{\partial W_{hh}}$, the exploding or vanishing gradients will lead to difficulties during training.

Exploding gradients may break the limitation of given memory (RAM). However, it is a relatively acceptable challenge, as the existing methods are able to handle this problem. A typical practice is the gradient clipping, which sets a threshold to avoid exploding gradients.

If a gradient $\boldsymbol{g}$ tends to exceed the predefined threshold r, i.e., $||\hat{\boldsymbol{g}}|| \geq r$, it will then be replaced by a new gradient as

$$\hat{\boldsymbol{g}} \leftarrow \frac{r}{||\hat{\boldsymbol{g}}||} \hat{\boldsymbol{g}}. \tag{2.21}$$

Meanwhile, vanishing gradients are usually a more severe problem. The vanishing gradients will literally stop the update of the corresponding parameters, since the optimization is usually along the direction of gradients. In such cases, the updates may stop at a local saddle point where the corresponding parameters produce a sub-optimal performance. Unlike the gradient clipping approach, it can be complicated to design a proper remedy for vanishing gradients. It is difficult to implement a universal approach for such problems, since a considerable amount of tuning and experiments is often required for one specific task.

However, according to Equations (2.16), (2.17), and (2.18), an RNN will continue to update its parameters, even if certain gradients are vanishing. In contrast to, for instance, the feedforward neural networks, an RNN will always involve new inputs along the direction of the (time) sequence. Since the weight matrices and bias matrices are shared between the hidden states inside an RNN model, the new inputs will always bring fresh updates to its weights and bias.

Nevertheless, an RNN can still be influenced by vanishing gradients to some degree. Considering the architecture of an RNN, theoretically, it is supposed to capture the long-range dependencies, while practically, an RNN may have difficulties in achieving it because of the vanishing gradients. It was thoroughly discussed by Bengio, Simard and Frasconi 1994, and can also be proven through interpreting the Equations (2.16) and (2.17).

2.3.4 Weight and bias initialization

The initialization of weights and biases is an essential part of the training procedure. An inefficient initialization may slow down the speed of training and worsen the performance of models. In certain extreme cases, the updating of weights and bias will not converge, and we hence cannot progress the training. Meanwhile, a proper initialization can prevent exploding and vanishing gradients, accelerate the speed of training, and improve the performance of the estimated optimal parameters.

The choice of the initialization method depends on the choice of the activation function, the architecture of the model, the size of the dataset, and many other factors, and thus, there exists a considerable number of different method configurations. The different initializations all have their own advantages and limits, but their common objective is to ensure that the training procedure will successfully advance to a satisfactory solution.

In this thesis, we have a specific task with a given dataset, which indicates that it is not worth demonstrating all possible methods. This section hence only introduces the chosen initialization method, i.e., the orthogonal initialization (Saxe, McClelland and Ganguli 2014).

Orthogonal initialization is one of the best-performing initialization methods for RNN (Dauphin et al. 2014). For a given square matrix with shape $m \times m$, orthogonal initialization first creates a matrix A, which is filled with random observations drawn from a standard normal distribution. Here, $A \in R^{m \times n}$.

We can then decompose matrix A using Singular Value Decomposition (SVD), such that

$$A = U \Sigma V^T, \tag{2.22}$$

where $U \in R^{m \times m}$, $\Sigma \in R^{m \times n}$, and $V \in R^{n \times n}$. Here, U and V are orthogonal matrices, and Σ is a diagonal matrix with its diagonal values are the singular values of A. The orthogonal matrix U can then be taken as the initial weight matrix or bias matrix.

2.4 Advanced recurrent neural nets: long short-term memory (LSTM)

As stated before, although RNN can to some extent overcome the vanishing gradient problem, the ability of capturing long-range dependencies is still rather limited. Therefore, several advanced recurrent neural nets have been designed for long-range dependencies.

Among these advanced methods, gated recurrent unit (GRU) (Cho et al. 2014) and long short-term memory (LSTM) (Hochreiter and Schmidhuber 1997) are the most popular and prominent ones, since both their practical performance and theoretical designs have been shown to be outstanding. In this section, we will introduce the LSTM, which is the core structure of our model in this thesis.

In order to improve the performance regarding long-range dependencies, LSTM introduces the so-called gate mechanism. Instead of having 1 hidden state, LSTM applies a cell, also named as a unit, which receives 2 states from the former time point $t-1$ and transfers 2 updated states to the next time point $t+1$. These states are named as cell state c and hidden state h as shown in Figure 2.7.

Rather than using a straightforward matrix multiplication in RNN, the corresponding mechanism inside an LSTM cell is more complicated, as illustrated in Figure 2.8. The most essential part of a cell are the gates. There are typically 3 types of gates, which are named as forget gate, input gate, and output gate.

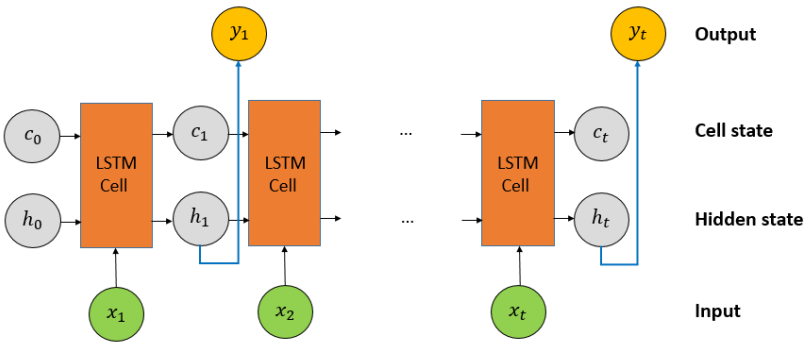


Figure 2.7: A recurrent neural network with long short-term memory (LSTM). Each circle represents a node in the neural network. Details of the LSTM cell are illustrated in Figure 2.8.

In this thesis, we represent the corresponding results of these gates at time point t respectively as f_t, i_t, $\tilde{c}_t$, and o_t.

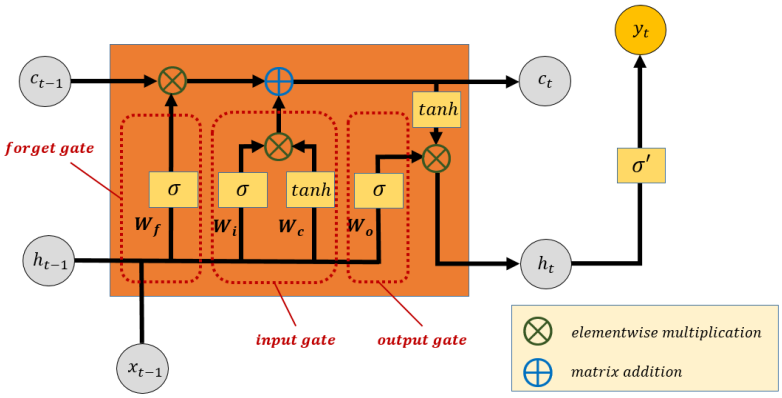


Figure 2.8: Schematic construction of an LSTM cell. Here, all three σ are the same activation function, while σ' can be different. c is the cell state, and h is the hidden state. x and y are the input and output, respectively. W_f, W_i, W_c, and W_o are weight matrices used for calculating the cell state and hidden state. Elementwise multiplication and the matrix addition are denoted by different notations as shown in the figure.

As described, the forget gate will control the information coming from the cell state c_{t-1} through

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f). \tag{2.23}$$

Here, $[h_{t-1}, x_t]$ is a concatenated vector of h_{t-1} and x_t. Since an activation

function σ is usually of a domain $[0, 1]$, f_t also has a domain $[0, 1]$. Then, by calculating the elementwise multiplication between c_{t-1} and f_t, the forget gate extracts certain features of c_{t-1} for further usage while abandoning other unimportant ones.

In the input gate, we filter the new inputs x_t through an elementwise multiplication as well, in order to remember the important input information

$$\begin{cases} i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{c}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \end{cases}.$$

Cooperating with the forget gate, LSTM is then able to update its cell state c from c_{t-1} into c_t,

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t. \tag{2.24}$$

The LSTM at time point t now has updated the cell state c. It hereafter aims at producing an updated hidden state h_t, so that we can provide an eventual result, i.e., the prediction y_t at time point t. We gain the information captured by the former hidden state h_{t-1} and the new input x_t through

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o). \tag{2.25}$$

Combining the updated cell state c_t, the former hidden state h_{t-1}, and the new input x_t, the LSTM cell is now able to update its hidden state by

$$h_t = o_t \odot \tanh(c_t). \tag{2.26}$$

The eventual prediction is typically a transformation of h_t through an activation function as

$$y_t = \sigma'(h_t). \tag{2.27}$$

As for the optimization of an LSTM, since it is essentially a type of recurrent neural network (RNN), the optimization also follows the backpropagation through time procedure (BPTT), as introduced in Section 2.3.2.

CHAPTER 3

Methodology development and validation

The principal objective of this thesis is to construct an optimal model for predicting the target (i.e., the motor temperature). However, the enormous amount of data in the given dataset gives an additional challenge to the construction of the model, since both the allowed time and computing resources are limited. The tuning of hyperparameters and the validation of possible models and algorithms exacerbate the problem even more.

In this chapter, we will incrementally seek solutions to the encountered challenges, demonstrate the methodology of developing the optimal model, validate possible candidates, and eventually determine the final approach for the task of this thesis.

3.1 The validation-holdout split

The dataset provided by ABB was stored in multiple files, where each file contains the sensor data of one motor throughout one calendar month. The motors belong to different ships, and each ship typically has three motors. For each motor, the recording period lasted for 16 months, and the total number of recordings (observations) of all motors is around 2.2 billion.

This chapter aims at developing an approach which fulfills the task of this thesis. Therefore, it is necessary to use part of the dataset to validate possible candidates, for instance, different model architectures, different hyperparameters, and possible training algorithms.

Notice that we are given a large amount of data, namely around 2.2 billion. It is hence reasonable to use half of the dataset as the validation set. Meanwhile, the remaining half of the dataset will be used as the holdout (test) set, which is used to evaluate our final model in Chapter 4.

Since the key method explored in this thesis is an LSTM, it is necessary to maintain the connections of consequent recordings (observations) with respect to time. We cannot randomly assign each observation into the validation set or

the holdout set, as it will destroy the time ordering. Therefore, we assign each file randomly, instead of each observation, into one of these two sets.

However, it is worth noticing that several files belong to the same motor, or the motors from the same ship. Therefore, if we randomly assign each file into the validation set or the holdout set, the distribution of these two sets can be different. We hence attempt to seek other solutions for the validation-holdout split.

Recall that the recording of each motor lasted for 16 months and was therefore stored in 16 files. A simple solution is to randomly assign eight files of each motor into the validation set, and the remaining eight files into the holdout set. The distribution and characteristics of both sets will then become similar to each other, and the optimal approach determined by the validation set should hence reserve a satisfying generalization.

3.2 The initial settings

In this section, we will introduce several initial settings of our approach for convenience. If not specified otherwise, all related parameters and methods in this thesis will by default follow the descriptions in this section.

3.2.1 The shapes of inputs and outputs of LSTM

In Section 2.4, we explained the structure of an LSTM model. According to the definition, the input to an LSTM model is a 3-rank tensor X, which is given by

$$X = (X_1, X_2, ..., X_n),$$

where

$$X_i = \begin{pmatrix} \boldsymbol{x_i} \\ \boldsymbol{x_{i+1}} \\ \vdots \\ \boldsymbol{x_{i+t-1}} \end{pmatrix} = \begin{pmatrix} 1 & x_{i,1} & \cdots & x_{i,p} \\ 1 & x_{i+1,1} & \cdots & x_{i+1,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{i+t-1,1} & \cdots & x_{i+t-1,p} \end{pmatrix} \quad for \quad i = 1, 2, ..., n.$$

Here, $x_{j,k}$ is the k-th feature of the j-th observation. In this thesis, we refer to each X_i within the tensor X as a time sequence of p features, and t, i.e., the length of each time sequence X_i, as the sequence length. Then, n is the number of given time sequences in the dataset. If denote the number of given observations as N, then n is given by $n = N - t + 1$. This way, each column of X_i is a time sequence of a given feature.

For an input tensor X, the corresponding output is given by

$$Y = (\boldsymbol{y_t}, \boldsymbol{y_{t+1}}, ..., \boldsymbol{y_{t+n}}),$$

where

$$\boldsymbol{y_{t+i}} = \begin{pmatrix} y_{t+i,1} \\ y_{t+i,2} \\ \vdots \\ y_{t+i,q} \end{pmatrix}.$$

Here, q denotes the number of units of an LSTM, which corresponds to the dimensionality of its output space. Note that although our target in this thesis is a single value, i.e., the overall motor temperature, q does not have to be 1. Instead, there are other specific output structures, such as a dense layer, which convert the output of an LSTM into the actual target, namely the temperature in such cases.

3.2.2 The model

Our model is initially elementary. It will incrementally become more complicated and eventually turn into the final model. However, note that such a process is bidirectional. Although certain procedures may have already been completed, e.g., the feature selection, we can still withdraw our decisions afterwards, since the optimal decision of a single procedure may not match the global optimization.

Specifically, the initial model is a 10-unit LSTM followed by a 1-node dense layer. The number of units is the dimensionality of the output space of the LSTM layer, while the number of nodes is the output dimensionality of the dense layer. A dense layer can be regarded as one single layer of a feedforward neural network. An important hyperparameter of our model is the sequence length t of LSTM, which was introduced in Section 3.2.1. The initial setting of t is 1800, according to prior physical knowledge and experiments. Details for the default settings will also be further explained in Section 3.5.3.

3.2.3 The training and validation scenario

The model is designed to predict the overall temperatures of motors. The corresponding performance is measured by the prediction error, namely the MSE. However, it is worth noticing that, in practice, there are multiple meaningful scenarios to apply the model, which correspond to different validation methods, as stated in Chapter 4. Therefore, we here need to specify the scenario employed in this chapter.

Since this thesis is most interested in training a general model which can predict the temperature of any motor. We therefore optimize our model based on a general scenario, where:

- For all 16 ships, all their motors have been working for a long time such that there are enough historical data stored.
- We aim at developing a generic model which is trained by all available data and can predict the temperatures of any of these motors.

As demonstrated in Chapter 4, the optimal model decided in this scenario shows great promise of generalization to handle other practical scenarios. We are hence confident about optimizing our model based on this assumed scenario.

3.2.4 The training and validation procedure

As for the specific training and validation procedure, 5-fold cross-validation (5-fold CV) is a reasonable choice. For the 5-fold CV, all available data will be randomly divided into 5 folds of equal size. However, since LSTM also takes into account historical observations rather than one current observation, we have to pay more attention during the fold-splitting procedure. Let us first introduce the definition of neighbor time sequences as a preparation.

Definition 3.2.1 (Neighbor time sequences)**.** Two time sequences of equal length t, i.e., $X_i = (\boldsymbol{x_i}, \boldsymbol{x_{i+1}}, \cdots, \boldsymbol{x_{i+t-1}})^T$ and $X_j = (\boldsymbol{x_j}, \boldsymbol{x_{j+1}}, \cdots, \boldsymbol{x_{j+t-1}})^T$, are **neighbor time sequences** if and only if $|i - j| < t$. The neighbor time sequences will comprise certain **common observations**. Specifically, they are $\boldsymbol{x_i}, \boldsymbol{x_{i+1}}, \cdots, \boldsymbol{x_{j+t-1}}$ if $i < j$, or $\boldsymbol{x_j}, \boldsymbol{x_{i+1}}, \cdots, \boldsymbol{x_{i+t-1}}$ if $j < i$.

During the 5-fold CV, each fold is once used as the test set, while the remaining folds are used as the training set. Therefore, if neighbor time sequences are located in different folds, one of them will once be in the training set whereas the other will be in the test set. Furthermore, since neighbor time sequences comprise certain common observation(s), they can be highly alike for our dataset. Therefore, if distributing neighbor time sequences into different folds, we may have similar observations in both the training set and test set. The results of validation can then become overoptimistic.

For the 5-fold CV, the loss function is always the mean squared error (MSE), i.e.,

$$Loss = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2, \tag{3.1}$$

where n is the number of observations, y_i is the actual output, and $\hat{y}_i$ is the predicted output. The optimization algorithm for this loss function is a mini-batch gradient descent algorithm with rmsprop. Rmsprop is an unpublished adaptive learning rate method proposed by Tieleman and G. Hinton 2012, which is widely in use as it provides a good performance.

While using the gradient descent algorithm, the gradients are calculated to update the parameters of our model. Theoretically, the gradients are the derivatives of the total loss of all given observations with respect to each parameter. In practice, however, it is more beneficial to use the summed loss of a partition of all observations. The size of this partition is a hyperparameter, defined as the mini-batch size, which requires tuning. This algorithm is referred to as mini-batch gradient descent (Bertsekas 1996).

However, when using the mini-batch gradient descent, we cannot exploit the information of all given observations through one update. In order to make the best use of all available information, we need to train the data through iterations, until all observations have been used. A straightforward approach is that the whole training set is divided into several partitions of equal size. We will use these partitions as mini-batches and sequentially update our model with them. Eventually, all these partitions will be used for training once. This procedure is referred to as one epoch. In practice, one epoch is typically not

enough for the training procedure to converge, and usually it is mandatory to run multiple epochs. The number of epochs is also an essential hyperparameter, which requires fine-tuning.

3.3 Simulated dataset for pretests

There are many ways to develop our model. Given the enormous amount of data, the corresponding training time for each attempt at a model can be very large, making the validation costly.

Also, it is difficult to forecast if the attempts will be meaningful and successful. Therefore, it is useful to run the pretests and feasibility analysis before conducting a full model validation. This section introduces an in-house simulator with system dynamics similar to the problem at hand. It is hence possible to produce simulated data that resemble some important characteristics and properties of the real data provided by ABB.

The simulated data follows a simplified physical model of the motors studied in this thesis, i.e., the thermal model of a wire. Therefore, it is simpler for our model to achieve the best possible performance on the simulated dataset. It indicates that, compared to the ABB dataset, fewer training data are required when using the simulated dataset. It then becomes practically achievable to pretest our models with much less training time. Furthermore, since the ground truth of the simulated dataset is known, it is more convenient to analyse and improve the model performance according to the test results on the simulated dataset. As one more advantage of using the simulated data, it is also possible to generate as much data as we want, which may help test the generalization of models.

3.3.1 Physical model

The simulator is based on a simplified physical model of the motor, i.e., the thermal model of a wire, such that there is only one feature in contrast to the five features reported in the original ABB dataset. For convenience, we refer to this single feature as the input and the target (temperature) as the output.

According to the physical model of electric marine motors provided by ABB, the main heat in the motor is generated by the current in circuits. We hence take the current as the input of this simulator. Assuming that the current is constant from timepoint 0 to timepoint T, according to the Joule's (first) law (Young, Freedman and Ford 2013), the heat generated by the input (current) during this period is

$$Q_i = \int_0^T I^2 R \, dt, \tag{3.2}$$

and its derivative form is

$$\frac{dQ_i}{dt} = I^2 R, \quad t \in (0, T), \tag{3.3}$$

where Q is the generated heat, I, R and t denote the current, resistance, and time, respectively. I may not be a constant, but instead, can be a function of t or other hidden variables.

Not all the generated heat is absorbed by the wire to increase its temperature. Some of the heat will be released to the surroundings following the heat transfer mechanism. Ignoring the radiation and natural convection effects, the heat transferred to the surroundings can be calculated by Newton's law of cooling (Young, Freedman and Ford 2013)

$$Q_o = -k \int_0^T (T_c - T_o) S \, dt. \tag{3.4}$$

The derivative form is then given by

$$\frac{dQ_o}{dt} = -k(T_c - T_o)S, \quad t \in (0, T). \tag{3.5}$$

Here, k denotes the thermal conductivity, which is the measure of the wire's ability to conduct heat to the surrounding material, and Q_o is the heat transferred from the wire to the surrounding material.

We are now able to derive the heat absorbed by the wire as

$$\begin{aligned} Q &= Q_i - Q_o \\ &= \int_0^T I^2 R \, dt + k \int_0^T (T_c - T_o) S \, dt, \end{aligned} \tag{3.6}$$

with its derivative form given by

$$\frac{dQ}{dt} = I^2 R + k(T_c - T_o)S, \quad t \in (0, T). \tag{3.7}$$

The derivative of its temperature T_o with respect to time t can then be given by

$$\frac{dT_o}{dt} = \frac{I^2 R + k(T_c - T_o)S}{mc}, \quad t \in (0, T), \tag{3.8}$$

where m is the mass of the wire, and c is the specific heat capacity of the wire.

The relationship between the input, i.e., the current I, and the output, i.e., the temperature T_o, is demonstrated by Equation (3.8), which enables the implementation of the simulator.

The initial condition assumes that there is no current flowing through the wire at $t = 0$ and the temperature of the wire T_o is equal to an initial state t_i. Given the initial condition, the differential Equation (3.8) can be solved numerically to obtain the changes of wire temperature with time for $0 < t < T$, namely

$$T_o = T_c + \frac{I^2 R}{kS} + (T_i - T_c - \frac{I^2 R}{kS}) e^{-\frac{KSt}{mc}}, \quad t \in (0, T). \tag{3.9}$$

3.3.2 Implementation

The current model assumes that I is constant from time 0 to T, whose transient state is governed by Equations (3.8) and (3.9). For a navigating ship, the current running through the wire varies with time. Recall that the data were recorded per second. Therefore, it is reasonable to assume that I is constant during each second. Defining the time interval between recordings as the sampling time h, Equation (3.8) can be approximated by Newton's Method (Solomon 2015), that is

$$\frac{T_{o,k} - T_{o,k-1}}{h} = \frac{I^2R + k(T_c - T_{o,k})S}{mc}, \quad for \quad k = 0, 1, 2, ... \tag{3.10}$$

For convenience, we rewrite $\frac{mc}{kS}$ as τ, $\frac{R}{kS}$ as α, and $T_{o,k}$ as y_k. Equation (3.10) can then be rewritten as

$$\frac{y_k - y_{k-1}}{h} = \frac{I^2\alpha + T_c - y_k}{\tau}. \tag{3.11}$$

It can be arranged to

$$y_k = \frac{\tau}{\tau + h}y_{k-1} + \frac{h}{\tau + h}(I^2\alpha + T_c). \tag{3.12}$$

The notation can be further simplified by assuming $\alpha = 1$ and $T_c = 0$, which leads to

$$y_k = \frac{\tau}{\tau + h}y_{k-1} + \frac{h}{\tau + h}I^2. \tag{3.13}$$

For convenience and simplicity, the input can be assumed as a range of random step functions that follows the Poisson process, and we consider only two alternative values for the input I, i.e., $I = 1$ or $I = 0$. The corresponding outputs can then be calculated by Equation (3.13).

Notice that τ and h are still unknown and need to be determined. In order to resemble the original ABB dataset, which were recorded at 1 Hz, the sampling time $h = 1$ second is used. To find the value for τ, it is useful to study it from another aspect, which is described below as a preparation.

The current output (temperature) at time step k is influenced by input at time step k and the output at time step $k - 1$, and the output at $k - 1$ is further influenced by the input at time step $k - 1$ and the output at time step $k - 2$, and so on. One can then infer that all historical inputs will eventually influence the current output.

Consider an extreme situation in which the input and output at $t \leq 0$ are 0 and the input becomes 1 when $t > 0$. According to Equation (3.13), the changes of output with time are shown in Figure 3.1. The output evolution indicates that 95.02% of the temperature increase has been achieved after a period of 3τ. Here, 95.02% is an approximated value of $1 - e^{-3}$, calculated by Equation (3.9) with $T_i = T_c = 0$, $I^2\alpha = \frac{I^2R}{kS} = 1$, and $t = 3\tau = 3 \cdot \frac{mc}{kS}$. Based on this observation, it can be assumed that the influence of historical input on the output is negligible after 3τ.

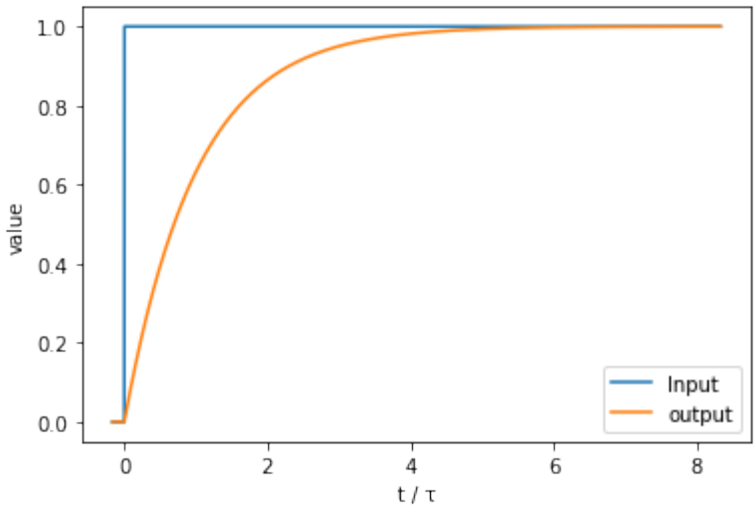


Figure 3.1: Outputs of the simulator. The inputs and outputs are both 0 when $t \leq 0$, and the inputs are 1 when $t > 0$. The x-axis represents the simulation time, and the y-axis represents the value of inputs and outputs. Note that the x-axis is denoted by t times τ, where τ is a predefined parameter and is equal to 600 for this plot.

According to the information provided by ABB, the historical observations during the past 1800 seconds are considered as correlated with the current prediction. The details will be further discussed in Section 3.5.3. Until then, we can start with an ad hoc assumption $3\tau = 1800$, i.e., $\tau = 600$, for the simulator. After obtaining the parameters of sampling time $h = 1$ and $\tau = 600$, we can now implement the simulator and obtain the simulated data which are used along with the ABB data in the following developments and considerations.

3.4 Data preprocessing

Multiple universal preprocessing methods have been introduced in Section 2.1, while in this section, we aim at proposing a concrete approach, which can be practically implemented on the ABB dataset and can deliver a satisfying performance.

3.4.1 Preprocessing

The dataset provided by ABB was divided into separate files, where each file contains the corresponding recordings of one motor from one ship throughout one calendar month.

The ships that are available in the data set were not always in the state of navigation. There hence exist blank recordings in the recorded data. The data provider, ABB, has already transformed these blank recordings into N/A values. There are 3 types of files with respect to the blank recordings. Some

files exclusively contain continuous recordings as shown in Figure 3.2, some comprise both valid recordings and invalid N/A values as shown in Figure 3.3, and the remaining ones only consist of N/A values.

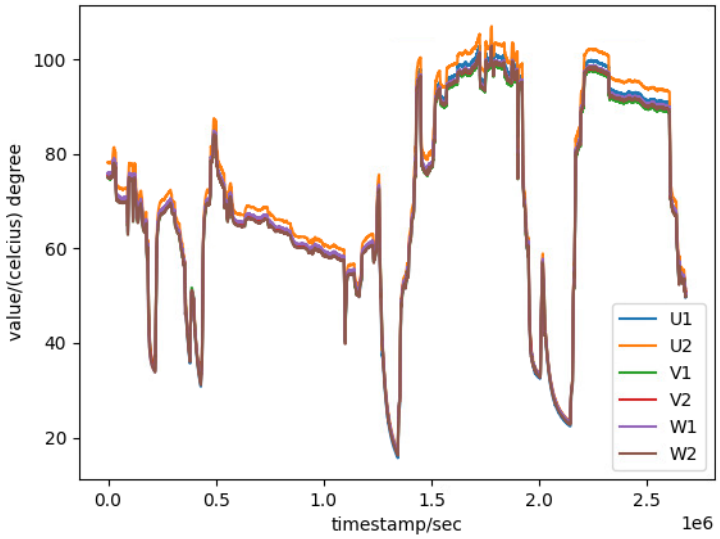


Figure 3.2: A continuous temperature recording of one motor during one month. The y-axis denotes the temperatures (degree Celsius), and the x-axis denotes the time (second). Here, $U1$, $U2$, $V1$, $V2$, $W1$, and $W2$ are the temperatures detected by six different sensors. These sensors are located in six different places of the same motor. There is no evident difference between them.

The files with continuous recordings are already well-structured, while the files without any valid data cannot be utilized in this thesis and are hence excluded from any further analysis. Efforts are only made to study the files that contain both valid and invalid N/A values. The invalid recordings cannot be simply deleted from the dataset. This is because, as described in Section 2.4, the LSTM emphasizes the connection between the data at adjacent time points. After removing these invalid N/A recordings, one needs to ensure that the recordings before each N/A value will not be connected to the recordings after these N/A values with respect to time.

According to Section 1.2 and Section 1.3, 5 features were collected from the motors, i.e., **CI** (the inlet cooling air temperature/$^\circ C$), **CO** (the outlet cooling air temperature/$^\circ C$), **PO** (the power of the motor/% of maximum nominal power), **SP** (speed of the motor/% of maximum nominal speed), and **TO** (mechanical torque /% of maximum nominal torque).

The given dataset was also found to comprise negative recordings, which occupy 10.52% of the data. The negative recordings indicate that the corresponding

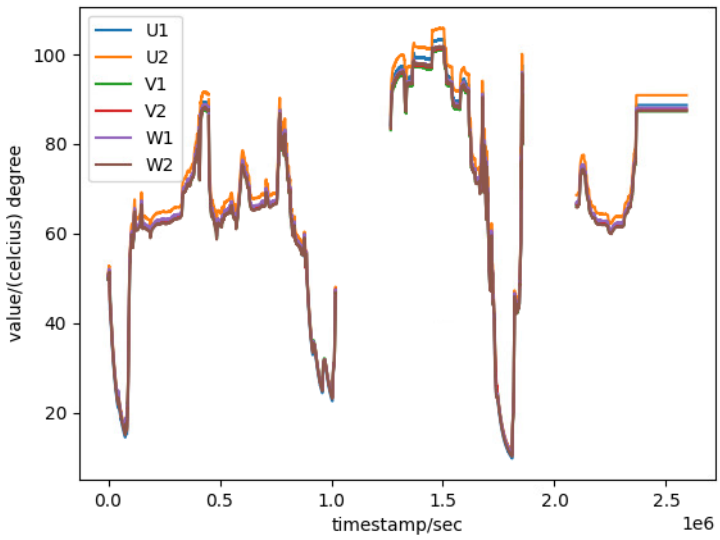


Figure 3.3: A temperature recording with blanks of one motor. The recording was recorded during one month. The blanks exist because the motor was not always working, i.e., the ship was not always navigating. The notations here have the same meanings as in Figure 3.2.

motor reversed its rotation. Because the current model aims to predict the temperatures of motors at a normal navigation state, these negative recordings might worsen the model behaviour.

Intuitively, it may be assumed that the value of negative recordings in our analysis can be negligible so that they can be excluded from the training data. However, as the mean squared errors shown in Table 3.1, simple removals of negative values tend to worsen the performance of the model.

As the LSTM can benefit from these negative recordings, we consequently suggest not to ignore these negative recordings, but include them in all analyses conducted in this thesis.

	MSE (LSTM)	MSE (Baseline)
With negative recordings	60.94	76.33
Without negative recordings	87.22	92.54

Table 3.1: MSEs with and without removing negative recordings. Here, the baseline model is the linear model.

Furthermore, the given dataset also contains 6 different targets, i.e., temperatures. They were collected through sensors located in two separate triplets of a motor, and are referred to as $U1$, $U2$, $V1$, $V2$, $W1$, and $W2$. However, as

shown in Figures 3.2 and 3.3, these temperatures have no clear difference with respect to their patterns and values.

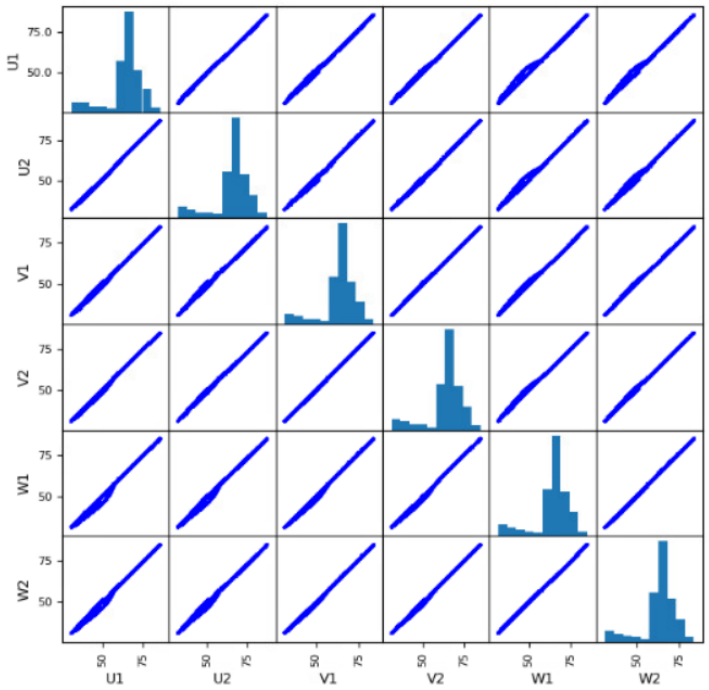


Figure 3.4: Correlation of the given targets (temperatures)

Figure 3.4 shows the scatter plots of each pair of the six temperatures, which indicates that there exists a strong linear correlation between these temperatures. It is therefore of little significance to attempt to predict all 6 targets. We instead transform the targets into their average values T, i.e.,

$$T = \frac{U_1 + U_2 + V_1 + V_2 + W_1 + W_2}{6}. \tag{3.14}$$

3.4.2 Exploratory analysis

Exploratory analysis usually contributes to statistical models, as one can select appropriate methods and features based on the analysis. However, exploratory analysis may be less relevant for neural networks, because it is typically difficult to interpret a neural network and select among different methods based on their functionalities. Nevertheless, the analysis can always help understand the characteristics of the data and is hence still valuable.

Since the given ABB dataset is not specifically made for this thesis, it is possible that some features can be redundant. We hence illustrate the correlations between each two features using a scatter plot, as shown in Figure 3.5.

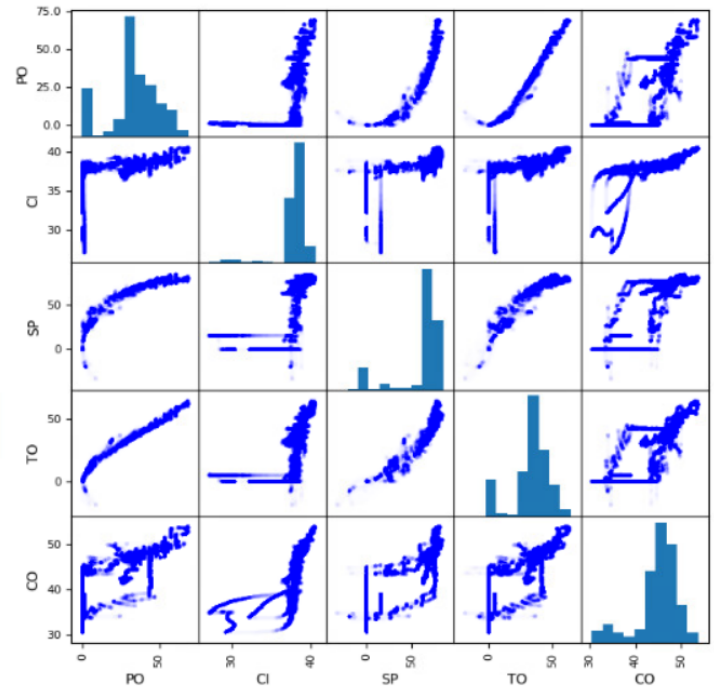


Figure 3.5: Correlation between the given features.

Considering its physical meaning, we temporarily neglect the outlet cooling air temperature (i.e., **CO**) in this section. The corresponding details will be discussed in Section 3.4.3. Regarding the remaining 4 features, the correlation indicates that **CI** does not provide redundant information, as its correlations with other features are weak. However, there seems to be a strong linear correlation between **TO** and **PO**, a non-linear correlation between **TO** and **SP**, and a non-linear correlation between **PO** and **SP**. From a statistical perspective, it is reasonable to consider removing **TO** or **PO**, since they are likely to provide equivalent information.

Removing one of these two features could benefit the performance of a linear model. However, according to the corresponding results shown in Table 3.2, the LSTM model seems to be robust enough even if including the redundant features. Also, it can be seen that the redundant features can still improve the performance of an LSTM.

3.4.3 Feature selection

There are 5 features in the ABB dataset, i.e., **CI**, **CO**, **SP**, **PO**, and **TO**. However, it is worth noticing that **CO**, i.e., the temperature of output cooling air, differs substantially from the others. According to the causality, **CO** is

Features	LSTM	Baseline
PO SP CI	61.27	76.23
TO SP CI	63.54	82.56
PO TO SP CI	56.77	87.12

Table 3.2: MSE for different feature combinations. The baseline model here is a linear model.

mostly influenced by the targets (the temperatures of motors) through the air circulation system, while the targets are not influenced by **CO**. Furthermore, as illustrated in Figure 3.6, there is an evident linear correlation between **CO** and other targets.

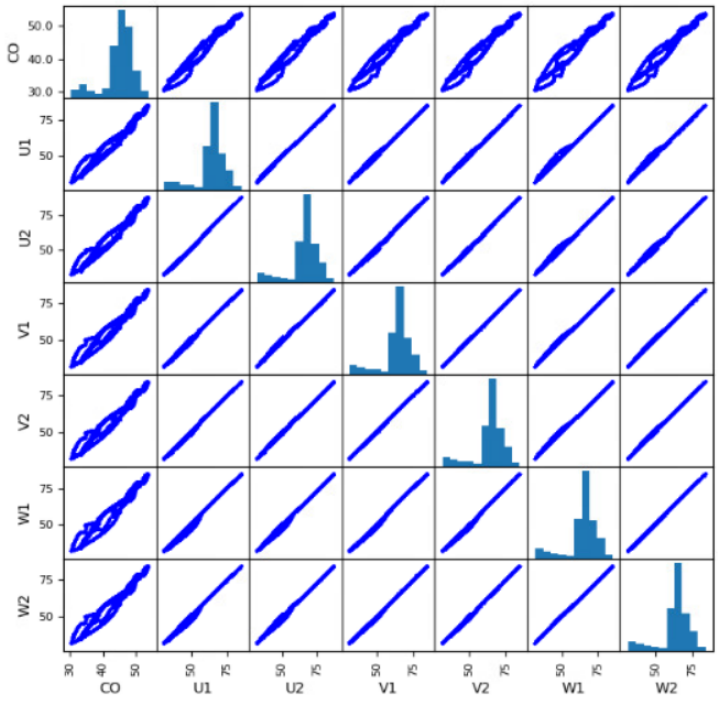


Figure 3.6: Correlation between the given targets (temperatures) and *CO*.

Note that the prediction model in this thesis should be used to detect overheating events before the temperature sensors detect them. Therefore, involving **CO** into our model will not contribute to better overheating detection, since the predicted overheating revealed by **CO** will be detected in advance by the sensors. In other words, the overheating alarms triggered by **CO** would have already been reported by the temperature sensors. It is therefore meaningless to take into account **CO** in the present model. The number of features can then be

reduced to four, i.e., **CI** (cooling air input), **PO** (power), **SP** (speed) and **TO** (torque).

As for these four features, it is difficult to eliminate any of them according to the prior knowledge given by ABB. Furthermore, it is also shown in Table 3.2 that the LSTM is robust against redundant features and can even benefit from them. Therefore, the feature selection based on the exploratory analysis does not improve the performance of our LSTM model. This thesis hence validates all possible feature combinations, since there exist merely $4 + 6 + 4 + 1 = 15$ combinations for the four selected features.

Features	MSE
PO	84.59
CI	127.20
SP	92.73
TO	89.21
PO CI	65.29
SP CI	66.36
TO CI	67.65
PO SP	72.55
PO TO	69.32
TO SP	65.08
PO TO CI	60.72
PO SP CI	58.89
TO SP CI	61.34
PO TO SP	68.57
PO TO SP CI	**57.30**

Table 3.3: MSEs of all possible feature combinations using the LSTM model. The MSE of the baseline model is 78.44.

According to Table 3.3, the performance of the combination **CI**, **PO**, **TO**, and **SP** are found to be superior to others, which is henceforth regarded as the optimal choice in this thesis.

3.4.4 Standardization

We have introduced 3 types of preprocessing methods in Section 2.1. Among these methods, normalization is typically applied to tasks which are based on distance metrics, e.g., clustering, and it would therefore not be beneficial in our situation. While examining the remaining preprocessing methods, the pretested performance on the simulated dataset is shown in Table 3.4.

The mean squared errors in Table 3.4 indicate that the min-max scaling provided the best performance, and the performance of all preprocessing methods is able to surpass that of the baseline model. However, it can be seen that the differences between the MSEs of these methods are not significant. We hence validate all preprocessing methods in Table 3.4 on the original ABB dataset, and their corresponding mean squared errors are shown in Table 3.5. The MSE

Preprocessing method	MSE(LSTM)
Robust scaling	0.0133
Min-max scaling	**0.0123**
Max absolute value scaling	0.0132
Standardization	0.0127

Table 3.4: MSE of different preprocessing methods on the simulated dataset. The MSE of the baseline model, namely the linear model, is 0.0215.

comparison shows that standardization is the optimal choice for the current model on the ABB dataset.

Preprocessing method	MSE(LSTM)
Robust scaling	96.77
Min-max scaling	68.35
Max absolute value scaling	75.66
Standardization	**59.20**

Table 3.5: MSE of different preprocessing methods on the ABB dataset. The employed features are **PO**, **TO**, **SP**, and **CI**. Here, the MSE of the baseline model, i.e., the linear model, is 77.64.

3.4.5 Shuffling

As described in Section 1.3, the order of the ABB data is arranged according to their temporal sequence. It is perceivable to train the model based on the original data sequence. However, always training the model with this given order may lead to a poor performance.

Recall that our model is a multi-layer neural network, and its loss function is the mean squared error (MSE). Therefore, this loss function is neither convex nor concave with respect to the parameters (H. Zhang, Shao and Salakhutdinov 2019). It is hence impossible to guarantee that our loss function will converge to a global minimum through training, but instead, it will most likely converge to a local minimum.

The performance can differ largely among local minimums, and some local minimums may provide much poorer performance than the others (Choromanska et al. 2015). It is crucial for the model to have a reduced likelihood of being trapped into these local minimums. Recall from Section 3.2.4 that there are typically multiple epochs needed. For the gradient descent algorithm employed in this thesis, a static order of training data will lead to a static loss surface for different epochs. The search for optimal parameters can then be trapped into a local minimum with unacceptable performance, although there may exist a better one in close proximity.

Shuffling is a simple but effective approach to solve this issue. The concept is to shuffle the original order into random ones for different epochs so that the loss surfaces of different epochs can vary. Theoretically, the search for optimal

parameters can be made more successfully, and one can avoid being stuck in a local minimum with poor performance. As a result, shuffling can often provide a better performance than keeping the original order (Meng et al. 2019).

To evaluate the effectiveness of shuffling, attempts were made to pretest the shuffling on the simulated dataset. The corresponding results are tabulated in Table 3.6, which shows that shuffling indeed enhances to the model performance.

	MSE(LSTM)
Shuffling	0.0126
Without shuffling	0.1277

Table 3.6: MSEs for simulated dataset with and without shuffling while training.

Furthermore, as described in Section 3.3.2, the inputs of the simulated dataset are randomly generated in accordance with the Poisson process. Therefore, on the original ABB dataset, the improvement brought by shuffling is expected to be even larger. As shown in Table 3.7, shuffling is indeed found to be an even more effective approach when applying to the ABB dataset.

	MSE(LSTM)
Shuffling	66.73
Without shuffling	217.42

Table 3.7: MSEs for ABB dataset with and without shuffling while training. The employed features are **PO**, **TO**, **SP**, and **CI**. The preprocessing method is standardization. The MSE of the baseline model is 76.71.

3.5 Approach for handling large amount of data

We are given over 1 billion observations in the validation dataset, but rather limited time and computing resources. Therefore, one main challenge in this thesis is to develop an efficient approach which is capable of handling the tremendous amount of data.

In general, the successful development of a neural network model requires a considerable number of experiments, which are crucial for the model to search for the optimal configurations and hyperparameters. In this thesis, each experiment typically consists of one training procedure and one corresponding validation. The execution of one experiment usually requires a long computing time, which makes it impossible to conduct all the desired experiments in the permitted time of this thesis. It is hence essential to ensure that the time cost for each experiment is affordable.

Furthermore, in view of the available computing resources, simultaneously loading 1 billion observations into the memory (RAM) is infeasible. Therefore, we have to divide 1 experiment into several parts, while trying to gain a similar conclusion or performance.

3.5.1 Data loading and training

Following the initial settings obtained in Section 3.2, a general algorithm for data loading and training is described as in **Algorithm 1**.

Algorithm 1: A general algorithm

1. Load all the data into memory ;
2. Preprocess the data without shuffling ;
3. Set $i = 0$, and denote the number of epochs as n_{epoch};

while $i < n_{epoch}$ **do**
- 4.1. Shuffle the data.;
- 4.2. Train our model for 1 epoch ;
- 4.3. Set $i = i + 1$;

end

However, considering the total size of over 1 billion observations and the limited amount of memory available (203 GB RAM), it is hence only viable to load a partition of the training set each time. Therefore, the corresponding training procedures need to be incremental. Below summarizes an incremental algorithm, referred to as **Algorithm 2**.

Algorithm 2: An incremental algorithm

1. Divide all training data into n partitions of equal size;
2. Set $i = 0$;

while $i < n$ **do**
- 3.1. Load the i-th partition into memory ;
- 3.2. Preprocess the data without shuffling ;
- 3.3. Set $j = 0$, and denote the number of epochs as n_{epoch};
- **while** $j < n_{epoch}$ **do**
  - 3.4.1. Shuffle the data.;
  - 3.4.2. Partially train our model on the data for 1 epoch;
  - 3.4.3. Set $j = j + 1$;
- **end**
- 3.5. Release only the memory loading this partition ;
- 3.6. Set $i = i + 1$;

end

One may notice that the shuffling procedure for **Algorithm 1** and **Algorithm 2** are implemented differently. In **Algorithm 1**, the data are shuffled within the whole training set, while in **Algorithm 2**, only the data within one partition of the training set are shuffled each time. For clarity, we define the shuffling in **Algorithm 1** as a global shuffling and the shuffling in **Algorithm 2** as a local shuffling.

As stated previously, the loss function in this thesis must be non-convex. It is then proven that the convergence rate for **Algorithm 2** can be significantly smaller than that for **Algorithm 1**, and the difference between convergence rate is determined by the number of partitions (Meng et al. 2019), i.e., n in **Algorithm 2**.

It is impractical to use **Algorithm 2** due to the lower convergence rate and increased time cost. We hence develop a more efficient algorithm, referred to as **Algorithm 3**, where simulated global shufflings are used instead of local shufflings in each partition.

Algorithm 3: An incremental algorithm with insufficient global shuffling

1. Divide all the training data into n partitions of equal size;
2. Create n empty files;
3. Set i = 0;

while $i < n$ **do**
- 4.1. Load the i-th partition into memory ;
- 4.2. Preprocess this partition without shuffling ;
- 4.3. Store each observation into a random file that we created before;
- 4.4. Set i = i + 1 ;

end

5. Set i = 0;

while $i < n$ **do**
- 6.1. Randomly load an unused file into memory ;
- 6.2. Set $j = 0$, and denote the number of epochs as n_{epoch};
- **while** $j < n_{epoch}$ **do**
  - 6.3.1. Shuffling the data.;
  - 6.3.2. Partially train our model on the data for 1 epoch;
  - 6.3.3. Set $j = j + 1$;
- **end**
- 6.4. Set i = i + 1 ;

end

The shuffling which produces a permutation following a uniform distribution is defined as an ideal (sufficient) shuffling, while the other shufflings are defined as insufficient shufflings (Meng et al. 2019). Considering it is impractical to generate an ideal global shuffling, **Algorithm 3** therefore employs an insufficient global shuffling. Although it will typically provide a worse performance than the ideal shuffling, the early studies found that the difference between insufficient shuffling and ideal shuffling can be insignificant with respect to the converging rates and performance (Meng et al. 2019).

Theoretically, each time a model is trained, one needs to carry out the whole procedure of **Algorithm 3**. The future training also have to start over from the beginning of **Algorithm 3**. Here, it is worth noticing that a considerable amount of time is spent in preprocessing and storing the loaded data, i.e., from step **1** to step **4** in **Algorithm 3**. Recall that the importance of shuffling is to ensure significantly different loss surfaces for different training epochs. Therefore, it is wise to reuse those stored preprocessed data several times for future training, as it will not lead to similar loss surfaces.

Using the simulated dataset, the pretests are carried out for these three different algorithms. Their corresponding mean squared errors and the mean epochs needed for convergence are tabulated in Table 3.8. Here, the number of epochs needed for convergence is used to approximate the cost of training time.

Algorithm	average number of epochs before convergence	MSE
1	**6**	**0.0126**
2	11	0.0183
3	8	0.0152

Table 3.8: MSEs of different training algorithms on the simulated dataset using LSTM. The MSE of the baseline model is 0.0215.

As expected, **Algorithm 1** provides the best performance regarding both the number of epochs and the MSE. The performance of all algorithms surpass that of the baseline model. The improved performance infers to extend the algorithms to the original ABB dataset. However, as stated before, it is impossible to train the model with **Algorithm 1** on the ABB dataset, because our available memory (RAM) cannot support such an algorithm. The corresponding results are marked as N/A, and all other MSEs are tabulated in Table 3.9.

Algorithm	average number of epochs before convergence	MSE
1	N/A	N/A
2	26	89.63
3	**6**	**61.55**

Table 3.9: MSEs of different training algorithms on the original ABB dataset using LSTM. The MSE of the baseline model is 78.44.

These pretests and validation have suggested the success of **Algorithm 3**, and it is hereafter applied in the model developed in this thesis.

3.5.2 Test procedure

Following the same philosophy as in Section 3.5.1, our test procedure also needs to be incrementally conducted. The exception is that no dynamic loss surfaces need to be accounted for, as they do not exist in the test procedure. Inspired by Section 3.5.1, a simple but efficient algorithm is developed as **Algorithm 4**.

Algorithm 4: An incremental test algorithm

1. Divide all the test data into n partitions of equal size;
2. Set $i = 0$;

while $i < n$ **do**

- 3.1. Load the i-th partition into memory ;
- 3.2. Test our model with this partition;
- 3.3. Record the corresponding mean squared error ;
- 3.4. Release only the memory loading this partition ;
- 3.5. Set $i = i + 1$;

end

3.5.3 Mini-Batch size, epochs and sequence length t

As described before, developing a model requires a considerable number of experiments, and each experiment consists of one training procedure and one corresponding validation procedure. The required time and memory for the training are dramatically larger than those for the validation. Therefore, one can ignore the expenses incurred by the validation when attempting to reduce the overall computing time and memory.

Regarding the training procedure, there exist 3 hyperparameters which have strong effects on the required time and memory. They are the size of mini-batch, the number of epochs, and the sequence length t.

Mini-batch size is a hyperparameter which is introduced by the mini-batch gradient descent. As described in Section 3.2.4, the mini-batch gradient descent is a gradient descent algorithm, which updates the parameters of the model according to the gradients calculated by the summed loss of a partition of all observations (Bertsekas 1996). It compromises the batch gradient descent, which employs the total loss of all observations instead of the summed loss, and the stochastic gradient descent (SGD), which employs the loss of a single observation.

A larger mini-batch size leads to more memory demand and less computing time for each epoch. We can hence achieve a well-balanced method to accommodate the time and memory constraints by tuning the mini-batch size.

There still exist two other hyperparameters, i.e., the number of epochs and time sequence length t, which also have an impact on the required time and memory. Regarding the sequence length t, the input of an LSTM model is a 3-rank tensor X as described in Section 3.2.1,

$$X = (X_1, X_2, ..., X_n),$$

where

$$X_i = \begin{pmatrix} \boldsymbol{x_i} \\ \boldsymbol{x_{i+1}} \\ \vdots \\ \boldsymbol{x_{i+t-1}} \end{pmatrix} = \begin{pmatrix} 1 & x_{i,1} & \cdots & x_{i,p} \\ 1 & x_{i+1,1} & \cdots & x_{i+1,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{i+t-1,1} & \cdots & x_{i+t-1,p} \end{pmatrix} \quad for \quad i = 1, 2, ..., n.$$

Here, $x_{j,k}$ corresponds to the k-th feature of the j-th observation. We refer to X_i within the tensor X as a time sequence of p features and t as the sequence length. Then, n is the number of given time sequences in the dataset. For an LSTM model, the sequence length t determines the number of historical observations which are considered as correlated with the current prediction. Notice that a larger sequence length will significantly increase the size of the input tensor X, and will hence also increase the corresponding required memory (RAM) and the cost of computing time.

After introducing the functionalities of these 3 hyperparameters, i.e., mini-batch size, the number of epochs, and sequence length t, we will determine their optimal values.

According to the information given by ABB, the historical observations within the past half an hour can be considered as informative for the current prediction, and it is reasonable to assume that the influence of earlier observations is negligible. Considering our observations were recorded every second, the starting assumption for the optimal sequence length t will be 1800, i.e., the number of recordings (observations) in half an hour.

However, it cannot be ensured that the optimal sequence length t indeed corresponds to this 1800-second period. We hence carry out a pretest on the simulated dataset. The corresponding performance is shown in Figure 3.7.

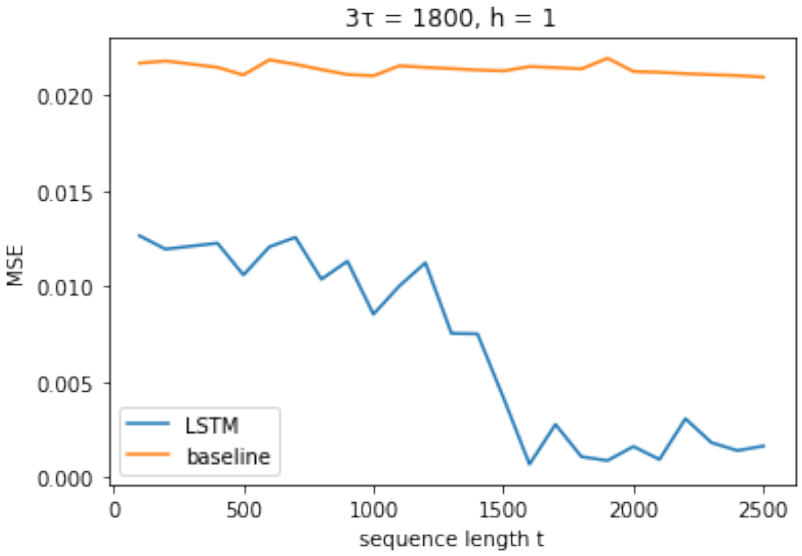


Figure 3.7: MSE for different sequence lengths t on the simulated dataset. Setting $3\tau = 1800$ and $h = 1$ ensures that the simulated data has the same characteristics as the original ABB data. Details were discussed in Section 3.3.

Figure 3.7 shows that the increase of sequence length t improves the model performance, and the MSE reaches its minimum when $t \in [1600, 2100]$. Our previous assumption of sequence length, $t = 1800$, can therefore be a reasonable choice, even if it is also implied that there may still be room for further fine-tuning.

Assuming $t = 1800$ produces a satisfying performance, the hyperparameters of mini-batch size and the number of epochs are first discussed and tuned. Afterwards, we revisit and fine-tune the sequence length t. In short, the following two-step tuning procedure is conducted:

1. First, tune the mini-batch size and the number of epochs for a constant sequence length, i.e., $t = 1800$.
2. After retrieving the optimal mini-batch size and epoch number, further tune the sequence length t with these optimized hyperparameters.

Note that setting $t = 1800$ requires a large amount of memory and time for training. It is hence impossible to train the model with all given observations in the validation set, i.e., over 1 billion. For practicality, we have exploited approximately $\frac{1}{10}$ of all observations in the remaining parts of this section. Note that it is essential to maintain the order of observations in a time sequence with respect to time. Therefore, we randomly select the time sequences instead of selecting each single observation. Less training data may lead to a worse model performance. However, the number of observations is still over 100 million, and it is reasonable to assume that they can represent the whole dataset. The decisions made by these observations can therefore still be considered as accurate and conclusive.

The optimal number of epochs is obtained by a dynamic algorithm, i.e., the early stopping algorithm. The given training set is split into two sets, namely a new training set and a validation set. Initially, an excessively large value is set as the number of epochs. As the training and validation proceed, if a pre-defined criterion is reached, the training process will stop before completing all epochs. The criterion selected in this study is the improvement of the validation error. A detailed description is given below as in **Algorithm 5**.

Algorithm 5: An early stopping algorithm for training

1. Denote the maximal possible number of epochs as n and set a threshold σ ;
3. Set $i = 0$;
while $i < n$ **do**
    4.1. Randomly split the given training set into a new training set and a validation set. ;
    4.2. Train our model on the new training set for 1 epoch ;
    4.3. Test our model on the validation set and record the corresponding mean squared error as e_i;
    if $i > 0$ *and* $e_{i-1} - e_i < \sigma$ **then**
        5.1. Stop training.;
    else
        5.2. Set $i = i + 1$;
    end
end

For the task and the dataset considered in this thesis, it was found that all training procedures can converge before 200 epochs. Therefore, an appropriate value of n in **Algorithm 5** is set as 200, which accompanies with the threshold $\sigma = 0.0005$. Note that 0.0005 is the change of MSE in one iteration, which is calculated by the standardized values. An 0.0005 improvement of MSE approximately represents an improvement of 0.1 degree Celsius for the temperature predicted by the model. Recall that the objective of this thesis is to forecast the possible occurrence of overheating through predicting the temperature of motors. Therefore, it is not worthwhile investing more time or computing resources in pursuing higher accuracy than measurement uncertainty.

By means of the dynamic algorithm (**Algorithm 5**) and a constant sequence

length $t = 1800$, it becomes viable to tune the remaining hyperparameter, i.e., mini-batch size. For each mini-batch size, the model is trained on a given number of observations using the given computing resources. The corresponding training time for different mini-batch sizes is shown as in Figure 3.8. It is shown that the training time of our model decreases as the mini-batch size increases. For the given time of this thesis, it is reasonable to consider 50 minutes as the maximal acceptable time, which means that the mini-batch size should not be smaller than 128.

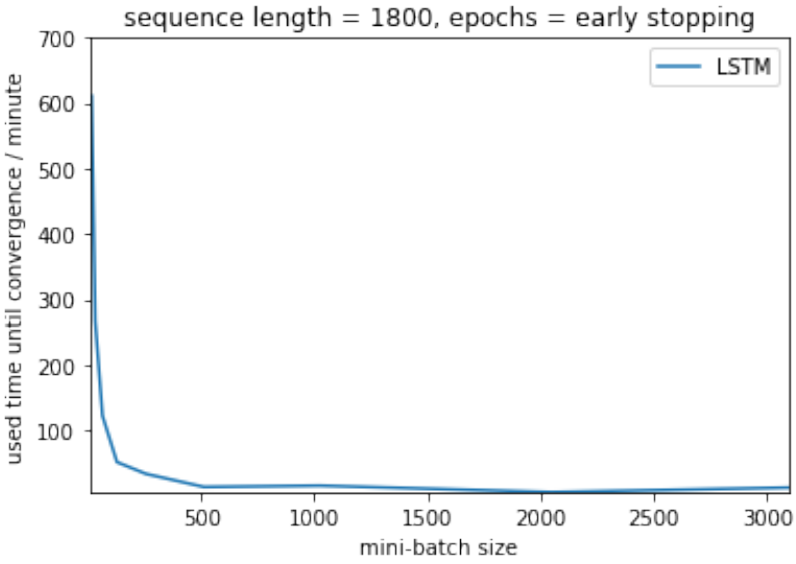


Figure 3.8: Used time before converging for different mini-batch sizes. Approximately 15 million observations are employed. All results are produced through a 5-fold cross-validation. The computing resource and hardware specifications can be found in Section 1.5.

However, different mini-batch sizes may also provide different model performance. A large mini-batch size may lead to a poor performance for certain tasks and models, since the corresponding training procedure will more likely converge to a sharp minimum rather than a flat one, which will reduce the ability of generalization of a model (Keskar et al. 2017). In order to verify that different mini-batch sizes can lead to different performance, we have carried out a pretest by using the simulated data. The corresponding mean squared errors (MSE) for different mini-batch sizes are shown in Figure 3.9. The MSE reaches its minimum, namely 0.0022, when the mini-batch size is set 16.

The figure indicates that different mini-batch sizes indeed result in different model performance. Therefore, it is important to also assess our model using different mini-batch sizes. For consistency, the same mini-batch sizes used in Figure 3.8 are investigated, and the corresponding model performance is shown in Figure 3.10.

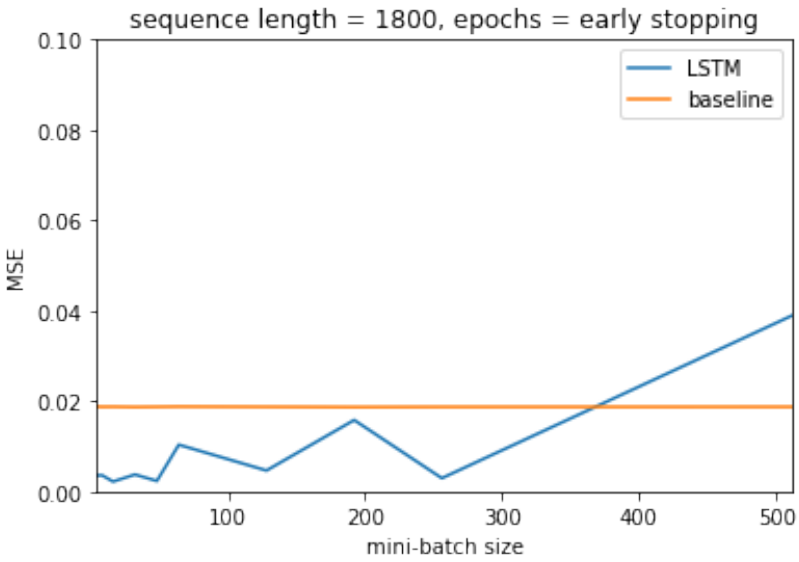


Figure 3.9: MSE for different mini-batch sizes on the simulated dataset. The MSE reaches its minimum, namely 0.0022, when the mini-batch size is set 16.

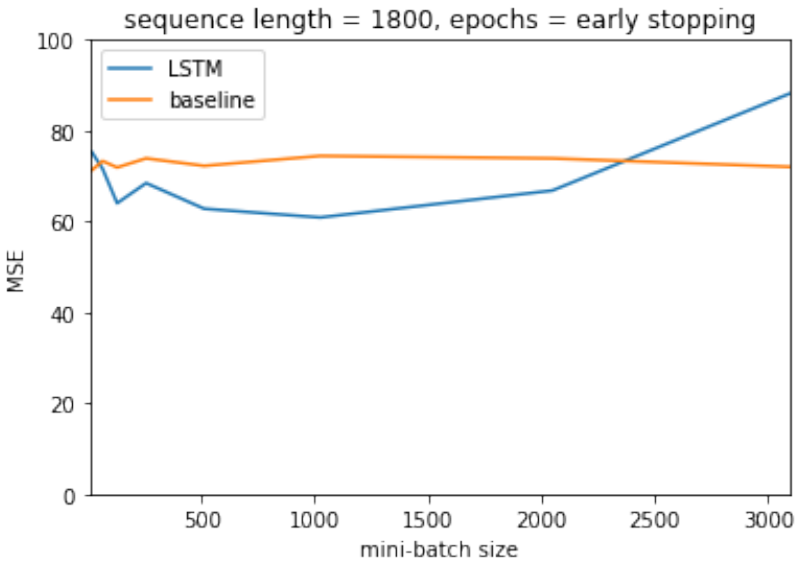


Figure 3.10: MSE for different mini-batch sizes on the ABB dataset. The MSE reaches its minimum, namely 60.89, when the mini-batch size is set 1024.

Among the mini-batch sizes used in Figure 3.8, the required training time is found to be acceptable if the mini-batch size is larger than 128. Furthermore, according to the results shown in Figure 3.10, we can conclude that 1024 is the optimal choice for the mini-batch size, which corresponds to a MSE of 60.89. Notice that the mini-batch sizes adopted in this thesis are typically multiples of 8. Such values are used to prepare for potential further optimization related to the use of parallelization and GPU.

According to the prior knowledge provided by ABB, this section assumed that the optimal sequence length t was 1800. However, as stated before, there can still be room for further fine-tuning. This section will then aim to evaluate the assumption's validity and search for a possible better value for the sequence length t. An approach that can be readily taken is to validate multiple different sequence lengths, until a local minimum with respect to MSE appears. The corresponding MSEs for different sequence lengths are shown in Figure 3.11. Here, the early stopping algorithm, i.e., **Algorithm 5**, is applied, and the mini-batch size is set to 1024.

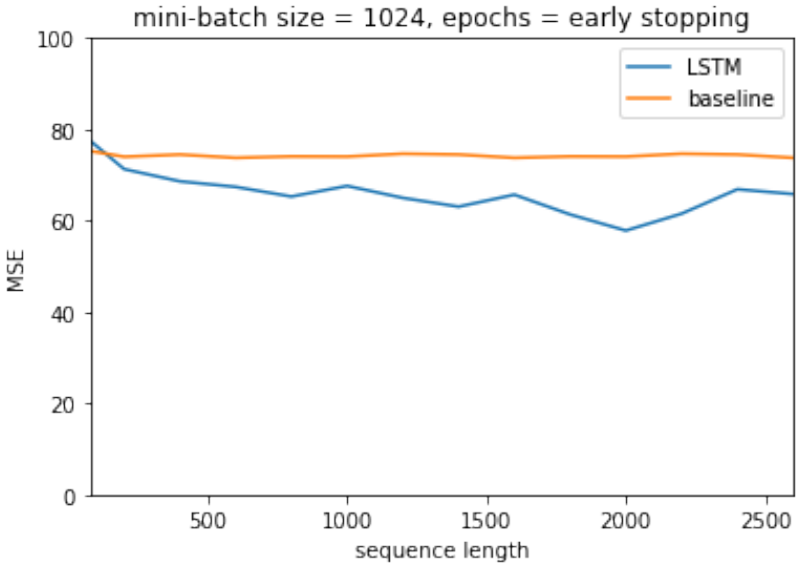


Figure 3.11: MSE for different sequence lengths on the ABB dataset. The MSE reaches its minimum, namely 57.84, when the sequence length is set 2000.

As expected, the MSE does reach its minimum around the sequence length $t = 1800$. However, it is also revealed that $t = 2000$ can deliver an even better performance, namely a MSE of 57.84. The optimal sequence length is thus determined as 2000 in replacement of 1800.

In summary, throughout the studies described in this section, it is concluded that the optimal mini-batch size is 1024, and the optimal sequence length is 2000. Furthermore, the optimal number of epochs is determined by an early stopping algorithm, i.e., **Algorithm 5**.

According to the findings from this section, the settings of our simulator are updated. The parameter τ is set to 666, in order to make sure that $3\tau \approx 2000$. The other parameters are kept unchanged.

3.6 Architecture

According to Section 3.2.2, the current study has been built upon a 10-unit single-layer LSTM followed by a 1-node dense layer. Despite its successes, it is conceivable that some adjustment of the default settings can further improve the performance of this model.

The optimal sequence length $t = 2000$ requires more training time and memory than what are available, which has hence exhibited a large bottleneck for the current model to exploit all given data. In this section, we aim to improve the current model performance by re-assessing its architecture.

For simplicity and clarity, the complete architecture of our model is divided into three parts: the front structure, the LSTM structure (core structure), and the output structure. Note that we will apply the optimal hyperparameters, training and validation algorithms, and preprocessing methods which were determined previously.

3.6.1 Front structure

As described in Section 3.5.3, the sequence length represents the number of historical observations that can influence the current prediction. In this thesis, the optimal sequence length, $t = 2000$, infers that the historical status of a motor in the past 2000 seconds are considered as influential on the current motor temperature.

The goal hence becomes to search for an alternative structure which can provide the same information as the sequence length $t = 2000$, and simultaneously lead to a reduced training time.

This section focuses on downsampling methods, as their functionality matches our objectives, namely reducing the training time while retaining as much information as possible. Prior to a detailed discussion, let us introduce the definitions for the primitive sequence length and downsampling rate.

Definition 3.6.1 (Primitive sequence length and downsampling rate)**.** Assume that a downsampling method a is applied. The method a takes n observations from the original ABB dataset, and produce n' observations as the downsampled data, i.e.,

$$a := f(x_1, x_2, ..., x_n) = (x'_1, x'_2, ..., x'_{n'}), \tag{3.15}$$

where x are original observations from the ABB dataset, and x' are downsampled observations. Here $n' < n$ holds, and the **downsampling rate** b is given by

$$b = \frac{n}{n'}. \tag{3.16}$$

Assume that an LSTM structure with sequence length t uses the downsampled observations provided by method a with downsampling rate b as input. The

primitive sequence length t_p is then given by

$$t_p = t \cdot b = t \cdot \frac{n}{n'}, \tag{3.17}$$

which represents the number of used observations before downsampling.

One downsampling method is sampling. For the dataset considered in this thesis, the given observations were recorded at a frequency of 1 Hz as illustrated in Figure 3.12. Considering the physical circumstances of a motor, the close recordings (observations), for instance, x_1 and x_2 in Figure 3.12, may be highly similar. Therefore, the information provided by recordings which are close in time may also be similar. One may then ignore $b-1$ recordings among each b recordings, expecting not to lose much information that exists in the recordings. A schematic is shown in Figure 3.13.

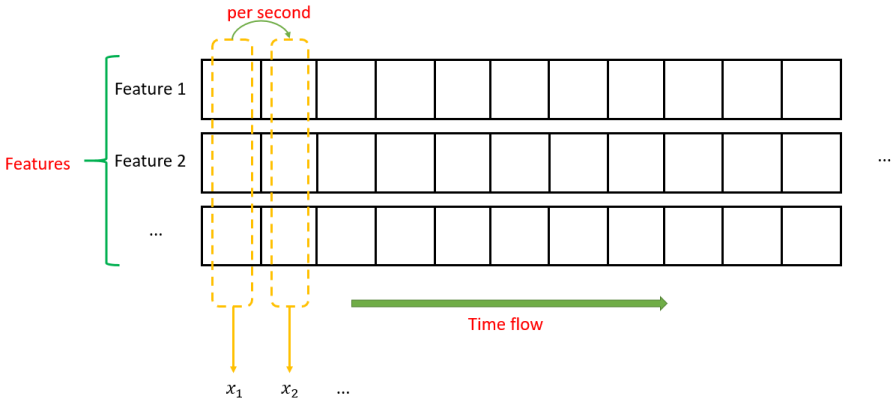


Figure 3.12: An illustration of the original recording procedure. The frequency of recordings here is 1 Hz.

Recall that the input 3-rank tensor X of an LSTM is given as

$$X = (X_1, X_2, ..., X_n),$$

where

$$X_i = \begin{pmatrix} \boldsymbol{x_i} \\ \boldsymbol{x_{i+1}} \\ \vdots \\ \boldsymbol{x_{i+t-1}} \end{pmatrix} \quad for \quad i = 1, 2, ..., n. \tag{3.18}$$

However, after sampling, the input tensor will become

$$X' = (X'_1, X'_2, ..., X'_{n'}),$$

where

$$X'_i = \begin{pmatrix} \boldsymbol{x'_i} \\ \boldsymbol{x'_{i+1}} \\ \vdots \\ \boldsymbol{x'_{i+t-1}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x_{bi-1}} \\ \boldsymbol{x_{bi-1+b}} \\ \vdots \\ \boldsymbol{x_{bi-1+b(t-1)}} \end{pmatrix} \quad for \quad i = 1, 2, ..., n'. \tag{3.19}$$

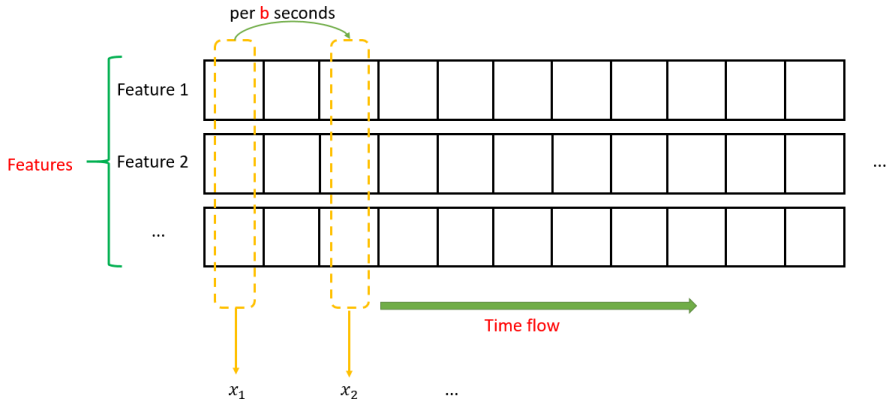


Figure 3.13: An illustration for the sampling method, based on the original recordings. For every b recordings, only 1 recording is now selected. The sampling is therefore also equivalent with increasing the frequency of recording into b Hz

Here, n' is the number of time sequences after downsampling in the given dataset.

As discussed earlier, the front structure should transmit the information during the past 2000 seconds to the LSTM structure. For convenience, we denote the floor function as $\lfloor \cdot \rfloor$. Then, after sampling, one only needs to take into account $\lfloor \frac{2000}{b} \rfloor$ observations instead of involving 2000 observations.

After sampling, a natural assumption for the optimal sequence length is then $t = \lfloor \frac{2000}{b} \rfloor$. However, the sampled dataset may have different characteristics in comparison with the original ABB dataset. After sampling, it is not of certain that $t = \lfloor \frac{2000}{b} \rfloor$, i.e., the primitive sequence length $t_p = b \times t = 2000$, provides the optimal performance, or at least an acceptable performance. Therefore, it is necessary to carry out a pretest on the simulated dataset to verify whether $t_p = 2000$ is a suitable choice.

During the pretest, a grid search is run with respect to b and t. The test model is a 10-unit LSTM followed by a 1-node dense layer. The 5-fold cross-validation is applied here, and the corresponding results are shown in Figure 3.14.

According to the figure, the minimum for a given b is always located around the curve $t_p = 2000$. It indicates that, for the simulated dataset, $t = \lfloor \frac{2000}{b} \rfloor$ can be regarded as the optimal choice. The same grid search on the original ABB dataset is impossible due to the demanded long training time. However, because the simulated dataset follows a physical model similar to the ABB dataset and has the same characteristics, it is then reasonable to also regard $t = \lfloor \frac{2000}{b} \rfloor$ as a suitable setting for the original ABB dataset.

Note again that, for the sequence length $t = 2000$, it is only possible to train the model with a limited amount of data within the feasible time. Reducing the sequence length from $t = 2000$ to $t = \lfloor \frac{2000}{b} \rfloor$ will shorten the training

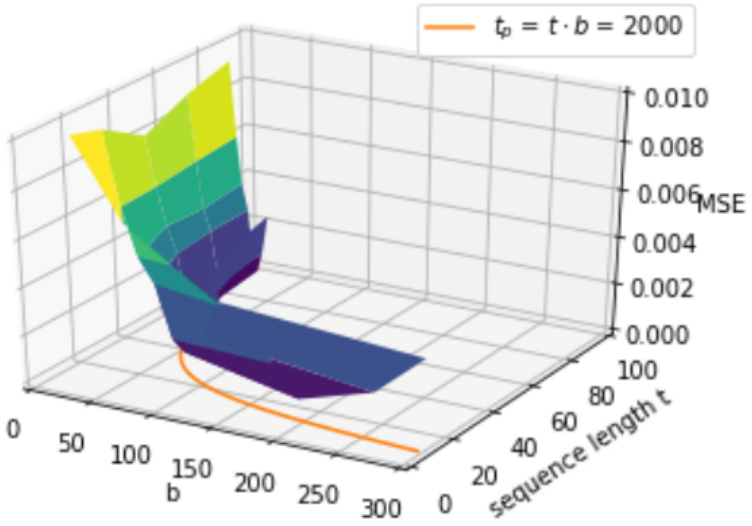


Figure 3.14: MSE for different downsampling rates (b) and different sequence lengths (t) on simulated data. Notice that b = 1 represents no sampling. The orange line is a reference line on the b-t surface. This line helps illustrate our assumption of the optimal t_p, i.e., $t_p = 2000$. MSE for the baseline model is 0.0201.

time, and one can then use a larger amount of data for training. The sampling would therefore hopefully improve the performance of our model. As expected, although the sampled $\lfloor \frac{2000}{b} \rfloor$ observations can to some extent represent the original 2000 observations, some information in small-scale patterns would be lost after sampling. The model performance will only be improved if the lost information after sampling is less valuable than the gained extra information due to the reduced training time.

The sampling also introduces a new hyperparameter b, which refers to the duration of sampling as shown in Figure 3.13. As b increases, the duration of sampling also increases, while the sequence length $t = \lfloor \frac{2000}{b} \rfloor$ and the training time decreases. Meanwhile, the number of omitted observations during sampling increases, and the number of new observations gained due to the reduced training time also increases. Correspondingly, the information lost after sampling will increase, while the information gained due to sampling will also increase. These trends indicate that there may exist an optimal b, which gives the best possible performance by balancing the amount of lost information and gained information.

One may notice that, in Figure 3.14, there is no significant difference among different minimums for different b. However, as stated in Section 3.3.2, the simulated dataset used to produce Figure 3.14 follows a simplified ideal physical

model. Therefore, non-optimal b-s can still provide a best possible performance, and a better b cannot further improve it. Nevertheless, the ABB dataset recorded by ships contains instrumental and measurement noises, hidden confounding variables, and other influential factors. The real data is therefore far from the ideal, and one can then expect an improvement given by an optimized b.

To fine-tune b, one approach is to gradually increase from its initial value, e.g. $b = 2$, until the corresponding mean squared error reaches a global minimum. As b increases, more data can be utilized in a fixed training time. Therefore, while searching for the optimal b, the amount of training data for each b are different. Here, for the given computing resources, the search for different b uses a fixed training time, i.e., 45 minutes. However, if the sequence length $t \leq 25$, i.e., $b \geq 80$, the entire training set can be utilized for training within 45 minutes. The detailed configurations of the computing resources can be found in Section 1.5.

Because the MSE may not be a convex function of b, it is therefore unlikely to always find its global minimum. However, for the case addressed in this thesis, the MSE does have a clear minimum as a function of b, as shown in Figure 3.15.

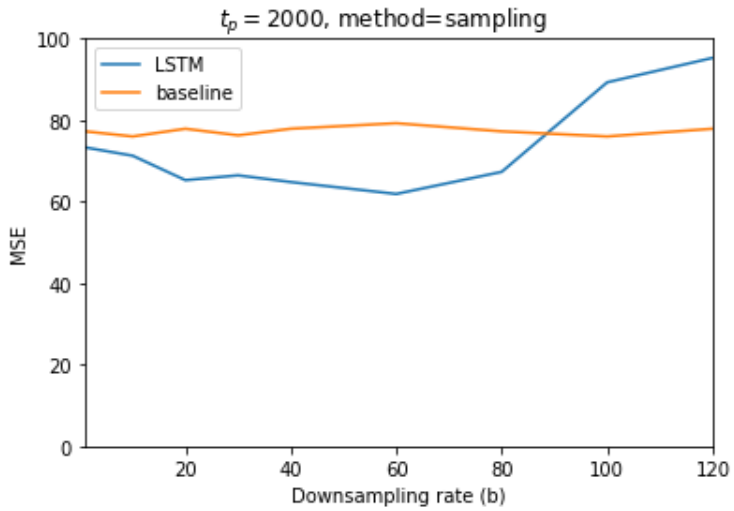


Figure 3.15: MSE for different downsampling rates (b) with using sampling. The model was trained for 45 minutes. However, if $b \geq 80$, the entire training set can be utilized for training within 45 minutes. Notice that b = 1 represents no sampling.

Referring to the MSE obtained from the sequence length $t = 2000$ (see Figure 3.11), a similar MSE value is expected from $b = 1$ in Figure 3.15. However, it is shown that the MSE from $b = 1$ is significantly larger than that in Figure 3.11. A close study found that the discrepancy was caused by the different training time used in the two scenarios. In Figure 3.11, the training time is about 2.5 hours, whereas the respective time in Figure 3.15 is only 45 minutes.

According to Figure 3.15, the MSE reaches its minimum at $b = 60$, and there indeed exists a significant improvement comparing with the MSE at $b = 1$, i.e., no sampling. However, the value of MSE suggests that it can be worth searching for an alternative approach, which provides even better performance. It is known that sampling omits a certain fraction of observations in the past 2000 seconds, and all the information contained by these observations is lost. It is hence conceivable that if one can preserve some of the lost information, a better model may be derived.

One simple but effective approach is averaging within each feature. Rather than completely omit certain observations, we take the average of b observations for each feature, as shown in Figure 3.16. The new 3-rank input tensor after averaging is now

$$X' = (X'_1, X'_2, ..., X'_{n'}),$$

where

$$X'_i = \begin{pmatrix} \boldsymbol{x'_{i,1}} \\ \boldsymbol{x'_{i,2}} \\ \vdots \\ \boldsymbol{x'_{i,t}} \end{pmatrix} = \begin{pmatrix} 1 & x'_{i,11} & \cdots & x'_{i,1p} \\ 1 & x'_{i,21} & \cdots & x'_{i,2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x'_{i,t1} & \cdots & x'_{i,tp} \end{pmatrix}$$

and

$$x'_{i,jk} = \frac{1}{b} \sum_{m=(j-1)b+1}^{jb} x_{i,mk},$$

for $i = 1, 2, ..., n$, $j = 1, 2, ..., t$, and $k = 1, 2, ..., p$. Here, $x'_{i,jk}$ denotes the k-th feature of the j-th observation in the i-th sequence after averaging, and n' is the number of time sequences after averaging in the given dataset.

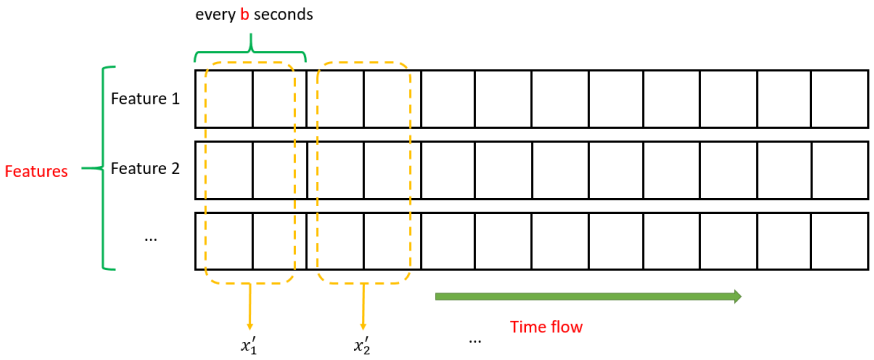


Figure 3.16: An illustration of the averaging of b observations. Note that each feature is averaged individually, and the averaging processes the observations in a non-overlapping manner. In other words, the averaging is based on a tumbling window, but not a sliding window.

Following the same idea as the sampling method, one reasonable assumption for the optimal sequence length is $t = \lfloor \frac{2000}{b} \rfloor$, which is equivalent to $t_p = 2000$. This assumption is further assessed with the simulated dataset in the similar manner as used in the sampling method. A grid search in the space of sequence length t and downsampling rate b is then carried out. The corresponding results are shown in Figure 3.17. The figure indicates that $t_p = 2000$ is the optimal choice on the simulated dataset, and averaging can indeed improve our model performance. This optimal t_p is the same as that found for the sampling approach. However, in addition to reducing the sequence length, the averaging also retains partially the information of all given observations. The averaging is hence expected to perform better than the sampling approach.

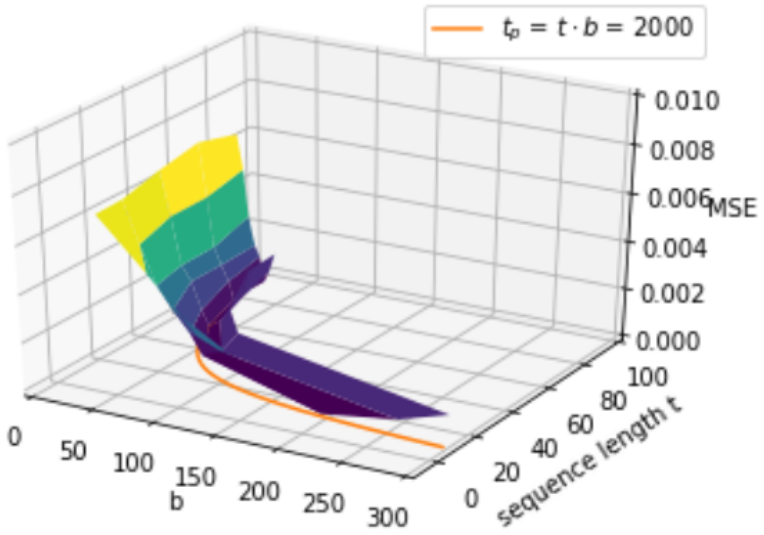


Figure 3.17: MSE for different downsampling rates (b) and different sequence length (t) with using averaging on the simulated dataset. Notice that b = 1 represents that averaging is not employed. The orange line is a reference line. MSE for the baseline model is 0.0201.

Such studies are therefore also carried out for the original ABB dataset. For different downsampling rates b, the corresponding MSEs are shown in Figure 3.18. All other settings here are the same as those for Figure 3.15, and the training time for each b is also fixed as 45 minutes. However, if the sequence length $t \leq 25$, i.e., $b \geq 80$, the entire training set can be utilized for training within 45 minutes. Comparing with the mean squared errors in Figure 3.15, the averaging indeed enhances our model more than sampling. The minimal MSE with sampling is 65.46 (at $b = 60$), whereas the minimal MSE with averaging becomes 54.01 (at $b = 40$), which indicates an improvement of 17.62%.

As described in Definition 3.6.1, the downsampling methods essentially create a

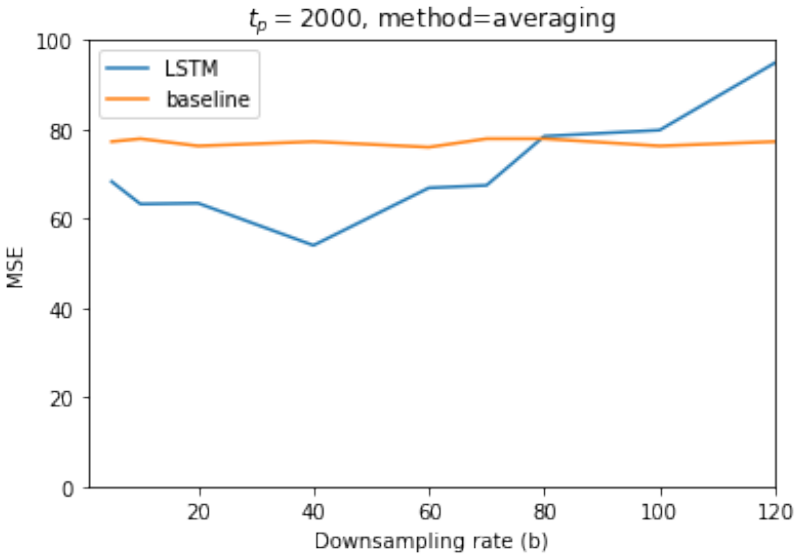


Figure 3.18: MSE for different downsampling rates (b) with using averaging. The model was trained for 45 minutes. However, if $b \geq 80$, the entire training set can be utilized for training within 45 minutes. Notice that b = 1 represents that averaging is not employed.

link function f such that

$$(x'_1, x'_2, ..., x'_{n'}) = f(x_1, x_2, ..., x_n), \tag{3.20}$$

where x' are the observations after downsampling, and x are the original observations provided by ABB. Here n must be an integer multiple of n', such that the downsampling rate is given by

$$b = \frac{n}{n'}. \tag{3.21}$$

There exist many different possibilities for such a link function f, and it is impractical to test all of them. One common solution is the convolutional neural network (CNN), since every possible link function f essentially corresponds to a specific convolutional neural network. The introduced sampling and averaging methods are also included. A concatenated CNN structure is hence introduced as shown in Figure 3.19. Note that the CNNs used here are 1-dimensional.

The outputs of different CNN layers within this structure are concatenated and afterwards taken as an input to the LSTM structure. Note that each CNN layer handles a single feature, and there is no communication between different features within each individual CNN.

The new structure introduces three new hyperparameters, i.e., the number of filters, kernel size, and strides. Typically, tuning is done by a grid search, which

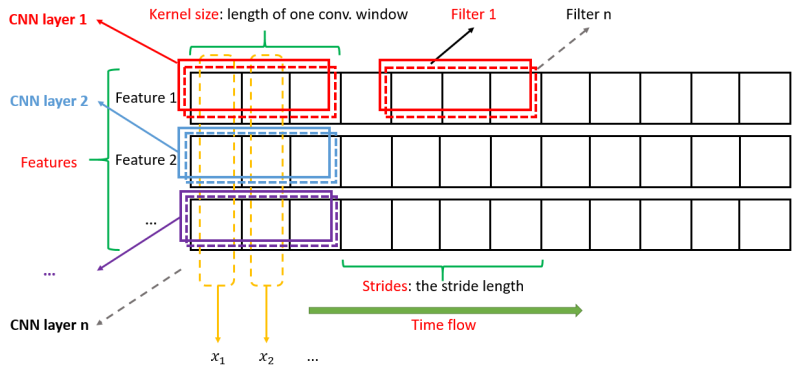


Figure 3.19: Concatenated CNN structure, where each CNN handles only one feature. The CNNs used here are 1-dimensional, and there is no communication between different features within CNNs. Kernel size is the length of one CNN window and strides is the stride length of CNN window.

investigates all possible combinations of the 3 hyperparameters and chooses the optimum in terms of MSE. However, such an algorithm requires an unacceptable amount of computing time.

According to Figure 3.19, the kernel size k and strides s determine the sequence length t by the expression

$$t = \left\lfloor \frac{2000 - k}{s} \right\rfloor + 1, \tag{3.22}$$

where $\lfloor \cdot \rfloor$ denotes the floor function. Since the sequence length t is the core hyperparameter of our structure, it is reasonable to assume that the two hyperparameters, i.e., kernel size and strides, are the most important parameters, and that the number of filters has a negligible influence on the model performance. Field experience indicates that five filters can deliver a reasonably good performance. It is hence used in this study as a starting value. The number of filters will be further fine-tuned after the kernel size and strides are optimized.

As described before, the training and validation on the ABB dataset requires a considerable amount of time. We have therefore pretested this method on the simulated dataset in advance. The subsequent tuning of the hyperparameters will be carried out for the ABB dataset, only if the concatenated CNN structure shows a satisfying performance on the simulated dataset.

We then process a grid search with respect to the kernel size and strides on the simulated dataset. The corresponding mean squared errors are shown as in Figure 3.20.

Because it is only a pretest prior to the actual studies, it is therefore unnecessary to proceed with further tuning for the number of filters. The objective of the pretest is to ensure that the concatenated CNN structure can deliver

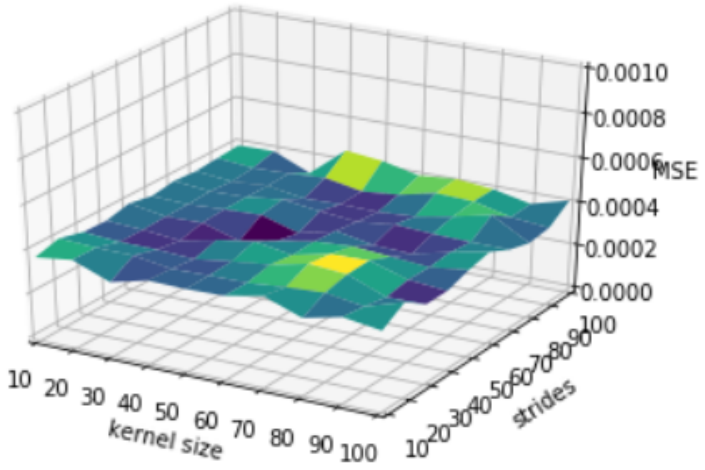


Figure 3.20: MSE of Concatenated CNN structure for different kernel sizes and strides on the simulated dataset. The MSE reaches its minimum, i.e., 0.0002 when $kernel\ size = 40$ and $strides = 50$. The MSE of the baseline model is 0.0203.

the expected performance, which is the case as illustrated by Figure 3.20. Furthermore, comparing Figure 3.20 with Figures 3.14 and 3.17, it can be found that the concatenated CNN does not lead to a better MSE than averaging. As stated before, the simulated dataset was created based on an ideal physical model. Therefore, the performance of our model has already reached its optimum when using sampling or averaging, and the possible better solution, i.e., concatenated CNN, may not further improve the MSE.

Up to now, it is evident that the concatenated CNN can deliver at least as good performance as the averaging, and so the aforementioned tuning procedure can be pursued further. First, the kernel size and strides are tuned through a grid search, and the corresponding results are shown in Figure 3.21. Note that the training time for each hyperparameter combination is still fixed as 45 minutes. However, if the sequence length $t \leq 25$, the entire training set can be utilized for training within 45 minutes.

The MSE surface is considerably noisy. However, it is possible to see that the mean squared errors around $kernel\ size = 50$ and $strides = 50$ are relatively smaller compared to other values. In Figure 3.21, the MSE reaches its minimum, namely 48.52, when $kernel\ size = 40$ and $strides = 40$. It is then reasonable to simply regard $kernel\ size = 40$ and $strides = 40$ as the optimal choice.

Then, given the kernel size and strides, we revisit the remaining hyperparameter, i.e., the number of filters, and optimize its value. Figure 3.22 shows the corresponding MSE for different numbers of filters with $kernel\ size = 40$ and $strides = 40$. Note that the training time for each number of filters is fixed as 45 minutes. It is shown that, when the number of filters is set 4, the MSE

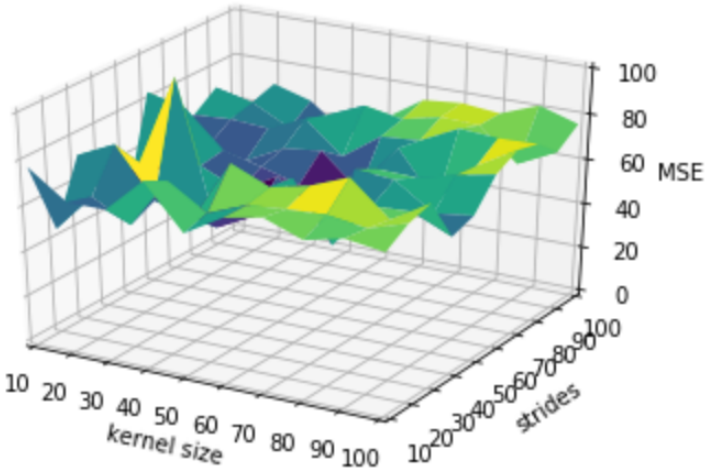


Figure 3.21: MSE of Concatenated CNN structure for different kernel sizes and strides on the ABB dataset. The training time for each hyperparameter combination is fixed as 45 minutes. However, if the sequence length $t \leq 25$, the entire training set can be utilized for training within 45 minutes. The minimal MSE is 48.52, when $kernel\ size = 40$ and $strides = 40$. The MSE of the baseline model is 78.81.

reaches its minimum, i.e., 46.18. Such a minimum has an improvement by 10.16% compared to that from averaging, as shown in Figure 3.18. However, considering that the concatenated CNN is a much more adaptive method than averaging, the improvement is expected to be even larger than 10.16%.

This result may be understandable from a physical point of view. Considering a given feature, the averaging of multiple observations of a constant size is proportional to the sum of these observations. Furthermore, because the given observations were recorded along the axis of time, so the sum of these observations in a certain period will have a similar effect as the time integral of this feature on the target (temperature).

The situation could be that, for certain features, instead of its original value, exploiting the time integral can improve the model performance. For example, the time integral of **PO**, namely the power of a motor, is the corresponding energy used to drive the motor during a period. Recall the Equations (3.4) and (3.6) in Section 3.3.1, one can then notice that the relation between the energy and the temperature should be linear, which indicates that the energy is a better predictor than the power.

We then attempt to search for a better front structure than the concatenated CNN layers. Recall that the exploited features in this thesis are **PO** (Power),

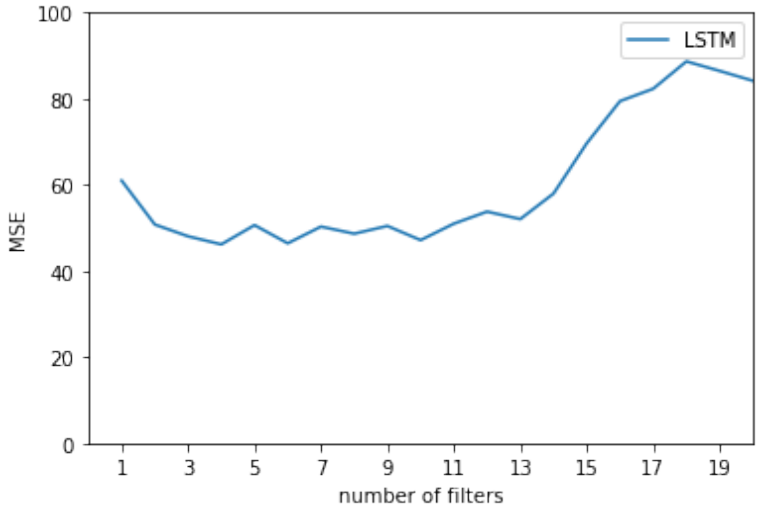


Figure 3.22: MSE of different number of filters for Concatenated CNN structure on the ABB dataset. The training time for each number of filters is fixed as 45 minutes. The MSE reaches its minimum, i.e., 46.18, when the number of filters is set 4. The MSE of the baseline model is 72.29

CI (temperature of input cooling air), **TO** (Torque), and **SP** (Speed). Given that these features come from one motor, there may exist possible interactions between them such that they collectively affect the output target (i.e., temperature). So far, the sampling, averaging, and concatenated CNN structure have not taken into account such interaction relationships between different features.

We hence introduce the 1-dimensional (1D) CNN structure, which also takes into account such interactions. A schematic of the 1D CNN is shown in Figure 3.23. The strong coupling of different features could improve the model performance.

Different from the concatenated CNN, it does not make sense to pretest the 1D CNN on the simulated dataset, since there is only one feature in the simulated dataset. We hence directly validate the 1D CNN on the ABB dataset. Following the same procedure as the concatenated CNN structure, the tuning first focuses on the kernel size and strides. According to field experience, the number of filters is initially set as 4. We then carry out a grid search to get the optimal values for the kernel size and strides. The training time for each hyperparameter combination is fixed as 45 minutes. However, if the sequence length $t \leq 25$, the entire training set can be utilized for training within 45 minutes. The corresponding MSE surface is illustrated in Figure 3.24.

As shown, most of the minimums are located around $kernel\ size = 60$ and $strides = 50$. A close comparison of these minimums reveals that the combination of $kernel\ size = 50$ and $strides = 50$ gives the smallest MSE, i.e., 32.53, and thus delivers the best performance. Given the optimal kernel size

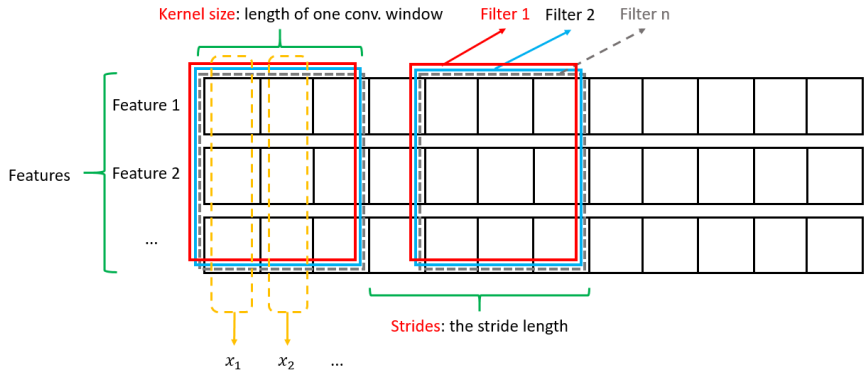


Figure 3.23: 1-dimensional CNN structure. Kernel size is the length of one CNN window and strides is the stride length of CNN window. Here, all features are handled by the same CNN layer.

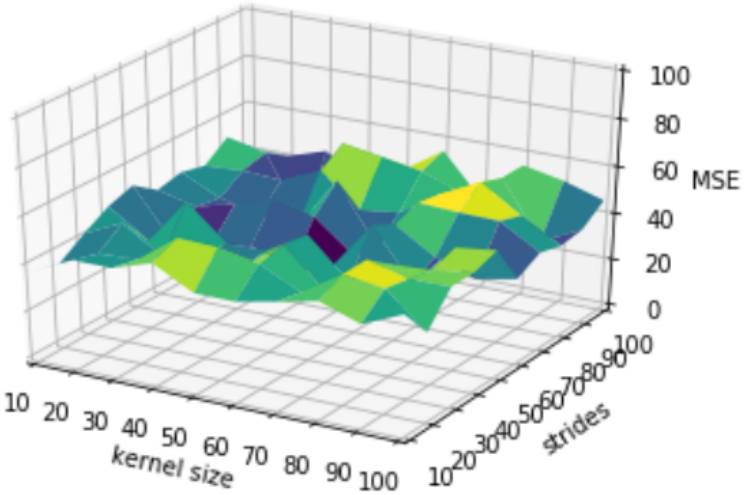


Figure 3.24: MSE of 1-dimensional CNN structure for different kernel sizes and strides on the ABB dataset. The training time for each hyperparameter combination is fixed as 45 minutes. However, if the sequence length $t \leq 25$, the entire training set can be utilized for training within 45 minutes. The MSE reaches its minimum, i.e., 32.53, when $kernel\ size = 50$ and $strides = 50$. The MSE of the baseline model is 76.55.

and strides, work is carried out to update the number of filters. The MSEs obtained from different number of filters are shown in Figure 3.25. The training time for each number of filters is fixed as 45 minutes.

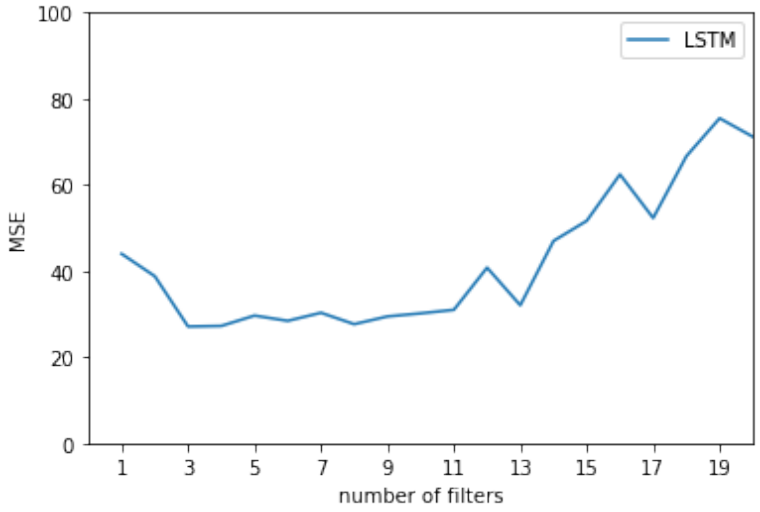


Figure 3.25: MSE of 1-dimensional CNN structure for different number of filters on the ABB dataset. The training time for each number of filters is fixed as 45 minutes. The minimal MSE is 27.11 when 3 filters are employed. The MSE of the baseline model is 75.67.

The trend illustrates that different numbers of filters provide similar performance in the range of 2-11. However, according to the structure of a CNN layer, a smaller number of filters leads to a shorter training time. Since the training time is one of the most important criteria in this thesis, it is thereby favorable to choose the smallest number of filters. However, it is difficult to assure that the optimal performance region in Figure 3.25 lies precisely in between 2 and 11. In order to avoid a possible non-optimal performance, 5 is selected as the optimal number of filters.

The discussions in this section indicate that the 1-dimensional CNN is the optimal front structure among the candidates. The optimal number of filters is suggested to be 5, the kernel size is determined as 50, and the optimal stride is 50. In the remaining parts of this thesis, the front structure with these selected settings has been used.

3.6.2 LSTM structure

This section demonstrates the LSTM structure, which is the core structure of the present model. As described in Section 3.5.3, one of the most important hyperparameters of an LSTM is the sequence length t. According to the previous studies in this chapter, if the front structure is determined, the optimal sequence length can also be known.

For the 1-dimensional CNN layer used in this thesis, the optimal front structure

defines the corresponding sequence length t as

$$t = \left\lfloor \frac{2000 - k}{s} \right\rfloor + 1, \tag{3.23}$$

where $\lfloor \cdot \rfloor$ denotes the floor function. Here k and s are the kernel size and strides, respectively. Given the optimal setting for this CNN layer, i.e., $k = 50$ and $s = 50$, the optimal sequence length t becomes

$$t = \left\lfloor \frac{2000 - 50}{50} \right\rfloor + 1 = 40. \tag{3.24}$$

As described in Section 3.2.2, the default LSTM structure used in this thesis is a 10-unit LSTM layer. For a given sequence length t, in order to achieve a better model performance, one can further fine-tune the two remaining hyperparameters, i.e., the number of LSTM layer(s) and the number of units of each layer. Here, the number of units of a layer corresponds to the output dimensionality of this layer. If defining the number of layers as n_l, the number of units $\boldsymbol{n_u}$ can then be given by a vector

$$\boldsymbol{n_u} = (n_{u,1}, n_{u,2}, ..., n_{u,n_l}), \tag{3.25}$$

where $n_{u,i}$ is the number of units of the i-th layer.

As shown, the number of layers n_l, needs to be optimized a priori before optimizing the number of units $\boldsymbol{n_u}$, because the latter is affected by the former. However, theoretically, there are unlimited possibilities regarding the number of layers. It is therefore necessary and preferable to first restrict its range based on both theoretical analysis and field experience.

In Section 2.4, as shown in Figure 2.8, each LSTM cell contains five nonlinear transformations. One is in the forget gate, two are in the input gate, one is in the output gate, and one is between the output gate and the new hidden state. Furthermore, since the optimal sequence length is 40 in the present model according to Equation (3.24), there will be maximum 40 cells and thus 200 nonlinear transformations within one LSTM layer. In this thesis, if we define the number of nonlinear transformations between an input i and an output o as N_{nl}, its value will then be given by

$$N_{nl} = 40(t_o - t_i). \tag{3.26}$$

Here, t_o and t_i are the time of this specific output and input, respectively.

For a feedforward neural network, there will only be one nonlinear transformation between any input and any output within a layer. Therefore, the measures of complexity of an LSTM and a feedforward neural network are different due to the different number of nonlinear transformations (S. Zhang et al. 2016). A feedforward neural net is regarded as a deep neural net (DNN) if it contains hundreds of layers. However, as stated before, even a single-layer LSTM can have a similar complexity as a DNN.

One can notice that the default single-layer 10-unit LSTM has been performing well. Adding more LSTM layers usually enhances the model performance.

However, in practice, more LSTM layers bring an enormous amount of extra training time. Therefore, it is not always worth adding extra layers and spending extra training time, since the gained improvement can be very limited and sometimes negligible. Therefore, the tuning procedure can begin with a single-layer LSTM, incrementally add more layers until the ratio of the increased training time and the decreased MSE reaches a threshold. For a better understanding, let us define what we term the MSE improvement rate:

Definition 3.6.2 (MSE improvement rate)**.** Denote the change of training time brought by the change of a model as Δt_{ex} and the corresponding variation with respect to MSE as Δe, the MSE improvement rate v is then defined as

$$v = \frac{\Delta e}{\Delta t_{ex}}. \tag{3.27}$$

The predictions of our model are the temperature ($^{\circ}C$) of motors, and the MSEs of our predictions are always less than 100. Therefore, it would be reasonable to consider the improvement less than 10% as insignificant, as it corresponds to an improvement of the predicted temperature by approximately $1^{\circ}C$. If such an insignificant improvement brings an extra training time which is larger than 40 minutes, we will stop adding more layers. This indicates that the threshold to be used can be given by

$$v_\sigma = \frac{10\%}{40min} = 0.25\%/min.$$

If the MSE improvement rate of adding an extra layer is less than v_σ, no more layers will be added. Here, the time threshold, i.e., 40 minutes, is predominantly governed by the available computing resources, as introduced in Section 1.5.

Note that one extra LSTM layer brings one extra hyperparameter, i.e., the number of units of this new layer. This new hyperparameter is highly correlated with the number of units of other layers, i.e., the vector $\boldsymbol{n_u} = (n_{u,1}, n_{u,2}, ..., n_{u,n_l})$. It indicates that all these hyperparameters should be tuned again if a new layer is added. Therefore, if n_l layers are employed, we are supposed to process a grid search with respect to the number of units vector $\boldsymbol{n_u}$. As n_l increases, the possibilities of $\boldsymbol{n_u}$ also increases exponentially, and it will become impractical to fulfill the grid search if n_l is too large. Specifically, with our given resource stated in Section 1.5, it takes months to run the grid search when $n_l = 4$, which is infeasible.

One may consider the pretests on the simulated dataset to be helpful in such cases. However, since the simulated dataset was created based on a simplified ideal physical model, its required model complexity is essentially lower than that for the ABB dataset. Therefore, if the model with extra LSTM layers does not perform well on the simulated dataset, it is still uncertain whether this model is suitable for the ABB dataset.

Nevertheless, as shown in Table 3.10, the MSE improvement rate with $n_l = 3$ has been found to be smaller than our threshold. Furthermore, the use of $n_l > 3$ should not bring better improvement than $n_l = 3$, since we see from

Table 3.10 that 2 layers can already produce enough complexity for the task in this thesis. Based on the trend illustrated by Table 3.10, the use of more than 2 layers can even lead to some overfitting and yield an adverse effect.

Number of LSTM layers	Minimal MSE	MSE improvement rate
1	26.93	N/A
2	22.49	0.84
3	24.57	−1.03

Table 3.10: MSE improvement rate for different number of LSTM layers on the ABB dataset. Note that the MSE improvement rate for $n_l = n$ is calculated by the MSEs of $n_l = n$ and $n_l = n - 1$.

Note that the results shown in Table 3.10 are produced by the optimal $\boldsymbol{n_u}$. These optimal $\boldsymbol{n_u}$ are determined through grid searches. The results of grid searches for $n_l = 1$ and $n_l = 2$ are illustrated in Figure 3.26 and Figure 3.27, respectively. As for the grid search of $n_l = 3$, it is difficult to clearly illustrate the 4-dimensional plot. We therefore directly give the final optimal $\boldsymbol{n_u}$ as $(15, 20, 20)$.

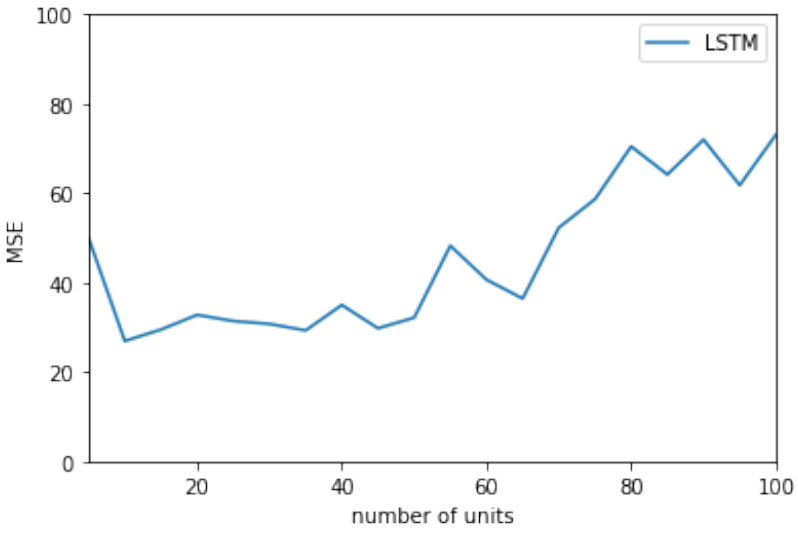


Figure 3.26: MSE with respect to $\boldsymbol{n_u}$ when $n_l = 1$. The MSE reaches its minimum, namely 26.93, when $\boldsymbol{n_u} = (10)$. The MSE of the baseline model here is 78.29.

The optimal hyperparameters of the LSTM structure are now determined. However, as mentioned before, there can be potential overfitting risks, namely, the lack of model generalization. Typically, field experience suggests two useful methods to improve the generalization. One is the batch normalization proposed by Ioffe and Szegedy 2015, and the other is the dropout technique introduced by Srivastava et al. 2014. However, batch normalization does not match the

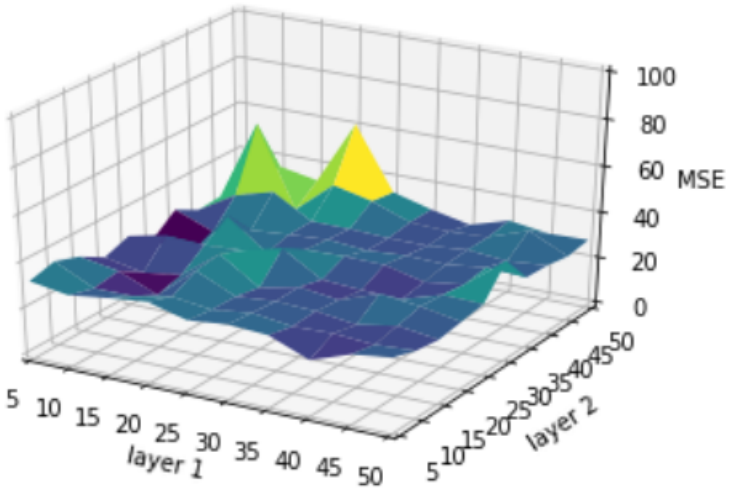


Figure 3.27: MSE with respect to $\boldsymbol{n_u}$ when $n_l = 2$. The MSE reaches its minimum, namely 22.49, when $\boldsymbol{n_u} = (10, 15)$. The MSE of the baseline model here is 78.29.

structure of an LSTM, and the retrofitted technique, i.e., layer normalization, is therefore employed instead. In what follows, we validated the model performance of both the dropout and layer normalization, in order to determine the optimum.

First, let us focus on the dropout technique. As discussed in Section 2.4, LSTM is different from feedforward neural networks, since there are recurrent connections within one LSTM layer, which does not exist in a feedforward neural network layer. Therefore, the dropout technique applied to an LSTM is slightly different from the original procedure. Previous studies found that the dropout can give even better performance if it is not only employed for the inputs to an LSTM layer, but also the recurrent connections within an LSTM (Gal and Ghahramani 2016).

Still, it is difficult to tell if there indeed exists overfitting. Following the same idea stated earlier, a pretest is carried out on the simulated dataset before working on the ABB dataset. As stated before, the required model complexity on the simulated dataset is lower than that on the ABB dataset, since the actual physical model of the simulated data is much simpler. Therefore, if a model does not raise overfitting concerns when being validated on the simulated dataset, there should not be overfitting when used on the ABB dataset. In other words, if the dropout technique deteriorates the model performance on the simulated dataset, it does not make sense to validate it on the ABB dataset.

Dropout brings a new hyperparameter into our model, i.e., the dropout rate r_d. It represents the fraction of the units to drop for the linear transformation of

the inputs and the recurrent state (Chollet et al. 2015), whose value is in the range of 0 to 1. The effect of different dropout rates on the model performance (MSE) for the simulated dataset is shown in Figure 3.28.

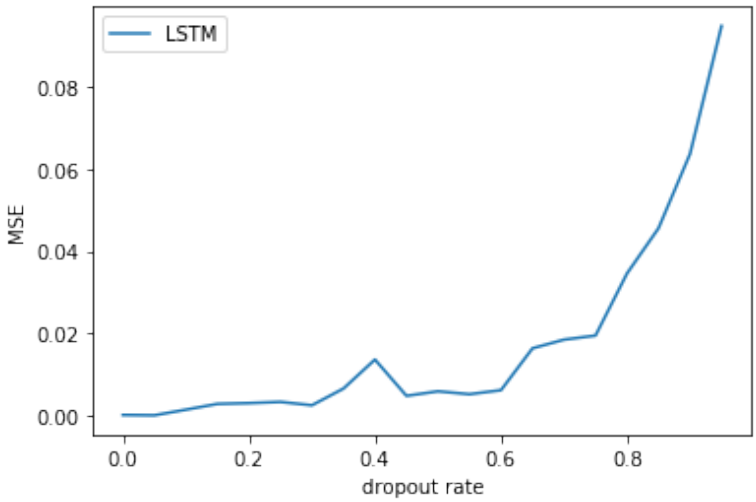


Figure 3.28: Effect of different dropout rates on MSE for the simulated dataset. If dropout rate is set 0, then no dropout will be carried out. The MSE reaches it minimum, i.e., 0.0001, when the dropout rate is 0.05. The MSE of the baseline model here is 0.0197.

The figure indicates that, for the simulated dataset, the improvement brought by the dropout is insignificant. However, recall that our model has almost reached the best possible performance on the simulated dataset, it is hence impractical to expect much improvement brought by the dropout. Then, we still validate the dropout on the ABB dataset. The effects of different dropout rates on MSE are illustrated in Figure 3.29. Here, the training time for each dropout rate is still fixed as 45 minutes. However, for a dropout rate larger than 0.15, all training data are utilized within 45 minutes. Compared to the best performance obtained before, i.e., 22.49, a small improvement, i.e., 10.49%, is made by the dropout at $r_d = 0.05$. The corresponding minimal MSE when employing dropout is given as 20.13.

After studying the dropout technique, we also need to assess the performance of layer normalization. There is no hyperparameter introduced in layer normalization. So unlike the dropout, it is unnecessary to carry out the pretests. One can then directly validate the performance of layer normalization using the ABB data. As shown in Table 3.11, the corresponding MSE given by a 5-fold cross-validation is 19.94, which indicates that the layer normalization also provides a slight improvement for the original model.

Previous work suggests that the layer normalization and dropout should not be employed on the same model (Li et al. 2019). It can therefore be useful to select the one that gives larger improvement in accuracy and computing speed.

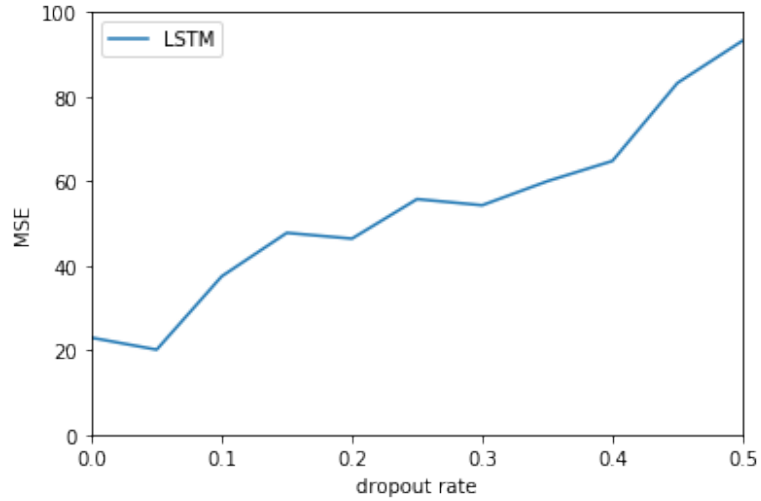


Figure 3.29: Effect of different dropout rates on MSE for the the ABB dataset. If dropout rate is set 0, then no dropout will be carried out. Here, the training time for each dropout rate is fixed as 45 minutes. However, for a dropout rate larger than 0.15, all training data are utilized within 45 minutes. The MSE reaches it minimum, i.e., 20.13, when the dropout rate is set 0.05. The MSE of the baseline model here is 78.47

Since the MSEs derived from both techniques are highly similar, the selection criterion therefore becomes their corresponding training time.

Theoretically, both techniques can accelerate the speed of convergence during training and therefore reduce the computing time. Given the same amount of data, the corresponding training time of the original model, the model with the dropout technique, and the model with layer normalization are listed in Table 3.11.

Training technique	MSE	Training time (hours)
No technique	22.49	2.44
Dropout rate = 0.05	20.13	2.60
Layer normalization	19.94	1.96

Table 3.11: The corresponding training time and MSEs using different training techniques. The training dataset consists of approximately 500 million recordings (observations).

According to the results shown in Table 3.11, the model benefits most from the layer normalization, as it gives the shortest training time. It is hence adopted as the method for our model.

Up to now, one can conclude that the optimal LSTM structure is a two-layer LSTM, with the layer normalization being applied between them. The optimal

number of units is given by $\boldsymbol{n_u} = (10, 15)$.

3.6.3 Output structure

It is essential to notice that the output dimensionality of an LSTM layer corresponds to the number of its units. The number of units is typically more than one. However, for our task in this thesis, the dimensionality of the target output is always one, i.e., the current temperature of a motor. It is hence necessary to build an output structure which converts the output of our LSTM structure into a single value, i.e., the target temperature.

Field experience has shown successful applications of a simple but satisfactory structure, namely the 1-node dense layer. The 1-node dense layer is essentially a fully connected feedforward neural network layer, which is also the default output structure that used in this thesis. Despite its popularity and simplicity, there may exist more complex and better output structures.

A possible alteration is to add extra layers into the output structure. It is worth noticing that more layers will bring more hyperparameters. There are two types of hyperparameters, i.e., the number of layers and the number of nodes in each layer. We refer to the number of layers as n_l and the number of nodes as a vector $\boldsymbol{n_u} = (n_{u,1}, n_{u,2}, ..., n_{u,n_l})$, where $n_{u,i}$ denotes the number of nodes of the i-th layer.

It is always possible to add more layers into the output structure. As we explained before, while tuning the LSTM structure, adding new layer(s) can possibly improve the model performance. However, it also results in a longer training time. The improvement of model performance may be too small to justify the extra training time achieved by the additional layer. Here, we refer to the metric defined by Definition 3.6.2, i.e., the MSE improvement rate, to assess the improved model performance and the increased training time.

The tuning procedure for n_l and its corresponding $\boldsymbol{n_u}$ are the same as those in Section 3.6.2. For each evaluation, the new layers is added incrementally. After a new layer is added, the corresponding MSE improvement rate is calculated by using the optimal $\boldsymbol{n_u}$, which is determined by a grid search. No layer(s) will be added if the MSE improvement rate is lower than a predefined threshold. To be consistent, it is reasonable to employ the same threshold as that used in Section 3.6.2, namely $0.25\%/minute$.

As explained in Section 3.6.2, it is not worth pretesting possible structures using the simulated dataset. We therefore directly assess our output structures on the ABB dataset. In this study, the target is a single value, namely the predicted motor temperature. Consequently, the last layer of the output structure has to be one-node. The corresponding MSE improvement rates on the ABB dataset are shown in Table 3.12.

The MSE improvement rates were calculated by the optimal MSE for each n_l. As described before, for each n_l, a grid search was carried out to tune the corresponding hyperparameter(s). We therefore illustrate the results of these grid searches for $n_l = 2$ and $n_l = 3$. The results for $n_l = 2$ and $n_l = 3$ are

Number of dense layers	Minimal MSE	MSE improvement rate
1	19.94	N/A
2	34.13	−0.68
3	43.57	−1.35

Table 3.12: MSE improvement rate for different number of dense layers on the ABB dataset. Note that the MSE improvement rate for $n_l = n$ is calculated by the MSEs of $n_l = n$ and $n_l = n - 1$.

shown in Figure 3.30 and Figure 3.31, respectively.

As opposed to the tuning of the LSTM structure in Section 3.6.2, the number of nodes of the last layer of the output structure is already determined as one. Therefore, for n_l dense layers, only $n_l - 1$ numbers of units need to be tuned. It is therefore feasible to illustrate the grid search results for $n_l = 3$.

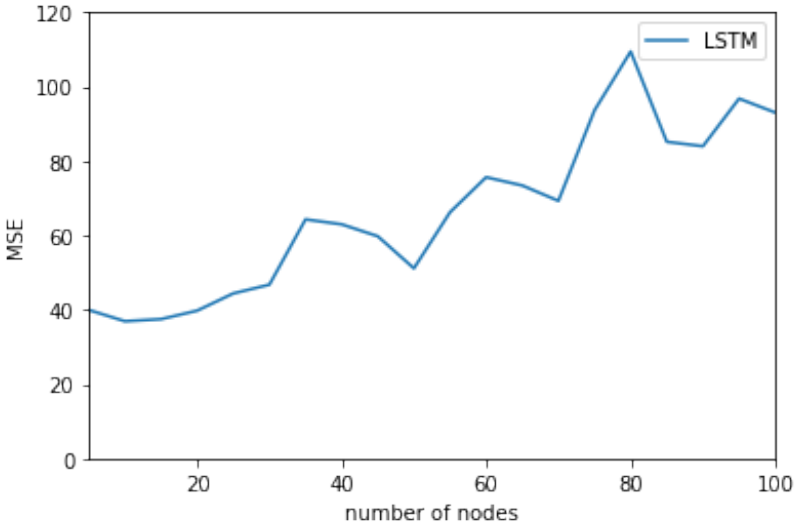


Figure 3.30: MSE as a function of $\boldsymbol{n_u}$ for $n_l = 2$. Note that the number of nodes of the last layer is fixed as 1. The MSE reaches its minimum, namely 34.13, when $\boldsymbol{n_u} = (5, 1)$. The MSE of the baseline model here is 81.52.

As illustrated in Table 3.12, for $n_l \geq 2$, the corresponding MSE improvement rates are less than our predefined threshold, and even a negative MSE improvement rate is given. It is then natural to regard $n_l = 1$ as the optimal number of dense layers in the output structure. The corresponding $\boldsymbol{n_u}$ is then a single-value vector, i.e., (1).

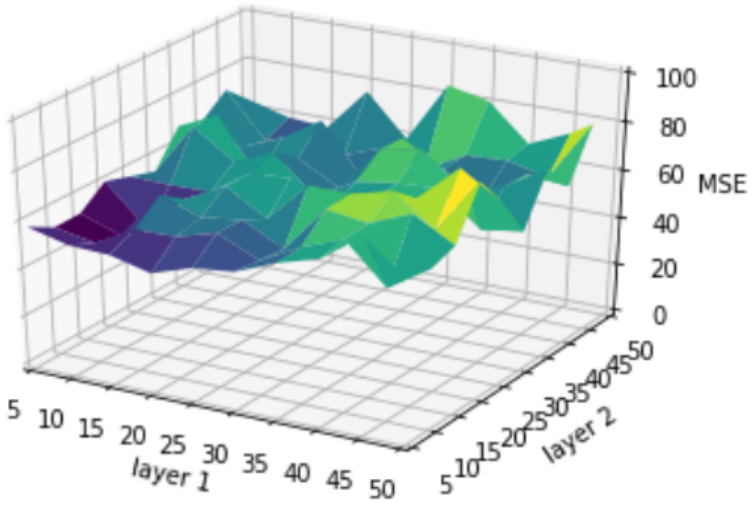


Figure 3.31: MSE as a function of $\boldsymbol{n_u}$ for $n_l = 3$. Note that the number of nodes of the last layer is fixed as 1. The MSE reaches its minimum, namely 43.57, when $\boldsymbol{n_u} = (5, 10, 1)$. The MSE of the baseline model here is 81.52.

3.7 Summary

In this chapter, the work has been largely concentrated on the optimization and validation of the approach, in order to solve the encountered challenges and accomplish the given task for this thesis. The optimization and validation procedures adopted in this chapter are considerably complicated. Therefore, this section aims to summarize the conclusions and results obtained from this chapter. From the six given features, four are eventually selected for further study:

- CI (inlet cooling air temperature [$^\circ C$]).
- PO (normalized motor power [%]).
- SP (normalized motor speed [%]).
- TO (normalized mechanical torque [%]).

The target (temperature) of the model is determined as the average temperature T of the original six given targets (temperatures), i.e.,

$$T = \frac{U_1 + U_2 + V_1 + V_2 + W_1 + W_2}{6}. \tag{3.28}$$

Also, the following conclusions are drawn from this chapter:

- Mini-batch gradient descent is employed to train the model, and the optimal mini-batch size is set 1024.
- The learning rate during training is determined by a dynamic algorithm, namely **rmsprop**.
- The number of epochs is dynamically determined by an early stopping algorithm, namely **Algorithm 5**.
- The training procedure follows **Algorithm 3**, and shuffling is employed for the training data.
- Both layer normalization and dropout are useful techniques for our model. Due to a better performance, the layer normalization is used.

The optimal model is schematically illustrated in Figure 3.32. The analysis showed that :

- The MSE of the developed model is 19.94.
- The MSE of the baseline model (linear model) is 76.53.
- The MSE of the alternative baseline model (feedforward neural network) is 74.32.

The MSEs are given by 5-fold cross-validation, and the entire validation dataset is employed, which means that the training time is not fixed as 45 minutes.

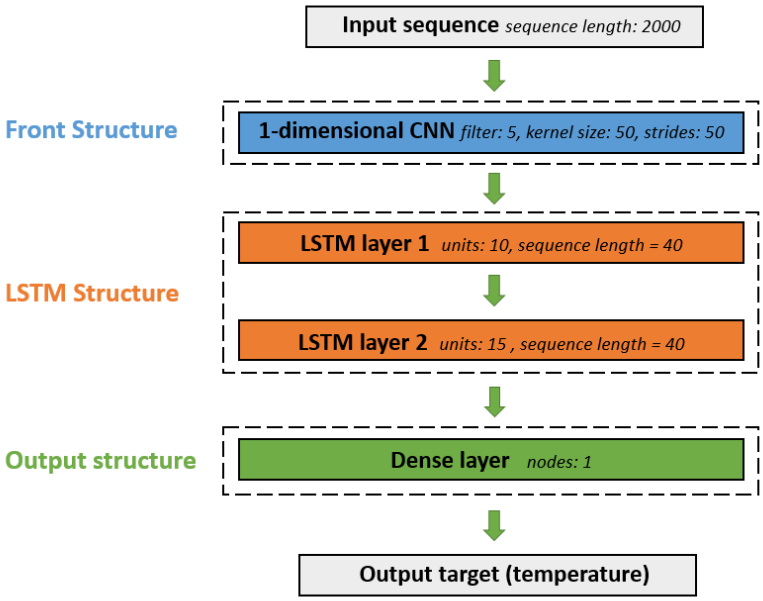


Figure 3.32: The architecture of the final model.

CHAPTER 4

Tests and analyses using the holdout dataset

The principal objective of this thesis was achieved in Chapter 3, where an approach is developed and validated on half of the ABB dataset. This chapter will use the remaining half dataset, i.e., the holdout dataset, to further test our developed approach.

For convenience, in this chapter, we describe the MSEs shown in the summary of Chapter 3, i.e., Section 3.7, as **the validation MSEs** or the **MSEs of Chapter 3**. The MSEs are calculated by employing a 5-fold cross-validation within the entire validation dataset.

The validation and holdout datasets are constructed with similar distributions and characteristics. The test performance produced by the holdout dataset is therefore expected to be similar to the one produced by the validation set, namely the validation MSEs. The comparison is shown in Table 4.1.

Model	MSE of the validation set	MSE of the holdout set
The developed mode	19.94	21.27
Baseline model	76.53	78.77

Table 4.1: MSEs of the developed model and the baseline model on the validation dataset and the holdout dataset.

Testing the optimal approach on the holdout dataset gives a MSE of 21.27. Compared to the validation MSEs, the holdout set delivers a similar performance. However, in practice, there still exist other meaningful tests as stated in Section 3.2.3. This chapter will focus on these tests and describe the associated analysis and results.

4.1 Tests using recordings of one motor

The test error reported previously in Table 4.1, i.e., MSE 21.27, is calculated by all the recordings in the holdout set. In this section, we focus on the recordings

of one motor instead of all motors, and test our optimal approach only with the recordings from each single motor.

4.1.1 Tests using one-month recordings of a motor

The given ABB dataset was originally divided into multiple files, where each file contains the recordings of one motor during one calendar month. For each motor, 16 files were provided by ABB, and eight were randomly assigned into the holdout dataset. To start with, we focus on a single file, i.e., the recordings of a motor during one calendar month.

For each single file, a 5-fold cross-validation is individually carried out, and the corresponding MSE is produced. The model performance can then be evaluated by the overall MSE, namely the average of the MSEs of all files, as listed in Table 4.2.

Model	Overall (averaged) MSE
The developed mode	11.47
Baseline model	14.36

Table 4.2: The overall MSEs of the developed model and the baseline model. For each file, a MSE is given by 5-fold cross-validation. The average of all MSEs is then used as the overall MSE, which measures the overall model performance.

The overall MSE of our model is given as 11.47, while the corresponding MSE of the baseline model is 14.36. Note that the dataset used for producing the validation MSEs is the entire validation set, the one used for Table 4.1 is the entire holdout set, whereas the dataset used for Table 4.2 is only one-month recordings of each motor. Comparing these MSEs, two intriguing findings can be discovered:

- The overall MSE of the developed model in Table 4.2 is smaller than the validation MSEs or the MSE in Table 4.1. The baseline model also performs better in Table 4.2 than in Table 4.1.
- Although the performance of our approach still surpasses that of the baseline model (linear model), their difference becomes much smaller than that shown by the validation MSEs. In comparison with the baseline, the improvement of our model becomes now 20.13%, while it is around 69.54% in Chapter 3.

According to our earlier studies, such findings can be attributed to:

- Hypothesis 1: The recordings within one calendar month of the same motor can be similar. During the 5-fold cross-validation, similar data were assigned to both the training sets and the test set. Therefore, the corresponding MSEs can be much smaller than those in Chapter 3 and Table 4.1.

- Hypothesis 2: The shortcomings of the baseline model, compared to our optimal model, are partially concealed by restricting the training and test data into the same month and the same motor. In such cases, the lack of generalization of the baseline model can be less exposed.

In order to further discuss and try to verify our hypotheses, we retrofit the previous tests and increase the size of test data. One can notice that in the holdout dataset, in addition to the selected one-month recordings, there still exist recordings for seven months of the same motor. We can then test the models, which are trained by the one-month recordings, on the remaining seven-month recordings. For a motor, one month's recordings corresponds to one MSE. Then, the average of the MSEs of all recordings of all motors in the holdout dataset can be referred to as the overall MSE, which measures the overall performance of a model, as shown in Table 4.3.

Model	Overall (averaged) MSE
The developed mode	43.21
Baseline model	78.28

Table 4.3: MSE of the developed model and the baseline model. The MSEs are provided by the same models as Table 4.2. It means that the models are trained by one-month recordings, but are tested with the remaining seven-month recordings of the same motor in the holdout dataset.

As shown, the overall MSE of our model is 43.21, while the MSE of the baseline is 78.28. It is indicated that the MSEs of both models are worse than those shown in Table 4.2. It clearly indicates that the previous MSEs are overoptimistic and also implies the lack of generalization for the corresponding models in Table 4.2, which to some extent confirms our Hypothesis 1.

It is also revealed that comparing with the baseline model in Table 4.3, the improvement of our model is 41.83%, which is much larger than 20.13% calculated from Table 4.2. It implies that the baseline model is more damaged by the increased size of test data, which means that its generalization with respect to time is worse than that of our optimal model. This can be regarded as an reinforcement for Hypothesis 2.

Only producing MSEs does not provide us much opportunity for further analysis. We hence also plot the predicted and the actual recorded temperatures. Such plots can lead to a more thorough understanding of the patterns, performance, and shortcomings of our approach, which could enlighten the discussion of the model and possible future work.

For each motor, eight-month temperatures are predicted. However, it is impractical to plot all predictions due to the following reasons:

- For a motor, its 8-month recordings in the holdout dataset are not continuous, as they were randomly assigned into it.

- The number of 8-month recordings reaches 15 million. It is not feasible to plot all of them. Imagine that Figure 4.1 is now zoomed out and becomes eight times narrower, then the existing patterns in this plot, for instance, the decreasing curve around 1.75 million seconds, will become invisible.

Therefore, it is favorable to illustrate one-month predictions in one plot. However, as there are around 50 motors, it is impractical to illustrate all plots. Hence, only one representative plot is selected, which corresponds to Table 4.2, and is shown in Figure 4.1. The other plot from the same motor during the same month, which corresponds to Table 4.3, is shown in Figure 4.2. Note again that the MSEs of these plots are different from the corresponding overall MSEs in Tables 4.2 and 4.3. However, as their difference is negligible, one could assume that these plots are also representative of the overall MSEs.

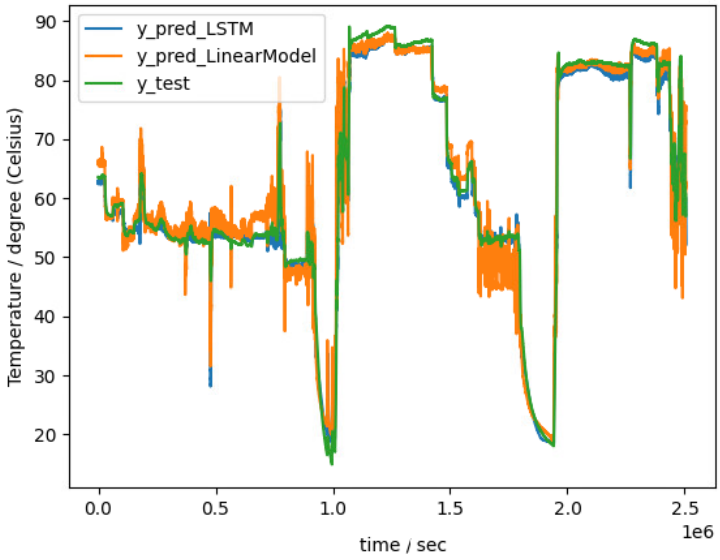


Figure 4.1: The predicted and recorded temperatures during one calendar month. The predicted temperatures were provided by the models trained with the recordings of a motor during the same month. The blue line illustrates the results predicted by our developed model, while the orange line represents the predicted temperatures given by the baseline model, i.e., the linear model. The green line is the actual recorded temperatures. The MSE of our model in this plot is 10.35, while the MSE of the baseline mode is 13.24. All MSEs are given by 5-fold cross-validation.

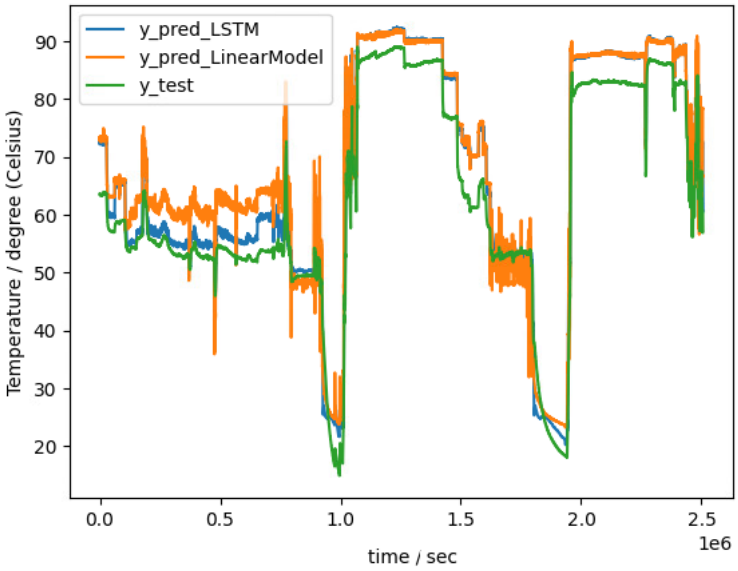


Figure 4.2: The predicted and recorded temperatures during one calendar month. The predicted temperatures were provided by the models trained with another one-month recordings of the same motor. The blue line illustrates the results predicted by our developed model, while the orange line represents the predicted temperatures given by the baseline model, i.e., the linear model. The green line is the actual recorded temperatures. The MSE of our model in this plot is 41.19, while the MSE of the baseline mode is 74.53. All MSEs are given by 5-fold cross-validation.

4.1.2 Tests using all 8-month recordings of each motor

Section 4.1.1 explored the performance of our optimal model and the baseline model by only using one-month recordings of each motor, and several interesting findings have been observed. In what follows, we will extend the evaluation by testing our optimal model under a few selected practical scenarios.

Regarding eight-month recordings of a motor, the scenario described as below is of practical importance:

- At present, only one motor is working and has been working for a long period such that there are enough data from this motor to train a model.
- The goal is to train a model which predicts the temperature of this motor.

In this situation, the 5-fold cross-validation can be directly applied to test the model performance. For each motor, an MSE can be calculated, and the overall model performance can afterwards be quantified by the average of the MSEs of all motors, as tabulated in Table 4.4.

Model	Overall MSE
The developed mode	33.27
Baseline model	74.11

Table 4.4: MSE of the developed model and the baseline model. For each motor, eight-month recordings are employed for a 5-fold cross-validation, and an MSE is provided. The overall MSE is calculated by taking the average of these MSEs.

As shown, the overall MSE of our optimal model is 33.27, while the corresponding MSE of the baseline model is 74.11. One can notice that the MSE of our model is still larger than the validation MSEs. It implies that six months of recordings are still not enough to train the models.

Following the same discussion in Section 4.1.1, the predicted and recorded temperatures are also plotted alongside the MSE values, which are shown in Figure 4.3. To make consistent comparisons between these figures, Figure 4.3 plots the temperatures of the same motor during the same month as Figures 4.1 and 4.2.

Comparing Figure 4.3 with Figures 4.1 and 4.2, one can notice that our model performs better in Figure 4.3 than in 4.1 and 4.2, whereas the baseline model performs worse in 4.3. However, it is also found that small oscillations of the predicted temperatures appear more frequently in Figure 4.3, for instance, during the period from around 1 to 1.4 (million seconds).

As described, the model used for Figure 4.3 is fed more data from the same motor than that for Figure 4.1 or 4.2. Our finding hence implies that, because all these increased data belong to the same motor, the trained models can suffer from the lack of generalization. It also raises the concern about overfitting.

4.2 Tests using recordings of one marine vessel

In this section, we consider one marine vessel, rather than one motor as before. Here, two types of tests are carried out, which correspond to two practical scenarios.

4.2.1 5-fold cross-validation

It is known that each ship has three motors. Here, we focus on one specific ship, where our optimal model is planned to be implemented onboard. Consider a practical scenario that:

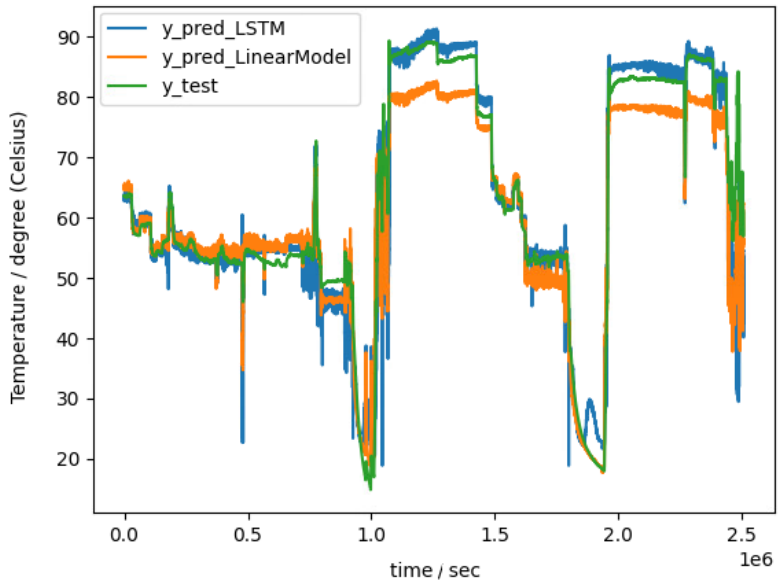


Figure 4.3: The predicted and actual recorded temperatures during one calendar month. Note that this figure illustrates the temperatures during the same month as Figures 4.1 and 4.2. The predicted temperatures were provided by the models trained with the recordings of a motor during around 6.4 months. The blue line illustrates the results predicted by our optimal approach, while the orange line represents the predicted temperatures given by the baseline model, i.e., a linear model. The green line illustrates the actual recorded temperatures. The MSE of our optimal approach in this plot is 31.45, while the MSE of the baseline mode is 74.11. All MSEs are given by 5-fold cross-validation.

- Only one ship is currently navigating. All three motors of this vessel have already been working and recorded for a substantial period, such that they have produced enough historical recordings.
- The goal is to train a generic model which can be applied to any motor on this vessel.

The tests are carried out by using 5-fold cross-validation. We obtain one MSE value for each vessel by using the 5-fold cross-validation, and the average of all MSEs of a model is used to evaluate its performance. For convenience and consistency, the average is also referred to as the overall MSE, which is the same as Section 4.1.1 and Section 4.1.2. These MSEs are shown in Table 4.5.

It is shown that the overall MSE of our model is given as 21.58, while the MSE

Model	Overall MSE
The developed mode	21.58
Baseline model	74.35

Table 4.5: MSE of the developed model and the baseline model. The overall MSEs are delivered by taking the average of MSEs given by 5-fold cross-validation with the recordings of each vessel.

of the baseline model is 74.35. Different from Section 4.1.2, the training data from the other motors are also employed in this section, and our optimal model performs significantly better than in Section 4.1.2.

However, after extending the size of data from one-motor recordings to one-vessel recordings, the generalization of our model could be challenged by a wider data diversity. Therefore, the model performance in this section is most likely worse than that in Section 4.1.2, which is however not the case according to the results shown in Table 4.5.

It is of interest to seek the explanation for this inconsistency. Notice that the motors within one marine vessel can often be homogeneous. Furthermore, since they all experience the same sea state and movement of the ship, the one-vessel recordings can be regarded as an expanded collection of its one-motor recordings. After involving more training data, it is not unreasonable that the model performance is improved.

Analogously, to further verify the explanations and conclusions discussed above. The predicted and actual recorded temperatures for the same month as Figures 4.1-4.3 are shown in Figure 4.4. As shown, for our optimal model, there are much less oscillations of the predicted temperatures. It indeed implies that the generalization of our model is improved. It is therefore reasonable to conclude that the model performance, the theories, and our explanations are self-consistent.

4.2.2 Leave-one-motor-out cross-validation

This section considers another practical situation other than that in Section 4.2.1. For a marine vessel, we assume that:

- One of its motors is newly replaced, and there is no historical recording of it. Meanwhile, the other two motors have been working for a long time and have produced enough historical recordings for training.
- The goal is to train a model which can be applied to the newly replaced motor.

In order to test the model performance under this scenario, a new type of test different from the 5-fold cross-validation is designed. During the test, the recordings of two motors in a vessel are used for training, while the recordings

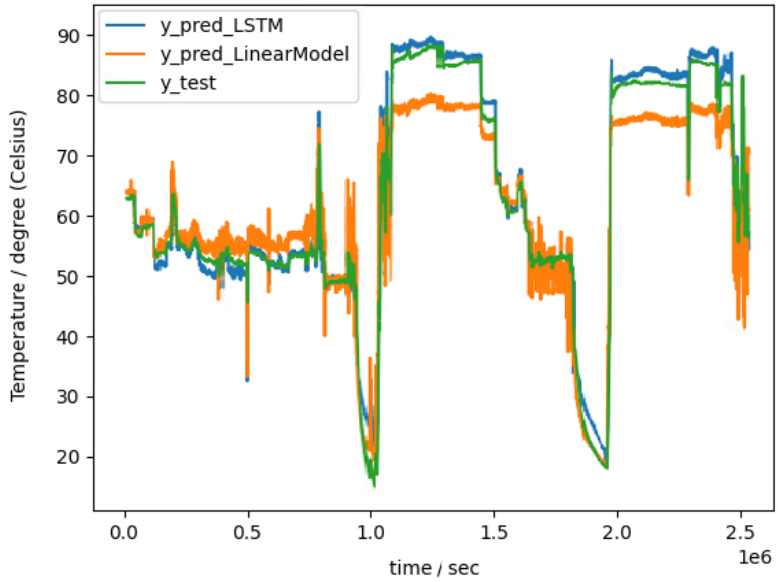


Figure 4.4: The predicted and actual temperatures of a motor during one calendar month. Note that this figure illustrates the temperatures during the same month as Figures 4.1-4.3. The predicted temperatures were provided by the models trained with the recordings of a marine vessel during around 6.5 months. The blue line illustrates the results predicted by our optimal approach, while the orange line represents the predicted temperatures given by the baseline model, i.e., a linear model. The green line illustrates the actual recorded temperatures. The MSE of our LSTM model in this plot is 21.33, while the MSE of the baseline mode is 73.69.

of the third motor are employed for testing. For convenience, we define the motors which provide training data as the training motors, while the motors which deliver the test data as the test motors.

For each marine vessel, three different tests can be carried out, which correspond to three different test motors. The averaged values of their MSEs are referred to as the overall MSE. Inspired by the leave-one-out cross-validation (LOOCV), we define such tests as leave-one-motor-out cross-validation (LOMOCV). The overall MSE is then regarded as the MSE of LOMOCV.

For each vessel, an LOMOCV is carried out to calculate the MSE value. The average of all LOMOCV MSEs is then defined as the overall MSE and becomes the measurement of the model performance. The overall MSEs of our model and the baseline model are shown in Table 4.6.

Model	Overall MSE
The developed mode	22.37
Baseline model	75.91

Table 4.6: The overall MSEs of the developed model and the baseline model. The LOMOCV is carried out for each ship and produces a MSE. The average of the MSEs of all ships is then taken as an overall MSE, which meansures the overall model performance. All the recordings of each vessel are employed.

As shown, the overall MSE of our optimal model is 22.37, while the baseline model delivers an MSE of 75.91. Compared with the corresponding MSEs in Section 4.2.2, one can notice that, for both our model and the baseline model, the difference of MSEs between using LOMOCV and 5-fold CV is insignificant.

Theoretically, the predicting for LOMOCV is supposed to be more challenging than that for 5-fold cross-validation, as there exists a motor which our model has no knowledge of. However, the results shown in Table 4.6 differ from this anticipation. It indicates that, for the motors of one vessel, their similarities are larger than expected. Therefore, unlike the previous sections, no redundant figures like Figures 4.1-4.4 are plotted, as it will almost be the same as Figure 4.4.

4.3 Tests using the whole holdout dataset

In this section, we carry out tests by employing the whole holdout set. Following the similar idea described in Section 4.2, two more experimental scenarios are simulated to test our model. Correspondingly, two testing methods are employed, namely the 5-fold cross-validation and a leave-one-vessel-out cross-validation.

4.3.1 5-fold cross-validation

Consider the most general scenario which assumes that:

- All motors in the 16 vessels have already been working for a long time, such that there are enough historical recordings for training.
- The goal is to train a generic model which can predict the temperature of any motor.

As stated in Section 3.2.3, the present model is optimized based on the same scenario above. Therefore, following the same approach as in Chapter 3, a 5-fold cross-validation can be employed to evaluate the model performance for such cases.

Furthermore, as described in Section 3.1, the validation set used in Chapter 3 and the holdout set used in this chapter have analogous characteristics, distributions, and sizes. Therefore, the MSE obtained in this section is expected to be similar as the validation MSEs found in Chapter 3, which is the case in Table 4.7. As shown, the test MSEs in this section show little difference from

the validation MSEs in Chapter 3.

Model	MSE in this section	MSE in Chapter 3
The developed mode	22.62	23.31
Baseline model (linear model)	75.91	76.53
Baseline model (neural net)	71.53	74.32

Table 4.7: MSEs of the developed model and the baseline model in this section and in Chapter 3. All MSEs are provided by 5-fold cross-validation. The MSEs in this section and in Chapter 3 are delivered by using the holdout set and the validation set, respectively.

Furthermore, after extending the size of dataset from the recordings of one vessel to all 16 vessels, it can be of great interest in examining the behavior of our model. Figure 4.5 plots the predicted temperatures in comparison with the actual recorded temperatures for the same month and the same motor studied in Figures 4.1-4.4.

One can notice that in Figure 4.5, the behaviour of our model, from around 1 to 1.4 million seconds and from around 2 to 2.3 (million seconds), is different from those in Figures 4.1-4.4. Here, the predicted values do not vary much with time, showing a smooth horizontal line. Although Figures 4.1-4.5 give similar trend plots without showing large structural changes in this period, some noticeable oscillations can be observed in Figures 4.1-4.4. It implies the validity of our generalized and robust model. For a much larger dataset, instead of fitting the vibrations and systematic errors, our model can capture the important characteristics and deliver an overall better performance.

4.3.2 Leave-one-vessel-out cross-validation

Consider another experimental scenario different from that in Section 4.3.1:

- All three motors in a marine vessel are newly replaced, and there is no historical recording of these motors. Meanwhile, the motors of other vessels have been working for a long period and have produced enough historical recordings for training.
- The goal is to train a model which can be applied to the three new motors in this vessel.

In order to simulate such a scenario, similar to Section 4.2.2, a new type of test is designed. For each test, the recordings of one marine vessel are taken as the test set, while all remaining recordings are the training sets. For convenience, we name the vessel, whose recordings are used as the test set, as the test vessel. The other vessels are defined as training vessels.

Each vessel is used once as the test vessel, and hence delivers one corresponding MSE. After all vessels have been tested, the average of their MSEs is regarded

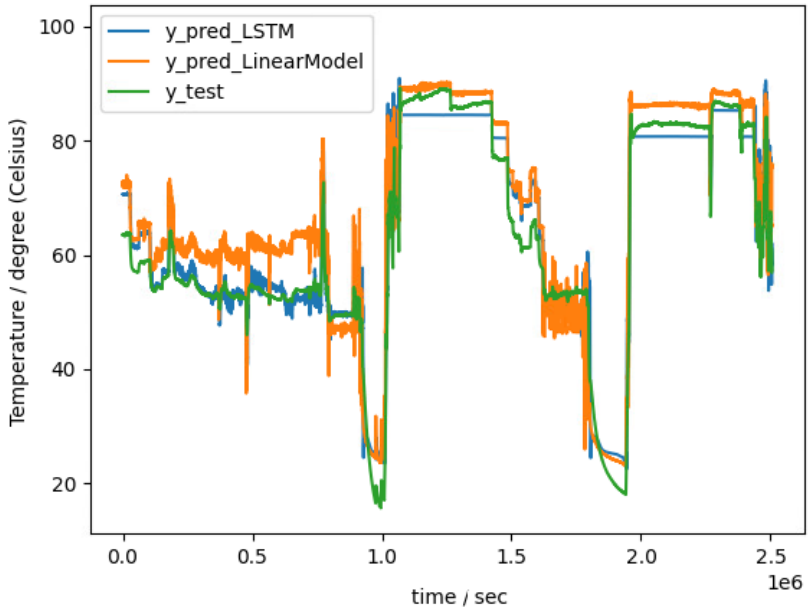


Figure 4.5: The predicted and actual temperatures of a motor during one calendar month. The predicted temperatures were provided by the models trained with six-month recordings of all motors. Note that this figure illustrates the temperatures during the same month as Figures 4.1-4.4. The blue line illustrates the results predicted by our optimal approach, while the orange line represents the predicted temperatures given by the baseline model, i.e., a linear model. The green line illustrates the actual recorded temperatures. The MSE of our LSTM model in this plot is 20.22, while the MSE of the baseline mode is 73.69

as the overall measurement of the model performance, namely an overall MSE. Inspired by the leave-one-out cross-validation, the procedure demonstrated above is named as a leave-one-vessel-out cross-validation (LOVOCV). The overall MSE can then be also referred to as the MSE of LOVOCV.

Following the similar arguments in Section 4.2.2, the LOVOCV can therefore be more challenging than the 5-fold cross-validation, as the trained model has no knowledge of the testing vessels. Therefore, it is expected that the MSE of LOVOCV can be larger than that of the 5-fold cross-validation. Observed interestingly, the difference between their MSEs is however insignificant, as shown in Table 4.8. The result implies that the homogeneity between the given ships is stronger than expected.

Model	MSE LOVOCV	MSE 5-fold CV
The developed mode	21.57	22.62
Baseline model	73.97	75.41

Table 4.8: MSE of the developed model and the baseline model for different testing methods. Here, the whole holdout set is employed.

However, it is worth addressing that these 16 ships do not have to be all homogeneous. When predicting one target ship, according to the LOVOCV, 15-ship recordings are employed to train the model. Therefore, among these 15 ships, even if a few are similar to the target ship, we would expect a good performance from the current model.

4.4 Summary

In this chapter, five relevant scenarios are simulated and tested, to further illustrate the generality and accuracy of the current model, starting from Section 4.1.2. However, notice that two tests are carried out in Section 4.1.1 and their corresponding scenarios were not described before. Since these tests indeed have practical meanings, their corresponding test scenarios are also summarized here to keep the consistency of all tests in this chapter. The tests which produce Table 4.2 and Table 4.3 in Section 4.1.1 correspond to Scenario 1 and Scenario 2, respectively. The seven scenarios (tests) considered in this chapter are then listed below:

- **Scenario 1 Short term prediction**: One motor is newly replaced, and the goal is to predict its temperature in the near future. There are only a limited number of historical recordings of the new motor, and there are no historical recordings of any similar motors which can be taken as a reference.
- **Scenario 2 Long term prediction**: One motor is newly replaced, and the goal is to predict its temperature in long-term future. There are only a limited number of historical recordings of it. Furthermore, there are no historical recordings of any similar motors which can be taken as a reference.
- **Scenario 3 Single motor prediction**: Only one motor is currently working, and the goal is to predict its temperature. This motor has been working for a long time, and there are enough historical recordings of it.
- **Scenario 4 Single ship prediction**: Only one ship is currently navigating, and the goal is to predict the temperature of all its three motors. These motors all have been working for a long time, and there are enough historical recordings of them.
- **Scenario 5 Leave-one-motor-out prediction**: Only one ship is currently working, and one motor is newly replaced. The goal is to predict the temperature of the new motor. There is no historical recording

of this motor, however, the other two motors of this ship have been working for a long time, and there are enough historical recordings of them.

- **Scenario 6 All motors and ships prediction**: All 48 motors in the 16 ships have been working for a long time, and there are enough historical recordings of them. The goal is to predict the temperature of all these motors.
- **Scenario 7 Leave-one-ship-out prediction**: All three motors of a ship are replaced. The goal is to predict the temperature of these three new motors. There are no historical recordings of these motors. However, there are enough recordings of 45 similar motors from 15 similar ships.

In order to address these practical situations, the corresponding tests are designed and carried out. The MSEs of these tests are employed to evaluate the model performance, which are summarized in Table 4.9. The anomalies of the MSEs of Scenarios 1 and 2, compared to other scenarios, have been thoroughly discussed in Section 4.1.1.

Scenario	MSE (the optimal model)	MSE (baseline model)
1	11.47	14.36
2	43.21	78.28
3	33.27	75.66
4	21.58	74.35
5	22.37	75.91
6	22.62	75.41
7	21.57	73.97

Table 4.9: MSEs of tests for different practical scenarios. The descriptions of these methods are summarized above.

These tests also bring a considerable number of intriguing findings, and the corresponding explanations are thoroughly discussed. The MSEs, figures, theories, and our explanations are all proven to be self-consistent. In short, the behavior of the present optimal approach was able to give accurate predictions of motor temperature under a variety of practically meaningful situations. It shows a promising aspect of generalization with respect to handling scenarios beyond the scope of this work. The outcomes of this chapter have further suggested the validity of the model derived in Chapter 3 and its broad extrapolation properties.

CHAPTER 5

Conclusion, discussion and future work

5.1 Conclusion

The studies in this thesis have investigated how machine learning techniques can be used to predict the temperatures of motors on a ship. The final objective of the current study is to develop an adaptive data-driven monitoring approach, which helps detect the possible overheating of motors.

Here, machine learning methods such as convolution neural network (CNN) and long short-time memory (LSTM) were reviewed and discussed. Along with this, the current work also developed several efficient algorithms for handling the massive amount of data made available by ABB.

In the validation of the model, two types of data have been used. One is the simulated data from a simulator which is based on the simplified thermal model of a wire. The other is the validation ABB dataset, which contains more than 1 billion recordings.

As stated in Section 4.4, multiple practical scenarios were investigated to test our developed model. The model has performed consistently well for these scenarios and has shown a satisfactory overall performance with respect to the prediction accuracy and speed. According to the validation and test results, the performance of our approach is much better than that of the baseline linear model regarding both the accuracy of predictions, namely the MSE, and the generalization. It indicates that the historical status of a motor can indeed help predict the current motor temperature. Furthermore, it also suggests the success of the architecture of our developed model, compared to the baseline model.

5.2 Discussion and future work

Despite the success of our approach, there still exist other possible approaches and methods, which can be helpful for the task given to this thesis. It is hence worth introducing them to prepare for future work or applications.

5.2.1 Re-balance the temperature distribution

One currently unexplored aspect is that the distribution of the target (temperatures) in the given dataset is unbalanced. In other words, if dividing the range of temperatures into intervals of equal size, the number of recordings in certain intervals can be significantly different from the rest.

It may be possible that our approach is influenced by this type of unbalance. Dividing the temperatures into small intervals (i.e., $10°C$), we can then illustrate the number of recordings in each interval by a histogram as shown in Figure 5.1. Meanwhile, the corresponding MSEs are also shown in this figure. It is indicated by the figure that the MSE varies with temperatures, and it seems that the unbalance can to some extent affect the model performance.

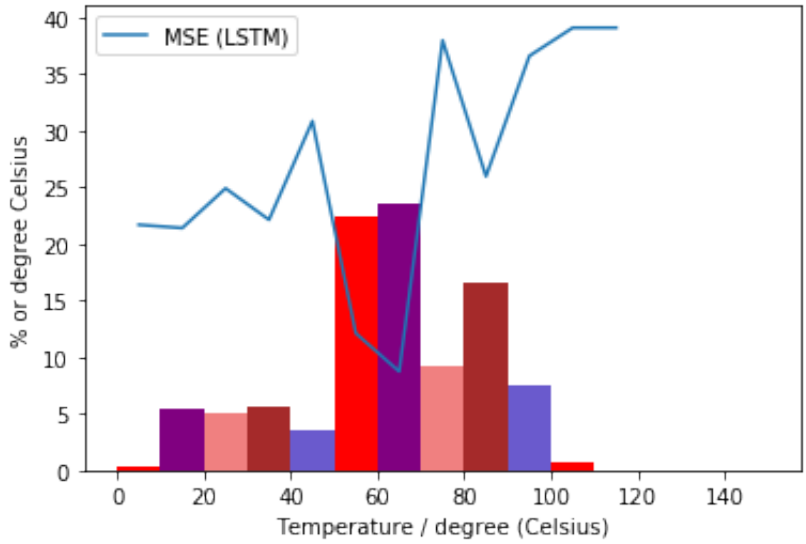


Figure 5.1: An illustration of the temperature distribution and the corresponding MSEs. The x-axis represents the actual temperatures given by ABB. These temperatures are divided into multiple intervals, and each interval consists of 10 degrees Celsius. The histogram illustrates the percentage of number of recordings in each interval, with respect to the number of all recordings. Note that there also exist temperatures between 110 and 120, but it is too small to be visible in this figure. The blue line illustrates the MSE of each temperature interval, which is calculated through 5-fold cross-validation. The entire holdout dataset and the developed model are used to produce these MSEs. The overall MSE, i.e., a weighted average of these MSEs, is given as 20.46.

However, the difference in MSEs may also be a result of other factors. For example, since different temperatures essentially represent different working modes of the motor. The lowest temperature interval was given when the motor was turned off or at idle, whereas the highest temperature interval represents

that the motor was engaged at its full load. Despite the unbalance, the model performance can also be influenced by these different working modes. For instance, if our model does not perform well for an idle or standby motor, then no matter how many recordings are used to train the model, the model still cannot predict well in this temperature interval.

In other words, the influence due to other factors (which are also correlated to the temperature) can be as large as that by the unbalance. Therefore, even if some improvements are made to solve the unbalance, it may be difficult to verify them.

5.2.2 Limitations of this study

In Chapter 4, multiple practical scenarios are simulated and tested. The developed approach of this thesis has shown a satisfactory performance through these tests. However, it is worth noticing that the test data provided by ABB has its own limitations. As demonstrated in Chapter 4, the motors and ships which provided the ABB data seem to be homogeneous. Therefore, although an enormous amount of data were provided, many of them can be similar, such that the generalization of the model is not challenged by the tests.

It is of great interest to investigate the model performance for heterogeneous motors and ships, which can be a useful future work. Nevertheless, the model is not likely to perform equally well for predicting the temperatures of heterogeneous motors as in this thesis. In order to provide a satisfactory and practical approach, the transfer learning introduced in Section 5.2.3 could be a possible avenue.

Besides the limitation of the dataset, this study also has limitations. Recall that predicting motor temperatures is determined as the objective of this thesis only because it helps prevent possible motor overheating. Although such prediction is an essential part of overheating prevention, more work still needs to be done to implement a prevention algorithm.

In order to complete an approach to prevent possible overheating, there are several possible approaches considered as future work. For example, as stated before, an alarm will be triggered once the deviation between the detected temperature and the predicted one reaches a warning limit. For convenience, we define such a deviation as the **prediction deviation**. The determination of the specific warning limit of the prediction deviation is an important and complicated task, since the prediction deviation does not correspond to a physical metric which directly reveals the overheating. Therefore, the relation between overheating and such prediction deviation requires further study and additional fault data, which are not necessarily easily available.

Furthermore, the possible overheating may not only correspond to one temperature anomaly, but instead, a combination of multiple anomalies, namely an anomaly pattern. It indicates that, although the prediction deviation does not reach the warning limit, some specific patterns may still imply a possible overheating. In such cases, a single data-driven machine learning approach does not match the requirements. In order to provide a real-time overheating pattern

detection, the well-known Complex Event Processing (CEP) (Leavitt 2009) can be applied in addition to the machine learning approach. CEP was developed in the early 1990s and was designed to detect complex events from data streams, which is promising for the real-time pattern detection in such cases.

5.2.3 Transfer learning

According to the findings in Chapter 4, the motors and vessels can be considered to have a high similarity. Therefore, the model trained by the recordings of selected motors could be highly informative for other motors. Such a case may not hold if different types of motors would be installed on different vessels. However, it is still possible to make use of the prior knowledge gained from the old motors following the analogy of the underlying physical models. One could therefore possibly retrofit the old model instead of completely abandoning it, and make it work on new motors. This process is the so-called transfer learning. According to the field experience, it has already been proven to be a practical and effective approach (Pan and Yang 2009). However, since the dataset of this thesis was collected from the same type of motors, no effort has been done in this thesis to investigate the effectiveness of the transfer learning.

Bibliography

Bengio, Yoshua, Simard, Patrice and Frasconi, Paolo (1994). 'Learning long-term dependencies with gradient descent is difficult'. In: *IEEE transactions on neural networks* vol. 5, no. 2, pp. 157–166.

Bertsekas, Dimitri P (1996). 'Incremental least squares methods and the extended Kalman filter'. In: *SIAM Journal on Optimization* vol. 6, no. 3, pp. 807–822.

Cantile, Carlo and Youssef, Sameh (2015). 'Nervous system'. In: *Jubb, Kennedy and Palmer's Pathology of Domestic Animals,* vol. 1, pp. 250–406.

Cauchy, Augustin et al. (1847). 'Méthode générale pour la résolution des systemes d'équations simultanées'. In: *Comp. Rend. Sci. Paris* vol. 25, no. 1847, pp. 536–538.

Cho, Kyunghyun et al. (2014). 'Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation'. In: *EMNLP*.

Chollet, François et al. (2015). *Keras*. https://keras.io.

Choromanska, Anna et al. (2015). 'The loss surfaces of multilayer networks'. In: *Artificial intelligence and statistics*. PMLR, pp. 192–204.

Dauphin, Yann et al. (June 2014). 'Identifying and attacking the saddle point problem in high-dimensional non-convex optimization'. In: *NIPS* vol. 27.

Gal, Yarin and Ghahramani, Zoubin (2016). 'A theoretically grounded application of dropout in recurrent neural networks'. In: *Advances in neural information processing systems* vol. 29, pp. 1019–1027.

Hellton, K. H. et al. (2021). 'Predicting propulsion motor overheating using machine learning'.

Hochreiter, Sepp and Schmidhuber, Jürgen (1997). 'Long short-term memory'. In: *Neural computation* vol. 9, no. 8, pp. 1735–1780.

Ioffe, Sergey and Szegedy, Christian (2015). 'Batch normalization: Accelerating deep network training by reducing internal covariate shift'. In: *International conference on machine learning*. PMLR, pp. 448–456.

Keskar, Nitish Shirish et al. (2017). 'On large-batch training for deep learning: Generalization gap and sharp minima'. In: *5th International Conference on Learning Representations, ICLR 2017*.

Leavitt, Neal (May 2009). 'Complex-Event Processing Poised for Growth'. In: *Computer* vol. 42, pp. 17–20.

Li, Xiang et al. (2019). 'Understanding the disharmony between dropout and batch normalization by variance shift'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2682–2690.

Meng, Qi et al. (2019). 'Convergence analysis of distributed stochastic gradient descent with shuffling'. In: *Neurocomputing* vol. 337, pp. 46–57.

Minsky, Marvin and Papert, Seymour A (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

Pan, Sinno Jialin and Yang, Qiang (2009). 'A survey on transfer learning'. In: *IEEE Transactions on knowledge and data engineering* vol. 22, no. 10, pp. 1345–1359.

Pedregosa, F. et al. (2011). 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* vol. 12, pp. 2825–2830.

Rosenblatt, Frank (1958). 'The perceptron: a probabilistic model for information storage and organization in the brain.' In: *Psychological review* vol. 65, no. 6, p. 386.

Rumelhart, David E, Hinton, Geoffrey E and Williams, Ronald J (1986). 'Learning representations by back-propagating errors'. In: *nature* vol. 323, no. 6088, pp. 533–536.

Saxe, AM, McClelland, JL and Ganguli, S (2014). 'Exact solutions to the nonlinear dynamics of learning in deep linear neural networks'. In: International Conference on Learning Represenatations 2014.

Solomon, Justin (2015). *Numerical algorithms: methods for computer vision, machine learning, and graphics*. CRC press.

Srivastava, Nitish et al. (2014). 'Dropout: a simple way to prevent neural networks from overfitting'. In: *The journal of machine learning research* vol. 15, no. 1, pp. 1929–1958.

Tieleman, Tijmen and Hinton, Geoffrey (2012). 'Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude'. In: *COURSERA: Neural networks for machine learning* vol. 4, no. 2, pp. 26–31.

Young, Hugh D, Freedman, Roger A and Ford, A Lewis (2013). *University Physics with Modern Physics Technology Update*. Pearson Education.

Zhang, Hongyang, Shao, Junru and Salakhutdinov, Ruslan (2019). 'Deep neural networks with multi-branch architectures are intrinsically less non-convex'. In: *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 1099–1109.

Zhang, Saizheng et al. (2016). 'Architectural complexity measures of recurrent neural networks'. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 1830–1838.

Appendices

APPENDIX A

Additional figures

As stated in Chapter 4, we wish to illustrate the predicted temperatures and the actual recorded temperatures. However, since the number of such figures in this thesis exceeds 5000, it is impractical to plot all of them. Nevertheless, here in the appendix, we are able to show more of these figures. As a representative scenario, **Scenario 6** in Chapter 4 and its corresponding test are chosen to be illustrated here. Approximately 10% of the plots of the tests in **Scenario 6**, namely 22 plots, are randomly selected. Note that they are produced by the same test as Figure 4.5.

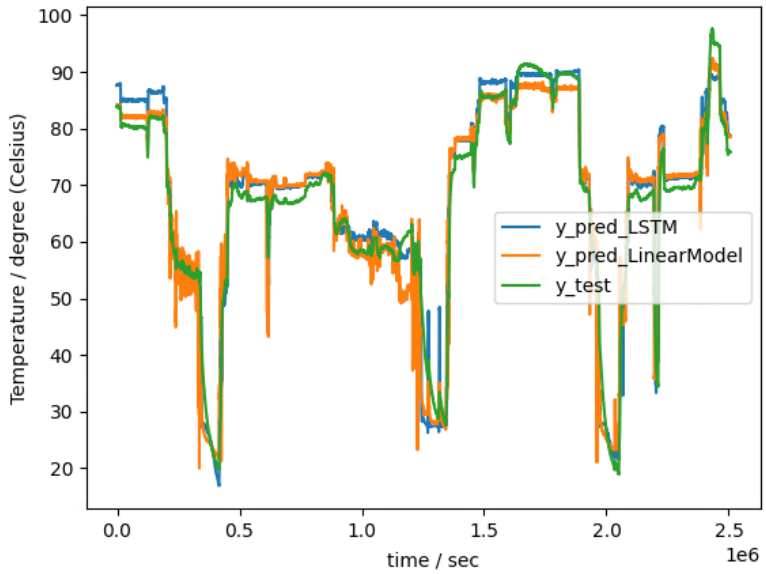


Figure A.1

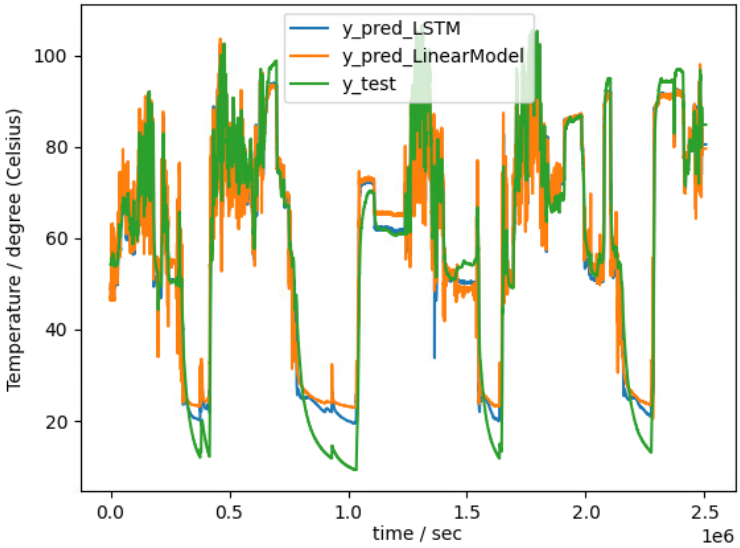


Figure A.2

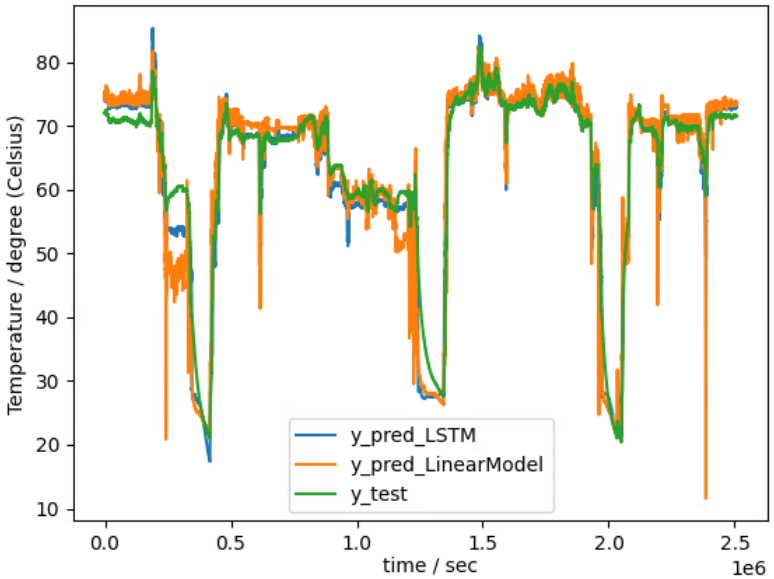


Figure A.3

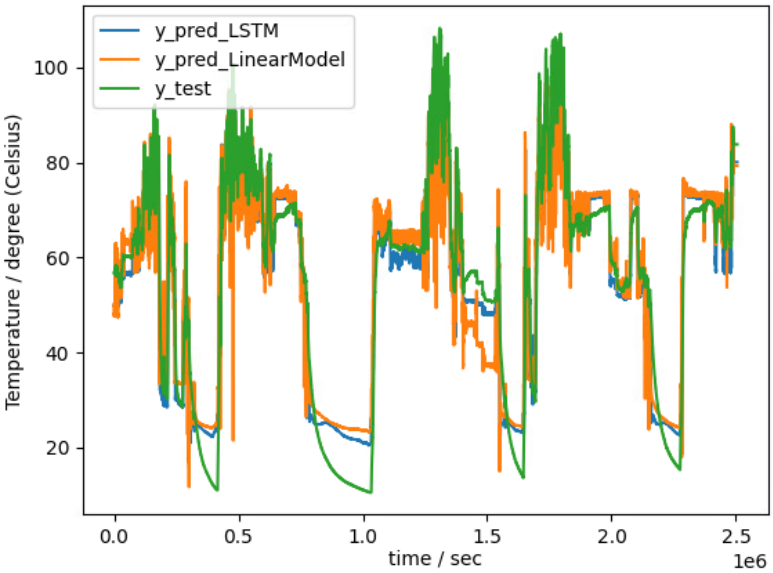


Figure A.4

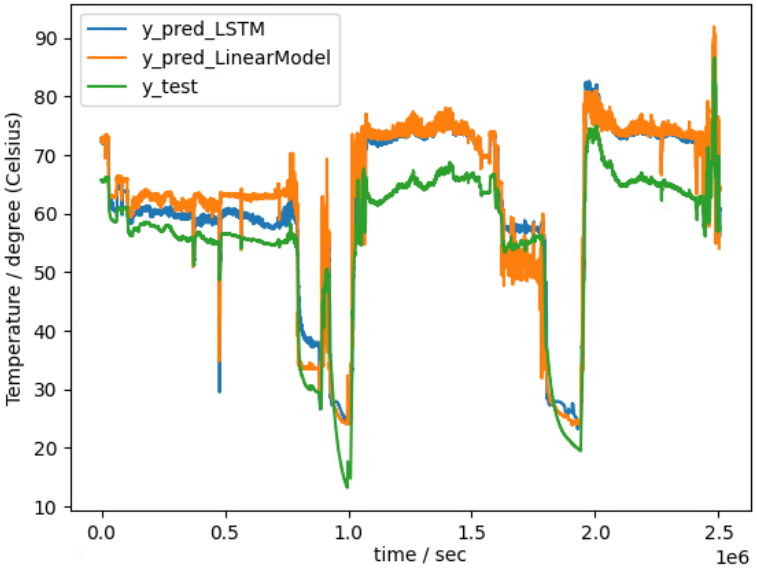


Figure A.5

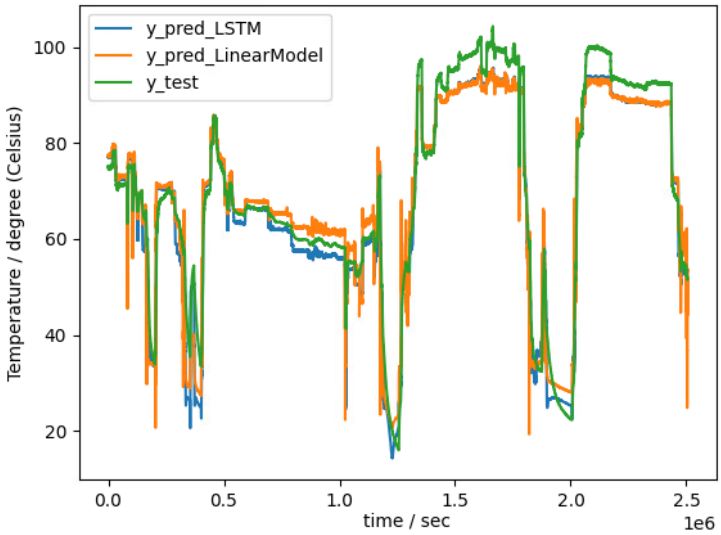


Figure A.6

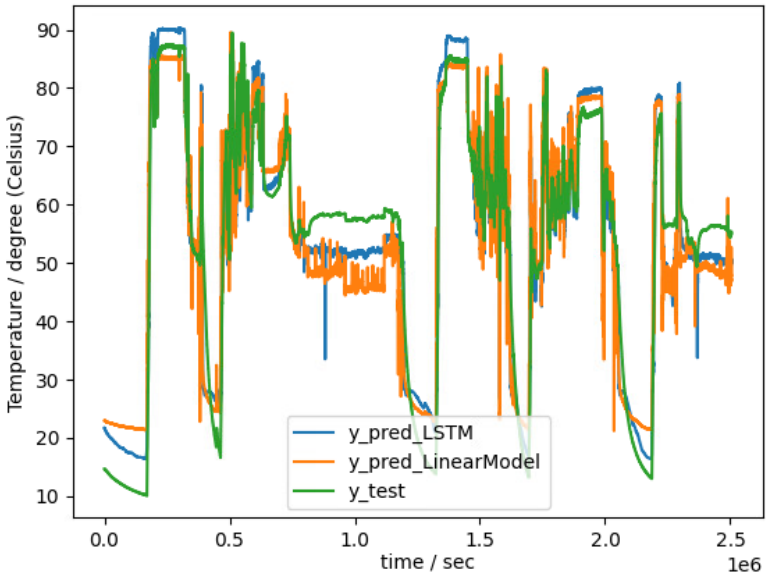


Figure A.7

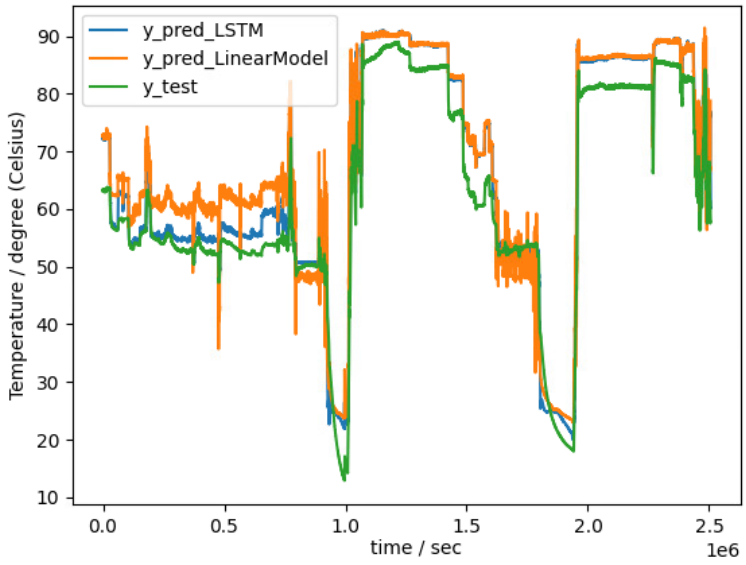


Figure A.8

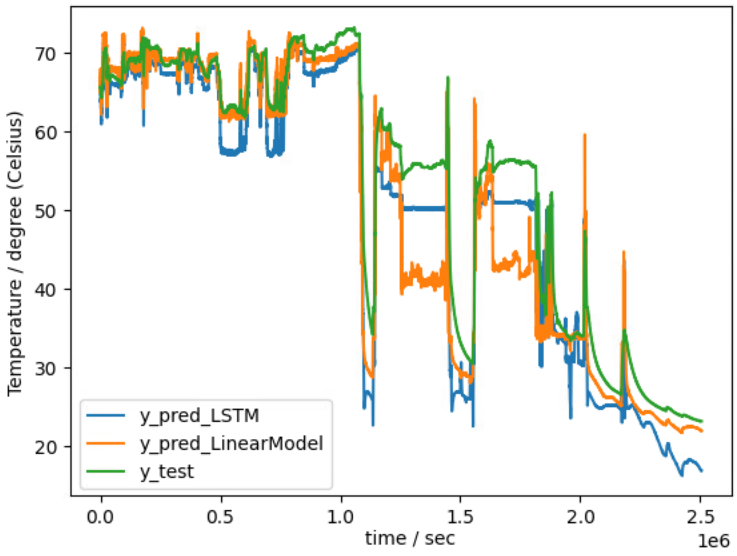


Figure A.9

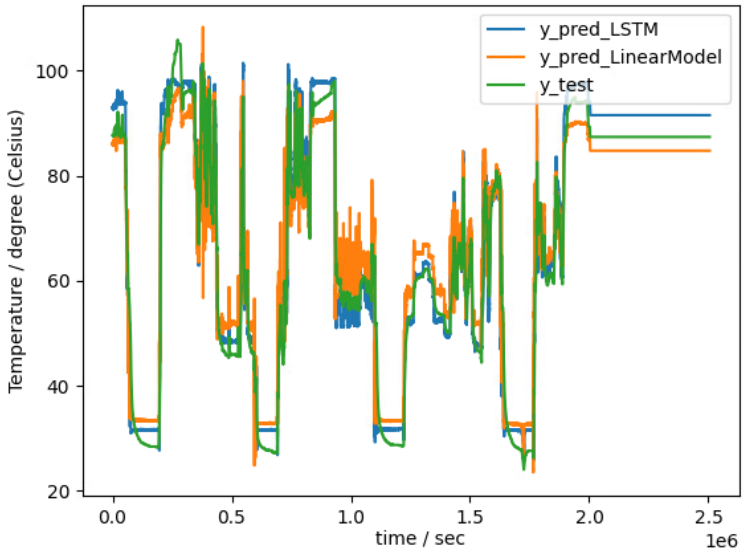


Figure A.10

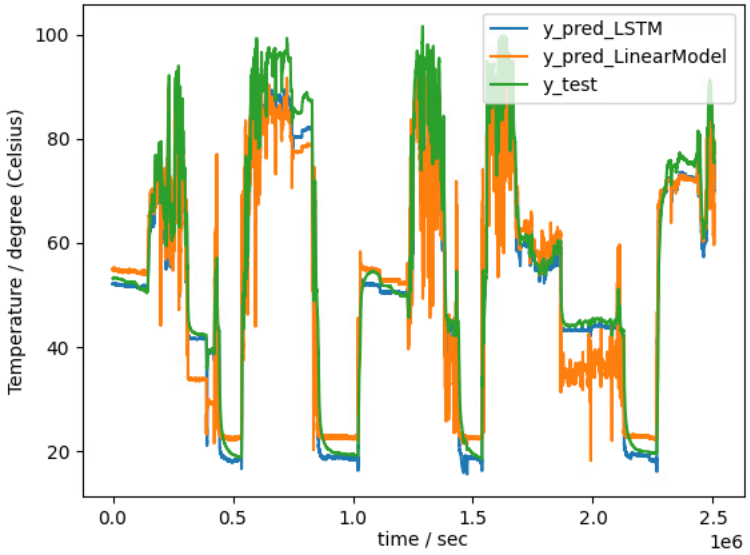


Figure A.11

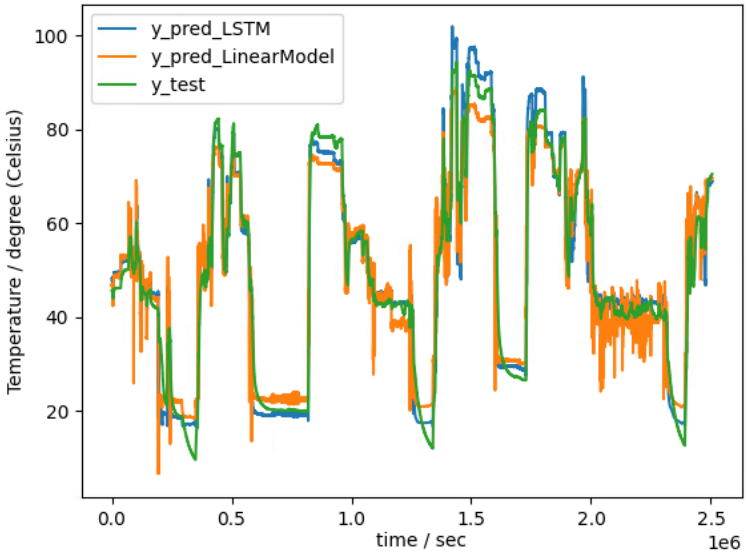


Figure A.12

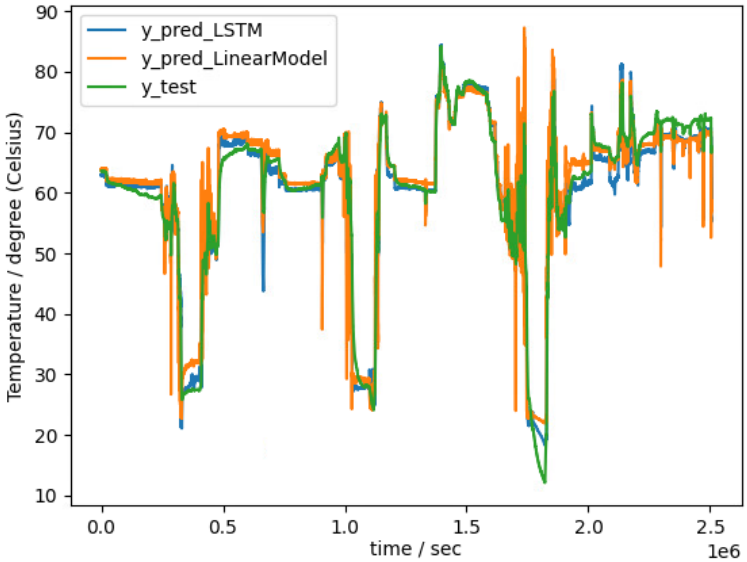


Figure A.13

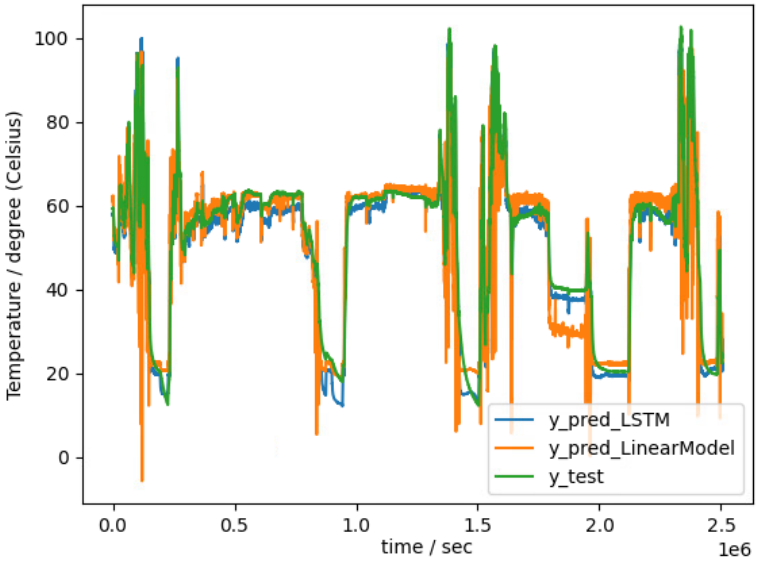


Figure A.14

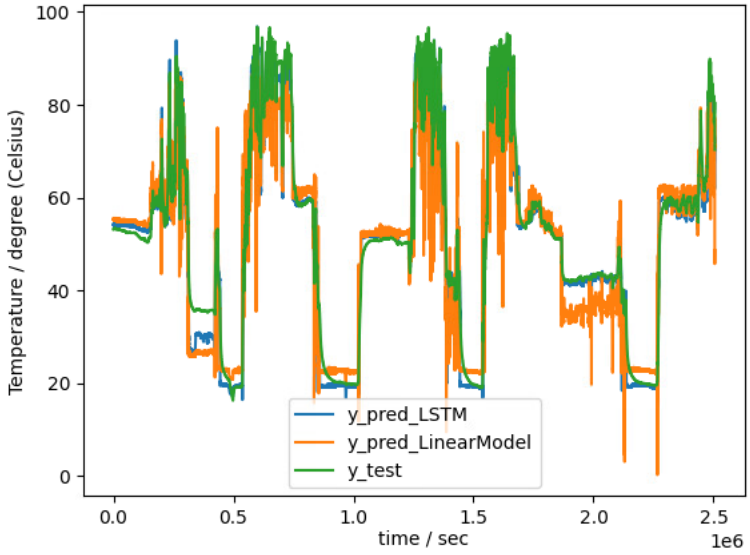


Figure A.15

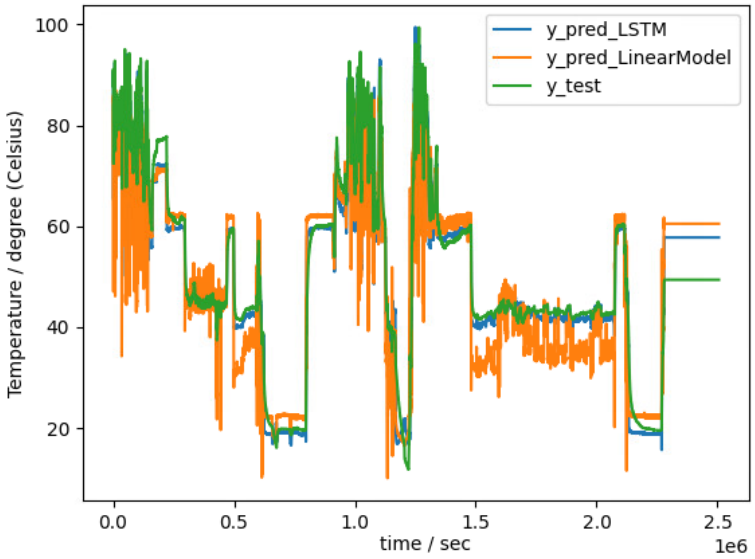


Figure A.16

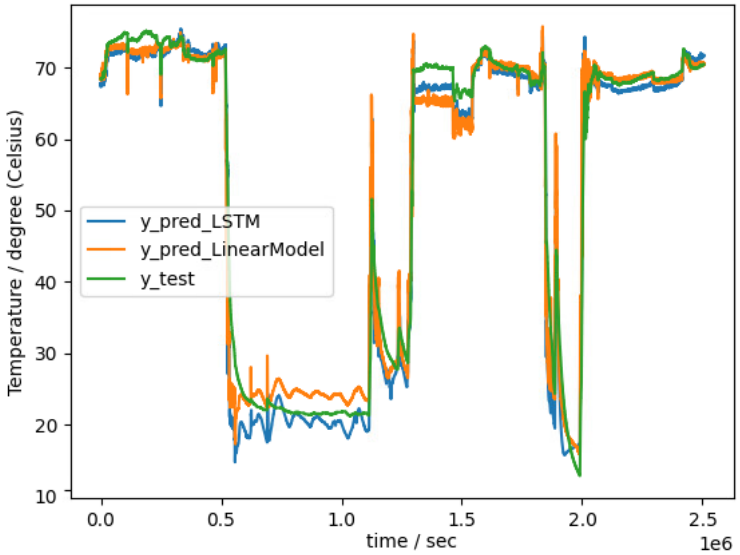


Figure A.17

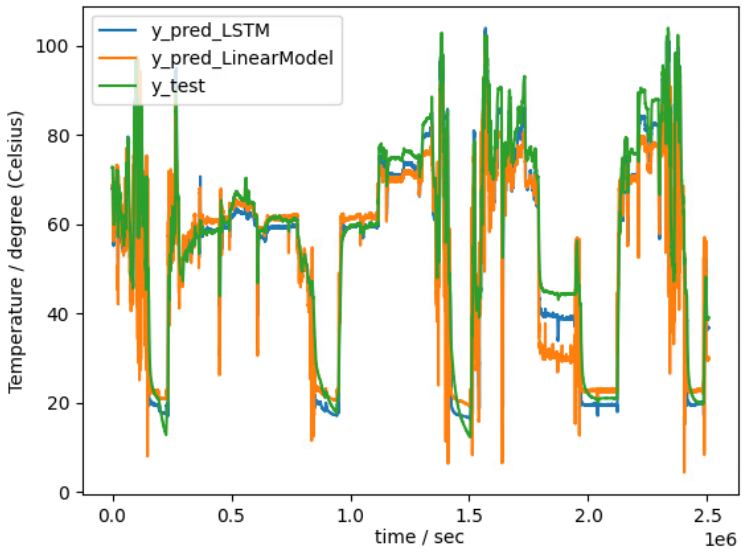


Figure A.18

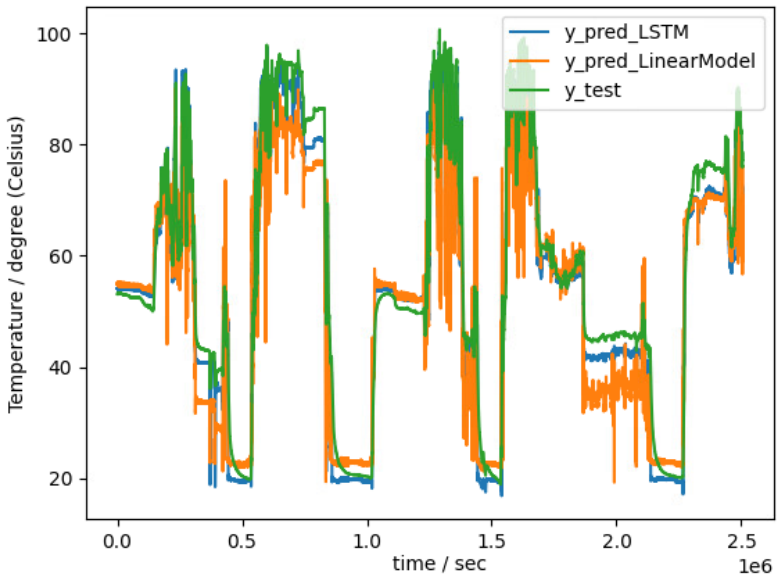


Figure A.19

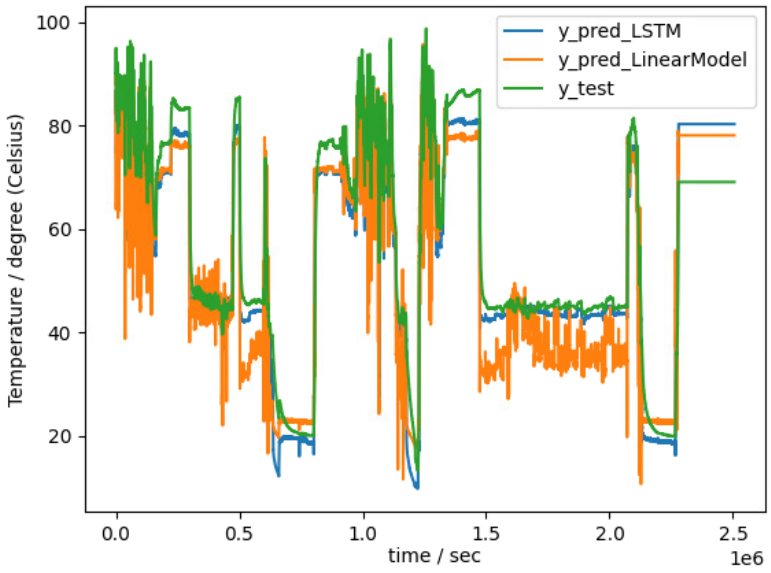


Figure A.20

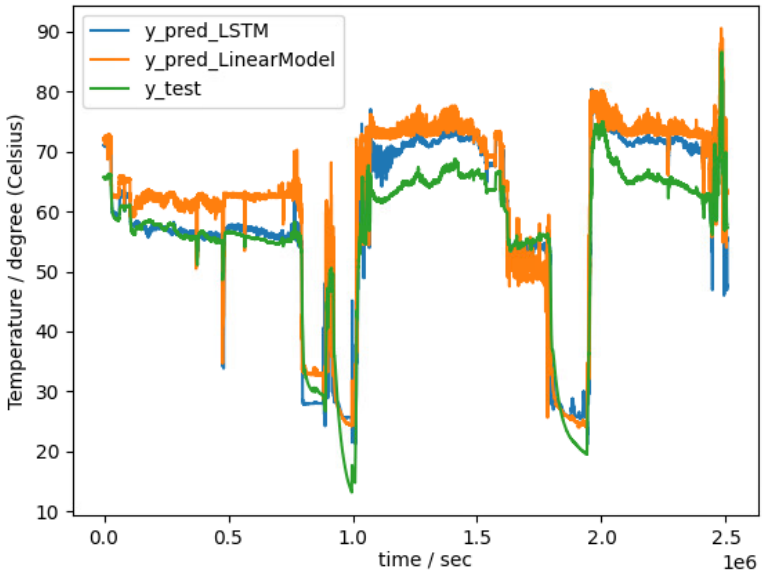


Figure A.21

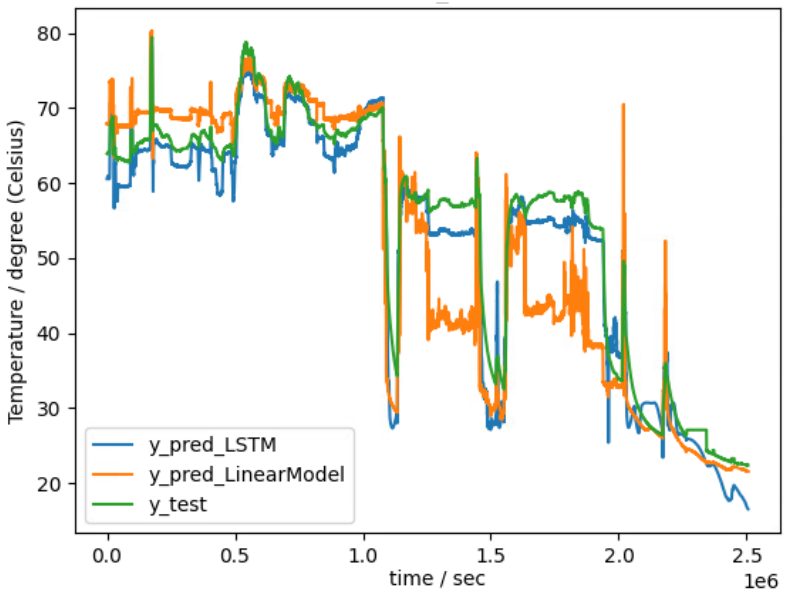

Temperature / degree (Celsius)
80
70
60
50
40
30
20
0.0
0.5
1.0
1.5
2.0
2.5
1e6
time / sec
y_pred_LSTM
y_pred_LinearModel
y_test

Figure A.22

APPENDIX B

Codes

B.1 Core code used for model optimization, validation, and test

The code used for model optimization, validation, and test will be shown in this appendix. The entire code consists of over 3000 lines. However, most of them are redundant and are essentially derived versions of a core source code. We therefore only provide the core code as below. One can easily derive other codes used in this thesis.

```
########################################
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import random
########################################
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import SGDRegressor
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import LayerNormalization, Dense
from keras.layers import LSTM, Conv1D

features = ['SP', 'CI', 'PO', 'TO']
target = ['T']

num_partitions = 700
n_folds = 5
########   1
plot_train = False
plot_pred = False
plot_hist = False

```

```
26 ########   2
27 add_conv = True
28 conv_output_len = 5
29 conv_window_len = 50
30 conv_strides = 50
31 lstm_layers = [10, 15]
32 dense_layers = [1]
33
34 ########   3
35 sequence_length = 2000
36 num_epochs = 200
37 batch_size = 1024
38 valid_split = 0.25
39
40 ########   4
41 baseline = SGDRegressor()
42
43 #################    standardize data    #################
44 MinMax_x = preprocessing.MinMaxScaler()
45 MinMax_y = preprocessing.MinMaxScaler()
46
47 ############################################################
48 overall_observation_counter = []
49 overall_error_lstm = []
50 overall_error_baseline = []
51
52 sorted_list = random.sample([i for i in range(num_partitions)],
       num_partitions)
53 n_test = int(num_partitions/n_folds)
54 #-----------------n fold CV------------------#
55 for fold_counter in range(n_folds):
56     print("Fold No.", fold_counter)
57     error_lstm = []
58     error_baseline = []
59     observation_counter = []
60     ########################       build folds
       ##########################
61     if fold_counter == (n_folds - 1):
62         test_list = sorted_list[fold_counter*n_test:]
63     else:
64         test_list = sorted_list[fold_counter*n_test:(fold_counter+1)*
       n_test]
65
66     print("Build test list : finished ", test_list)
67     ########################       build model
       ##########################
68     #-----------------baseline model------------#
69     model_baseline = baseline
70     #-----------------LSTM model-----------------#
71     model_lstm = Sequential()
72     if add_conv:
73         model_lstm.add(Conv1D(conv_output_len, conv_window_len,
       strides = conv_strides, activation='relu'))
74     for i in range(len(lstm_layers)):
```

```
75          if i == len(lstm_layers) - 1:
76              model_lstm.add(LSTM(lstm_layers[i], return_sequences =
    False))
77          else:
78              model_lstm.add(LSTM(lstm_layers[i], return_sequences =
    True))
79      for i in range(len(dense_layers)):
80          model_lstm.add(Dense(dense_layers[i]))
81      opt = tf.keras.optimizers.RMSprop(learning_rate = 0.001)
82      model_lstm.compile(loss = "mse", optimizer = opt)
83
84      print("build model : finished")
85      ######################      train model
        ##########################
86      for i in range(num_partitions):
87          if i not in test_list:
88              file_name = "/all_data/" + str(i) + ".parquet"
89              print(file_name)
90              data = pd.read_parquet(file_name, engine = 'pyarrow')
91              X = data[features].values
92              y = data[target].values
93              MinMax_x.partial_fit(X)
94              MinMax_y.partial_fit(y)
95      for i in range(num_partitions):
96          counter = 0
97          if i not in test_list:
98              counter = counter + 1
99              file_name = "/all_data/" + str(i) + ".parquet"
100             print(file_name)
101             data = pd.read_parquet(file_name, engine = 'pyarrow')
102             X = data[features].values
103             y = data[target].values
104             X = MinMax_x.transform(X)
105             y = MinMax_y.transform(y)
106             length = len(X)
107             #---------------fit LR---------------#
108             # partial fit
109             model_baseline.partial_fit(X, np.ravel(y))
110             #--------------fit LSTM--------------#
111             # build sequence
112             X_time, y_time = [], []
113             for index in range(0, len(X) - sequence_length):
114                 X_time.append(X[index: index + sequence_length])
115             X_time = np.array(X_time)
116             y_time = np.array(y[(sequence_length-1):(length-1)])
117             print("build train sequence: finished")
118             #fit
119             early_stopping=tf.keras.callbacks.EarlyStopping(monitor='
    val_loss', min_delta = 0.0001, mode='auto', restore_best_weights =
     True)
120             hist = model_lstm.fit(X_time, y_time, epochs = num_epochs,
     batch_size = batch_size, callbacks=[early_stopping],
    validation_split = valid_split)
121
```

```
122             if plot_train:
123                 y_pred_train = model_lstm.predict(X_time)
124                 y_pred_inverse = y_pred_train.reshape(-1, 1)
125                 y_train_inverse = y_time
126                 t = np.linspace(0, len(y_pred_inverse), len(
    y_pred_inverse))
127                 plt.figure()
128                 plt.plot(t[0:1000], y_pred_inverse[0:1000])
129                 plt.plot(t[0:1000], y_train_inverse[0:1000])
130                 plt.legend(['y_pred_lstm', 'y_test'])
131                 plt.savefig("plot_train_" + test_props + str(j))
132 
133             #plot fit history, i.e. val_loss and train loss - number
    of epochs
134             if plot_hist:
135                 plt.figure()
136                 t = range(len(hist.history['loss']))
137                 plt.plot(t,hist.history['loss'],'b',label='Training
    loss')
138                 plt.plot(t,hist.history['val_loss'],'r',label='
    Validation val_loss')
139                 plt.title('Traing and Validation loss')
140                 plt.legend()
141                 plt.savefig("plot_model_loss" + str(plot_loss_counter)
    )
142                 plot_loss_counter = plot_loss_counter + 1
143 
144             print("fit(train) model part %d/%d: finished"%(counter,
    num_partitions-len(test_list)))
145 
146     ########################      test model
    ##########################
147     for test_index in test_list:
148         # read test data
149         file_name = "/all_data/"+str(test_index)+".parquet"
150         data_test = pd.read_parquet(file_name, engine = 'pyarrow')
151         X_test = data_test[features].values
152         y_test = data_test[target].values
153         X_test = MinMax_x.transform(X_test)
154         y_test = MinMax_y.transform(y_test)
155         length = len(X_test)
156         observation_counter.append(length)
157 
158         # prepare test data
159         X_time_test = []
160         for index in range(0, len(X_test) - sequence_length):
161             X_time_test.append(X_test[index: index + sequence_length])
162         X_time_test = np.array(X_time_test)
163         X_test = X_test[(sequence_length-1):(length - 1)]
164         print("test data: ready")
165 
166         # test
167         y_pred_LSTM = (model_lstm.predict(X_time_test)).reshape(-1, 1)
168         y_pred_baseline = (model_baseline.predict(np.array(X_test))).
```

```
    reshape(-1,1)
169         y_test = np.array(y_test[(sequence_length-1):(length- 1)])
170         print("predict: finished")
171         y_pred_LSTM_inverse = MinMax_y.inverse_transform(y_pred_LSTM)
172         y_pred_baseline_inverse = MinMax_y.inverse_transform(
    y_pred_baseline)
173         y_test_inverse = MinMax_y.inverse_transform(y_test)
174         mse_lstm = mean_squared_error(y_pred_LSTM_inverse,
    y_test_inverse)
175         mse_baseline = mean_squared_error(y_pred_baseline_inverse,
    y_test_inverse)
176         error_lstm.append(mse_lstm)
177         error_baseline.append(mse_baseline)
178         print ("Test ERROR(LSTM) = ", mse_lstm)
179         print("Test ERROR(SGDLR) = ", mse_baseline)
180
181         # plot results
182         if plot_pred:
183             t = np.linspace(0, len(y_pred_inverse_minmax), len(
    y_pred_inverse_minmax))
184             plt.figure()
185             plt.plot(t, y_pred_LSTM_inverse)
186             plt.plot(t, y_pred_baseline_inverse)
187             plt.plot(t, y_test_inverse)
188             plt.title("mse_lstm:"+str(mse_lstm)+" mse_OLRwithSGD: "+
    str(mse_baseline))
189             plt.legend(['y_pred_lstm', 'y_pred_baseline', 'y_test'])
190             plt.savefig("plot_" + str(test_index))
191
192         observation_counter_temp = np.array(observation_counter)
193         print("MSE of LSTM for now:")
194         print("MSE:", sum(error_lstm*observation_counter_temp)/sum(
    observation_counter_temp))
195         print("MSE of baseline for now:")
196         print("MSE:", sum(error_baseline*observation_counter_temp)/sum
    (observation_counter_temp))
197
198     overall_observation_counter.append(sum(observation_counter))
199     counter_temp = np.array(observation_counter)
200     print("Fold %d/%d: finished"%(fold_counter + 1, n_folds))
201     print("MSE of LSTM")
202     print("MSE:", sum(error_lstm*counter_temp)/sum(counter_temp))
203     overall_error_lstm.append(sum(error_lstm*counter_temp)/sum(
    counter_temp))
204     print("MSE of baseline")
205     print("MSE:", sum(error_baseline*counter_temp)/sum(counter_temp))
206     overall_error_baseline.append(sum(error_baseline*counter_temp)/sum
    (counter_temp))
207
208 overall_temp = np.array(overall_observation_counter)
209 print("Final MSE of LSTM")
210 print("MSE:", sum(overall_error_lstm*overall_temp)/sum(overall_temp))
211 print("Final MSE of baseline")
212 print("MSE:", sum(overall_error_baseline*overall_temp)/sum(
```

```
    overall_temp))
```

B.2 Code of the simulator

Here, we demonstrate the source code of the simulator introduced in Section **3.3**. The basic idea of this code is provided by ABB, and the detailed theories are discussed in Section **3.3**.

```
#import
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import signal

# parameters
tau = 10.0 * 60
h = 1.0
N = 10000 # Number of simulated epochs of time
n = 3 # Multiplication factor, explained in Section 3.3.

beta = n*tau
Imax = 1

################### A test ###################
# create inputs
t = pd.Series(np.arange(-100.0, 5000.0, h))
dt = pd.DataFrame(t, columns = ['t'])
dt['I'] = 0.0
dt['I'].loc[dt['t']> = 0] = 1

# Analytic solution for the corresponding outputs
dt['a'] = 0.0
dt['a'].loc[dt['t']>0] = dt['I']*dt['I']*(1 - np.exp(-dt['t']/tau))

# Numerical solution for the corresponding outputs,using the lfilter
    function for convenience
alpha = h/(tau + h)
a = [1,-(1 - alpha)]
b = [alpha]
dt['f'] = signal.lfilter(b, a, np.square(dt['I'].values))
dt.set_index('t', inplace = True)

################### Creat simulated data ###################
# Draw intervals
d = np.random.exponential(beta, N)
stepTimes0 = np.cumsum(d) - d[0]
maxTime = stepTimes0[-1] + 3*tau
stepTimes = np.append(stepTimes0, maxTime)
t = np.arange(0, maxTime, h)
```

```
41
42 # Draw amplitudes
43 Is0 = np.random.uniform(0, 1, N)
44 Is = np.append(0.0, Is0) # Start with 0
45 inds = np.digitize(t, stepTimes)-1
46 I = np.take(Is, inds)
47
48 # Create dataframe
49 dt = pd.DataFrame({'t':t, 'I':I})
50 alpha = h/(tau + h)
51 a = [1,-(1 - alpha)]
52 b = [alpha]
53 dt['O'] = signal.lfilter(b, a, np.square(dt['I'].values))
54
55 ################### Illustrate simulated data ###################
56 # illustrate inputs and outputs
57 # dt[['I','O']].loc[0:100000].plot()
```