TUHH
Technische Universität Hamburg

Investigating Artificial Neural Networks for Monitoring an Unmanned Air Vehicle

Master's Thesis submitted in partial of
the requirements for the degree of
Master of Science
in
Mechatronics

Emin Çagatay Nakilcioglu
Matr. No. 21756438

Institute of Embedded Systems
Hamburg University of Technolgy

July 2020

1. Examiner : Prof. Dr. Görschwin FEY
2. Examiner : Prof. Dr.-Ing. Rolf-Rainer Grigat

Abstract

In this thesis, a custom fault injection model for an unmanned air vehicle (UAV) running in software-in-the-loop (SITL) simulation is proposed together with five distinctive deep learning based time series prediction model combined with an anomaly detection algorithm to address anomaly detection problem in a UAV. First, SITL simulation of a UAV is performed using a combination of three software tools; an open source control software PX4, an open-source robotics simulator Gazebo and a ground control station software QGroundControl. Via SITL, sensor related data of the simulated UAV is logged and stored. This feature is exploited for applying further analysis on the collected data. Moreover, a custom fault injection (FI) model specifically designed for UAVs running in SITL simulation is developed. Via this FI model, faulty behaviours of a target system are simulated. Combining the SITL mechanism with the FI model provides data source for anomaly detection on a UAV suffering from faults in its system. Data provided by the SITL simulation and FI model consist of 2 different type of datasets; datasets corresponding to the normal behaviour of the system and datasets corresponding to the behaviour of the system in the presence of any fault in the system. Datasets are further split into two subcategories where for the first category, a UAV follows a circular path in a simulation run whereas second category contains data regarding the UAV following a non-circular path. Provided data is utilized as input data for deep learning based models proposed for anomaly detection. Proposed models are mainly categorized in two categories; convolutional neural network (CNN) based and long-short term memory(LSTM) network based models. CNN-based approaches consist of a 1-dimensional CNN model and temporal convolutional network (TCN) whereas LSTM-based approaches are a stacked LSTM model and two LSTM models with two different attention mechanisms. Models are first trained on the dataset with normal behaviour of a UAV with the intention of learning the normal behaviour of a UAV. A secondary dataset containing only normal behaviour of a UAV is used as a validation dataset during training in order to contribute to the generalization ability of the models. During training, models are expected to predict a future data point given the previous related data points. For anomaly detection, a summary prediction model is implemented to the time series predictive models. In the anomaly detection algorithm, prediction errors of the models on the training data is computed and later fitted to a multivariate Gaussian distribution. An anomaly score function is derived using the parameters of the Gaussian distribution. Anomaly score is used for classifying whether a given data point is an anomaly point or a normal data point. A fixed threshold of anomaly score for each experiment is individually set. In threshold calculation, anomaly score function is applied to another validation dataset containing both normal and anomalous data points. Anomaly score value yielding the fewer false positives are determined as the threshold value. Finally, prediction error of the models on a test dataset containing normal and abnormal points is computed for evaluation of the models' performance on anomaly detection. Comparison between the performance of the models is made in order to conclude the experiments. The results shows that LSTM-based models yields better performance on flagging detections in a given test dataset as compared with CNN-based models. However CNN-based models provides similar precision values with less computational power.

Contents

1 Introduction

Use of the unmanned air vehicles (UAV) and drones has started to widely spread in various aspects of the life. The purpose of their usage varies from recreational to academical goals. The developments that have been made in this industry in the recent past have played a major role in this ever-lasting increase of their usage. Thus, there is a significant number of advancements emerged from academical researches and they continue to grow [1].

Automatic and pilot-in-the-loop flight control systems are widely implemented in UAVs in order for UAVs to perform an automatic flight without an external interruptions by a user. Though, these control systems are proven to be highly reliable to handle unexpected sensor readings resulting from an external or internal interruptions [2], these unexpected behaviours are not further addressed but rather taken care of during the exposure. If an unexpected interruption, that a system experiences and suffers during an automated flight, is an indication of a more serious malfunctioning in the internal system of the UAV and not further inspected, the flight control systems may not be able to handle the next occurrence of the same malfunctioning depending on the complexity of the malfunctioning. Applying an anomaly detection method in an analysis on the data of a UAV provides an early detection mechanism for such malfunctionings.

1.1 Anomaly Detection

There has been wide range of studies conducted on anomaly detection algorithms since anomaly detection problem is considered a concern in many application domains such as fraud detection, health monitoring and mechanical diagnostic [3]. In general, patterns in data that deviate from the expected normal behaviour are referred to as anomalies. In terms of discrete values, anomalies are defined as samples that are further away from the majority of the samples. Anomalies for both discrete and continuous values are shown in Figure 1.1. In literature, these abnormal behaviours are referred to as anomalies, outliers or novelties [4]. Throughout this thesis, the term anomaly was used to refer to these abnormalities.

There are three main types of anomalies in literature [4]:

- **Point anomalies:** A single data sample is considered an anomaly if it is anomalous relative to the remaining data samples.
- **Contextual anomalies:** Data samples are referred to as contextual anomalies if the samples are anomalous only in a particular context. For example, a data representing the temperature measurement during the year where minus degrees during August is considered a contextual anomaly, whereas the same measurements would be considered expected measurements.
- **Collective anomalies:** Data samples are considered collective anomalies when a set of grouped samples are normal sample points individually but anomalous relative to the entire dataset if taken together. For instance, time series sequences which differ from the usual pattern of the data are considered a collective anomaly.

Anomaly detection algorithms focus on finding abnormal behaviour in data. Due to several factors, anomaly detection may be considered a complex task [4]. For instance, in both cases illustrated in Figure 1.1, abnormal behaviours can be seen but the representation of these anomalies are clearly different. Considering that an anomaly is defined by not conforming to the expected behaviours, detection of an anomaly becomes difficult to define generalized normal cases and cases with anomalies spanning multiple application domains. Anomaly detection methods are dependent on the type of data at hand as the anomalies themselves show dependency to the data type. This has resulted in many field- and type-specific solutions [3].

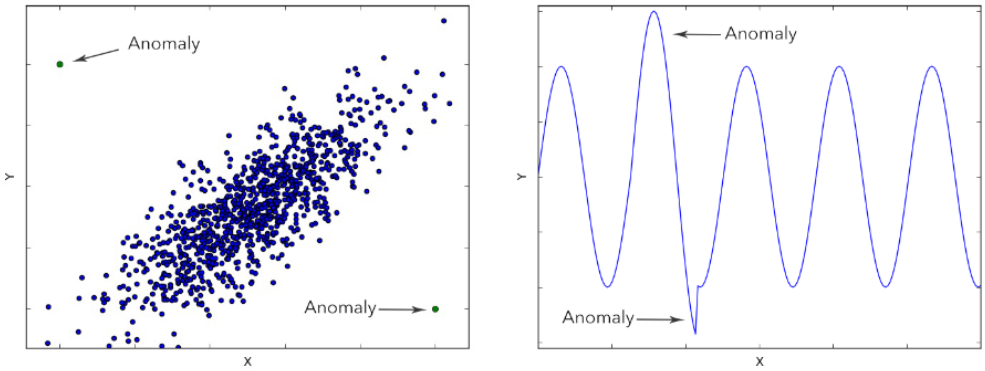

Figure 1.1: Representation of anomalies in a dataset. On the left hand side, two points are further away from other sample point in the dataset and thus, are considered anomalies. On the right hand side, in two occasions, data shows a different pattern then the pattern seen in the rest of the data. Thus, both instances are flagged as anomalies [5].

A close relationship between time series prediction and anomaly detection has been established in the time series domain. As showed in [6], an instance of anomaly detection can be transformed into a time series prediction problem. Considering that an anomaly is a pattern or a individual sample point deviated from a normal instance of a input pattern in the data, a time series predictor can model these patterns. Any breach in a predicted pattern will present itself an anomaly in the normal data model therefore can be flagged as an anomaly.

Machine learning models are able to learn patterns by training on data and later perform forecasting on a given test input data. In recent years, deep learning approaches has been considered as one of the most popular machine learning techniques, yielding successful results for a range of supervised and unsupervised tasks due to its capability to learn high-level representations related to the given task. Learning process of these representations are performed automatically from data with a relatively little of domain expertise knowledge. Regarding the time series and sequential data domain, recurrent neural networks (RNN) are gaining popularity due to their ability to recognize patterns in sequences of data. Furthermore, several researches have been conducted using Long-short term memory (LSTM), a variant of RNNs, in case of the need of learning log range patterns in time series data [7]. In [8], stacked Bidirectional self-attention LSTM was proposed to detect and predict system failures. As reported in the paper, model was able to capture the complex representations of the anomaly, and obtain promising results as compared to the other existing anomaly detection algorithms. There has been also implementation of convolutional neural networks (CNN) on learning temporal relationship in data [9]. In a related study conducted by Google DeepMind [10], a generative model of a dilated causal convolutional neural network (CNN) operating on audio with a very high resolution was developed. Though as stated in [10], application of this model was for audio generation, possibility of using

the proposed model for classification was also discussed in [10]. These works and other related works are further examined in 2.5.

In order to address this issue of anomaly detection in UAVs, an UAV vehicle running in the software in the loop (SITL) simulation is employed in this project. Via simulation environment, datasets that consists of sensor parameters of the UAV is gathered for further investigation. Additionally, fault injection (FI) techniques are studied to develop a FI model for simulating possible anomalies that can occur in an UAV during a flight. Using FI techniques, anomalies can be recreated in a simulation environment and datasets containing information of the behaviour of the simulated UAV can be collected for further analysis.

Many forms of FI have been employed by the designers. These forms include hardware-based, software-based and simulation based FI, where each type presents its own trade-off [11]. Though hardware-based FI is considered to be the most representative of the harsh environment of space [12], device under test (DUT) can be permanently damaged during a hardware-based FI. On the other hand, software-based FI does not pose any threat of damaging to the DUT. However, in software-based FI, target system's software is required to be altered in such a way that this alterations may accidentally affect how the faults are manifested in the target system. Considering that simulations offers a high level of visibility into and control over the DUT, it offers a unique technique [13]. Nonetheless, integrity of the simulation models highly affects the accuracy of results. The simulation-based FI techniques is chosen for the thesis due to the mentioned advantages over other FI techniques. FI models and techniques as well as the SITL mechanism employed in the thesis are further discussed in 3.3.

1.2 Contributions

Primary focuses of the thesis are categorized into two major topics; SITL and FI mechanism, and deep learning based anomaly detection approaches derived from time series prediction models. To this end, a custom FI model for a UAV running in SITL simulation is designed. Using the FI model, particular faults are injected into a UAV in SITL simulation in order to emulate the behaviour of the UAV in the presence of faults. Through SITL simulation, information regarding the sensor parameter values of the targeted UAV is collected for further off-line inspection. This information consists of the UAV's behaviour both with and without the presence of faults in its system. This information, also referred to as datasets in this thesis, serves as data source for anomaly detection approaches also proposed in this thesis.

With regards to anomaly detection, approach of combining time series prediction models and an anomaly detection algorithm is pursued in this thesis. Time series prediction models are modelled based on conventional and state-of-the-art deep learning architectures which are further discussed in 2. These model are collected under two main categories; CNN-based models and LSTM-based models. An anomaly detection approach based on summary prediction is implemented in the models to apply these models in anomaly detection problem. Only the short term anomalies and deep learning based models for detection are considered in this thesis. Thus, conventional machine learning models for predictive anomaly detections, such as support machine vectors are beyond the scope of this thesis and not examined as related works. Additionally, study of this thesis is limited to investigating time series predictive models and anomaly detective algorithms that are suitable for the datasets used in this thesis, which are datasets gathered through aforementioned mechanism of SITL and FI model.

This thesis resulted in four main contributions:

- An FI model, focusing on recreating anomalies on a AUV running in SITL simulation, was

designed in this thesis. The FI model is derived from a simulation-based FI technique and consists of its custom fault library.

- Two 1-dimensional (1-D) CNN based time series prediction model, a vanilla 1-D CNN model and a a temporal convolutional network (TCN)model was designed. Via further extension of the model with implementing an anomaly detection approach, proposed models are used to address detection problem in a simulated UAV.
- Three different LSTM-based time series prediction model, stacked LSTM network and two LSTM networks with additional attention mechanisms, were proposed. Similar to CNN-based models, these models are also employed in anomaly detection problem via attaching an anomaly detection algorithm to the proposed models.
- Following the experiments on the data of a simulated UAV, a comparison of the proposed models is carried out for further analysis. Through the results of the comparison, individual performance of the proposed models in both time series prediction problem and anomaly detection problem are examined.

The structure of this paper is laid out as follows: Chapter 2 gives a detailed theoretical introduction of artificial neural networks and relevant additional theory as well as related work regarding anomaly detection in time series data. In Chapter 3, the tools employed for the SITL simulation of the UAV along with the embedded architecture of the considered flight control software are provided. Additionally, examination of related FI techniques, followed by the introduction of the FI environment and fault scenarios used in the thesis are explained in detail. A detailed explanation of the proposed deep learning based architectures, comprised of a time series prediction algorithm and an anomaly detection algorithm, is given in Chapter 4. In Chapter 5, first, datasets employed in the thesis are explored and pre-processing operations applied on the dataset for the experiments are introduced. Furthermore, design space exploration of the proposed models is discussed and, the experimental set-up and results of the experiments are presented. In the final section of Chapter 5, results and general prospect of the thesis are further discussed. Finally, conclusions are drawn in Chapter 6 along with suggestion for future work.

2 Artificial Neural Networks

Artificial neural networks (ANNs) are a set of algorithms that are designed to find linear and non-linear relationship between the input and output variables in a given dataset. The purpose of an ANN is to map a set of input patterns against a corresponding set of output patterns. Depending on the arrangement of neurons and connection patterns of layers, different neural networks can be modelled. In some of the ANN applications, expected output variables are available together with the input whereas in some other applications expected output is unknown. In machine learning terminology, first instance is referred to as a supervised learning problem and the latter as an unsupervised learning problem.

A neural network (NN) is formed using simple computational units called nodes or neurons. When input data is fed to a neuron, the neuron then multiplies the input data by corresponding weights. Finally, the neuron applies a non-linear function called activation function, to the weighted sum to produce a final output. The functioning of a neuron is illustrated in Figure 2.1. Given the input vector $\vec{x} \in \mathbb{R}^n$, weight vector $\vec{w} \in \mathbb{R}^n$, bias vector $\vec{b} \in \mathbb{R}^n$, a neuron computes an output as follows

$$y = f(\sum_{j=0}^{n}(w_j x_j + b_j)), \tag{2.1}$$

where $x_j, w_j, b_j, f(\cdot)$ and y respectively denoting the j-th element of the input vector $\vec{x}$, j-th element of the weight vector $\vec{W}$, j-th element of the bias vector $\vec{b}$, activation function and output value of the neuron.

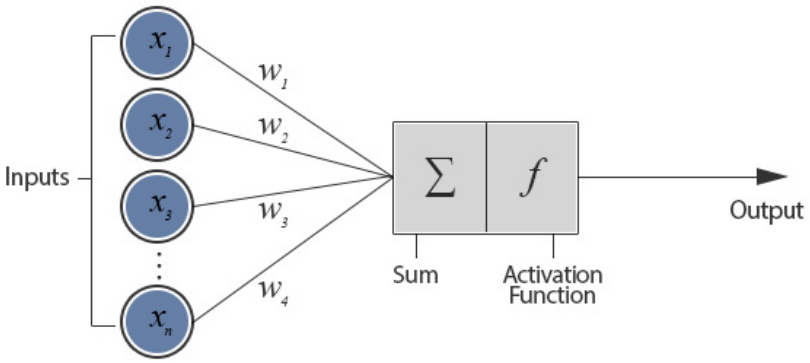

Figure 2.1: Process in a neuron. A single neuron generates a non-linear function of the weighted sum of its inputs as an output. The activation function introduces non-linearity [14].

2.1 Feed-forward ANNs

Feed-forward NNs are artificial neural networks where the connections between units do not form a cycle. The basic feed-forward network with one intermediate layer, also called hidden

layer, comprising of several neurons, also known as units, is shown in Figure 2.2. As the name suggests and Figure 2.2 illustrates, the information only travels forward in these NNs, starting from the input nodes, then through the hidden layer and finally to the output nodes. Given the input vector $\vec{x} \in \mathbb{R}^n$ and corresponding output vector $\vec{y} \in \mathbb{R}^m$, equation of the feed-forward NN model in Figure 2.2 can be written as

$$\vec{y} = f(W \cdot \vec{x} + \vec{b}), \tag{2.2}$$

where $W \in \mathbb{R}^{m \times n}, \vec{b} \in \mathbb{R}^m$ and $f(\cdot)$ respectively denote the weight matrix, bias vector and activation function of the network.

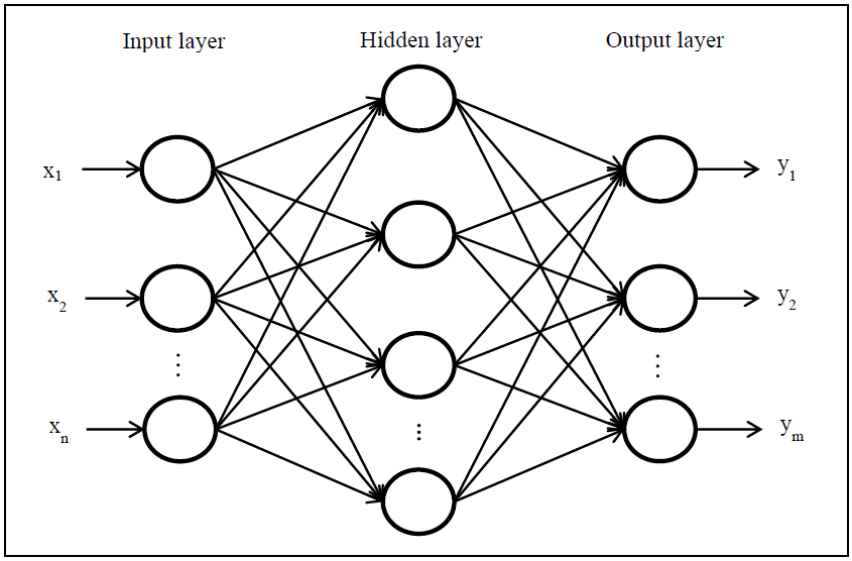

Figure 2.2: The structure of a two layered feed forward NN. Given an input vector $\vec{x} \in \mathbb{R}^n$, the feed-forward NN above generates the output vector $\vec{y} \in \mathbb{R}^m$. Each neuron in a layer is connected to all the neurons in the previous layer. The input layer is not considered as one of the functioning layer of the feed-forward NN since the input layer only forwards the input variables without any processing and thus, the input layer nodes are not technically neurons [15]

NNs which has one hidden layer are referred to as shallow neural networks whereas neural networks formed by stacking multiple NNs together are referred to as deep neural networks as shown in Figure 2.3. Deep NNs consist of multiple hidden layers and each layer is a non-linear module in which the output of the preceding layer is fed. Starting from the first layer to last layer, more complex features are learned progressively. As reported in [16] and [17], deep NNs with an appropriate architectures has achieved better results than shallow NNs which have the same computational power, i.e. number of neurons. Results have showed that by using their intermediate hidden layer, the deep NN models performed better at extracting features that are useful for the learning process than shallow NN models.

The goal of a feed forward NN is to approximate a certain function $f^*(\cdot)$ given an input vector $\vec{x}$. In its most general form, feed forward NNs define a mapping $\vec{y} = f(\vec{x}; \theta)$ and learns the value of the parameters θ that generates the best approximation of function $f^*(\cdot)$. Feed-forward neural networks are formed by composing many functions together to obtain a chain-like structure as

$$f(\vec{x}) = f^{(n)}(f^{(n-1)}(\cdots f^{(2)}(f^{(1)}(\vec{x})))), \tag{2.3}$$

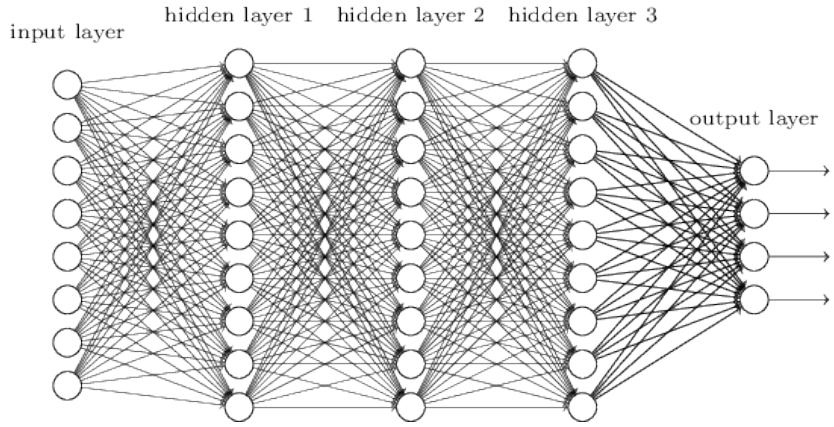

Figure 2.3: A deep NN with 3 intermediate hidden layers. [18]

where $f^{(1)}(\cdot)$ being the first layer of the network, $f^{(2)}(\cdot)$ being the second layer and so on. In ANN terminology, first layer $f^{(1)}(\cdot)$ is called input layer and final layer $f^{(n)}(\cdot)$ is called output layer while all the other remaining layers are called hidden layers [19].

For a given input vector $\vec{x}^{(0)}$, output vector $\hat{\vec{y}} = \vec{x}^{(n)}$ of a n-layered feed-forward ANN is computed as

$$\vec{x}^{(i+1)} = f^{(i)}(W^{(i)} \cdot \vec{x}^{(i)} + \vec{b}^{(i)}) \qquad i \in \{0, \ldots, (n-1)\}, \tag{2.4}$$

where $W^{(i)}, \vec{b}^{(i)}$ and $f^{(i)}(\cdot)$ respectively denote the weight matrix, bias vector and activation function of the layer i. Weight matrix and bias vector are specialized version of the parameters, denoted as θ above, that networks trains to learn in order to generate the best possible approximation of the function $f^*(\cdot)$. Non-linear activation functions employed in the thesis are sigmoid function, hyperbolic tangent (tanh) function and rectified linear activation unit (ReLU) function [20] which are defined as follows

$$f_{sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp^{-x}} \tag{2.5}$$

$$f_{tanh}(x) = \tanh(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}} \tag{2.6}$$

$$f_{ReLU}(x) = \max(0,x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}. \tag{2.7}$$

In addition to the non-linear activation functions, linear activation function is also used in the thesis since linear activation function is used specifically for regression problems where main focus in on predicting continuous values. Linear activation functions apply the identity function which is derived as

$$f_{linear}(x) = x. \tag{2.8}$$

During training, both input and output data is introduced to the network. In order to facilitate learning, a loss function $L(\vec{y},\hat{\vec{y}})$, which takes the values of expected output vector $\vec{y}$ and derived output vector $\hat{\vec{y}}$ and computes a distance score, is constructed.

One of the loss functions commonly used for regression problems and also applied in this thesis is mean squared error (MSE). MSE computes the averaged squared distance between

the expected output vector $\vec{y}$ and the derived output vector $\hat{\vec{y}}$. This computed distance is also referred to as the error or residual. Given the output vectors $\vec{y}, \hat{\vec{y}} \in \mathbb{R}^N$, MSE is computed as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{2.9}$$

where N is the number of observations , y_i is i-th value of the expected output vector and $\hat{y}_i$ is the i-th value of the derived output vector.

During the training of a feed forward NN, learning is accomplished by performing an optimization (error minimization) of every W_i and $\vec{b_i}$ for a given pair of $\vec{x}_0$ and $\vec{y}$ with the focus of minimizing the loss function. The optimization algorithm used in NNs is called gradient descent. The gradient is a measure of the change in the loss value corresponding to a small change in a network parameter. Gradient descent algorithm involves computing the gradients of the loss function with respect to the network parameters, θ. The method used to compute the gradients is called back-propagation and is based on the chain rule of derivatives, computing the gradient one layer at a time, iterating backwards from the last layer to avoid redundant calculations of intermediate terms in the chain rule [21]. A scalar value called the learning rate η is used to update the parameters θ in opposite direction of the gradient as derived in 2.10 Entire process is performed iteratively through several passes over the training data. A complete pass over training data is referred to as an epoch and after every epoch the updated parameters approach their optimum values which minimizes the loss function of the network.

$$\theta = \theta - \eta\frac{\partial L(\theta)}{\partial \theta}. \tag{2.10}$$

In case of large sized datasets, calculating the loss and gradient over the entire dataset may be too slow and computationally infeasible. Thus in practice, there are several variants of gradient descent and one of the commonly used one is called stochastic gradient descent (SGD). In SGD, data is divided into subsets called batches, and the update of the parameters are performed after calculating the loss function over one batch. One of the other popular variants is Adam optimizer introduced in [22]. Adam optimizer is designed to keep track of an exponentially weighted moving average of the gradient and to update the network parameters accordingly by using the weighted moving average.

One of the common problems encountered frequently when training NNs is overfitting. Overfitting occurs as a result of applying a more complex model than required since more complex models will try to fit the noise in training data. In case of overfitting, though the model performs well on training data, it performs poorly on new data. In order to avoid overfitting, several methods were introduced. One of these methods is early stopping. In early-stopping procedure, a validation dataset is used in addition to training dataset. The loss value of the validation set is monitored in each epoch to stop the training procedure if it is not improving within a certain number of previous epochs. Number of previous epochs to consider for monitoring is determined before starting the training. Another commonly used method is dropout [23]. In this method, a certain percentage of NN connections in the model are randomly removed in each training epoch. By dropping a unit out, the unit is temporarily removing it from the network, along with all its incoming and outgoing connections [24].

Weights and biases are learned and tuned by the training algorithm through backpropagation whereas parameters such as learning rate, dropout rate, batch size and decay are part of the learning algorithm and require to be tuned appropriately by the user. In the literature, these

learning parameters are called hyper-parameters or design parameters. Throughout the thesis, they are referred to as design parameters.

2.1.1 Shortcomings of Feed-Forward NNs for Sequential Data

A variety of domains including natural language processing, speech recognition, forecasting and computational biology commonly work on sequential data. In its most general form, sequential data is divided into time series and ordered data structures. Time series data exhibit changes over time and remain consistent in the clips such as daily prices of stocks or the time frames for speech or video analysis. In addition to time series data, ordered data structure can also be found in the sequence, such as text and sentence for handwriting recognition, and genes. For instance, in order to successfully predict protein-protein interactions, knowledge of the secondary structures of the proteins is required [25].

Sequential data consist of time steps. In its most general form, main goal of the network that trains on sequential data is to define a mapping $f : x \to y$ where x denoting an input sequence $x = \{x_1, x_2, \ldots, x_T\}$ with T time steps and y denoting the corresponding output sequence $y = \{y_1, y_2, \ldots, y_T\}$,

One of the main assumption for NNs and many other machine learning models in general is that there is no dependency among input data samples. The problem arises when training on sequential data since sequential input data samples exhibit dependence between their individual elements across time. Given that NNs treat each element of the input data individually, they cannot make use of this sequential dependence that can be extracted by exploiting this sequential information.

One approach to address the dependency in a sequential data is to concatenate a fixed number of consecutive data samples together and treat them as one data point. [26] applied this method for time series prediction using NNs, and [27] for acoustic modelling. However, as reported in [26], finding the optimal window size plays a major role on the success of this method. Applying a larger window size than needed would add noise to the training whereas a small window size fails to capture the longer dependencies. Furthermore, in case of long-range dependencies in data ranging over hundreds of time steps, a window-based method would fail to scale.

RNNs' ability to process the element of a given sequence individually by a hidden state vector which resembles a memory for the past information. Hidden states learn to store the relevant information and to further pass along the information inside RNN. Through this internal information flow, network can use not only the current input information but also information regarding the past can be used for generating future predictions. Therefore, RNNs are widely used in application domains where sequential data is common.

RNNs try to capture the temporal relations from the first element in the given sequence to the last element. In other words, RNNs assume that every point in the sequence depends on every previous information. However, it is not applicable to many practical cases [28]. Applying looser assumption is an approach to counter this issue. For instance, modelling a projectile motion does not require more than few consecutive data points. Likewise, in certain cases, a model does not require to know every preceding text to guess a word in a sentence. For this purpose, CNN models are also implemented in when processing sequential data.

2.2 Recurrent Neural Networks

In feed-forward neural network models, there are no feedback connections where outputs of the models are fed back into itself. When the feedback connections are introduced to feed-forward neural networks, they are called recurrent neural networks. RNNs are a class of neural networks which exhibit temporal behaviour due to directed connections between units of an individual layer. As reported by Pascanu et al. [29], recurrent neural networks maintain a hidden state vector $\vec{h}$ in the hidden units, which holds information of every previous element in the sequence. Figure2.4 illustrates a standard RNN and its feedback connection that connects the hidden neurons across time. At time t, both the current element of the sequence x_t and hidden state from the previous time step h_{t-1} is received as in put by RNN. Thenceforth, the next hidden state at times t is updated and the final output o_t is computed as follows

$$\vec{a}_t = W \cdot \vec{h}_{t-1} + U \cdot \vec{x}_t + \vec{b} \tag{2.11}$$

$$\vec{h}_t = \tanh(\vec{a}_t) \tag{2.12}$$

$$\vec{o}_t = V \cdot \vec{h}_t + \vec{c}, \tag{2.13}$$

where tanh is the hyperbolic tangent function 2.6, $\vec{b}$ is the first bias vector, W is the recurrent weight matrix for hidden-to-hidden connections and U is the recurrent weight matrix for input-to-hidden connections, $\vec{c}$ the second bias vector and with V is the recurrent weight matrix for hidden-to-output connections.

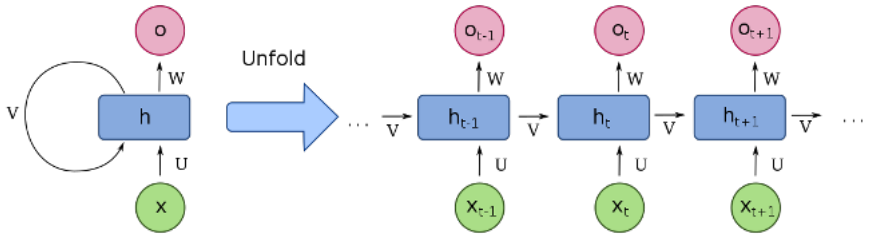

Figure 2.4: A standart RNN maps an input sequence to an output sequence of the same length. On the right hand side of the figure is a standard RNN. h denotes the hidden states in the hidden units. On the right hand side is the same RNN unfolded in time to portray how the hidden states are built over time given an input sequence $x = \{\ldots, x_{t-1}, x_t, x_{t+1}, \ldots\}$ where t denoting the corresponding time step [30].

Computation of the current hidden state h_t and output o_t is carried out at each time step. Hidden state $h_t$can be considered as the memory of the network since it captures information regarding all the previous time steps. The output o_t is calculated solely based on the memory at time t. Moreover, unlike a traditional feed-forward deep NN, which utilizes different parameters at each layer, a RNN shares the same parameters, namely U, V and W in 2.11, across all time steps. Thus same task is performed at each step, just with different inputs. This considerably reduces the total number of parameters that the network needs to learn.

Depending on the task, initial output o_t can be put through an activation function to generate an appropriate output. For instance, common choice for multi-class classification problem is to

apply softmax activation function which is derived as

$$f_{softmax}(\vec{z})_i = \text{softmax}(x) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.14}$$

$$\tag{2.15}$$

where $\vec{z}$ being the input vector of the softmax function, z_i being the i-th element of the input vector $\vec{z}$ and K being the number of classes in the multi-class task.

A standard RNN such as the one shown in Figure 2.4, can be considered a deep NN where number of time steps in the input sequence corresponds to the number of layers, when the NN is unfolded in time. Due to the use of the same weight for each time step, an RNN has the ability to process variable length sequences. Information can flow in the RNN for an arbitrary number of time steps, allowing the network to keep the memory of the past time steps, since the hidden state $h_t$is updates at each time step the network received a new input.

Treating the unfolded RNN as a deep NN, RNN can be trained in a similar fashion to back-propagation using back-propagation through time (BPTT) where generalized back-propagation algorithm 2.10 is applied to the unrolled computational graph [19]. Though, in theory standard RNNs are expected to be able to learn long-range dependencies over arbitrarily long sequences and tune weights accordingly, as concluded in [31] and [32], RNN performs poorly when trained to learn dependencies across long intervals. When BPTT is applied during training of an RNN, back-propagation of error gradients are performed over several time steps. Due to the chain rule, error gradients are calculated by continuous multiplications of derivatives. In the case of back-propagating gradients over a long sequence, gradients either decay to zero or become too large. These problems are respectively referred to as vanishing gradients and exploding gradients problem. In the presence of such issues, model does not converge during training and thus, learning can take disproportionate amount of time.

Several approaches addressing the issue of learning long-term dependencies of RNNs were proposed. These approaches include introducing modifications to the learning procedure as well as architectural variants of RNN. For exploiting gradients problem, [33] proposed an approach called proposed gradient clipping where the gradient is scaled down if the norm of the gradient goes beyond the predefined threshold. Another approach proposed in [33] to address vanishing gradients problem ism vanishing gradient regularization where a regularization term that represents a preference for parameter values such that back-propagated gradients neither increase or decrease in magnitude, is introduced to the network. However, as stated in [33], the disadvantage of using the regularization method to avoid vanishing gradients increases the chances of exploding gradients. As for architectural variants of RNN, LSTM network was introduced in [34] to overcome the vanishing gradients problems.

2.2.1 Long Short Term Memory Model

Standard RNNs suffer from the gradient vanishing, where after each iteration of training, gradients get vanishingly small and effectively preventing the neural network model weights from changing their value, or exploding problems where large loss gradients accumulate and cause very large updates of the neural network model weights during training. In order to overcome these vanishing and exploding gradient problems, LSTM network was developed and successfully addressed these issues .

The main contribution the initial LSTM model [34] was introducing self-loops to produce paths where the gradient can flow for a longer time. Another addition was introduces in [35] by

making the weight on this self-loop conditioned on the context, rather than fixed. By making the weight of this self-loop gated, i.e. controlled by another hidden unit, the time scale of integration can be changed dynamically. In this case, even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself.

An LSTM unit is illustrated in Figure 2.5. Instead of a unit that simply applies an element-wise non-linearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have a self-loop in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The main components of the LSTM units are; input, input gate, forget gate, memory cell, output gate and output. In the input component, LSTM unit receives the current input vector x_t and the hidden state output of the previous time step, denoted by h_{t-1}. Weighted inputs are summed and fed through tanh activation function, producing z_t. Similar to input component, input gates takes x_t and h_{t-1} to compute their weighted sum and to later apply sigmoid activation function. Resulting vector i_t is multiplied by the z_t to feed the input into the memory cell. Another component providing data to memory cell is forget gate. Using forget gate LSTM can learn to reset memory contents in case of relevant or old information. This cases occur when the network begin processing a new sequence. Similar to input gate, forget gates takes x_t and h_t to compute their weighted sum and to later apply sigmoid activation function. Result f_t is then multiplied with the previous cell state s_{t-1}. Inside the memory cell, cell state at the current time step is computed by accepting new relevant information from x_t while forgetting irrelevant information from previous time step, if required, via the output of the forget gate f_t. Output gate receives the weighted sum of x_t and h_{t-1} and applies sigmoid activation function. Through this activation, output gate controls the kind of information flowing out of the LSTM unit. Output of the LSTM h_t is generated by applying tanh function to cell state s_t and multiplying the output of the activation with the output gate o_t. Corresponding equations for the functioning of a LSTM unit are derived as

$$z_t = tanh(W^z \vec{x}_t + U^z \vec{h}_{t-1} + \vec{b}^z) \tag{2.16}$$

$$i_t = \sigma(W^i \vec{x}_t + U^i \vec{h}_{t-1} + \vec{b}^i) \tag{2.17}$$

$$f_t = \sigma(W^f \vec{x}_t + U^f \vec{h}_{t-1} + \vec{b}^f) \tag{2.18}$$

$$o_t = \sigma(W^o \vec{x}_t + U^o \vec{h}_{t-1} + \vec{b}^o) \tag{2.19}$$

$$s_t = z_t \odot i_t + s_{t-1} \odot f_t \tag{2.20}$$

$$h_t = tanh(s_t) \odot o_t, \tag{2.21}$$

where W^*, U^* and b^* denoting input weights, recurrent weights and bias vectors of the corresponding gates, respectively.

2.2.2 Attention Mechanism

The attention mechanism, first proposed in [37], was born to help memorize long source sentences in neural machine translation. Rather than only using the LSTM's last hidden state for the output, the new approach is introduced to the network by attention mechanism in order to exploit the relevancy between a derived context vector and the entire source input. The weights of these connections are customizable for each output element. The attention mechanism is involved in learning the weight distribution of different parts, which leads to different parts

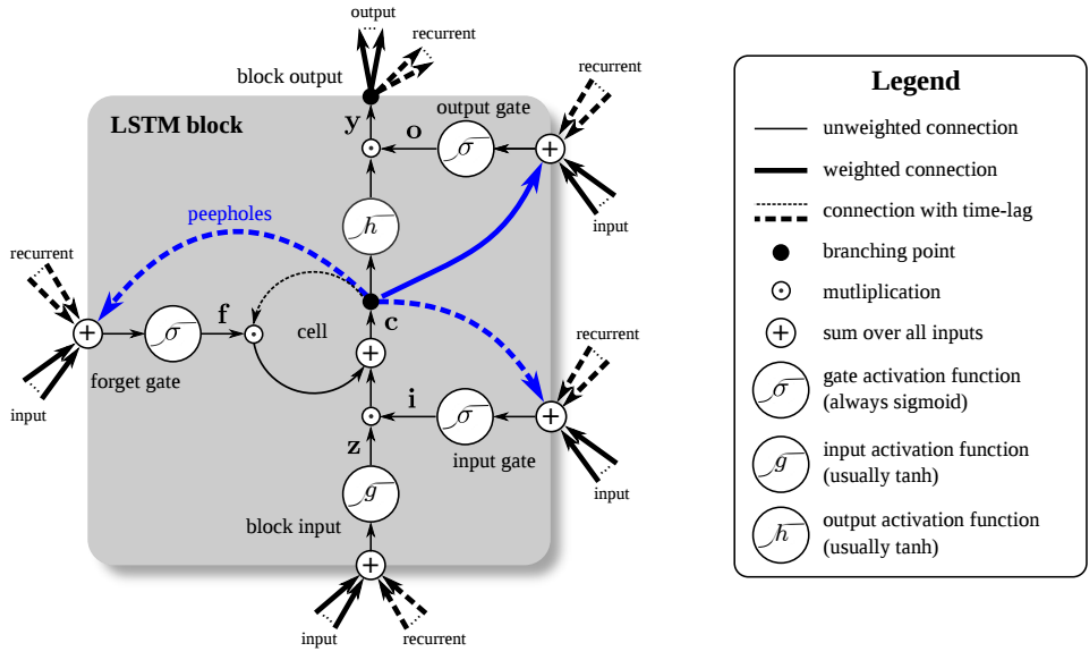

Figure 2.5: A schematic diagram of a LSTM unit [36].

corresponding to different degrees of concentration. The benefits of this property have been proven in many tasks, ranging from machine translation [37] and text summarization [38] in sequence-based tasks to classification and segmentation in computer vision [39].

In its most general form, an attention mechanism computes an alignment score between elements from two sources. In particular, given a uni-variate input source sequence $\vec{x} = \{x_1, x_2, \ldots, x_T\}$, represented as keys in Figure 2.6, and vector representation of query $\vec{q}$, represented as query in Figure 2.6, an attention mechanism computes the alignment score between x_t and $\vec{q}$ applying a score function $f(x_t,\vec{q})$ which computes the dependency between x_t and $\vec{q}$, also referred to as the attention of $\vec{q}$ to x_t. Then, a softmax function 2.15 is applied to transform the scores $[f(x_t,\vec{q})]_{t=1}^{T}$ to a probability distribution $p(z|\vec{x},\vec{q})$, referred to as alignment score above, by normalizing over the T samples of $\vec{x}$. Here z is an indicator which of the sample values in $\vec{x}$ is important to $\vec{q}$ on a specific task. Large $p(z = t|\vec{x},\vec{q})$ means x_t contributes important information to $\vec{q}$. Equations of the above process are provided as follows

$$a = [f(x_t,\vec{q})]_{t=1}^{T} \tag{2.22}$$

$$p(z|\vec{x},\vec{q}) = softmax(a), \tag{2.23}$$

where

$$p(z = t|\vec{x},\vec{q}) = \frac{\exp(f(x_t,\vec{q}))}{\sum_{t=1}^{T}\exp(f(x_t,\vec{q}))}. \tag{2.24}$$

The output of the attention mechanism is a weighted sum of the samples in $\vec{x}$, where the weights are calculated using the probability function $p(z|\vec{x},\vec{q})$. It assigns large weights on the samples that are important to $\vec{q}$, and can be written as the expectation of a sample according

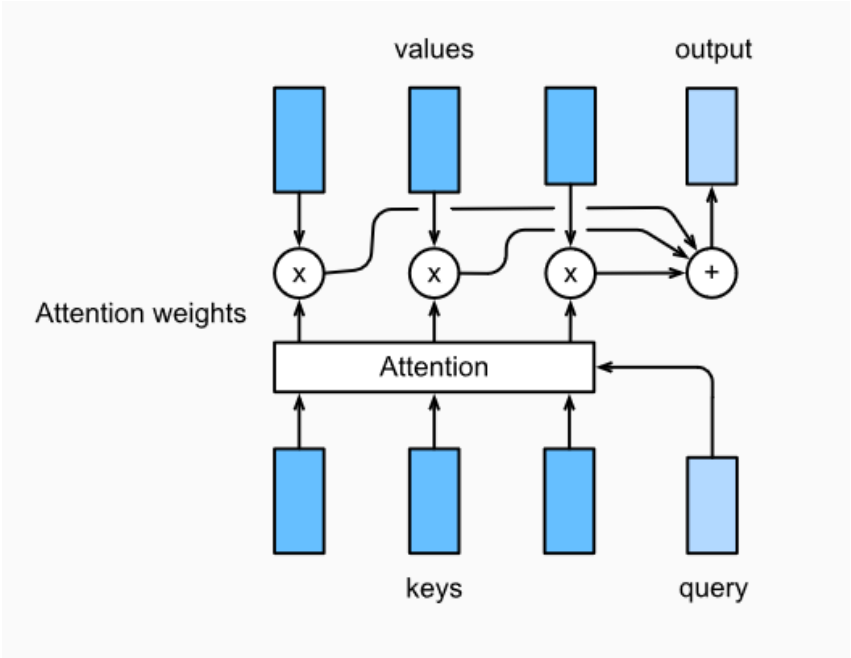

Figure 2.6: Block representation of attention mechanism. [40].

to its importance as

$$\vec{s} = \sum_{t=1}^{T} p(z = t|\vec{x},\vec{q}) x_t, \tag{2.25}$$

where $\vec{s}$ is referred to as context vector [37].

Additive attention [37] and multiplicative attention [41] are the two most commonly used attention mechanisms. They both are grounded on same and unified form of attention introduced above, but differ in terms of the score function $f(x_t,\vec{q})$. Score function for additive attention mechanism is derived as

$$f(x_t,\vec{q}) = w^T \sigma(w^{(1)} x_t + W^{(2)} \vec{q}), \tag{2.26}$$

where $\sigma(\cdot)$ being sigmoid activation function and w^T,$w^{(1)}$ and $W^{(2)}$ being weight vectors and a weight matrix, respectively. On the other hand, multiplicative attention applies inner product or cosine similarity for its score function, which is derived as

$$f(x_t,\vec{q}) = \langle w^{(1)} x_t, W^{(2)} \vec{q} \rangle. \tag{2.27}$$

As reported in [42], additive attention often outperforms multiplicative attention in prediction quality. However, the latter is faster and more computational efficient due to optimized matrix multiplication.

Furthermore, there is also a special case of an attention mechanism introduced above, self attention. Self-attention is an extended version of additive attention. It replaces the vector representation of a query $\vec{q}$ with a vector of samples $\vec{x}_j$ from the source input itself. It relates elements at different positions from a single sequence by computing the attention between each pair of samples, $\vec{x}$ and $\vec{x}_j$. It can be applied for both long-range and local dependencies by

exploiting latent correlation from elements at different positions for both long-range and local dependencies. Self-attention mechanism can also be applied in a multi-dimensional level [42]. Main difference in multi-dimensional level is that attention is explored between two vectors instead of individual samples in a vector as performed in uni-variate case. Thus, the input sequences now have an additional dimension which is referred to as feature dimension in this thesis. Multi-dimensional self-attention explores the dependency between $\vec{x}_i$ and $\vec{x}_j$ from the same source $\vec{x}$ over their feature dimension and produces context-aware representations. Instead of computing a single scalar score $f(\vec{x}_t,q)$, in multi-dimensional attention, feature-wise score vector for $\vec{x}_t$ by replacing weight vectors w^T and $w^{(1)}$ in 2.26 with a weight matrix W^T and $W^{(1)}$. Therefore, score function for multi-dimensional self-attention is derived as

$$f(\vec{x},\vec{x}_j) = W^T\sigma(W^{(1)}\vec{x} + W^{(2)}\vec{x}_j). \tag{2.28}$$

With a further addition of bias terms, score function becomes

$$f(\vec{x},\vec{x}_j) = W^T\sigma(W^{(1)}\vec{x} + W^{(2)}\vec{x}_j) + \vec{b}. \tag{2.29}$$

Multi-dimensional attention then computes a categorical distribution $p(z_k|\vec{x},\vec{x}_j)$ over the entire T samples for each feature k. Similar to uni-variate case, larger $p(z_k = t|\vec{x},\vec{x}_j)$ indicates feature k of sample t is important to $\vec{x}_j$. Same procedure in 2.22 and 2.23 is employed to the k-th dimension of $f(\vec{x},\vec{x}_j)$. Each input sample $\vec{x}_j$ is the corresponds to a probability matrix P^j, such that

$$P^j_{kt} \triangleq p(z_k = t|\vec{x},\vec{x}_j). \tag{2.30}$$

Ignoring the subscript k, which indexes feature dimension, for concision, the output s_j can be written as an element-wise product such that

$$\vec{s}_j = \sum_{t=1}^{N} P^j_t \odot \vec{x}_t. \tag{2.31}$$

The context vector of a self-attention layer for all elements from source sequence $\vec{x}$ can be written as

$$\vec{s_j} = \{s_1, s_2, \ldots, s_T\}. \tag{2.32}$$

2.3 Convolutional Neural Networks

CNNs, first introduced in [43], are a specialized type of ANNs to process data which has a known grid-like topology. Examples of their use include time-series data, which can be perceived as a one dimensional (1-D) grid taking samples at regular time intervals, image data, which can be thought of as a 2-D grid of pixels, and video data, which can be thought of as a 3-D grid having image at regular time intervals. As the name "convolutional neural networks" suggests, CNNs applies a mathematical operation called convolution. CNNs are basically neural networks that use convolution as an alternative to general matrix multiplication (2.4) in at least on of their layers.

Convolution, in its most general way, is a mathematical operation on two functions of real-valued arguments in order to express the amount of overlap of one function as it is shifted over another function. Convolution is derived as follows [19]

$$conv(t) = (x * w)(t) = \int x(t)w(t-a)da, \tag{2.33}$$

where a being the shifting measure, x and w being two functions with real-valued arguments, and asterisk denoting the convolution operation.

In CNN terminology, first argument to the convolution function, $x(\cdot)$ in 2.33, is often referred to as the input whereas the second argument as the kernel. Often when working with data on a computer, time will be disretized. Therefore, it is considered to be more applicable to employ discrete convolution to CNNs. The discrete convolution is defined as

$$conv(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{2.34}$$

In machine learning, the input is usually a multidimensional array of data, also referred to as tensor, and the kernel is usually a multidimensional array of parameters that are applied by the learning algorithm. And in case of multidimensional input, convolutions are used over more than one axis at a time. For instance, for a 2-D input array I and thus a 2-D kernel K, convolution can be written as

$$conv(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i-m,j-n)K(m,n), \tag{2.35}$$

where m and n denote the shifting measure in two dimensions. Since the convolution operation is commutative and thus it can also be written as

$$conv(i,j) = (K * I)(i,j) = (I * K)(i,j). \tag{2.36}$$

The commutative property of convolution arises because we have flipped the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. Despite its practicality for writing proofs, the commutative property is not usually considered an important property for ANN implementations [19]. Because of that, many neural network libraries, including the one adopted in this project, implement a function called cross-correlation, which is the same operation as convolution but without flipping the kernel:

$$conv(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i+m,j+n)K(m,n). \tag{2.37}$$

As shown in Figure 2.7, a typical layer of a CNN is composed of three stages. The first stage is where the layer generates a set linear activation by performing several convolutions in parallel. Following that, each linear activation is fed through a non-linear activation function in the second stage. In the third stage, pooling function is applied to modify the output of the layer further. A pooling function replaces the output at a certain location with a summary statistic of the nearby outputs, essentially downsampling the output. Some of popular pooling functions include max pooling, returning the maximum output within a rectangular neighbourhood, average pooling, returning the mathematical average within a rectangular neighbourhood, and weighted average pooling, returning the weighted average based on the distance from the central input. The use of pooling can be thought of as adding a strong prior that the learnings of the function or the layer must be invariant to small translations. Depending on the task at hand, a final layer is added to the network for generating an output. The most commonly applies layer of choice for such final layer is a feed-forward NN layer which is also referred to as fully connected layer. Fully connected layer applies an activation function to the output of the previous layer and generates the final output of the network.

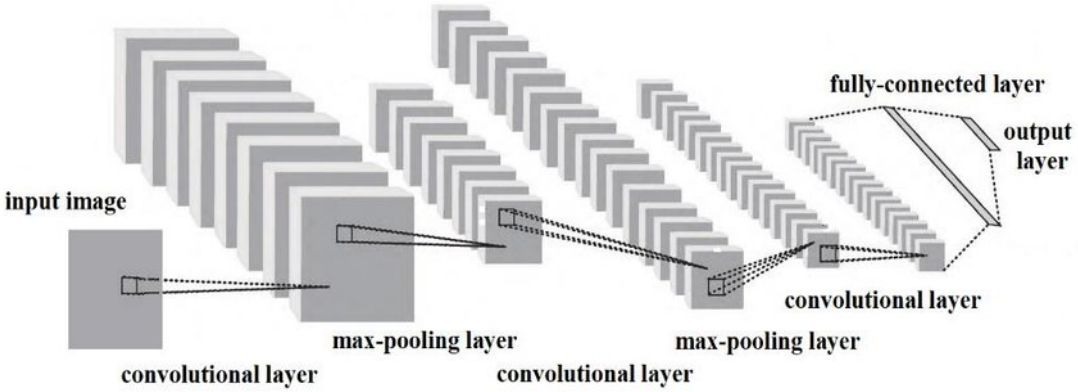

Figure 2.7: Typical CNN formation. [44].

2.3.1 Dilated Causal Convolution

Process of applying convolution where kernel is skipping values within a certain step size larger than one, instead of applying convolution to each input data as in 2.34, is called dilation. Convolution operation in dilated layers are equivalent to a regular convolution with larger kernel size where kernel replaces the elements within the step size with zero [10]. In [10], this particular step size is referred to as dilation rate. Dilation shown similarity to pooling operation. Difference is that dilation performs upsampling unlike the downsampling effect of pooling layer [45].

The causality of a layer, on the other hand, indicates that a prediction at time t only depends on values from previous times. Therefore, causality does not allow model to exploit dependencies of future time steps when computing a prediction at time t. Due to this feature of causality, it is commonly applied in time series modelling in order to employ only the past behaviour of a data when predicting the next time step.

Figure 2.8 illustrates a CNN with stacked dilated causal convolutional layers where dilation rates increases exponentially at each layer. Due to stacking dilated layers, CNN model has a large receptive field with only a few layers. Since the illustrated CNN has a receptive field of $2^3 = 8$, every output of the network depends of eight previous layer of the input compared to a convolutional layer without dilation which would have a receptive field of 4

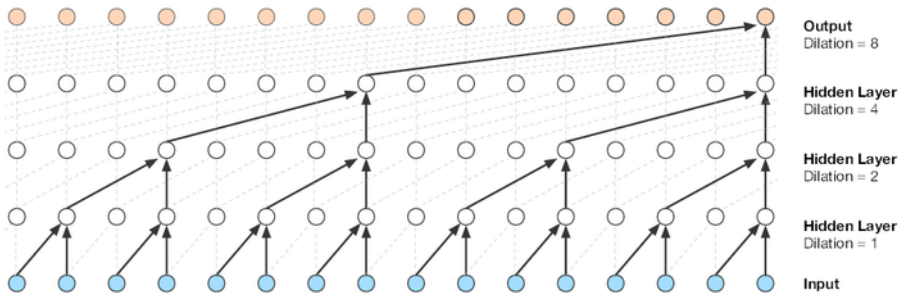

Figure 2.8: Dilated causal convolution in a CNN where dilation rate ranges between 1 and 8. [46].

2.4 Batch Normalization

As mentioned in [47], convergence is proven to be faster when the average of each input variable of the training set is close to zero and they are scaled to have the same covariance. In [47], this operation is referred to as normalizing the inputs. However, input variables may become denormalized as they go through each layer and thus, activation functions of each layer in a NN. To address this issue, there are several learning techniques that performs normalization on the given data were proposed. One of the common normalization techniques is batch normalization.

Batch normalization [48] is a supervised learning technique that converts interlayer outputs of a neural network into a standard format by normalizing. Batch normalization affects the output of the previous activation layer by subtracting the batch mean, and then dividing by the batch's standard deviation. Through batch normalization, the distribution of the output of the previous layer is reset to be more efficiently processed by the subsequent layer.

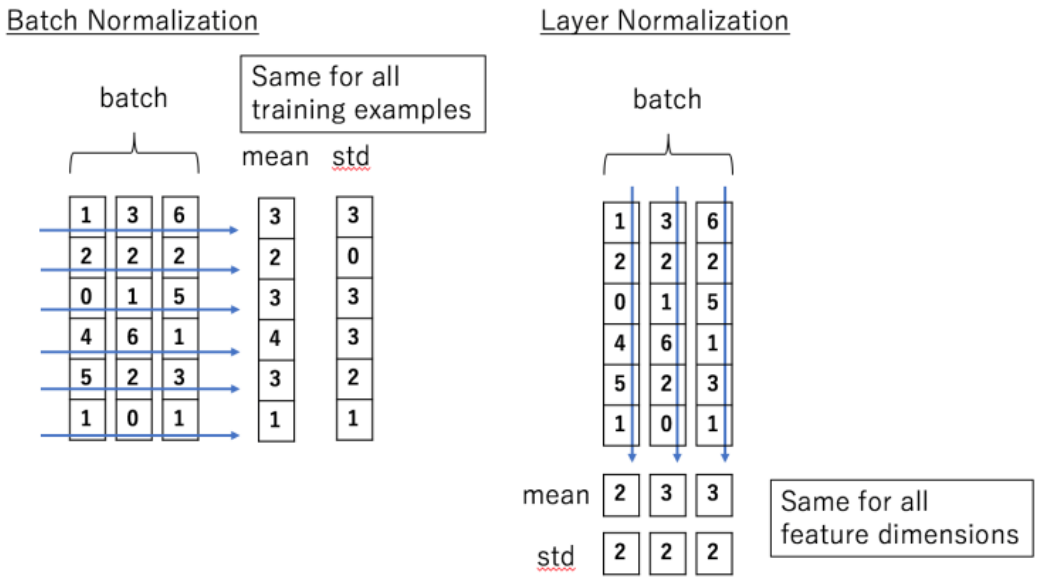

Figure 2.9: Visual representation of normalization operation performed by two different techniques. On the left, the batch is normalized across its batch dimension using batch normalization. On the right, the batch is normalized across its feature dimension using layer normalization [49].

Given that the mini batches are tensors where one axis corresponds to the batch and the remaining axis (or axes) corresponds to the feature dimension, batch normalization normalizes the input features across the batch dimension as shown in Figure 2.9 and the statistics in the batch normalization are the same for each example in the batch. In any non-recurrent network, performing batch normalization to each layer adjusts the incoming scale and mean and therefore, changes in incoming distribution for each layer are avoided [48]. However, applying batch normalization on recurrent NNs raises a storing issue. In a recurrent neural network, the recurrent activations of each time step will have different statistics. Thus, batch normalization layer must be separated for each time step. This requirement results in a more complicated model and necessity of storing the statistics for each time step during training. In [50], these concerns were addresses and an alternative to batch normalization was proposed; layer normalization. Similar to batch normalization, layer normalization is used to normalize the interlayer output

of a NN. In contrast to batch normalization, in layer normalization, normalizing is carried out along each feature dimension as shown in Figure 2.9.

Given that x_{ij} is the i,j-th element of the input where i representing the batch dimension and j representing the feature dimension, the equations of batch normalization is derived as [48]

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij} \tag{2.38}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2 \tag{2.39}$$

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}, \tag{2.40}$$

whereas the equations of the layer normalization are given as [50]

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij} \tag{2.41}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_i)^2 \tag{2.42}$$

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}. \tag{2.43}$$

Since, the statistics are computed across each feature and the statistics are independent of other examples. Due to the independence between input, each input has a different normalization operation, allowing arbitrary mini-batch sizes to be used. Moreover, as experimentally shown in [41], [50] and [51], layer normalization performs well for recurrent neural networks and thus, is facilitated into LSTM-based models of the thesis. Additionally, as a result of the experiments conducted to observe the effects of batch and layer normalization on CNN-based models, layer normalization is also applied to CNN-based models proposed in this thesis. Experiments on normalization layers are further explained in 5.5.1.

2.5 Related Works

There has been a considerable amount of work focusing on time series data analysis [52], specifically in terms of removing noise and smoothing time series for more accurate prediction and regression [53]. In most of the application domains, anomaly detection is considered as an unsupervised problem where the target labels of anomalies are not provided prior to model during training. However, in [54],a Replicator feed-forward NN trained on a dataset to predict the data dependencies was investigated as an instance of using supervised models for an unsupervised problem. The network performed its training by minimizing the reconstruction error of the training data and adjusting the network weights accordingly. Then, the trained model is tested on all the data to compute an outlier factor OF_i, which is the average of reconstruction error overall the input variables also referred to as features, at i-th sample. Using OF, the data was sorted in a descending order where highest rank corresponds to the largest amount of anomalies. The results of testing the model on a datasets containing data for Intrusion and Breast Cancer detection showed that the model was able to flag anomalies where the top 1% of the ranked

data contains all the intrusions, and the top 40 ranked cases contains 77% of all malignant cases which are the anomalies.

In another anomaly detection related study was done in [55], a real-time anomaly detection system was developed for time series data gathered from environmental sensors. In the paper, a method based on autoregressive model of the data and its prediction interval was proposed. Only one step further predictions were considered in the paper using 4 different prediction models: Naive predictor, nearest cluster (NC), multilayer perceptron (MLP) and single layer linear network (LN). A sample was considered to be a normal data point if the corresponding prediction was within the prediction interval,which is computed by the standard deviation of the model residual. Otherwise, the sample is flagged as an anomaly. In another study in data domain, [56] has proposed an approach where autoregressive integrated moving average (ARIMA) model and RNN architecture was combined for anomaly detection in power consumption data. For each method, a predictive model was modelled to calculate the error between the predicted and true consumption values. Error gathered from both models was matched at same time steps in order to predict a potential anomaly. It was reported that the combination of ARIMA and RNN has generated better results in terms of anomaly detection performance compared to their individual performance.

In [57], stacked LSTM networks was proposed to detect anomalies in electrocardiogram (ECG) time signals. A model was trained on normal ECG data for computing an errors vector between the predicted and expected values. Then, the error vectors was fitted to a Gaussian distribution to determine a threshold for classifying normal or anomalous behaviours. According to the results, LSTM models was considered a reliable option for anomaly detection in ECG signals. As mentioned in 1, a similar approach was also pursued in [7] for detecting anomalies in three different time series datasets. Best model performance was observed on power consumption data where model achieved a precision of 0.94 and recall of 0.17. In [58], an LSTM-based model was developed for detection of collective anomalies in an intrusion dataset. A simple LSTM architecture was trained on the given dataset to detect individual point anomalies at each given time step. A circular array that stores the most recent prediction error, was employed to detect the anomalies. The array gives the sum of prediction errors and the percentage of anomalous observations in the data. Furthermore, the metrics are used to determine a threshold to be used for the classification of a sub-sequence as a collective anomaly. LSTM was able to detect collective anomalies with a true positive rate of 86%. In another LSTM related study, a LSTM-based Encoder-Decoder scheme for anomaly detection, called EncDec-AD, was developed. The model was able to learn to reconstruct normal time series behaviours and to use the reconstruction error for anomaly detection. Model was performed on five different dataset; power demand dataset, space shuttle dataset, ECG, and two real world engine datasets with both predictive and unpredictable behaviours. EncDec-AD was able to detect anomalies from a wide variety of time series; predictable, unpredictable, periodic, aperiodic, and quasi-periodic time-series. Additionally, EncDec-AD has performed well on detect anomalies from short time-series with a length as small as 30 as well as long time-series with length as large as 500. In addition to only LSTM-based studies, another study [8] was conducted with the aim of predicting and detecting system anomalies via stacked Bidirectional self-attention LSTM networks in certain time intervals. The model was able to learn and encode entire log messages such as timestamps, Transmission Control Protocol (TCP) statistics, and packet values. Further investigation has resulted that the model has shown a good network performance and outperform existing works, such as LSTM, stacked LSTM and stacked bidirectional LSTM, on a wide range of anomaly detection and prediction tasks.

In [9], a CNN-based anomaly detection approach, called DeepAnt, was proposed. The model was able to detect point anomalies as well as contextual anomalies and anomalous sub-sequences. Moreover, DeepAnt was reported to be robust to a small fraction of anomalies in the training data, which is equivalent to a smaller fraction than 5%, while only being trained on a small amount of data. Considering that CNN has the ability of parameter sharing within convolutional layers, execution time of training and inference on CNN are faster as compared to comparable RNNs [9]. In another CNN related study [59], Temporal convolutional network (TCN), which is a framework which employs casual convolutions and dilations in their convolutional layers, was proposed. The model was first trained on normal sequences to predict the trend in a number of time steps. After training, a multivariate Gaussian distribution was fitted to prediction errors and the anomaly scores of points were calculated using the parameters of the multivariate Gaussian distribution.Results of the experiments showed that TCNs were able to learn inherent patterns in sequential data automatically. Therefore, it was concluded that the model provided a feasible method to learn normal time series behaviours and could be used for anomaly detection. In another related study conducted by Google DeepMind [10], a generative model of a dilated causal CNN operating on audio with a very high resolution was developed. Though as stated in [10], application of this model was for audio generation, possibility of using the proposed model for classification was also discussed in [10].

3 Software in the Loop and Fault Injection

In this chapter, tools employed in SITL mechanism of the simulated UAV are introduced. Additionally, architecture of the flight control software and internal communication systems of the software are investigated, followed by literature research on FI techniques, detailed explanation of the proposed FI environment for the thesis and the type of faults implemented for the experiments.

3.1 SITL

The UAV chosen for this thesis runs on an open-source flight control software called PX4. PX4 is an open-source flight control software for UAVs and other unmanned vehicles [60]. PX4 software provides a modular and configurable control software for UAVs by providing basic navigational functionalities such as landing, taking off, hovering and way point navigation. To validate the controller and also to observe the new behaviour of the system after implementing certain changes inside its control software, testing needs to be performed on the control software. Physical testing with the actual system under actual conditions is expensive, difficult and can also result in physical damage of the system. Therefore, Hardware in the Loop (HITL) and Software in the Loop (SITL) simulations can be performed to aid the testing, verification and validation of the control software without having to resort to pure, fully-physical testing with actual products.

In HITL, two main elements are co-simulated; physical hardware of the component in question and a mathematical model representing the complete system which the physical hardware is a part of. On the other hand, in SITL simulations, instead of a physical hardware, a software simulating the hardware is used together with a mathematical model of the system that the hardware in question is attached to. Testing on HITL offers only limited applicability. Due to safety hazards or costs of assembling a physical component, some reactions of controller to certain faults can only be investigated in a SITL simulated environment [61]. As compared with SITL, HITL facilities are often limited and expensive resources since a computer suffices to complete SITL-based testing. Additionally, there is no requirement of simulating in real-time in SITL and therefore, SITL simulations can run faster than the real-time simulations, allowing comprehensive logic tests, debugging and improvements that generates fast results [61]. Therefore, SITL simulation is explored and implemented in this thesis.

In Fig 3.1, the SITL simulation of a UAV running on a local computer is shown. SITL simulation consists of three main components; control software (PX4), ground control station software (QGround Control) and the simulator (Gazebo). QGroundControl is a software package designed for the applications of monitoring and mission planning for any MAVLink enabled UAV [62]. Gazebo is an open-source robotics simulator utilized for designing robots, environments and perform realistic rigid body dynamics [63]. One feature of simulation in Gazebo is that objects in Gazebo are provided with mass, velocity, friction, and other physical properties that result in more realistic behaviour in simulation. Communication between the components is carried out using either Micro Air Vehicle Link (MAVLink) messaging protocol or simulator MAVLink API.

MAVLink is a lightweight messaging protocol that was designed for communicating with small unmanned vehicles. The protocol defines a number of standard messages and microservices for exchanging data, many of which are implemented in PX4's Firmware [64]. PX4 uses MAVLink to communicate with QGroundControl. MAVLink is further investigated in 3.2.1. On the other hand, simulator MAVLink API defines a set of MAVLink messages that supply sensor data from the simulated world in Gazebo to PX4 and return motor and actuator values from the flight code that will be applied to the simulated vehicle in Gazebo. In addition, the communication between PX4's internal components (modules) is carried out by uORB message bus which is an asynchronous publish-subscribe messaging API [64]. uORB message bus is also explained in detail in 3.2.1.

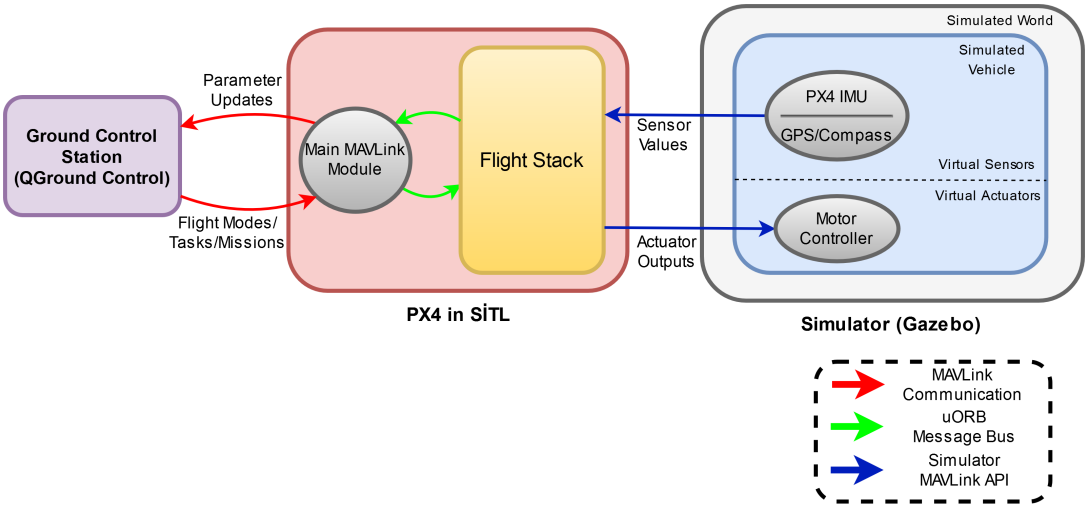

Figure 3.1: Software in the loop simulation running on a local computer [64].

Main processing part of the PX4 control software is the flight stack. Flight stack is a collection of modules that are responsible for guidance, navigation and control algorithms for the UAV. Simulation in Gazebo, on the other hand, is composed of two major parts, simulated environment and the simulated vehicle. During SITL simulation, simulator generates virtual environmental data for virtual sensors and then, the sensor data is transmitted to PX4 via virtual sensor drivers. These sensor values, along with the flight mode information from QGround Control, are taken as input by PX4. Flight mode data transmitted by QGround Control defines how the autopilot responds to user input and controls the vehicle's movement. Flight tasks and missions are used within flight modes to provide specific movement behaviours such as follow me, flight smoothing, orbiting, scanning a designated area etc. PX4 control software transitions between flight modes via input gathered from a ground control station. After the flight mode information is converted to a readable format for other internal modules of the flight stack by the main MAVLink module in PX4, it publishes this data for flight stack to subscribe and consume via uORB message bus. Simultaneously, sensor values from the simulator also fed into the flight stack. As a result of its internal processing, flight stack generates two outputs; actuator controls for the simulator and parameter update for QGround Control. Then, the outputs are directed to the corresponding part of these two components.

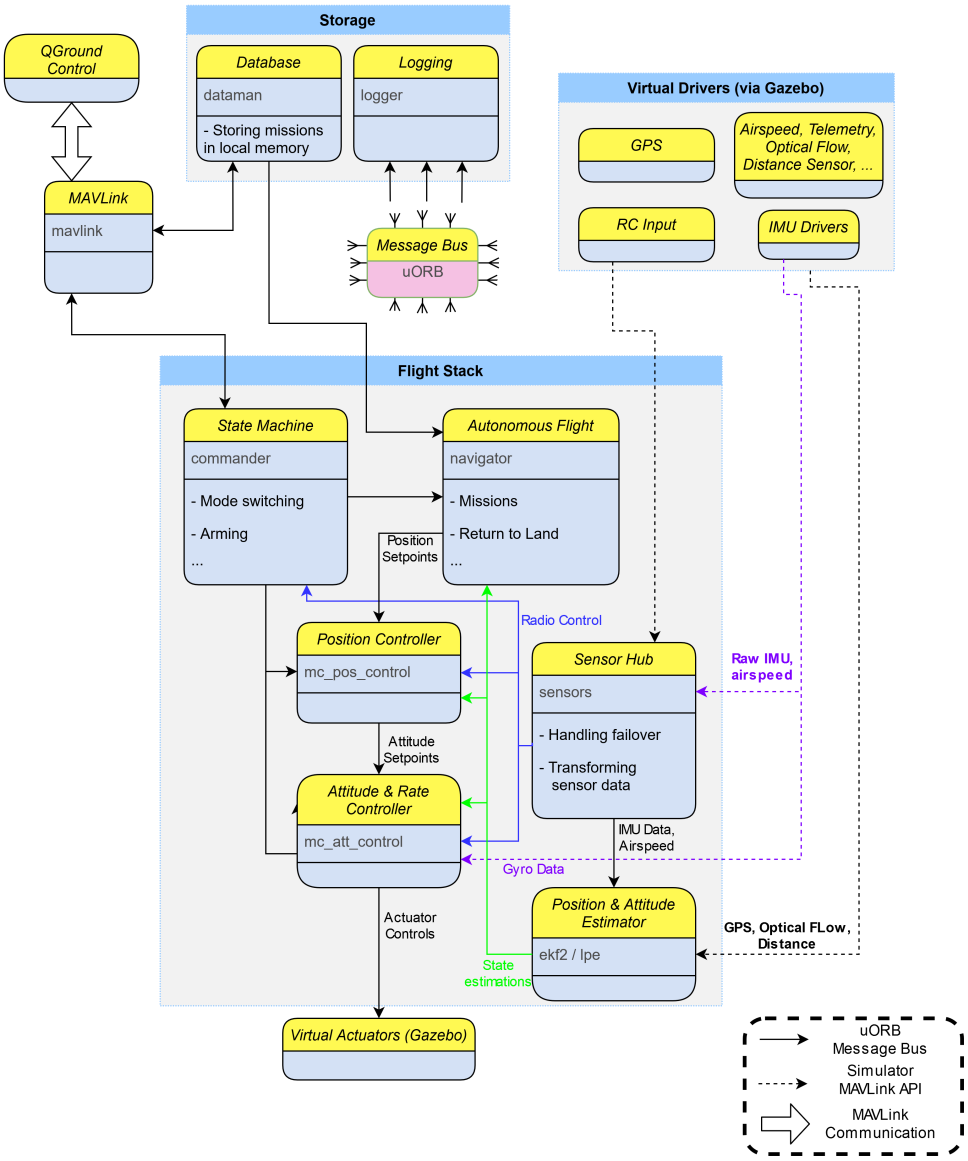

Figure 3.2: PX4 Architecture in SITL [64]

3.2 PX4 Architecture

As mentioned in 3.1, PX4 flight stack is a collection of modules that are liable for motion and control algorithms for the UAV. A detailed explanation of the flight stack and its main modules is given in this section in order to provide the reasoning behind why a particular FI approach was pursued in this thesis. In Figure 3.2, the internal connection mechanism of PX4's modules during SITL simulation is shown along with their interaction with other components in SITL. Module names are presented in a yellow background and their corresponding PX4 firmware module names are written in a light gray colour. Entire inter-module communication is done using uORB message bus module while the inter-component communication is achieved by MAVLink messages.

Mavlink module is the main module for the communication between PX4 flight stack and

QGround Control. It receives flight modes, tasks and missions from QGround Control and transfers them to the corresponding modules. Database module in storage block, dataman, is responsible for storing the mission information, which is initially gathered from QGround Control and fed into PX4 via mavlink module, inside the local memory. The other module in this block, logger, logs any ORB (object request broker) topics with all included fields.

Flight stack receives input from mavlink module, virtual drivers in Gazebo and if a mission is requested and stored in the local memory, also from dataman module. Input from mavlink module contains flight mode information and consumed by commander module. Commander module is the main module that overlooks the entire control algorithm modules and it can interrupt the system immediately when such input is read from QGround Control. Other input of the flight stack is transmitted by virtual drivers in the simulator Gazebo. Virtual drivers sample the sensor data at 1kHz, integrate it and then publish it at the rate of 250Hz [64]. Since the modules wait for the message update, the drivers define how fast a module updates. After a message update transmitted by the drivers, sensor values are consumed by position & attitude estimator and sensor hub modules. Estimator modules, ekf2 and lpe, use an Extended Kalman Filter (EKF) algorithm to process sensor measurements and provide necessary state estimates for the controller modules while sensor hub module, sensors, transforms the sensor data and handles failover in case of any initial failure during SITL simulation. Moreover, another possible input of the flight stack is the flight mission information stored by dataman module when a mission is defined and transmitted by QGround Control. In this case, autonomous flight module, navigator, obtains flight mission information together with state estimations from estimator modules and flight mode information from commander module. As a result of its internal processing, navigator outputs position setpoints for the position controller module. Controller algorithm in the controller modules works using a cascaded loop method [64]. For instance, in the controller inside the attitude & rate controller module, the outer loop computes the error between the attitude setpoint vector and the estimated attitude vector that, multiplied by a gain factor, generates a rate setpoint vector. The inner loop then computes the error in rate vectors and uses a proportional integral (PI) controller to generate the vector of desired angular acceleration. After that, actuator controls are generated through mixer [64]. Final output of the flight stack, actuator outputs, is then sent to the virtual actuators inside the simulator.

3.2.1 Communication Protocols

Point-to-point Communication

As illustrated in Figure 3.3, the point-to-point channel pattern allows only a single receiver to consume a message send by the producer. In case of multiple receivers trying the consume the message, point-to-point channel ensures that consuming attempt of only one of the consumers will be successful. However, multiple receivers can be included in the channel and these receivers can received multiple messages concurrently with the criteria of only one receiver receiving a specific message.

Publish/Subscribe Communication

Publish/subscribe messaging is a form of asynchronous service-to-service communication generally used in serverless and microservices architectures. In a publish-subscribe messaging model, subscribers of a topic instantly receive the messages once the messages are published to a topic. In Figure 3.4, a publish-subscribe messaging model with two publishers and three subscribers is

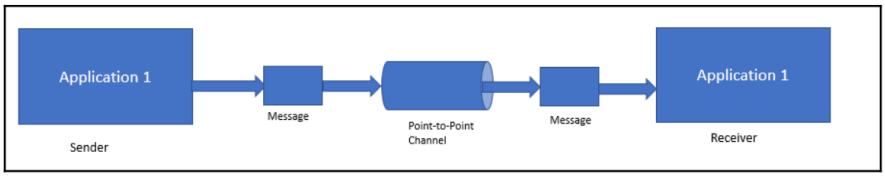

Figure 3.3: Schematic representation of a point-to-point communication[65].

illustrated.

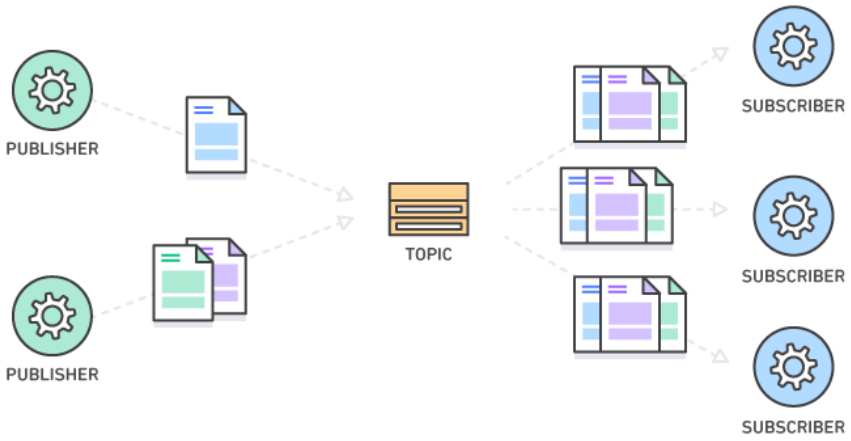

Figure 3.4: Schematic representation of a publish/subscribe communication[66].

In a publish-subscribe messaging model, messages are asynchronously broadcast to different parts of the system. Message topics are utilized in publish-subscribe models. A message topic provides a lightweight mechanism to broadcast asynchronous event notifications and also endpoints by which software components connect to the topic. Through this connection, the components can send and receive messages. A message is broadcast by a component, called publisher. By sending a message to the topic, publisher broadcast a message. While message queues batch the received messages until they are fetched, there is no or very little queuing involved in message topics when the topic transfer the messages. Once message topics receive the messages, they simply push the messages out immediately to all subscribers. Thus, every broadcast message is received by all component that subscribe to the topic.

In a publish-subscribe messaging model, different functions are often performed by the subscribers of a message topics and thus, different functions with the same message can be performed in parallel. When a message is broadcast, publisher information is not included in the published information and therefore, subscribers receive no information about the publishers. Similarly, publishers are also uninformed about the subscribers. Due to this structure, publish-subscribe messaging model differs from the message queues which is another form of asynchronous service-to-service communication pattern where destination of a message is known by the component that send the message.

uORB

The uORB is an asynchronous publish-subscribe messaging API. The uORB is used during the communication between the internal threads and processes of PX4. Modules introduced in 3.2 are facilitated as publishers and subscribers. In PX4, an uORB topic can have multiple publishers as well as multiple subscribers.

In Figure 3.5, two topics with multiple publishers and subscribers are illustrated where dashed line indicates publishing and straight line subscribing to a topic. Both navigator and mc_pos_control modules can send message to the topic "vehicle_attitude_setpoints" which is subscribed by mavlink module and mc_att_control module where received messages are used to compute the actuator outputs. Another topic illustrated in Figure 3.5 is the topic "vehicle_land_detected". While land_detector and mavlink modules publishes messages to the topic, other associated modules receive these messages. Another attribute of this topic is that mavlink module acts as both publisher and subscriber of the topic. Since the publishers and subscribers are decoupled in the publish-subscribe messaging model, publishers and subscribers work independently from each other. Therefore, one module can both be a publisher and a subscriber of a topic.

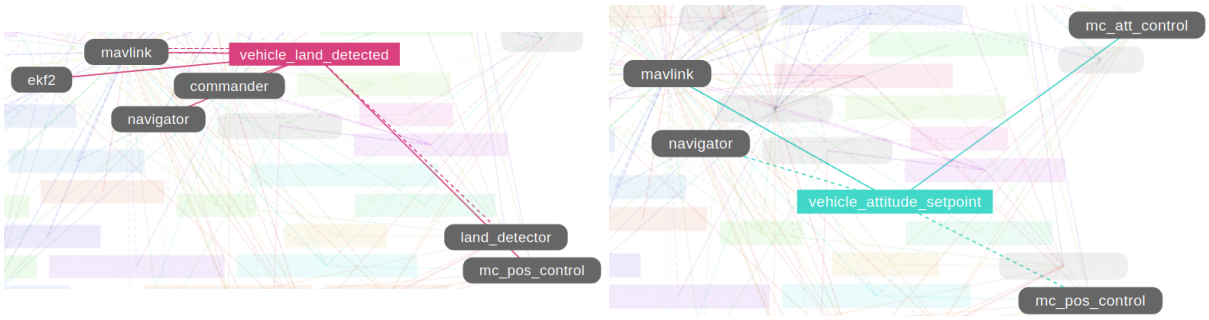

Figure 3.5: Two topics from PX4 with their publishers and subscribers[64].

MAVLink

MAVLink is a lightweight messaging protocol designed for the drone ecosystem [67]. MAVLink is designed as a hybrid protocol of publish-subscribe and point-to-point design patterns where data streams are sent/published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission. Figure 3.6 illustrates the uORB topics implemented in mavlink module of PX4 that are used to transfer the data streams between PX4 and QGroundControl.

On the other hand, configuration sub-protocols, also referred to as microservices, define higher-level protocols implemented in MAVLink systems for better inter-operation. Many types of data are exchanged via microservices, including parameters, missions, trajectories, images and other files. Though the microservices include several sub-protocols, two main protocols of MAVLink is further explained in this section; mission protocol and parameter protocol.

Via the mission sub-protocol, QGroundControl exchanges mission data with a UAV in simulation. Types of missions implemented in MAVLink systems are flight plans, geofences and rally/safe points. In Figure 3.7, the diagram of communication sequence to upload a mission to a UAV is shown where GCS corresponds to QGroundControl software used in the SITL simulation of this thesis and name of protocol-related messages are written in capital letters such as

Figure 3.6: uORB topics of mavlink module implemented in PX4 [64].

MISSION_COUNT and MISSION_REQUEST_INT.

Uploading a mission to a UAV in simulation is executed as follows [67]:

1. QGroundControl sends MISSION_COUNT which is the variable containing the number of mission items to be uploaded. Since the communication is carried out following a point-to-point protocol, a timeout is started for QGroundControl in order to wait on the response from the UAV.

2. UAV receives message and responds with MISSION_REQUEST_INT requesting the first mission item. Following the UAV's response, a timeout is started for the UAV in order to wait on the MISSION_REQUEST_INT response from the QGroundControl.

3. After receiving MISSION_REQUEST_INT, QGroundControl responds with the requested mission item in a MISSION_ITEM_INT message.

4. The MISSION_REQUEST_INT/MISSION_ITEM_INT cycle is repeated until every item is uploaded.

5. The UAV receives the last mission item and responds with MISSION_ACK with the type of MAV_MISSION_ACCEPTED indicating successful upload of the mission.

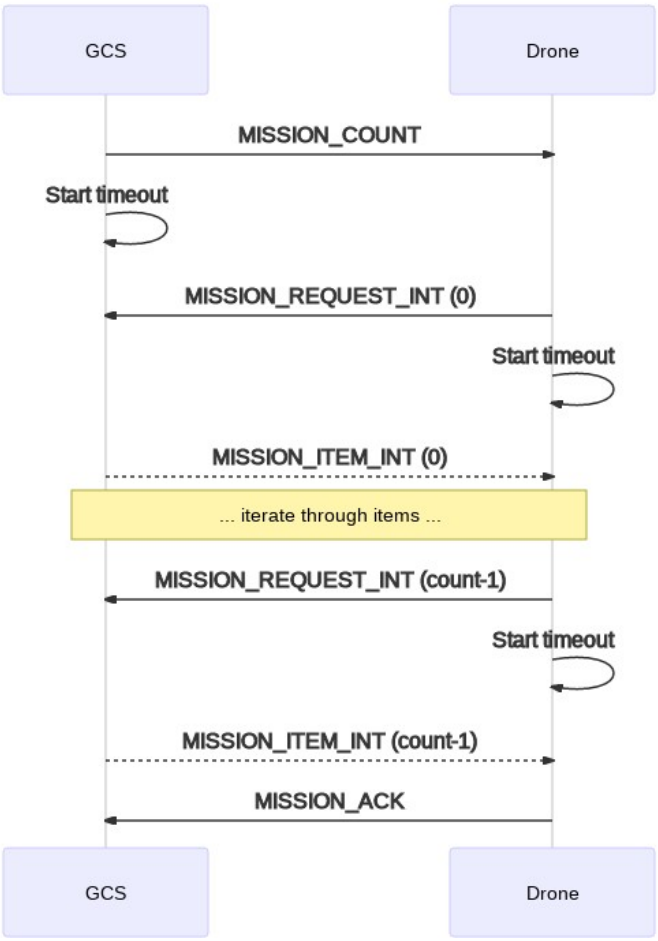

Figure 3.7: Communication sequence to upload a mission to a UAV [67].

6. Lastly, QGroundControl receives MISSION_ACK containing MAV_MISSION_ACCEPTED indicating the completion of the operation.

Another microservice is parameter protocol which is used in exchanging configuration settings between PX4 and QGroundControl. Each parameter in the protocol is represented as a key/value pair. While the key is the human-readable name of the parameter, a values is the corresponding value written as an integer or a floating number. In Figure 3.8, communication sequences for reading a parameter from a UAV and writing a parameter to a UAV is presented where the same notation for the name of the protocol-related messages is also utilized as in Figure 3.7.

The sequence of operation for reading a parameter from a UAV is as follows [67]:

1. QGroundControl sends PARAM_REQUEST_READ which specifies either the parameter id (name) or parameter index
2. A timeout for QGroundControl is started to wait on the information in the form of a PARAM_VALUE message
3. UAV responds with PARAM_VALUE which contains the parameter value.

QGroundControl writes a parameter to a UAV following the sequence of operation below [67]:

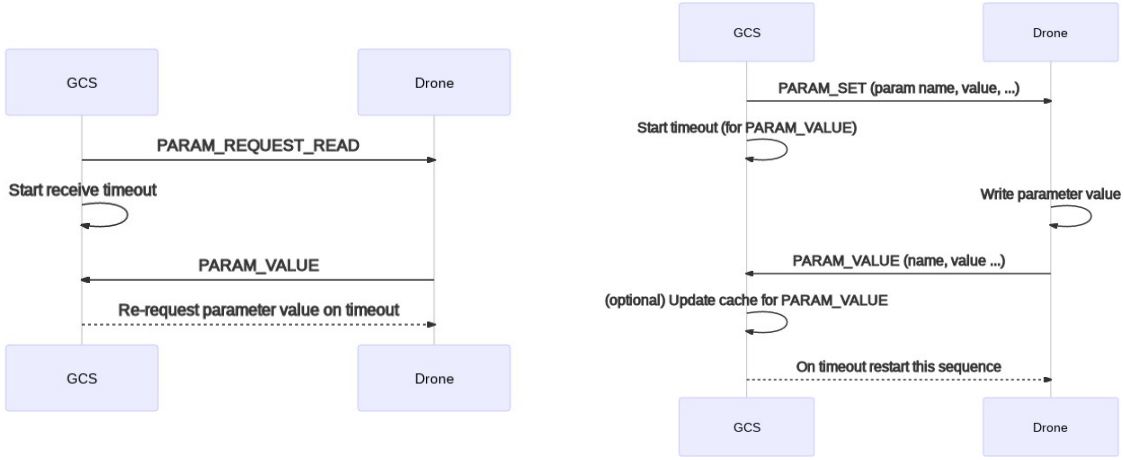

Figure 3.8: On the left side, the diagram for the communication sequence to read parameter from a UAV is shown whereas on the right side, communication sequence to write a parameter to a UAV is illustrated [67].

1. QGroundControl sends PARAM_SET that specifies the name of parameter to be updated and its new value
2. A timeout for QGroundControl is started to wait on the information in the form of a PARAM_VALUE message
3. After writing the parameter, UAV responds with broadcasting a PARAM_VALUE, which contains the updated parameter value, to all components/systems.
4. As an optional step, QGroundControl updates the parameter cache with the new value if QGroundControl uses the parameter cache.

3.3 Fault Injection

A fault is described as an event inside the system which is suspected to have caused a failure [68]. According to [69], faults can be divided into three categories based on their pattern of manifestation; permanent, intermittent and transient. Permanent faults corresponds to persistent faults which continuously affect the system functionality. Permanent faults remain in the system indefinitely until they are corrected or repaired. Stuck-at faults which occurs as a result of a short or open circuit where the lines, that always carry logical signals, are stuck at one logical signal, either ”0” or ”1”, is an example of a permanent fault. Intermittent faults are the faults that occurs and reoccurs in a system at irregular intervals. Intermittent faults are often early indicators of forthcoming permanent faults. Intermittent faults are usually transformed to permanent faults over time. Finally, transient fault refers to the faults that only occurs at a certain time and transient fault are most likely to disappear once the affected part of the system is re-initialized. Bit flips is an example of a transient fault. Permanent faults are beyond the scope of this thesis and thus, simulating only the transient and intermittent faults are considered in this thesis.

Fault injection refers to any hardware-level modification that may cause a change in normal program execution [70]. Fault injections can either be intentional or unintentional. In intentional fault injection, an attacker performs the injections to change the program execution. For instance, in [71], an FPGA-based, which is a semi-conductor device designed to be reprogrammed

to desired application by a customer or designer after manufacturing, transient fault injection system was introduced and employed to perform fault injection attacks on microprocessor of a software filter used in aviation for probabilistic sensor data fusion. On the other hand, unintentional fault injection is generally associated with the environment [72]. One of the first observed fault injections is an example of an unintentional fault injections where radioactive elements existent in packing materials caused bits to flip in chips [73].

3.3.1 Fault Injection Model

The FARM model, proposed in [74], serves as an effective approach to characterize a fault injection environment. FARM model consists of for main attributes:

- The set of faults F: F is comprised of faults that are intentionally injected into the system. Faults are characterized by their type/model (e.g., transient, permanent, etc.), a location and injection time. Therefore, fault space size is $M \times L \times T$ where M,L and T respectively denote the set of possible fault models, the set of possible fault locations and the set of possible fault injection times. Unlike the traditional hardware testing in which the fault space size is $M \times L$, fault space size is often assumed to be infinite in a fault injection experiment. Due to infinite sized fault space, working with comprehensive fault lists are considered impossible in a fault injection experiment [75]. Thus, the main issue with defining F is selecting a subset fault space derived from the entire fault space where this subset is injected in a reasonable time and able to produce statistically important results.
- The set of activation trajectories A: A specifies the activation domain used to functionally exercise the system during the experiment. The set of functional inputs fed into the system during each experiment is also established by A. Thus, choice of A has a direct influence on the length of the experiments and therefore, the fault space size ($M \times L \times T$) and target fault list. FARM model is also often improved by the addition of the set of Workloads W. A set of software benchmarks is an example of W when the target system is a microprocessor-based system.
- The set of readouts R: R refers to the system's behaviour logged for each fault injection experiment to characterize the system's behaviour in present of faults. Target system and the mechanisms used for observing the behaviour of the system strongly affect the recorded data in R. To illustrate,in a microprocessor-based system, memory accesses or the system exceptions may be included in the data recorded in R [75].
- The set of measures M corresponds to the experimental measures, such as latency estimate and fault dictionary entries. M is obtained by analyzing and evaluating the elements of the FAR sets.

In Figure 3.9, basic components of a fault injection environment based on FARM model is illustrated. In the FARM model, a fault injection campaign is a collection of experiments where each experiment needs a injection of a fault f of the set F while an activation trajectory a from A exercise the system in a workload w from W. M is acquired by processing the R collected during each experiment.

3.3.2 Fault Injection Techniques

According to [76], three main fault models and FI techniques are commonly used: Hardware-based models, software-based models and simulation-based models.

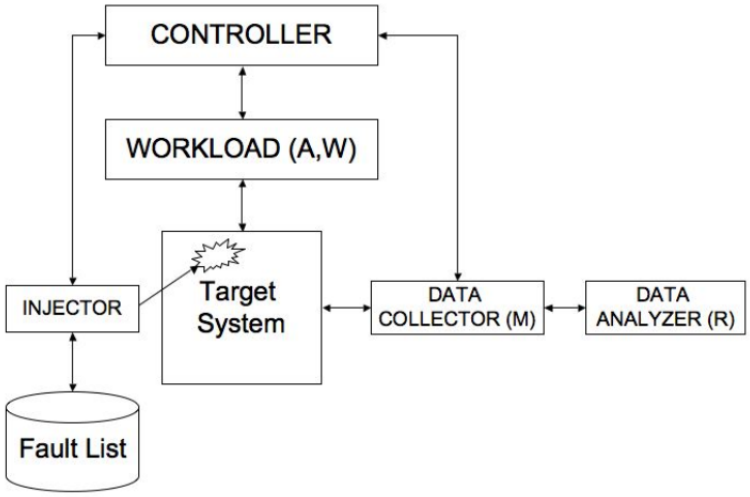

Figure 3.9: A typical fault injection environment[75]

Hardware-based FI is consist of two sections; FI with and without physical contact [77, 13]. In the latter sub category, FI results in a similar to the experiences of a device in a space environment whereas with physical contact, voltage and current changes is introduced to the DUT using either a active probes or socket insertion. Main drawback of hardware-based FI is to cause possible damage on the DUT and the necessity of modifying the DUT.

Since software-based FI shows no risks of damaging the DUT, does not require any hardware modifications and provides portability, it is a popular choice among FI techniques [13]. However, software-based FI shows disadvantages over hardware-based FI. One of the disadvantages is software's inability to access certain components, such as caches, for injection. Moreover, possibility of disturbing the processing workload in unintended ways arises when software-based FI is applied. For instance, the scheduling and timing of system tasks may be altered when additional software required for performing injection is added [77].

Simulations are also used for fault injection. Using simulation-based FI, where faults are injected in a simulation model, shows several advantage over FI techniques injecting faults in a physical system. Since simulations operate at different levels of abstraction, multiple fault model can be used in a simulation-based FI. Fault injections then becomes transparent from the target system's point of view, since fault-injection mechanisms and system-simulation models are tightly integrated [78]. Moreover, the most visibility into and control over the FI mechanism and the target is provided in simulation-based FI among mentioned FI models [78]. Taking its advantages over other FI techniques and its less expensive requirements, simulation-based FI is used in this thesis.

There are two types of injection used by the FI techniques: runtime FI and compile-time FI. In run-time FI, faults are injected while target system is loaded and running whereas in compile-time injection, faults are injected by modifying the source code of the target system before the target system is loaded and executed. Due to fault effect being hard-coded inside the source code, it is mostly used to emulate permanent faults. Since permanent faults are excluded in this thesis, only run-time type fault are used.

During runtime, FI needs to be triggered. Three different triggering mechanism that are commonly employed in FI techniques are time-out, software trap and code insertion [79].

- Time-out. In this technique, a timer is set to trigger injections at a predetermined time. Specifically, the time-out event generates an interrupt to trigger a FI. The timer used in this project is a software timer embedded in the source code. Due to injecting faults on a time-basis rather than event or system state based faults, this technique results in unpredictable faults effects and system behaviour. However, it is considered an appropriate technique for imitating transient and intermittent faults.
- Software Trap. In this technique, a software trap transfers control to the fault injector. Unlike time-out, in software trap, fault is injected whenever particular events or conditions happen. For example, a trap instruction inserted into the source code will trigger the FI before the program executes a certain instruction. As a result of the trap execution, an interruption is generated and the interruption transfers control to a fault injector.
- Code Insertion. In this technique, instructions are added to the target system in order for FI to occur before particular instructions, much like the software trap method. This technique differs from software trap in that the fault injector can exist as part of the target system since there is no control transfer as in software trap.

3.3.3 Related Work

As mentioned in the previous section, fault injection methods are divided into two main class; hardware-based (physical) and software-based fault injection methods. Software-based methods are further divided into two sub-categories; software-implemented fault injection where change of data and the timing of an application is controlled by software while running on a real hardware, and simulation-based fault injection, where behaviour of a target system is modelled and emulated using simulation. In this section, previous studies related only to the simulated-based fault injection is further explained since simulated-based method is implemented in this thesis.

One of the proposed simulator for simulation-based fault injection is a simulator called LIFTING [80]. LIFTING is an open-source fault simulator with an object-oriented architecture where both logic and fault simulation for stuck-at faults and single event upset (SEU), which is essentially an alter of state caused by one single ionizing particle strikes that discharge the charge in storage elements [81], on digital circuits can be simulated. LIFTING based on an event-driven logic simulation engine and performs a fault injection via fault simulation. LIFTING differs from other fault injection tools due its capability of providing features for analysing the fault simulation results, which is useful for research purposes. To simulate faults in complex microprocessor-based systems, the hardware system including the memory can be described via LIFTING to define the software stored in the memory, and to inject faults in all elements of the hardware model.

Another tool, called MEFISTO, employing the simulation-based fault injection method is proposed in [82]. Since MEFISTO is simulation-based, it provides an advantage of high level of controllability over where and when a fault is injected. Another advantage of using a simulation-based tool is that set of fault models or component types to simulate are not limited to any predefined sets as users can model specific behaviour using the simulator. Faults can either injected in three ways; using the command language of the simulation engine, using mutants modelling a behaviour different to model component, that is fault-free, or using saboteurs changing signals and variables.

To address to major problem of using simulation-based fault injection which are time and effort needed for developing a simulation model and performing simulation, a Mixed-Mode Fault

Injection, where software-implemented and simulation-based fault injection techniques are combined, is proposed in [83]. Main idea is to combine software-implemented and simulation-based fault injection methods to utilize their individual advantages of these techniques. By using software-implemented fault injection, hardware/software state of the system under investigation (SUI) can be modified under software control, thus an abnormal system behaviour can be forced to the system. However, software-implemented fault suffers from its inability to inject faults to locations that are not accessible by the software. To overcome this issue, simulation-based fault injection can be applied since it offers high level of controllability over the SUI. Components of SUI that are inaccessible by the software are simulated via a model of that component. Therefore, in a simulated model, data and timing of an application can be changed, and any kind of fault can be injected into a SUI. It was reported in the paper that Mixed-Mode Fault Injection approach was fast and accurate since fault simulation is restricted only to the fault injection phase while otherwise real hardware was used [83].

3.3.4 Proposed FI Model

Figure 3.10 shows the FI environment model adopted this thesis. The environment model is based on the FARM model with an addition of a monitor element to the FARM model. The environment model consists of the target system, a fault injector, fault library, controller, simulation/workload generator, monitor, data collector, and data analyser. The fault injector is a custom-built software module implemented inside the PX4's Firmware and was designed to support and implement the desired fault types and scenarios which are drawn from the fault library. The fault injector injects faults into the target system over uORB message bus while the target system executes commands from the simulation/workload generator. The monitor is responsible for tracking the execution of the commands and initiating data collection which is essentially logging the data gathered from the target system. Though the data collector performs data collection during the experiments, the data analyser, which is designed to perform data processing and analysis, is the ANN models. The ANN models remain off-line during the experiments. After an experiment is complete, ANN architectures analyse the data collected by the data collector. The controller controls the experiment.

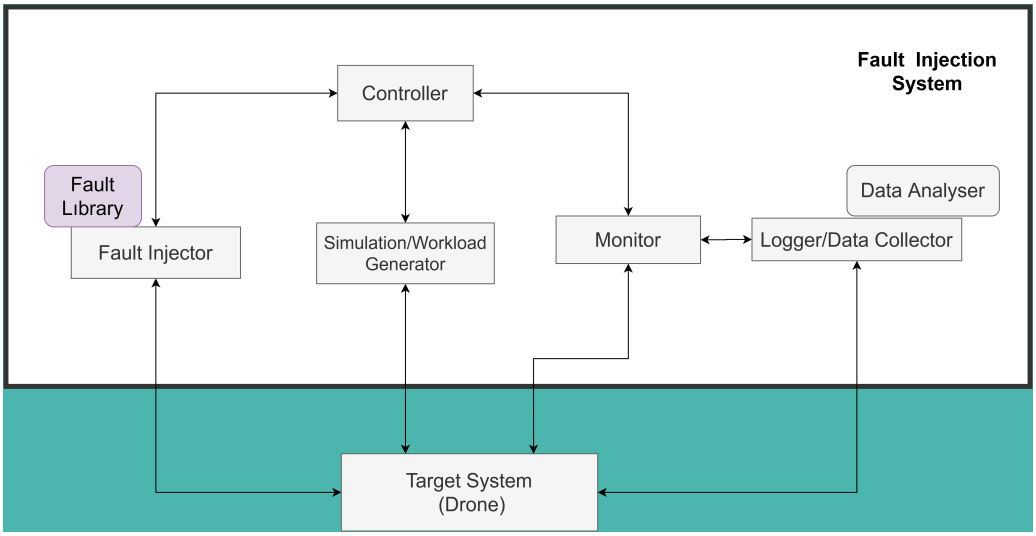

Figure 3.10: Fault injection environment

Controller mechanism implemented in PX4's firmware is designed to handle certain type of faults such as infrequent misreadings of the sensors. Moreover, "sensors" module is able to perform failover in case of a failure or unexpected termination of an active application. Though these two features provides a wide-scale fault handling capability, the faults which the system is exposed to during a flight are not addressed or examined in any particular way but rather only taken care of when they occur. By applying simulated faults into a UAV, this thesis aims to imitate the scenarios where a UAV faces a failure or unexpected fault in its system during its operation and to store the data for a further analysis. Main objective of this analysis to successfully detect the faults that are present in the given data.

In this thesis, 2 fault scenarios were simulated. Triggering mechanism of time out is employed for these scenarios. Each scenario is repeated for a certain number of times in order to acquire an exhaustive dataset for the further step of the analysis.

- Rapid jumps or drops in sensor reads (F1): In this FI scenario, rapid jumps and drops in the reading of a pre-selected sensor parameter is simulated after a certain time. Though the number of faults injected into the target model is manually decided and implemented into the source code, sensor selection is carried out by the computer using a random number generator. Before running the simulation, random number generator generates a value within the range of total number of sensor parameters of UAV. Additionally, a timer is initialized for initiating of the fault injections. Triggering time of the timer is randomly decided by the computer given the number of faults to be injected. Timer is restarted after each FI. Algorithm of this fault scenario is represented in 1 where $:=$ sign denotes an assignment operator whereas $=$ sign denotes a boolean operator to test equality of values.

- Sensor confusion (F2): In this FI scenario, sensor reading of two pre-selected sensors are mixed, where a sensor reading from one sensor is fed to the other sensor after a certain time. Similar to the previous fault scenario, sensors are selected by the computer using a random number generator. A timer is also initialized for initiating of the fault injections. Number of faults are manually decided and added to the fault library. Triggering time of the timer is randomly decided by the computer given the number of faults to be injected and timer is restarted after each FI. Corresponding algorithm is provided in 2.

Algorithm 1 F1 algorithm

```
1: procedure FAULT INJECTION
2:     initialization;
3:     n_faults ← number of faults
4:     rand_gen ← random number generator
5:     timer
6:     sensors[ ] ← array of sensors
7:     time_trigger ← time to trigger the injection
8:     fault_rate ← rate of the drop or jump in sensor readings
9:     start;
10:    time_trigger := rand_gen.generate(n_faults)
11:    timer.start()
12: loop:
13:    while n_faults ≠ 0 do
14:        if timer.read() = time_trigger then
15:            sensor_no := rand_gen.generate(range(len(sensors[ ]))).
16:            fault_rate := rand_gen.generate(range(10))
17:            if n_faults is even then
18:                sensors[sensor_no].write() := sensors[sensor_no].read() * fault_rate.
19:            if n_faults is odd then
20:                sensors[sensor_no].write() := sensors[sensor_no].read()/fault_rate.
21:            n_faults := n_faults − 1
22:            timer.restart()
23:    close;
```

Algorithm 2 F2 algorithm

```
1: procedure FAULT INJECTION
2:     initialization;
3:     n_faults ← number of faults
4:     rand_gen ← random number generator
5:     timer
6:     sensors[ ] ← array of sensors
7:     time_trigger ← time to trigger the injection
8:     start;
9:     time_trigger := rand_gen.generate(n_faults)
10:    timer.start()
11: loop:
12:    while n_faults ≠ 0 do
13:        if timer.read() = time_trigger then
14:            sensor_no1 := rand_gen.generate(range(len(sensors[ ]))).
15:            sensor_no2 := rand_gen.generate(range(len(sensors[ ]))).
16:            sensors[sensor_no1].write() := sensors[sensor_no2].read().
17:            n_faults := n_faults − 1
18:            timer.restart()
19:    close;
```

4 Proposed Approach

The anomaly detection method proposed in this thesis has two main steps. In the first step, a time series prediction model is trained to learn time series patterns and predict future time steps. Then, anomalies are detected by computing anomaly score from the prediction errors. In this chapter, mathematical notation of the input sequence is first introduced to achieve uniform notation for the illustrations. Following that, five different ANN models proposed in the thesis are explained and illustrated. Lastly, the anomaly detection algorithm and evaluation metrics used in the thesis is described.

4.1 Time Series Predictors

Input sequence fed into the NN models is a multivariate time series sequence which consist of multiple channels where each channel corresponds to parameter extracted from the drone in SITL. Detailed explanation of the parameters are provided in 5.2.

Given a sequence with T consecutive time steps, input sequence matrix X at time t is written as

$$X_{t-1} = \{\vec{x}_{t-T}, \vec{x}_{t-T+1}, \ldots, \vec{x}_{t-2}, \vec{x}_{t-1}, \}, \tag{4.1}$$

where T denoting the number of time steps and $\vec{x}_t$ denoting the multi-variate time series sequence at time step $t-1$. Given a input matrix with T consecutive time steps, models are expected to generate a prediction of adjacent vector at time step t.

$$\vec{x}_{t-T}, \vec{x}_{t-T+1}, \ldots, \vec{x}_{t-2}, \vec{x}_{t-1} \rightarrow \vec{x}_t. \tag{4.2}$$

Mean squared error (MSE) 2.9 is applied as the loss function in every model to calculate the difference between the ground truth value $\vec{y}$ and the predicted output $\hat{\vec{y}}$. In order to determine optimal values for the design parameters of each model, design space exploration of the models are further inspected in 5.4.

4.1.1 1-Dimensional CNN

Figure 4.1 illustrates the process of proposed a 1-D CNN model generating a prediction for the time step t given a multi-variate input sequence with T time steps. The model has two 1D convolutional layers, each followed by a max pooling layer. Each convolutional layer is grouped consecutively with a max pooling layer in order to give the model a good chance of learning features or patterns from the input data. Before each max pooling layer, layer normalization is applied to the output of the convolutional layers. The reasoning behind choosing layer normalization over batch normalization is further explained in 5.5.1. Convolutional layers consist of F number of filters with a kernel size of K, followed by an element-wise activation function ReLU 2.7. Number of filters and size of the kernels applied in each convolutional layers are design parameters that are further inspected in 5.4 in detail.

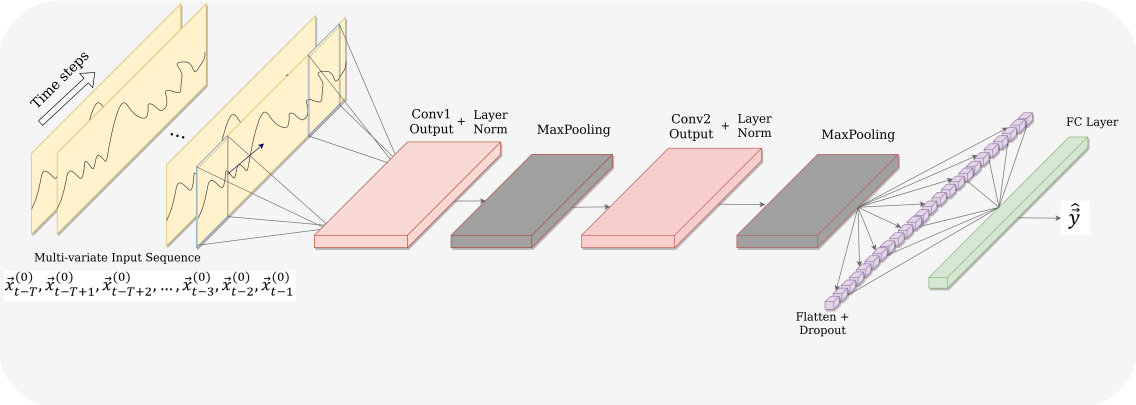

Figure 4.1: 1-D CNN Model

While the convolutional layers initiate the general learning process, the max pooling layers reduce the learned features to a certain size derived by the design parameter pooling window P and therefore, consolidating the features to only the most essential elements. Last layer of the model is a fully connected layer in which each neuron is connected to all the neurons in the previous layer by flattening and converted into a single vector. Fully connected layer is a basic feed-forward NN layer, also referred to as dense layer. This layer computes the network prediction for the next time step applying a linear activation function to the flattened vector. The fully connected layer ideally acts as a buffer between the learned features and the output with the intent of interpreting the learned features before the prediction was carried out. However, the fully connected layers tend to overfit on the training data, therefore damage the generalization ability of the overall network. To counter this problem, a dropout layer is added after flattened layer as a regularizer which randomly sets half of the activations to the fully connected layers to zero during training. Addition of a dropout layer results in improvement in the generalization ability of deep convolutional neural networks by preventing overfitting [84].

4.1.2 Temporal 1-Dimensional CNN

Proposed temporal CNN model is illustrated in Figure 4.2. Model is formed of 4 consecutive residual block followed by a 1-D convolutional layer and a global average pooling layer which generates the prediction for the adjacent time step. Residual block used in this model is inspired by the residual block introduced in [85]. Similar to residual block in [85], input data of the block goes through a dilated causal convolution operation, followed by a normalization layer, ReLU activation and a dropout layer for regularization. This process is performed consecutively twice in a residual block. Considering that in TCN, the input and output could have different widths, in order to account for discrepant input-output widths, an additional 1×1 convolution is introduced to the output of the second dropout layer to ensure that element-wise addition receives tensors of the same shape.

Model proposed in the thesis differs from the residual block in [85] in terms of the normalization layer and dropout layer since in this thesis, layer normalization and global dropout layer are implemented in the model as opposed to weight normalization and spatial 1-D dropout layer in [85].

Another addition of this model is to introduce a combination of a convolutional layer and a global average pooling layer, similar to the network proposed in [86], instead of fully connected

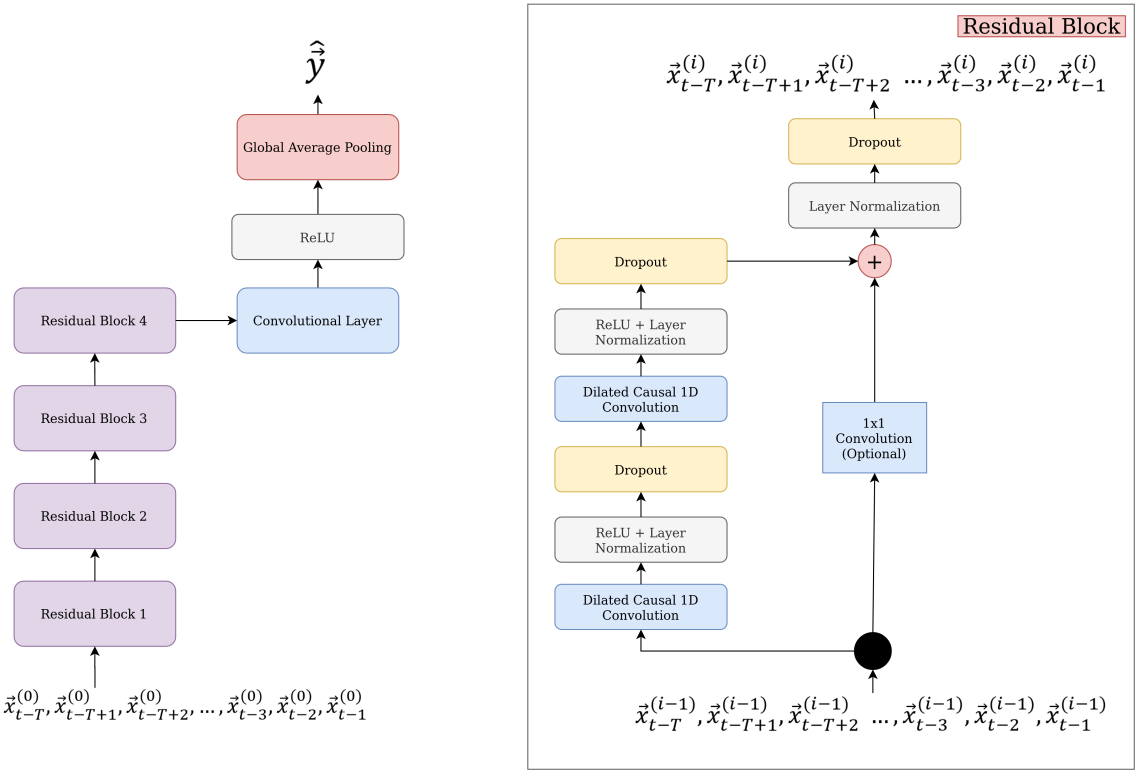

Figure 4.2: TCN Model

layers commonly used in CNNs. The objective of replacing fully connected layers with a combination of a convolutional layer and global average pooling layer is to produce one feature map for each corresponding output value. Thus, model take advantage of each feature map and the resulting vector is fed directly into the last activation layer which is linear in above case. As reported in [86], one advantage of global average pooling over the fully connected layers is that global average pooling is found to be affiliated with the convolution structure more by enforcing correspondences between feature maps and categories. Thus, the feature maps can be easily interpreted as categories of confidence maps. Moreover, global average pooling has no parameter to be optimized and therefore, overfitting is avoided at this layer. Representation of the parameters transmitted between layers was carried out following the pattern where in $\vec{x}^{(i)}_{t-1}$, i indicates the i-th layer and $t-1$ the corresponding time step in the input data sequence of the corresponding layer.

4.1.3 LSTM

In Figure 4.3, snapshot of ongoing process in a LSTM-based model whose task is to generate a prediction for the time step t given a time-series sequence with T time steps as input sequence data.

The model consists of a stacked LSTM layer where hidden states of the primary LSTM layer is fed into subsequent LSTM cells of the secondary layer. Layer normalization and a dropout are applied before transferring the hidden states to the secondary LSTM layer. As discussed in 2.4, layer normalization is proven to be very effective at stabilizing the hidden state dynamics in RNNs. As concluded in [50], layer normalization can substantially reduce the training time

compared with RNN models where layer normalization is not applied. Additional dropout layer is applied to the output of the the secondary LSTM layer, which the last hidden state of the layer, in order to reduce overfitting and improve the model's generalization error. In addition to hidden state vectors, cell states vector $\vec{s}$ is transmitted through the cells in each corresponding layer. As explained in 2.2.1, cell states were only shared among the cells over the recurrent edge. Last layer of the model is a dense layer which is basic feed forward layer with a linear activation function. This layer computes the model's vector prediction for time step t. Same notation as in Figure 4.2 is used in Figure 4.3 where i and $t-1$ in $x_{t-1}^{(i)}$ denoting the order of layer and the corresponding time step in the the data sequence.

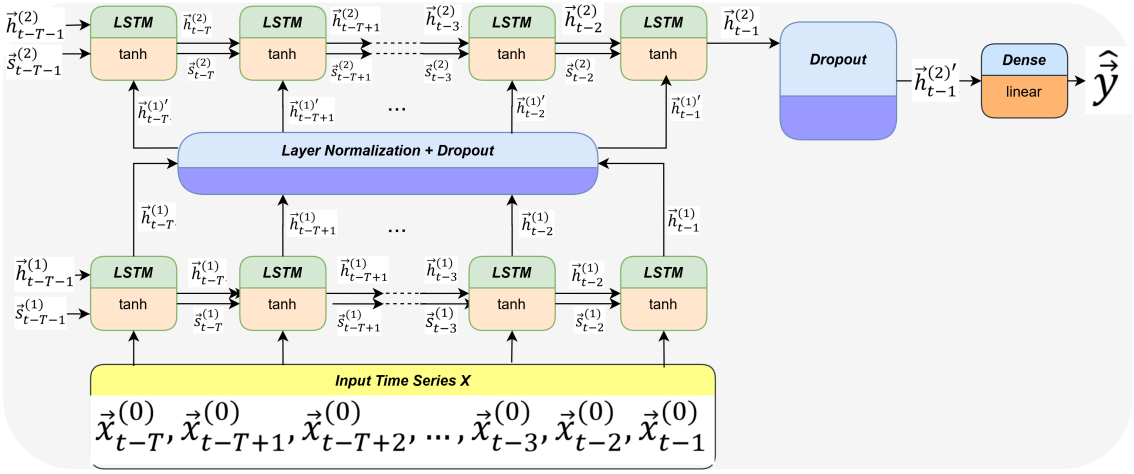

Figure 4.3: LSTM Model

4.1.4 LSTM with Attention Module

The stacked LSTM architecture, explained in 4.1.3, are further extended with an attention mechanism implemented after the second stacked layer of the LSTM as illustrated in Figure 4.4. Only addition is to apply another layer normalization layer to output hidden states of the secondary LSTM layer before the dropout layer. For exploiting the attention mechanism, second LSTM layer returns its entire hidden state vector instead of only returning its last state. Hidden states are fed into the attention layers are processed using self attention mechanism as explained in 2.2.2. Given the input sequence of $H = \{\vec{h}_{T-t}, \vec{h}_{T-t+1}, \ldots, \vec{h}_{t-2}, \vec{h}_{t-1}\}$, sequence is split into two parts as the last hidden state $\vec{h}_{t-1}$ and the rest of the sequence $\vec{h}_s$. $\vec{h}_s$ is then scored using last hidden state, then the scores are normalized using softmax function to compute alignment weights. Two different score functions are considered individually as two different models in the thesis; additive score function, as derived in 2.26 and multiplicative score function, as derived in 2.27. The alignment weights are applied to $\vec{h}_s$ by calculating the weighted sum to result in the context vector $\vec{c}_{t-1}$ using 2.32. As a final step of the attention layer, the context vector $\vec{c}_{t-1}$ and the last hidden state $\vec{h}_{t-1}$ are concatenated and weighted, followed by applying a tanh activation function to compute the final output of the attention layer h'_{t-1}. Then, h'_{t-1} is passed through a dropout layer connected to a dense layer wherein a linear activation function is applied to generate the prediction $\hat{\vec{y}}$.

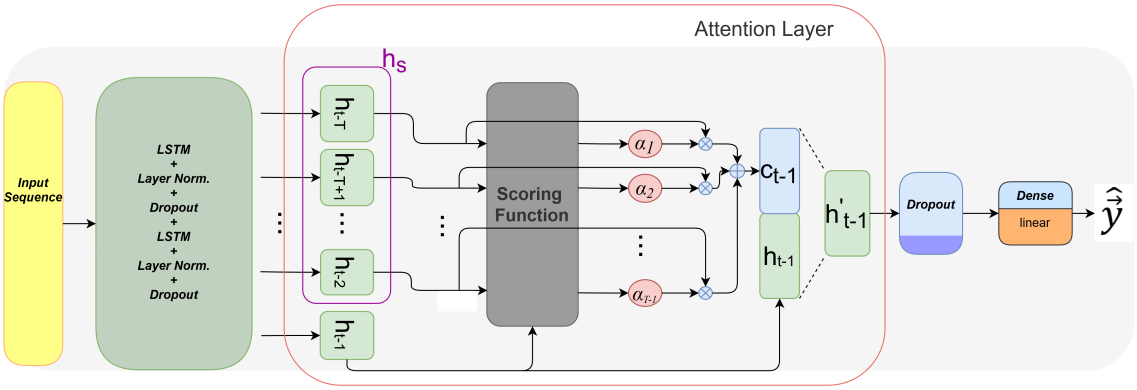

Figure 4.4: Attention mechanism attached to a stacked LSTM network

4.2 Anomaly Detector

Using the prediction errors as anomaly indicators, anomaly detection is implemented in point-wise. Prediction errors are residuals of the prediction made at time step $(t-1)$ and expected true value at time step (t). The prediction errors distribution on training data is modelled using a multivariate Gaussian distribution [87], where parameters of the Gaussian distribution, mean and variance, are calculated using maximum likelihood estimation (MLE) [87].

For obtaining the anomaly scores which describes the probability of a point be anomalous in the test dataset, the anomaly score of an observation prediction error vector e_t is defined as follows

$$a_t = (e_t - \mu)^T \Sigma^{-1} (e_t - \mu), \tag{4.3}$$

where μ and Σ denoting mean vector and covariance matrix of the prediction errors distribution and e_t denoting the prediction error vector in the test dataset at time step t. A point x_t is considered an anomaly if $a_t > \rho$ where ρ is a threshold value, otherwise is classified as a normal point. A validation dataset containing normal data along with anomalies is utilized in order to determine the appropriate threshold values which can separate anomalies from normal observations and incur as few false positives as possible. A separate test set is used to evaluate the model.

Models proposed in 4 are first trained on dataset with only normal values with the goal of learning normal time series patterns and optimized to increase prediction accuracy. For this purpose, four different dataset for each experiment is used: a training dataset N consisting of only normal values; primary validation dataset N_{V1} with only normal values, secondary validation dataset N_{V2} with both normal and abnormal values, and a test dataset N_T with both normal values and anomalies. The anomaly detection algorithm pursued in the thesis is explained as follows:

1. The time series prediction models are trained on training dataset N and use N_{V1} to prevent overfitting of the model on training data by using early stop method.

2. The prediction errors distribution on training dataset N are modelled using a multivariate Gaussian distribution. Corresponding parameters of Gaussian distribution, mean and covariance are determined using MLE.

3. Trained prediction models are applied on secondary validation dataset N_{V2} and anomaly scores $a_{(t)}$ are computed by the parameters of the Gaussian distribution calculated in the previous step. It is followed by setting a threshold value ρ that can distinguish the anomalies with as few false positives as possible.

4. The prediction models and their corresponding threshold values are evaluated on test dataset N_T and results are obtained.

It is expected that the models should have higher prediction errors on time steps with anomalies compared to time steps with no normal behaviour when generating predictions on a new dataset. Thus, values derived from 4.3 can be used as anomaly scores and for computing an appropriate threshold for separating anomalies from data points with normal behaviours.

4.3 Evaluation Metrics

For accurate evaluation of the models in its anomaly detection capabilities, it is essential to use quantitative metrics to provide a proper comparison. Three metrics are used to measure performance of the model on anomaly detection; precision, recall and the F1 score. The metrics are derived as

$$Presicion = \frac{True\,Positives}{True\,Positives + False\,Positives} \tag{4.4}$$

$$Recall = \frac{True\,Positives}{True\,Positives + False\,Negatives} \tag{4.5}$$

$$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}. \tag{4.6}$$

The confusion matrix [88] where these metrics are derived are illustrated in Table 4.1.

		Actual data	
		Anomalous	Normal
Predictions	Anomalous	True Positive	False Positive
	Normal	False Negatives	True Negatives

Table 4.1: Confusion matrix.

Recall is synonymous to accuracy in anomaly detection, as a recall of 1 implies that the model is able to flag each of anomalies located in the actual data. On the other hand, precision is a measure of the distribution of true positives and false positives in the detected anomalies. Therefore, a precision of 1 means that there are no false positives among identified anomalies. To provide accurate comparison of these two metrics, another metric called F-score which combines precision and recall was introduced [89]. The F1 score is the harmonic mean of precision and recall where equal weights were applied to precision and recall. There are other F measures applied in case of applying greater weights to one of the two metrics [89]. In this thesis, only F1 score was used.

5 Experiments

In this chapter, datasets employed in the experiments are further examined. Pre-processing of the datasets, followed by the design space exploration of the design parameter of the models are provided in detail. Furthermore, experimental procedure and the results of the experiments are provided and discussed.

5.1 Hardware and Software

Implementation of the fault injection algorithm in PX4's source code is programmed in C++ [90] while NN models are implemented using Keras package [91] with Tensorflow [92] backend. Keras is a Python-based [93] high-level application programming interface (API) of Tensorflow that is used for fast prototyping and implementation of many DNNs. In order to increase the training speed, Keras package is configured to utilize GPU by implementing CUDA library [94] which is a collection of GPU-accelerated libraries specialized to deliver higher performance compared to CPU-only alternatives. Trainings are performed on a machine with Intel Core i7@2.9 GHz, Nvidia GeForce 930MX and 16GB RAM.

5.2 Datasets

Datasets used in the thesis are collected by the logger/data collector block of fault injection environment, following the procedure explained in 3. When collecting training datasets and primary validation datasets, the fault injector block remains off-line during the simulation runs in order to collect datasets containing only normal values. On the other hand, the fault injector block goes on-line during injecting each of the faults, introduced in 3, in their own simulation sessions. Datasets gathered from these sessions include normal data values along with anomalies. Simulation runs are carried out to perform two separate tasks of the drone, circular path task and non-circular path task. Motivation behind performing two different tasks in the simulation is to investigate whether model trained on datasets of a particular task was able to generalize on datasets corresponding to the other task.

For the circular task, drone following a circular path, shown in Figure 5.1, is simulated in Gazebo. One of the two collected datasets containing only normal values is used as training dataset N by each prediction model and the other one is the primary validation datasets N_{V1} for the models during training. Datasets collected from the simulation sessions where the faults are injected to the target drone include secondary validation datasets N_{V2} and test dataset N_T for each of the individual anomaly detection tests.

For the non-circular task, drone follows a non-circular path in the simulation environment as illustrated in Figure 5.1. Same procedure as in the circular task is followed while collecting datasets. Thus, datasets gathered from this task also consist of training datasets, test datasets, primary and secondary validation datasets.

In order to collect the datasets for training, validation and testing, total of 12 simulation run was executed. Details of the datasets gathered from both tasks are presented in Table 5.1.

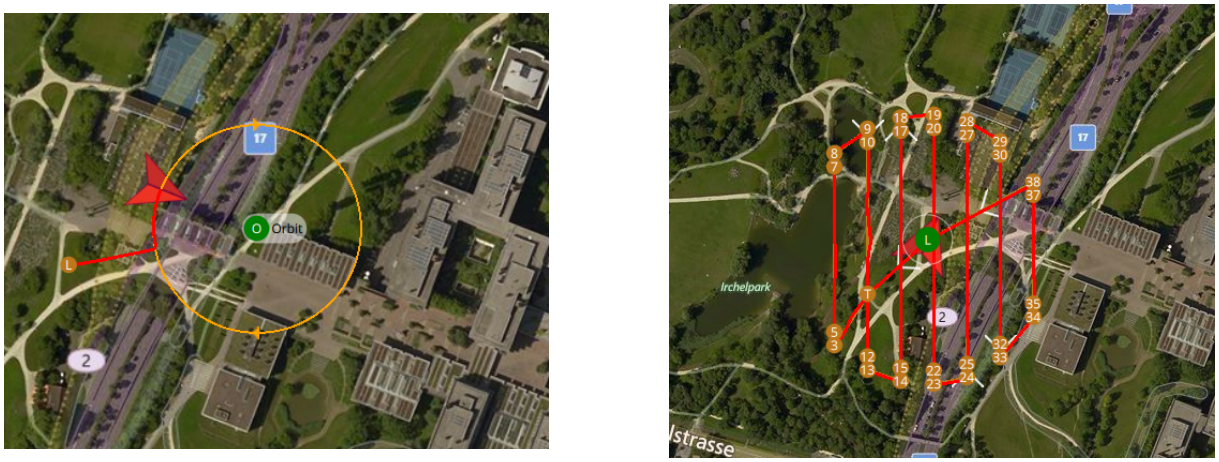

Figure 5.1: second figure

Timestamps of the simulation are used as the index of the datasets. Number of timestamps in Table 5.1, refers to the duration of the corresponding simulation, and number of data points refers to the total number of data sampled by the logger during simulation. 659 different sensor parameters are logged during the simulation runs and the number of the types of sensor parameters remain constant for every set of dataset. These parameters include IMU and GPS sensor readings, values regarding the simulated physical components, such as voltage readings for the motors and battery level of the drone, internal parameters for the control algorithms such as setpoints, state estimations and control outputs as well as constant identification parameters used by the simulator such as sensor and mission identification number as explained in 3.2 in detail. In Figure 5.2, two plots of the same parameter, namely parameter 520, was illustrated. Above plot represent the parameter readings from the datasets collected through non-circular task whereas the plot below presents the values of the same parameter in the datasets of circular task. Plots indicates different temporal patterns in each task where parameter 520 has a temporal pattern occurring approximately at each 10000 time steps during a non-circular path whereas the said parameter follows a patters resembling to a sine wave during a circular path. Plots are taken from training datasets and thus, does not contain anomalies.

Time steps where the faults were injected to the target model in simulation is additionally logged using the logger module in order to label the anomalies in the testing datasets. These labels are externally added to the datasets containing anomalies in order to evaluate the performance the models on flagging anomalies.

5.3 Data Pre-processing

Parameters of the datasets are individually updated and logged. Parameters of a simulation run are merged into one single dataset after the completion of each simulation by taking timestamps as a reference point. During the logging session, another pre-processing step is needed regarding the SITL simulation. Due to the implementation of PX4 software, the sensor parameters that are activated during a simulation run is selected based on the type of UAV in simulation. These sensor selection algorithm is hard-coded into the source code of PX4 software. However, every sensor parameter available in the simulation is directly logged by the logger module regardless of their states of activation. This results in a situation where unwanted parameters are included

		number of timestamps	total number of data point
Circular	N	172,252	113,514,068
	N_{V1}	135,655	89,396,645
	$N_{V2,F1}$	88,965	58,627,935
	$N_{V2,F2}$	65,398	43,097,282
	$N_{T,F1}$	84,323	55,568,857
	$N_{T,F2}$	74,262	48,938,658
Non-Circular	N	149,123	98,272,057
	N_{V1}	86,352	56,905,968
	$N_{V2,F1}$	100,458	66,201,822
	$N_{V2,F2}$	88,595	58,384,105
	$N_{T,F1}$	103,565	68,249,335
	$N_{T,F2}$	118,662	78,198,258

Table 5.1: Size of the datasets used during training and testing of the models. Number of timestamps denotes the duration of the corresponding simulation run in terms of seconds and number of data points illustrates total number of samples applied both in training and testing.

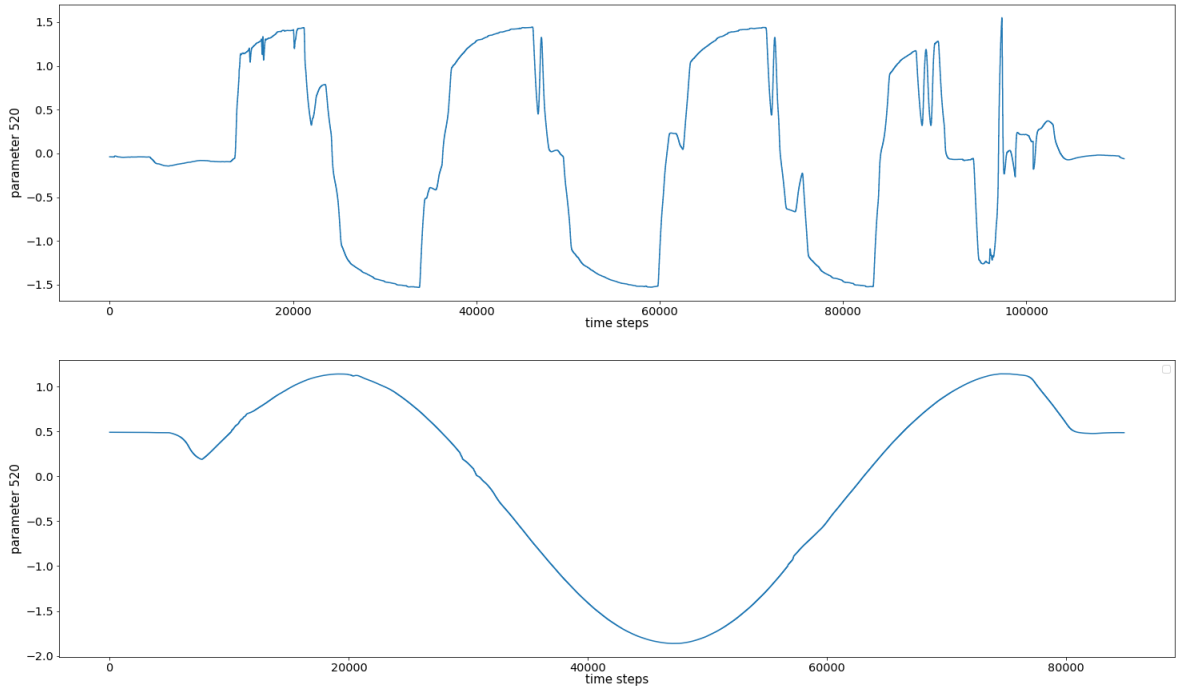

Figure 5.2: Sampled values of parameter 520 in two different tasks. In the above plot, parameter 520 values corresponds to the datasets of non-circular task while in the below plot to the dataset of the circular task.

in the final logged data. Since these parameter are not activated during the simulation of the chosed UAV, their values for the time steps are logged as not-a-number (NaN) inside the log file.

Using dataset with NaN values, also referred to as missing values, for training the models results in poor performance of the models for training. Commonly applied approach is to handle the missing value problem before initiating a training. In this thesis, instead of gathering dataset with missing values, source code was reprogrammed to discard the parameters, that are inactive during a simulation run, for the logging session. Thus, collected datasets does not contain any missing value resulted from the PX4's hard-coded implementation and datasets are further passed over to the normalization process.

As stated in [47], convergence was found to be faster if the average of each input variable over the training set is close to zero. Scaling the training set values to unit variance also affect the convergence of the model in a positive manner. This process is commonly referred to as standardization or normalization. Therefore, normalization is performed in the training dataset as follows

$$x_{norm,t} = \frac{x_t \mu_x}{\sigma_x}, \tag{5.1}$$

where x_t is the input value at time step t, μ_x is the mean over entire sequence $\vec{x}$ and σ_x of the standard deviation of $\vec{x}$. For concision, assumption of all x_t being normalized is made and therefore, x_t is used to annotate the normalized values instead of $x_{norm,t}$ for the rest of the thesis. Given the univariate time series sequence $\vec{x} = \{x_1, x_2, \ldots, x_{T-1}, x_T\}$, x_t denotes the normalized data point at time step t and T is the total number of time steps in the time series sequence. In this thesis, normalization is performed on a multi-variate level due to the nature of the datasets used in the experiments. Each datasets are normalized before training, validation and testing in order to provide a uniform basis for the evaluation.

Mini batch approach is pursued for the training of the models. Given multivariate input time series sequence $X_t = \{\vec{x}_t, \vec{x}_{t+1}, \ldots, \vec{x}_{t+T-2}, \vec{x}_{t+T-1}\}$, batches are constructed as

$$Batch_j = \{X_{jm}, X_{jm+1}, \ldots, X_{jm+m-2}, X_{jjm+m-1}\}, \tag{5.2}$$

where j indicates j-th batch and m denotes the batch size which is basically the number of sequences in a given batch. Batch size and time steps are design parameters of the models and further examined in 5.4.

5.4 Design Space Exploration

Design parameters of the ANN models are listed as follows: number of time steps T, ,batch size B, size of the convolutional window K, also referred to as kernel size, number of output filters used in the convolution operations F, dilation rate for causal convolution operation D, size of the max pooling window P, number of hidden units in LSTM layer H_{lstm}, number of output units of the attention layer H_{att}, dropout rate R and learning rate for the optimizer L_r.

Setting the design parameter T accurately plays a significant role in the learning process for the LSTM-based models. If there is a high correlation between the consecutive time steps of a training dataset, this correlation may have an effect on the learning process, since it can be picked up by the model, and therefore, projections of the model will suffer from this correlation, resulting in a biased prediction model and thus, yielding possible poor generalization performance on new data with different correlation rate. This internal correlation is called autocorrelation. Autocorrelation is the correlation between time series data and its delayed copy. Autocorrelation is important since it can be applied to uncover patterns in the dataset, successfully select the best time step value for the model and correctly evaluate an unbiased prediction model.

In order to determine the appropriate value for the design parameter T, autocorrelation analysis were applied. Autocorrelation analysis is used to measure the extent of a linear relationship between lagged values of a time-series data. Given measurements, $x_1, x_2, \ldots, x_T$ at time $t_1, t_2, \ldots, t_T$, the lag k autocorrelation function is defined as [95]

$$r_k = \frac{\sum_{i=k+1}^{T}(x_i - \mu_x)(x_{i-k} - \mu_x)}{\sum_{i=1}^{T}(x_i - \mu_x)^2}, \tag{5.3}$$

where μ_x being the mean of T measurement. Analysis was individually conducted columns-wise over every dataset and the analysis yielded that starting from the 380th time step for circular datasets and the 365th time step in non-circular datasets, autocorrelation coefficient in the datasets falls within a range of values between 0.12 and 0, which can be considered a negligible amount of autocorrelation [95]. Thus, T is set to 400 for every model.

As shown in [96], large batches tend to degrade a model's ability to generalize due to their tendency to converge to sharp minimizer of both the training and testing functions. Besides, larger batch size requires more computational power for each epoch since the number of parameter to be trained for an epoch increases. Thus, different batch sizes were applied during the experimental exploration of the appropriate batch size for the models and the most effective batch size, optimal batch size B for the models were concluded to be 64.

As presented in 4, dilation rates D for the causal convolutional layer inside the residual block follows an exponential pattern where the dilation rate increases by a power of 2 between each convolutional layer in the residual blocks. For example, two convolutional layers in the first residual block consecutively have a dilation rate of 1 and 2 whereas convolutional layers in the second residual block have a consecutive dilation rate of 4 and 8. The same exponential pattern was applied to the rest of the residual block. The remaining design parameters of the models were finalized after an individual experimental exploration. According to the experimental results, applying kernel size, K, of 32 for each convolutional layers in TCN resulted in the the most optimal outcome for Temporal CNN model. After a series of experiments, number of filters, F, for 4 residual blocks of the TCN was found to be optimal in the pattern of [64,32,64,128]. For convolutional layers in 1-D CNN model, the length of kernel is increased to 64 while the number of filters, F, was concluded to be 32. Additionally, a max pooling window size P was set to 2 for the 1-D CNN model. For LSTM-based models, it was observed that as the number of hidden units H_{lstm} increases, models are tend to overfit on training data and also training process gets more computationally expensive. 400 hidden units were concluded to be the optimal number of units for the stacked LSTM model whereas the LSTM layers in Attention-LSTM models has 300 hidden units in addition to 400 and 450 output units ,H_{att}, of multiplicative and additive attention layers , respectively. As discussed in 2.1, Adam optimizer requires an initial learning rate to perform its optimization algorithms. Learning rate, L_r was tuned experimentally for each model. Another consideration for design space exploration was to achieve the same number of total trainable parameters for each model during learning. Design parameter values yielded the best results for the models are presented in Table 5.2 together with other design parameters of the models.

5.5 Results

Experimental procedure consists of 2 stages. In the primary stage each model was trained on the both of training dataset N using a secondary dataset with no anomaly points as a validation

1-D CNN	$F = [64,64], K = 64, P = 2$ $R = 0.15, L_r = 0.00001$
TCN	$F = [64,64,32,32,64,64,128,128]$ $K = 32, R = 0.1$ $D = [2^0,2^1,2^2,\ldots,2^6,2^7], L_r = 0.000001$
LSTM	$H_{lstm} = 400, R = 0.15, L_r = 0.000001$
LSTM with Additive Attention	$H_{lstm} = 300, H_{att} = 450, R = 0.1, L_r = 0.00001$
LSTM with Multiplicative Attention	$H_{lstm} = 300, H_{att} = 400, R = 0.1, L_r = 0.00001$

Table 5.2: Finalized design parameters of the proposed models

dataset N_{V1}. Prediction error vector of the prediction for the training dataset is then fitted to a multivariate Gaussian distribution. Corresponding parameters of the Gaussian distribution are used to conduct anomaly scores for the prediction error vector of the same model on the test datasets. A point x_t is considered an anomaly if its corresponding anomaly score is above the predetermined threshold value, otherwise is classified as a normal point. Anomaly score function is applied to the secondary validation dataset N_{V2}, containing normal data along with anomalies, in order to determine the appropriate threshold values which separates anomalies from normal observations with few false positives as possible. Total number of 4 test datasets N_T and 4 secondary validation dataset N_{V2} are utilized during the experiments. As mentioned in 5.2, time steps of fault injections and therefore the point location of the anomalies are logged during the fault injection process. A fixed sized anomaly windows with each window centred around an anomaly is implemented. The earliest detection inside a window is considered as a true positive while any detection flagged outside the window is considered a false positive. Size of the anomaly windows is denoted with Δt. Optimal window size is determined by experimenting with five different window sizes, where $\Delta t = [100,120,150,200,250,300]$. It is concluded that the best performance was achieved when $\Delta t = 200$. $\Delta t = 200$ indicates that in a window of 200 time step, an anomaly is present in the dataset. Same window size is implemented for the evaluation of each experiment. Experiments, conducted for the each model individually, are structured as follows

- Experiment 1 (E1): Model trained on a circular dataset performs time series prediction on a circular dataset with anomalies resulted from fault injection scenario F1. .
- Experiment 2 (E2): Model trained on a circular dataset performs time series prediction on a circular dataset with anomalies resulted from fault injection scenario F2.
- Experiment 3 (E3): Model trained on a circular dataset performs time series prediction on a non-circular dataset with anomalies resulted from fault injection scenario F1.
- Experiment 4 (E4): Model trained on a circular dataset performs time series prediction on a non-circular dataset with anomalies resulted from fault injection scenario F2.
- Experiment 5 (E5): Model trained on a non-circular dataset performs time series prediction on a non-circular dataset with anomalies resulted from fault injection scenario F1.
- Experiment 6 (E6): Model trained on a non-circular dataset performs time series prediction on a non-circular dataset with anomalies resulted from fault injection scenario F2.

- Experiment 7 (E7): Model trained on a non-circular dataset performs time series prediction on a circular dataset with anomalies resulted from fault injection scenario F1.
- Experiment 8 (E8): Model trained on a non-circular dataset performs time series prediction on a circular dataset with anomalies resulted from fault injection scenario F2.

After the completion of the experiments, anomaly score function is applied to prediction error vectors on the corresponding secondary validation and test datasets, $N_V 2$ andN_T. Using the anomaly scores yielded from the validation dataset prediction, an appropriate threshold for each experiment is determined individually, followed by detection of anomaly windows using the computed threshold and anomaly scores for the test datasets.

In Figure 5.3, prediction for parameter 131, performed by the LSTM model during E1 and corresponding anomaly scores are illustrated. Windows shown in yellow in the anomaly score plot indicate false positive prediction of the given model while green windows are indication of true positives and red windows are false negatives. Anomaly score of the model for experiment E1 is computed on the anomaly score function, as derived in 4.3. Taking into account the entire dataset, threshold value that yields maximum overall F1 score for the model is determined to be 9.7 . According to the plots, LSTM model is able to detect 4 of the anomaly windows among the total number of 5 anomaly windows that are present in parameter 131. The model produces false positives for time steps that are close the start and end of the time series data. Though the model is able to detect anomalies at the time they first occurred, model results in smaller prediction error for the later part of the anomaly windows since model tries to fit its predictions to the observed anomaly pattern after the re-occurrence of the similar patterns in the dataset.

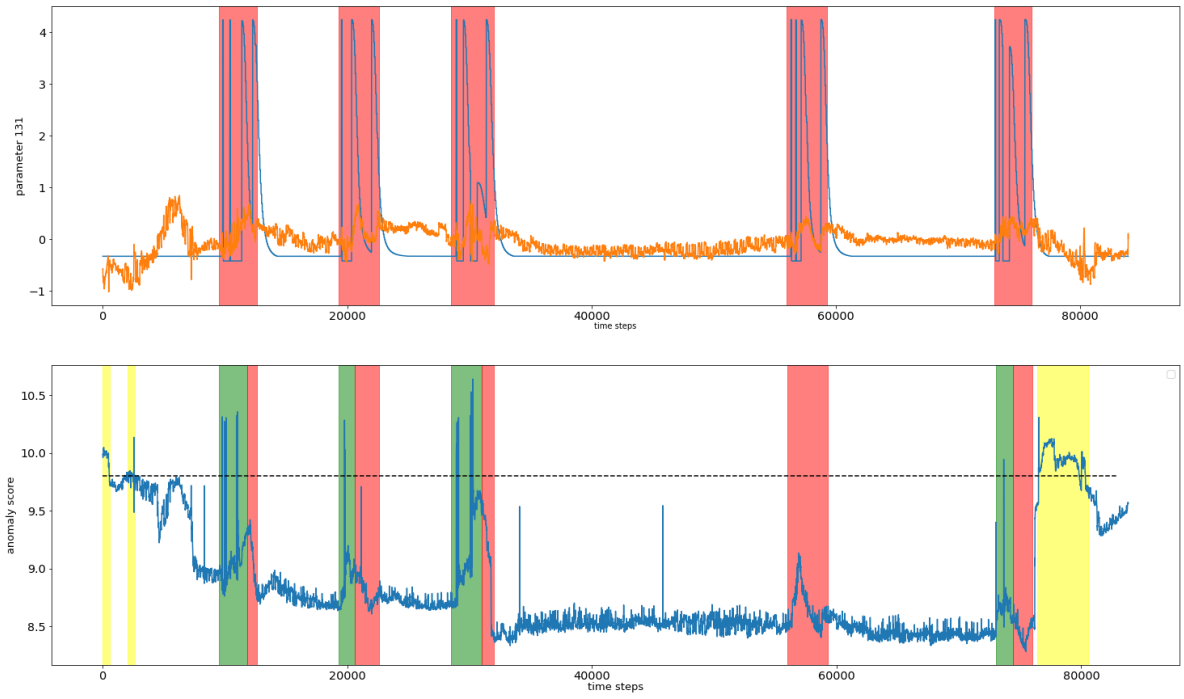

Figure 5.3: In the plot above, ground truth values and models' prediction of parameter 131 in circular test dataset is shown. Predictions was performed by the LSTM model. In the below plot, computed anomaly scores of the LSTM model for the same dataset are illustrated.

Figure 5.4 illustrates true values of the parameter 131 and predictions made by the TCN

model during the experiment E1. Optimal threshold value for the model is set to 80.3 . Though, similar to LSTM model, TCN model is able to flag anomalies for 4 different anomaly windows out of 5 anomaly windows, the model generates more false positives than the LSTM model. Similar pattern is also seen in TCN's predictions where the models tries to fit the prediction to re-occurring anomalous pattern.

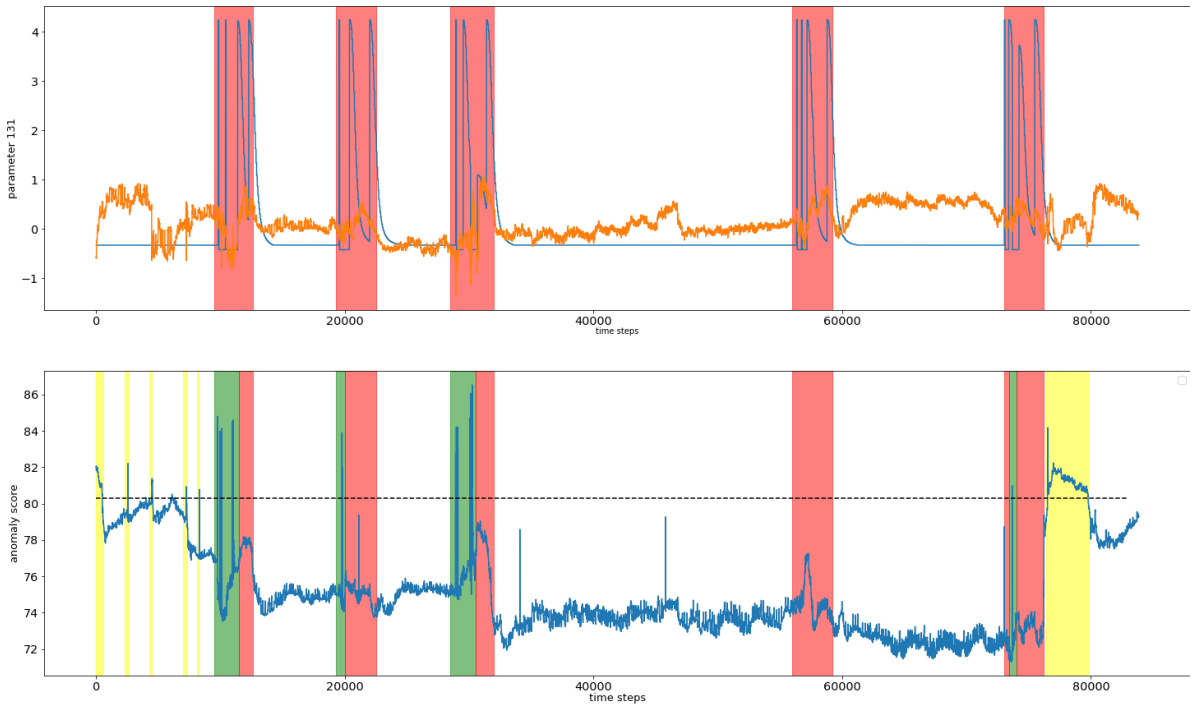

Figure 5.4: In the plot above, ground truth values and models' prediction of parameter 131 in circular test dataset is shown. Predictions was performed by the TCN model. In the below plot, anomaly scores of the model for the same dataset are illustrated.

Figure 5.5 shows prediction performance of LSTM model with additive attention mechanism along with anomaly scores of the model during the experiment E1. Appropriate threshold value was computed as 10.4 . Similar to the previous models' performance, 4 out of 5 anomaly windows was detected. Furthermore, this model also suffered from false positive prediction at time steps close to beginning and end of the flight as the TCN models.

In Figure 5.6, true values of parameter 131 together with predicted output of LSTM model during the experiment E4 are illustrated in order to evaluate the generalization ability of the model. Anomaly score of this experiment is also provided in the below plot of Figure 5.6. Threshold value of 9.5 results in best F1 score for the model. For this experiment, LSTM model trained on a dataset of a circular task is tested on a dataset of a non-circular task. Though model's prediction performance is similar to its performance on experiment E1, the model shows poorer performance in terms its anomaly detection ability. Similar to its performance in E1, model's prediction error decreases in case of re-occurrence of anomalous pattern in a short time span and therefore, model is unable flag the succeeding anomaly windows. It is concluded that using fixed threshold values resulted in poorer anomaly detection performance of the models even though the same model performs similarly to its performance in E1.

In order to evaluate the overall performance of each model during the experiments, F1 score is calculated after each experiment. Table 5.3 shows the precision and recall values of the LSTM

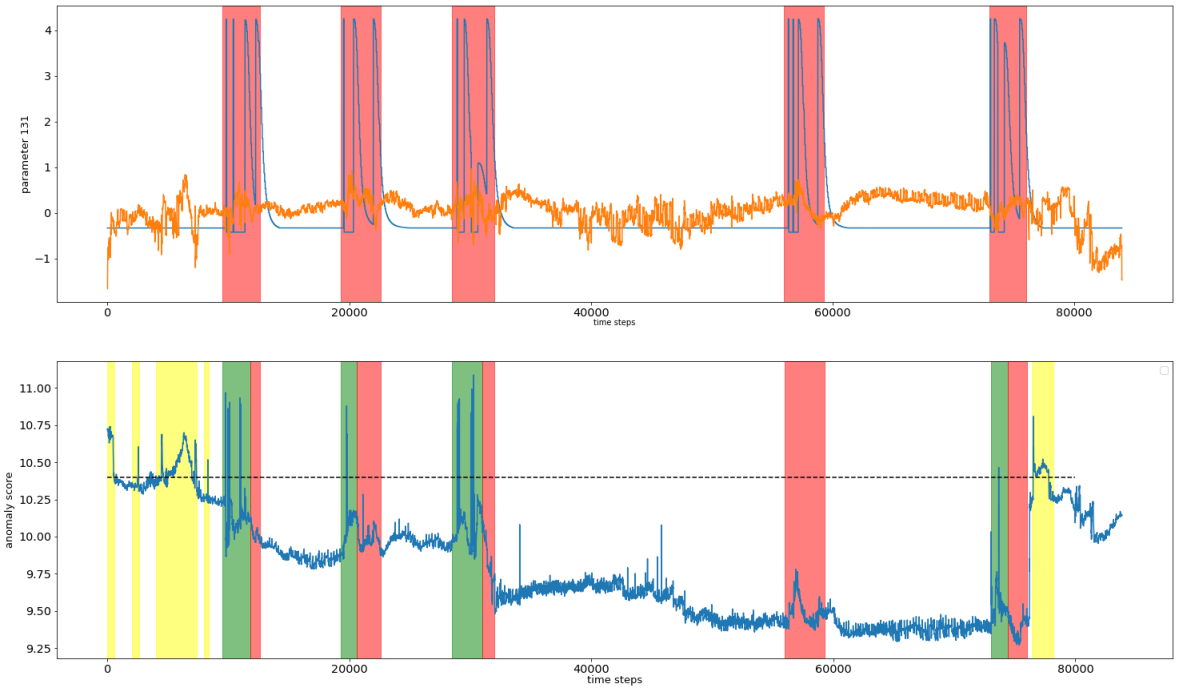

Figure 5.5: In the plot above, ground truth values and prediction of the LSTM model with additive attention mechanism for parameter 131 in a circular test dataset is shown. Predictions was performed by the LSTM model. Corresponding anomaly scores of the model are also illustrated in the below plot.

model for each experiment along with the corresponding F1 scores. LSTM model is able to detect anomalies with an average precision of 82% when they are tested on the datasets that are similar to the datasets used to train the model. For concision, experiment E3, E4, E7 and E8, are referred to as crossed experiment while the remaining experiments are referred to as non-crossed experiments. Detection performance of the LSTM model is dropped to average precision of 70% during crossed experiments.

$\Delta t = 200$	Precision	Recall	F1 Score
E1	0.852	0.320	0.465
E2	0.803	0.266	0.399
E3	0.716	0.2136	0.328
E4	0.711	0.102	0.178
E5	0.834	0.188	0.306
E6	0.793	0.173	0.284
E7	0.691	0.312	0.429
E8	0.686	0.236	0.351

Table 5.3: Precision, recall and F1 scores of the LSTM model.

Overall precision and recall values, and F1 score of the LSTM models with attention mechanism are listed in Table 5.4 and Table 5.5. Results shows that the attention mechanisms do not show any significant superiority over each other since they perform similarly on anomaly detection task. Models are able to detect anomalies with an approximate average precision of

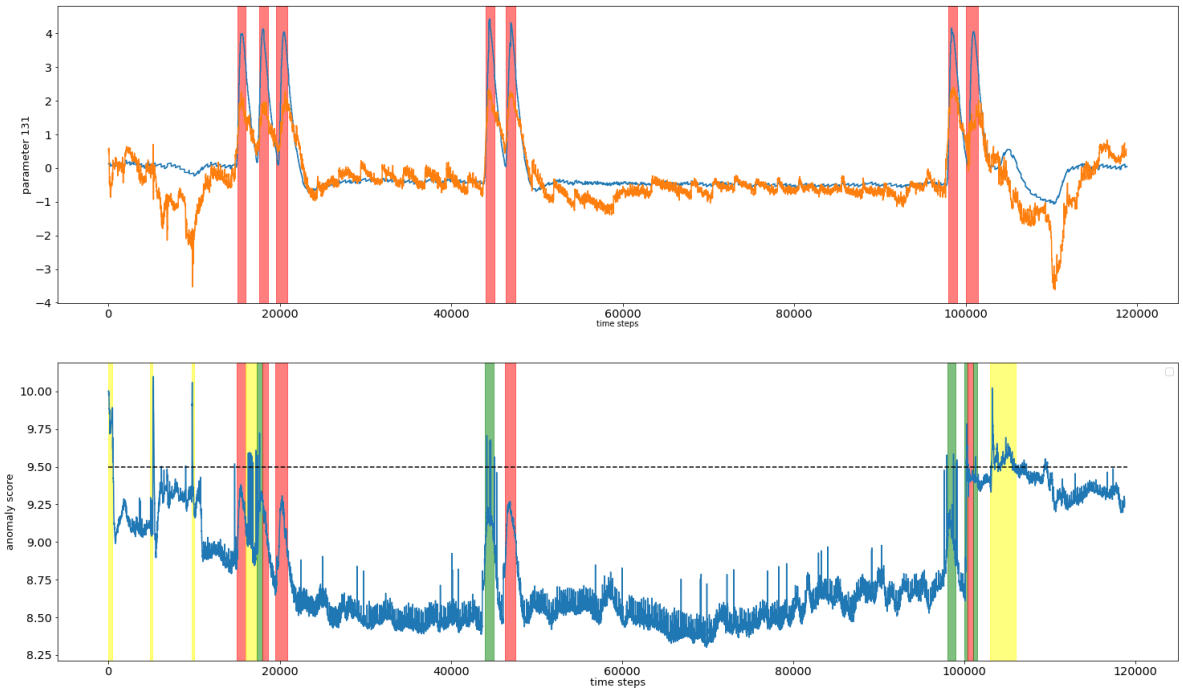

Figure 5.6: In the plot above, true value and predicted values for parameter 131 in non-circular test dataset is shown. Prediction was preformed by the LSTM model. In the below plot, corresponding anomaly scores were plotted.

80.% on datasets of non-crossed experiments. Though, similar to LSTM model, a decrease on detection performance is observed when testing on datasets of crossed experiments, the detection performance is higher than the LSTM model during crossed experiments.

$\Delta t = 200$	Precision	Recall	F1 Score
E1	0.821	0.151	0.255
E2	0.796	0.173	0.284
E3	0.742	0.482	0.591
E4	0.731	0.331	0.455
E5	0.815	0.1290	0.222
E6	0.790	0.382	0.514
E7	0.718	0.152	0.250
E8	0.7096	0.169	0.272

Table 5.4: Precision, recall and F1 scores of the LSTM model with additive attention mechanism.

Overall precision and recall values, and F1 score of the CNN models with attention mechanism are listed in Table 5.6 and Table 5.7. According to the results, though TCN model performs slightly better than 1-D CNN model, no significant difference in terms of detection is observed. As observed in previous models, a performance drop for both models occurs during crossed experiments. 1-D CNN and TCN respectively achieve an average precision of 78% and 77% during non-crossed experiments whereas TCN outperforms the 1-D CNN on crossed experiments with an average precision of 68% as opposed to 1-D CNN's precision rate of 64%.

$\Delta t = 200$	Precision	Recall	F1 Score
E1	0.818	0.156	0.262
E2	0.795	0.177	0.289
E3	0.736	0.367	0.489
E4	0.722	0.294	0.417
E5	0.810	0.210	0.333
E6	0.785	0.366	0.499
E7	0.719	0.155	0.255
E8	0.705	0.203	0.315

Table 5.5: Precision, recall and F1 scores of the LSTM model with multiplicative attention mechanism.

$\Delta t = 200$	Precision	Recall	F1 Score
E1	0.765	0.133	0.220
E2	0.78	0.231	0.356
E3	0.697	0.345	0.463
E4	0.680	0.212	0.323
E5	0.788	0.209	0.330
E6	0.794	0.322	0.458
E7	0.678	0.342	0.458
E8	0.669	0.420	0.516

Table 5.6: Precision, recall and F1 scores of the TCN model.

$\Delta t = 200$	Precision	Recall	F1 Score
E1	0.789	0.203	0.322
E2	0.758	0.308	0.438
E3	0.659	0.336	0.445
E4	0.642	0.234	0.342
E5	0.772	0.128	0.219
E6	0.759	0.305	0.435
E7	0.648	0.136	0.224
E8	0.640	0.366	0.465

Table 5.7: Precision, recall and F1 scores of the 1-D CNN model.

5.5.1 Sub-experiments

In addition to the main anomaly detection experiments, several other sub-experiments were also conducted. These experiments aim to investigate the effects of the certain individual elements of the time series predictor models proposed in this thesis.

Firstly, effect of adding layer normalization and attention layers to the LSTM model are investigated. In Figure 5.7, process of the validation loss values of the models over number of epochs is illustrated. In this experiment, a stacked LSTM model, a stacked LSTM model with the addition of layer normalization and, stacked LSTM models with the addition of additive and

multiplicative attention mechanisms are employed. In order to perform a fair comparison, each models are trained on the same dataset and early stopping are implemented in each models using the same validation dataset containing only normal values. Moreover, the number of parameters that needs to be trained by each model are kept close with a margin of approximately 6,000 parameters over a total number of approximately 2 million parameters in each model. Experiment yields that addition of layer normalization decreased the model loss by 12%. Furthermore, it is shown that though addition of attention mechanism together with layer normalization reduced the model loss value compared to simple stacked LSTM model, stacked LSTM with only layer normalization still yields better loss values compared to attention-based LSTMs. However, it is shown in the anomaly detection algorithm that attention-based mechanism provided better performance on cross-experiments compared to LSTM with only layer normalization. Thus, it is concluded that via attention mechanisms, model is able to establish a better relation between variables of the datasets in cross-experiments re-delegating the importance of the variables for better resulting predictions. Taking into account the positive effect of the layer normalization, layer normalization is implemented in the final proposed LSTM models.

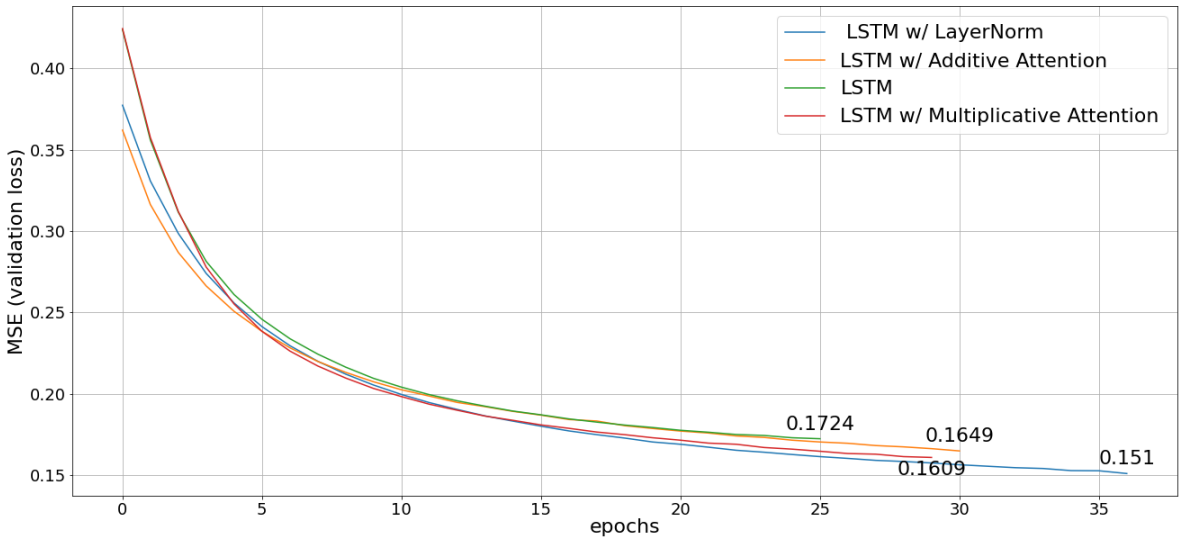

Figure 5.7: Validation loss values of different LSTM-based models with respect to the corresponding epoch

Second sub-experiment is conducted on CNN-based models to further investigate the effect of modifications proposed in these models such as global average pooling layer and normalization layer. As in the previous experiments, same training and validation datasets are applied to the models and the number of parameters of the CNN-based models are kept in the same range. In Figure 5.8, validation loss values of the different CNN-based models are shown. Initially, two different normalization methods are compared, batch normalization and layer normalization. CNN model with batch normalization yields poorer results compared to the CNN-model without any normalization layer. Batch normalization estimates the mean and variance through mini-batch statistics. These estimations contain a certain amount of error and vary from mini-batch to mini-batch since the statics are computed across the batch and same for each training example in the batch. In the experiment, batch size, which was 64, is relatively small compared to the entire dataset and therefore, it is though that the model suffered from the error of computed mean and variance through each batch. On the other hand, replacing the batch normalization with layer normalization results in smaller loss values for the model. Since the layer normalization

is performed the feature dimension instead of the batch dimension, computed statistics are independent from the other training examples and consequently, it is concluded that the statistics computed through layer normalization did not suffer from the error occurred after every batch as the batch normalization. In addition, applying layer normalization to TCN model with a global average pooling layer results in a significant drop in the validation loss which corresponds to approximately 22% decrease in the loss value. This drop also ensures that the TCN model with layer normalization and global average pooling layer has the smallest validation loss among CNN-based models.

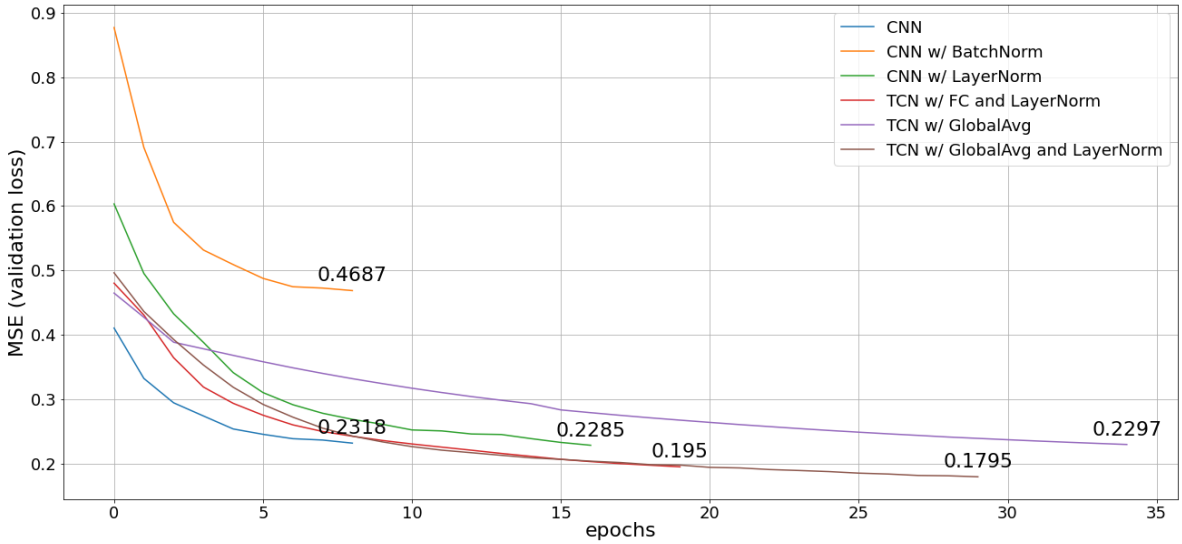

Figure 5.8: Validation loss values of different CNN- and TCN-based models with respect to the corresponding epoch.

Additionally, effect of replacing the fully connected layer in TCN model with a global average pooling layer and a convolutional layer is studied. Using a combination of convolutional layer and global average layer results in a considerable drop in model's validation loss value. This TCN model is able to achieve a 7% smaller loss value as compared with the TCN model with a fully connected layer. Thus, the fully connected layer is replaced with a convolutional layer followed by a global average pooling layer when implementing the proposed TCN model of this thesis.

5.6 Discussion

According results of the experiments, stacked LSTM model yields the best overall performance on anomaly detection task. While LSTM model is able to predict anomalies in datasets of non-crossed experiments with the overall precision of 82%, a performance drop was observed in each model when tested on the datasets of crossed experiments. Results also shows that LSTM models with additional attention mechanism outperforms the both LSTM model and CNN-based models on crossed experiments. Results indicate that models with attention mechanisms result in the best generalization ability among the proposed models. It is though that attention mechanism was able to establish the better resulting relation between the time series variables on new dataset due to its ability to re-delegate the importance of variables for efficient prediction.

CNN-based models also result in similar performance with a slight difference compared to

LSTM-based models. Using a causal dilated convolution operation, TCN model is able to extract patterns on the sequence data. Moreover, as experienced during the trainings, LSTM-based models use up more memory to store the partial results for their multiple cell gates as compared with CNN-based models, resulting in longer training times for LSTM training. An training epochs of CNN model and TCN model are completed in approximately 4 and 9 minutes, respectively, whereas LSTM-based models need approximately 27 minutes to complete an epoch and therefore required more computational power for both training and testing. In addition, convolutions are calculated in parallel applying same filter in each layer unlike the LSTM-based model where predictions for future time steps waits for their predecessors to complete. Thus, in both training and evaluation, a long input sequence is processed as a whole, instead of the sequential approach of LSTM-based models. Considering these advantages and the results of the experiments, CNN-based models provides a reliable alternative to LSTM-based models.

During the experiments, trade-off between prediction performance and the anomaly detection performance of the models are experienced. As shown in Figure 5.9, when the LSTM model is optimized for minimizing the loss function, the time series prediction performance of the model on the test data increases. However, models optimized for prediction yield poorer results in anomaly detection. Due to its high prediction performance, prediction errors on dataset N_{V2} and N_T are relatively small, causing difficult computation for finding the optimal threshold for flagging anomalies from the normal data points without resulting in too many false positives. On the other hand, LSTM model tuned for anomaly detection yields the prediction results shown in Figure 5.10. Model is unable to entirely learn the expected pattern and rather generates higher prediction errors. However, the same model is applied in the experiment and results better performance on anomaly detection with a overall precision of 79% . It is concluded that when optimizing to achieve better prediction results, a certain overfitting to the data occurs, on which early stopping will have no effect since loss (MSE) continues to decrease on both N and N_{V1}. During learning of the model, tuning of the weight is performed in such a way that only recent observations are included in the model predictions and model tends to ignore any temporal information from its state. Consequently, even the anomalous points are fit by the model in order to achieve low prediction error. Since model is trained in a unsupervised fashion with regards to anomaly and no feedback is provided to the model for protecting the model against fitting the anomalies. Model is manually tuned to find the optimal trade-off point where the model is able to flag anomalies on dataset N_{V2}. After the manual tuning of the model, MSE values expectedly increase. Every parameter discussed in 5.4 is included in the tuning process.

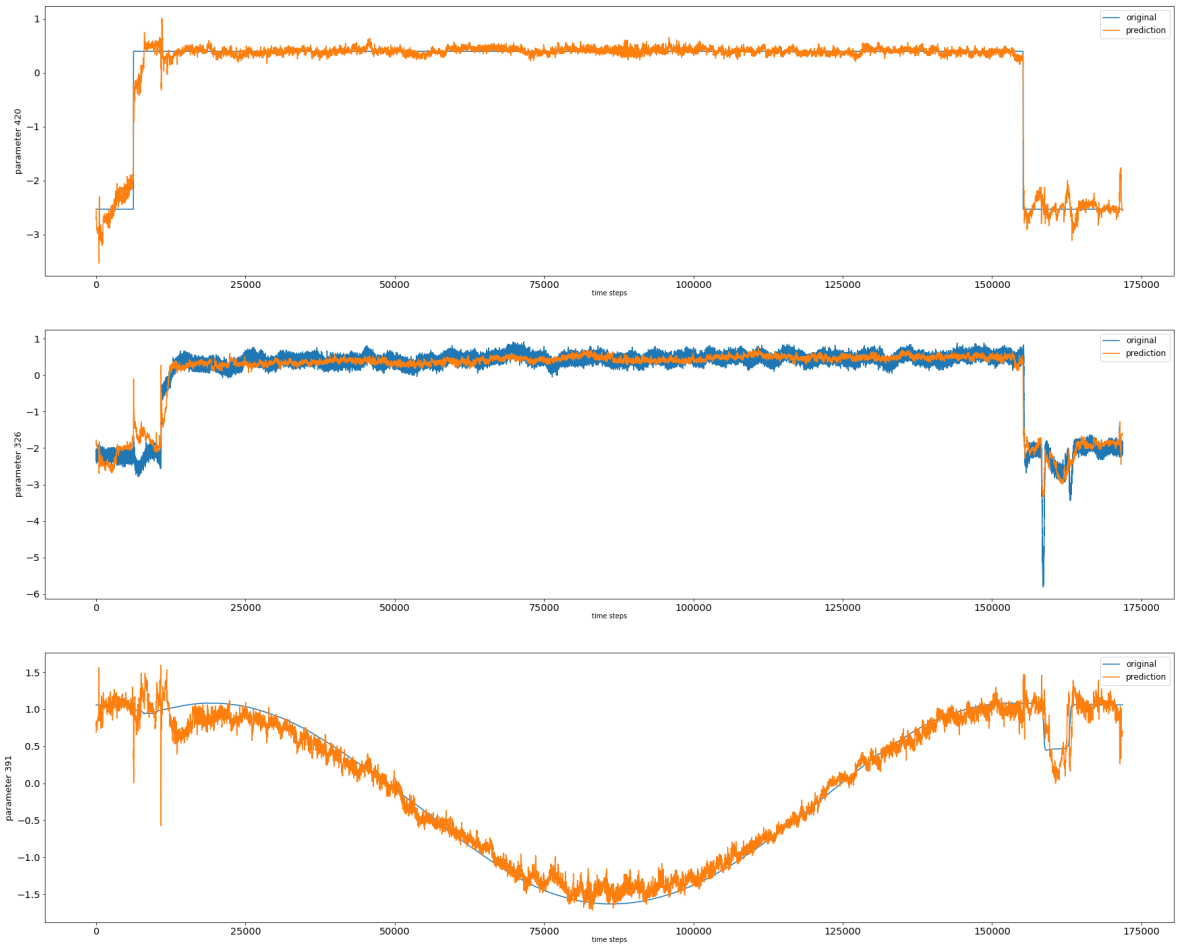

Figure 5.9: Prediction of the LSTM models for three different parameters with different patterns. LSTM optimized for time series prediction was able provide robust result on the validation data in terms of future time steps prediction.

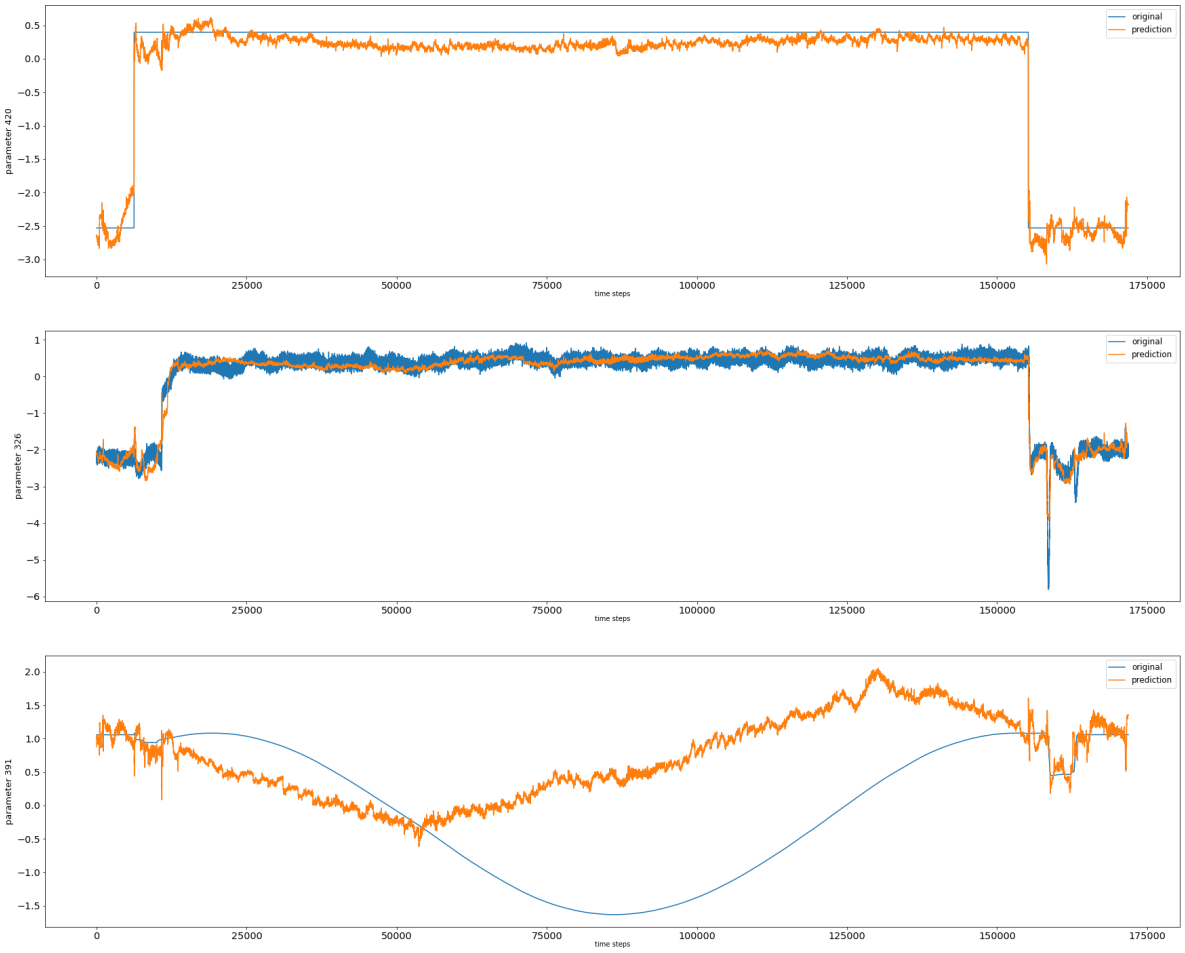

Figure 5.10: Prediction of the LSTM models optimized for anomaly detection for three different parameters with different patterns. LSTM optimized for anomaly detection resulted in prediction with higher predictions errors on validation data as compared to the case of optimizing the model for time series prediction.

6 Conclusion and Future Work

In addition to a custom FI model for a UAV running in SITL simulation, five different ANN-based models, formed by individually combining the time series prediction models with the anomaly detection algorithm, were proposed and compared in terms of their anomaly detection performance in this thesis. Firstly, theoretical background for the employed ANN type and the reasons of their usage for sequential data was explained in details. Additionally, related studies on anomaly detection in time series data were examined. Next, SITL simulation of an UAV was modelled and simulated to perform two separate tasks where the UAV completed a circular and non-circular path in a simulation environment. Using the FI model, particular faults are injected into a UAV in SITL simulation in order to emulate behaviour of the UAV in the presence of faults. Through SITL simulation, datasets containing the sensor parameter values of the targeted UAV is collected for off-line inspection. During dataset collection, datasets containing the UAV's behaviour both with and without the presence of faults in its system were considered. Gathered datasets were used as input data for ANN-based time series prediction. Proposed models are consists of two different type of approaches; CNN-based models and LSTM-based models. These time series prediction models are extended by an implementation of an anomaly detection algorithm based on summary prediction. Finally, the experiments were conducted using the collected datasets. Based on the results, it was concluded that LSTM-based models were proven to be effective time-series predictors and anomaly detectors. Addition of attention mechanism to LSTM increased the LSTM's performance on new datasets. Furthermore, CNN-based models produced similar overall results with lower computational power and thus, provides a possible alternative approach to LSTMs on anomaly detection..

Anomaly detection algorithm used in this thesis was the summary prediction anomaly detection algorithm which is widely used among researchers. However, further study on other anomaly detection algorithm and its implementation on the proposed models are worth considering. For instance, the detection algorithm proposed in [97], where error threshold are dynamically computed and fed back to the detection algorithm for further computation, can help model to adjust accordingly to flag anomalies. Additionally, extending this work to perform the anomaly detection on-line would also be interesting and worthwhile study for the future. Performing online detection also provides a possibility of adding fault handling feature to the models.

For creating data containing faulty behaviour of the UAV, two fault scenarios were simulated and corresponding data was implemented in anomaly detection task. In this scenarios, code insertion and software trap trigger mechanisms were excluded and only transient and intermittent results were considered. In a future work, scope of the faults and trigger mechanism can be widened to include more diverse fault scenarios. Thus, models are evaluated on a broader aspect and can appeal to a broader scope of application domains. Moreover, combining this advancement with online detection feature could provide a model that is applicable to and functional in a real-time environment. For that purpose, Tensorflow Lite can be utilized. Tensorflow Lite is set of tools for running TensorFlow models on mobile, embedded, and IoT devices [92]. It allows on-device machine learning inference with low latency and a small binary size using its two main components; the Tensorflow Lite which is designed to run specially optimized models on mobile, embedded, and IoT devices, and the TensorFlow Lite converter, which is a tool for

converting TensorFlow models into a form that can be use by the interpreter while applying optimizations for improving binary size and performance.

Moreover, a NN model based on GRUs instead of LSTM networks can be considered for further study due to their relatively simplified architecture. Simplification in GRUs are achieved by merging the input and forget gates into a single gate. In addition, GRUs does not contain explicit memory cells. Their simplified architecture also make GRUs computationally more efficient as compared to LSTM networks. In a related study [98], it was found that GRUs generate similar results to LSTM networks. It was stated in [99] that typical attention mechanisms employed in this thesis selects the relevant information for output generation by reviewing the information at each previous time step and thus, cannot capture the temporal patterns across multiple steps. According to the paper, temporal attention mechanism produced efficient result in multivariate time series forecasting. Further implementation of the mechanism for anomaly detection can provide the boost to attention based models of this thesis for outperforming the other models and therefore worth considering for a further research. It is also worth considering developing a model combining CNNs and RNNs in a single model. In a related study [100], a model combining a CNN and RNN was used for supervised multi-time series anomaly detection. Study concluded that proposed model was suitable for multi-time series anomaly detection since the model showed promising results on the real industrial scenario.

Lastly, in this thesis, design parameters were manually optimized through several experiments using different values of the parameter in question. However, manual optimization is expensive in terms of time and computational power since the number of experiments required to determine the final value of a parameter is unclear. Instead of manually optimizing the design parameters, an open-source Python library, called GpyOPT, which was developed for Bayesian optimization by the Machine Learning group of the University of Sheffield [101], can be used in a further study of this thesis. Advantage of Bayesian optimization over other search methods, such as gradient descent, which was employed in this thesis, random search and grid search, is that Bayesian optimization requires typically requires fewer function evaluations to effectively cover the search space as compared with other mentioned search methods. Bayesian optimization incorporates information learned in previous function evaluations to choose an optimal set of coordinates for the next evaluation. It performs this by computing the posterior predictive distribution for the value of the function at each point. Hyper-parameter tuning, also referred to as design parameter tuning, is one of most common applications of Bayesian optimization [102] considering that deep learning models frequently contain a large number of design parameters for models which subsequently require a considerable amount of training time.

Bibliography

[1] R. Cho. How drones are advancing scientific research. https://phys.org/news/2017-06-drones-advancing-scientific.html, 2017.

[2] A. Zulu and S. John. A review of control algorithms for autonomous quadrotors. volume 04, 01 2014.

[3] M. A. Hayes and M. A. M. Capretz. Contextual anomaly detection in big sensor data. In *2014 IEEE International Congress on Big Data*, 2014.

[4] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. 2009.

[5] A. Rosebeock. Intro to anomaly detection with opencv, computer vision, and scikit-learn. https://www.pyimagesearch.com/2020/01/20/intro-to-anomaly-detection-with-opencv-computer-vision-and-scikit-learn/, 2020.

[6] H. Paulheim and R. Meusel. A decomposition of the outlier detection problem into a set of supervised learning problems. 2015.

[7] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal. Long short term memory networks for anomaly detection in time series. 04 2015.

[8] C. You, Q. Wang, and C. Sun. sbilsan:stacked bidirectional self-attention lstm network for anomaly detection and diagnosis from system logs. 2019.

[9] M. Munir, S. Siddiqui, A. Dengel, and S. Ahmed. Deepant: A deep learning approach for unsupervised anomaly detection in time series. volume PP, 12 2018.

[10] A. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. 2016. cite arxiv:1609.03499.

[11] I. Koren and C. Krishna. Chapter 10 - simulation techniques. In Israel Koren and C. Mani Krishna, editors, *Fault-Tolerant Systems*, Burlington, 2007. Morgan Kaufmann.

[12] Y. Crouzet and K. Kanoun. Chapter 3 - system dependability: Characterization and benchmarking. In *Dependable and Secure Systems Engineering*, volume 84 of *Advances in Computers*. Elsevier, 2012.

[13] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. volume 30, Washington, DC, USA, 1997. IEEE Computer Society Press.

[14] L. Jacobson. Introduction to artificial neural networks - part 1. http://www.theprojectspot.com/, 2013.

[15] S. Ahmadian and A. Khanteymoori. Training back propagation neural networks using asexual reproduction optimization. 2015.

[16] H. Mhaskar, Q. Liao, and T. Poggio. When and why are deep networks better than shallow ones? 2017.

[17] H. Mhaskar and T. Poggio. Deep vs. shallow networks : An approximation theory perspective. 2016.

[18] M. Nielsen. Neural networks and deep learning. Determination Press, 2018.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. The MIT Press, 2016.

[20] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. pages 436–444, 2015.

[21] D. Rumelhart, G. Hinton, and R. Williams. Learning Representations by Back-propagating Errors. volume 323, pages 533–536, 1986.

[22] S. Ruder. An overview of gradient descent optimization algorithms. 2016.

[23] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. 2012.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. volume 15, pages 1929–1958, 2014.

[25] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. 2016.

[26] R. J. Frank, N. Davey, and S. P. Hunt. Time series prediction and neural networks. 2001.

[27] A. Mohamed, G. E. Dahl, and G. Hinton. Acoustic modeling using deep belief networks. 2012.

[28] H. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller. Deep learning for time series classification: a review. volume 33. Springer Science and Business Media LLC, Mar 2019.

[29] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. 2013.

[30] C. Nyugen. Deep learning for information extraction. `https://blogs.itemis.com/en/deep-learning-for-information-extraction`, 2018.

[31] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. 1994.

[32] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[33] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2013.

[34] S. Hochreiter and J. Schmidhuber. Long short-term memory. pages 1735–1780. MIT Press, 1997.

[35] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. volume 12, pages 2451–2471, 2000.

[36] K. Greff, R. Srivastava, J. Koutnik, B. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. volume 28. Institute of Electrical and Electronics Engineers (IEEE), Oct 2017.

[37] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. 2014.

[38] T. Shi, Y. Keneshloo, N Ramakrishnan, and C. Reddy. Neural abstractive text summarization with sequence-to-sequence models: A survey. 2018.

[39] K. Xu, J. Ba, R. Kiros, A. Cho, K.and Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. 2015.

[40] M. Mu Young. Attention mechanism and seq2seq model of tak04-2 attention camp for datawhale group. `https://programming.vip/docs/5e4cadd75dc1d.html`, 2020.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, 2017.

[42] T. Shen, T. Zhou, Long G., J. Jiang, S. Pan, and C. Zhang. Disan: Directional self-attention network for rnn/cnn-free language understanding. 2017.

[43] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. 1989.

[44] X. Sun, L. Liu, C. Li, J. Yin, J. Zhao, and W. Si. Classification for remote sensing data with improved cnn-svm method. 11 2019.

[45] L. Chen, G. Papandreou, I. Kokkinos, Murphy K., and A. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. 2016.

[46] Fully connected layers in convolutional neural networks: The complete guide. `https://missinglink.ai/guides/convolutional-neural-networks/fully-connected-layers-convolutional-neural-networks-complete-guide/`.

[47] Y. LeCun, L. Bottou, G. Orr, and K. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*. Springer, 2012.

[48] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[49] K. Kurita. An overview of normalization methods in deep learning. `https://mlexplained.com/`, 2018.

[50] J. Ba, J. Kiros, and G. Hinton. Layer normalization. 2016.

[51] J. Xu, X. Sun, Z. Zhang, G. Zhao, and J. Lin. Understanding and improving layer normalization. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[52] M. Gupta, J. Gao, C. Aggarwal, and J. Han. Outlier detection for temporal data. Morgan & Claypool Publishers, 2014.

[53] I. Chang, G. Tiao, and C. Chen. Estimation of time series parameters in the presence of outliers. 1988.

[54] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. 2002.

[55] D. Hill and B. Minsker. Anomaly detection in streaming environmental sensor data: A data-driven modeling approach. Elsevier Science Publishers B. V., 2010.

[56] K. Hollingsworth, K. Rouse, J. Cho, A. Harris, M. Sartipi, S. Sozer, and B. Enevoldson. Energy anomaly detection with forecasting and deep learning. 2018.

[57] S. Chauhan and L. Vig. Anomaly detection in ecg time signals via deep long short-term memory networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2015.

[58] L. Bontemps, J. Cao, V.and McDermott, and N. Le-Khac. Collective anomaly detection based on long short term memory recurrent neural network. 2017.

[59] Y. He and J. Zhao. Temporal convolutional networks for anomaly detection in time series. volume 1213. IOP Publishing, jun 2019.

[60] L. Meier, D. Honegger, and M. Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[61] T. Baumann, B. Pfitzinger, T. Jestädt, and D. Macos. The role of simulation performance in software-in-the-loop simulations. In *Advances in Business ICT: New Ideas from Ongoing Research*, pages 17–26. Springer International Publishing, 2017.

[62] E. Zürich. Qgroundcontrol: Ground control station forsmall air land water autonomous unmanned systems. `http://qgroundcontrol.com/`, 2013.

[63] Koenig N. and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, 2004.

[64] the Dronecode Project, Inc., a Linux Foundation Collaborative Project. Px4 development guide. `https://dev.px4.io/master/en/index.html`, 2018.

[65] P. Raj, A. Raman, and H. Subramanian. Architectural patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture. 2017.

[66] What is pub/sub messaging? `https://aws.amazon.com/pub-sub-messaging/`, 2020.

[67] the Dronecode Project, Inc., a Linux Foundation Collaborative Project. Mavlink developer guide. `https://mavlink.io/en/`, 2020.

[68] P. Bernstein and E. Newcomer. Chapter 1 - introduction. In *Principles of Transaction Processing (Second Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 1 – 29, San Francisco, 2009. Morgan Kaufmann.

[69] Chapter 1 - introduction. In S. Mukherjee, editor, *Architecture Design for Soft Errors*, pages 1 – 41, Burlington, 2008. Morgan Kaufmann.

[70] T. Given-Wilson, N. Jafri, and A. Legay. The state of fault injection vulnerability detection. 08 2018.

[71] A. Spilla, I. Polian, J. Müller, M. Lewis, V. Tomashevich, B. Becker, W. Burgard, and Albert-Ludwigs-University. Run-time soft error injection and testing of a microprocessor using fpgas. 2011.

[72] R. Ecoffet. In-flight anomalies on electronic devices. pages 31–68, 01 2007.

[73] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. volume 94, pages 370–382, 2006.

[74] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. volume 42, pages 913–923, 1993.

[75] A. Benso and S. DiCarlo. The art of fault injection. volume 13, pages 9–18, 2011.

[76] J. Clark and D. Pradhan. Fault injection: a method for validating computer-system dependability. pages 47–56, 1995.

[77] H. Ziade, R. Ayoubi, and R. Velazco. A survey on fault injection techniques. pages 171–186, 01 2004.

[78] M. Kooli and G. Natale. A survey on simulation-based fault injection tools for complex systems. 05 2014.

[79] Me. Yang, G. Hua, Y. Feng, and J. Gong. Fault-tolerance techniques for spacecraft control computers. Wiley Publishing, 2017.

[80] A. Bosio and G. D. Natale. Lifting: A flexible open-source fault simulator. In *2008 17th Asian Test Symposium*, pages 35–40, 2008.

[81] F. Wang and V. D. Agrawal. Single event upset: An embedded tutorial. In *21st International Conference on VLSI Design (VLSID 2008)*, pages 429–434, 2008.

[82] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, pages 66–75, 1994.

[83] J. Guthoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 196–206, 1995.

[84] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[85] S. Bai, J. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. 2018.

[86] M. Lin, Q. Chen, and S. Yan. Network in network. 2013.

[87] M. Baron. Probability and statistics for computer scientists, second edition. Chapman and Hall/CRC, 2013.

[88] M. Neira, M. Gomez, F. Suarez-Perez, D. Gomez, J. Reyes, M. Hoyos, P. Arbelaez, and J. Forero-Romero. Mantra: A machine learning reference lightcurve dataset for astronomical transient event recognition. 2020.

[89] Y. Nan, K. Chai, W. Lee, and H. Chieu. Optimizing f-measure: A tale of two approaches. 2012.

[90] B. Stroustrup. The c++ programming language. Addison-Wesley Professional, 2013.

[91] F. Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[92] M. Abadi, A. Agarwal, P. Barham, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015. Software available from tensorflow.org.

[93] Python software foundation. python language reference, version 2.7. `http://www.python.org`.

[94] NVIDIA Corporation. Nvidia cuda compute unified device architecture programming guide. NVIDIA Corporation, 2007.

[95] G. Box and G. Jenkins. Time series analysis: Forecasting and control. Holden-Day, 1976.

[96] N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 2016.

[97] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. Jul 2018.

[98] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[99] S. Shih, F. Sun, and H. Lee. Temporal pattern attention for multivariate time series forecasting. 2018.

[100] K. Canizo, I. Triguero, A. Conde, and E. Onieva. Multi-head cnn-rnn for multi-time series anomaly detection: An industrial case study. volume 363, 2019.

[101] The GPyOpt authors. Gpyopt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[102] N. Knudde, J. an der Herten, T. Dhaene, and I. Couckuyt. Gpflowopt: A bayesian optimization library using tensorflow. 2017.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel ,,Investigating Artificial Neural Networks for Monitoring an Unmanned Air Vehicle" selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Hamburg, August 18, 2020

Ort, Datum

Emin Çagatay Nakilcioglu