Universität Stuttgart
Institut für Automatisierungstechnik
und Softwaresysteme

Master's Thesis Report

Deep Learning based Monitoring of the Urban Traffic using MOBATSim

Submitted by

Nikita Jhawar

Degree Program: M.Sc. Electrical Engineering
Enrollment ID: 3441310
Examined by: Jun.-Prof. Dr.-Ing. Andrey Morozov
Supervised by: Sheng Ding, M.Sc.
Submitted on: 13 June, 2022

Abstract

The concept of autonomous vehicles adapting in a traffic scenario is expanding at a rapid pace due to the increasing advancement of information, communication, and sensor technology applications, thus, offering a broad range of opportunities in terms of safety [1]. A technique, completely based on Simulink MATLAB, called MOBATSim [2] helps to analyze and ensure the safety of autonomous driving systems on component, vehicle, and traffic levels by using fault injection method. This framework helps in designing various driving scenarios and supports fully fault-error free chain analysis. MOBATSim is an extended version of autonomous vehicle system, consisting of multiple vehicles in different traffic scenarios, thus showcasing a real-time behaviour of fault-injected vehicles [3].

In this thesis, we focus mainly on the safety analysis of autonomous vehicles by using Simulink-based Fault Injection (FI) block [4] in sensor level, on traffic scenarios in MOBATSim . In order to perform fault injection at both speed and distance sensor target variables, the simulink model is subjected to some modifications in two of the vehicle models out of 10 (vehicle 2 and vehicle 6) that includes injecting sensor faults like noise, stuck-at, and bias/offset to the sensor module of the system. MOBATSim model collects data for every feature of the vehicle like speed, rotation and translation. Since, speed is one of the majorly affected feature due to sensor faults, faulty and fault-free data is collected separately for just the speed feature analysis. Therefore, this model is repeatedly simulated to collect faulty and fault-free for two types of data i.e. uni-variate(only speed) and multivariate(speed, rotation, position and translation). After this, Deep Learning models to detect and classify faults are proposed such as Simple RNN, GRU-Simple RNN, BiGRU-BiSimpleRNN, Transformers, Autoencoders and LSTM-Autoencoders, to detect the sensor-level faults and classify them based on their fault category for both uni-variate and multivariate data.

Finally, different results of fault detection and classification for both types of data is represented based on metrics such as accuracy, precision, recall, and F1 score for all the implemented DL models with extensive training and testing. After a comprehensive tabular comparison between the results obtained by different models, it is shown that, the **LSTM-Autoencoders** performs the best out of all the proposed DL models for both types of data with **83%** test accuracy for multivariate data and **77%** test accuracy for uni-variate data.

Contents

Nomenclature

AE Auto-encoder

AI Artificial Intelligence

ANN Artificial Neural Network

$BiGRU$ Bidirectional Gated Recurrent Unit

$BiSimpleRNN$ Bidirectional Simple Recurrent Neural Network

CPS Cyber Physical System

$DLAD$ Deep Learning Anomaly Detection

DL Deep Learning

FI Fault Injection

GRU Gated Recurrent Unit

$LSTM$ Long Short-Term Memory

ML Machine Learning

$MOBATSim$ Model-based Autonomous Traffic Simulation

RNN Recurrent Neural Network

Chapter 1

Introduction

Traffic models are essential tools for planning and operating road traffic infrastructure. Modeling of a traffic system helps in understanding the aggregated phenomena arising from complex interactions and also in predicting the behavior of the traffic system in the near or far future, depending on the application [5]. It involves descriptive study of the traffic (road) infrastructure, and assumptions on the road users' behavior. Discussions about the future of autonomous and connected vehicles and intelligent traffic systems are encountered more and more frequently in recent years [6]. One main use of automation is to optimize the transport system and make it more safer.

MOBATSim is a Model-based Autonomous Traffic Simulation Framework for Safety Assessment, completely based on MATLAB Simulink. Customization of decision and control algorithms for the modeled autonomous vehicles and observing their effects on the overall safety of the high-level urban traffic environment is obtained by this model. Safety goals, derive functional safety requirements and driving scenarios can be defined, and also have the capability to verify if these goals are achieved for the tested autonomous vehicle functions [2]. These requirements for safety and control of autonomous vehicle functions are derived from the safety standard ISO 26262 [7] [8]. Automatically generated reports saves user time and costs in the early design phase through comprehensive testing and logging of data during simulation itself [9]. The output from traffic simulation is used for traffic engineering analysis and also for developing and testing various forms of traffic control, e.g. variable speed limit systems and ramp metering, which in the simulation include vehicle-to-vehicle communication and automation [10].

The functional safety assessment for autonomous driving system becomes an urgent necessity for the transition to full autonomy [11]. Testing decision and control algorithms with a lot of variables and parameters in a unified manner is a tiresome task and threat assessment has to be made for vehicles to actively avoid hazardous situations. This requires the analysis of complex operational profiles such as routing, intersection management and collision prediction in an environment where multiple vehicles are in different positions, and traveling at different speeds [12]. So, in order to accomplish this, comprehensive traffic simulation framework like MOBATSim which models not only the functionality of the vehicles but also the interactions between them is required. The simulation-based fault injection performs tests with various types of random or predetermined faults of low-level vehicle components such as sensors and communication modules[13].

In case of MOBATSim, faults may occur at the sensor level, hardware level, or network level. Interpretation and Classification of these faults in order to determine the root cause is highly contextual and uncertain. MOBATSim is used to simulate urban city traffic, design intersection management algorithms for the infrastructure or path planning algorithms for vehicles and test the efficiency of your algorithms by PC-based simulations and also simulink 3D Animation with V-Realm is used to visualize the driving scenarios.

In this thesis, an extensive investigation is done on various sensor faults that can occur especially in the speed and distance targets. A Simulink-based MOBATSim model is used to generate data for serving our purpose i.e. by introducing fault injection to distance and speed sensors. Also, the automated data generation and collection processes are designed and the data sets are stored finally in two types of formats explained later.

Finally, deep learning models like Simple RNN, Autoencoder based Classifier, Transformer based Classifier, Hybrid GRU- Simple RNN, Bidirectional GRU- Bidirectional Simple RNN and Hybrid LSTM-Autoencoder architecture in order to detect and classify faults [14] are used. The obtained results on the test data are then compared based on the performance metrics (accuracy, precision, recall, and F1 score).

The rest of this thesis is comprehended in the following way: Chapter 2 (Related Work) discusses some related works regarding MOBATSIM and fault injection as well as anomaly detection in timeseries data . Chapter 3 (Theory) discusses MOBATSim as a Cyber-Physical System and the types of faults that occur in the sensor unit. Also, an introduction is provided to Deep Learning algorithms that are used to evaluate the performance of data sets. In Chapter 4 (Model-based Autonomous Traffic Simulation Framework), MOBATSim model is discussed in detail and also showing different scenarios existing in the model. Chapter 5 (Thesis Workflow shows the overall workflow of this thesis. In Chapter 6 (Data Generation), the data collection using faulty scenarios is shown . All the data pre-processing steps in order to implement the Deep Learning algorithms are noted in Chapter 7 (Data Pre-processing). Chapter 8 (Fault Classification Training of Deep learning Models) shows all of the implemented Deep Learning algorithms to detect and classify faults in different types of data, and Chapter 9 (Results) shows the comparative results between the deep learning model techniques for both types of data. Finally, Chapter 10 (Conclusion and Future Scopes) provides a conclusion to this thesis with remarks and future scopes.

Chapter 2

Related Work

Anomaly detection is ultimately crucial to ensure the security of cyber-physical systems (CPS). But due to the increasing complexity of these systems and more sophisticated attacks, conventional anomaly detection methods like state estimation (e.g., Kalman filter), statistical model (e.g., Gaussian model, histogram-based model) based methods, which face the growing volume of data and need domain-specific knowledge, cannot be directly applied to address these challenges [15]. Since these methods cannot capture unique attributes and fail to detect and classify the anomalies, different Deep Learning based models for fault classification come into picture [16]. Since the main feature of this thesis is to generate fault injected data from MOBATSim Urban Traffic Model and evaluate the performance for fault classification using Deep Learning models,related work regarding MOBATSim and anomaly detection using deep learning models is discussed further.

One of the prominent research works on safety analysis and fault injection in autonomous driving system in MOBATSim is performed in [3]. The simulation framework, designed in MATLAB Simulink, provides the building blocks for modeling various real-world scenarios. It consists of different types of fault were injected at simulation run-time and the outputs were examined to verify the safety criteria. An illustrative case study is shown to analyze safety criteria after injecting fault in one of the 10 vehicles in urban traffic model. In this thesis, the method of fault injection and target variables are changed as well as fault injection is performed in more than one vehicle in different scenarios.

Different researches on Deep learning models have proved successful in order to solve complex fault detection and classification problems with a reduction the limitations that exist in the classical techniques [17] [18]. Research on detecting anomalous sub-sequences in time series data have proved to be quite successful with different deep learning algorithms using various scoring metrics and also proves that no single algorithm achieves perfect scores [19]. It is also observed that anomalies on periodic time series are easier to detect than on non-periodic time series and anomaly detection on univariate time series is on average easier than on multivariate time series [20].

Other important research works [21] [22] shows comprehensive evaluation of unsupervised and semi supervised deep-learning-based methods for anomaly detection and diagnosis on multivariate time series data from cyberphysical systems and also propose the composite F-score for evaluating time-series anomaly detection [23]. The

effect of models and scoring functions are studied independently to gain a deeper understanding of what makes a good time-series anomaly detection algorithm [24]. It is also found that the choice of the scoring function can have a large impact on anomaly detection performance, and dynamic scoring functions Gauss-D and Gauss-D-K work better than the static scoring function, Gauss-S.

The importance on forecasting multi-step ahead both multivariate and univariate time series by implementing different algorithms is clearly visible [25]. Another research on Anomaly detection multivariate time series [26] shows experiments being conducted on different datasets, significantly outperforming recent state-of-the-art MTS anomaly detection methods. This work provides important insights into the design and evaluation of methods for anomaly detection and diagnosis, and serves as a useful guide for future method development. Anomaly diagnosis has been approached primarily from a supervised classification perspective. In the unsupervised context, studies mention that ranking of scores or errors can be used to diagnose the cause of anomalies [27] . Therefore, in this thesis, a comparison between different deep learning algorithms for both uni-variate and multivariate data sets showing F1 score ,accuracy, precision and recall is represented.

Chapter 3

Theory

In the previous chapters, we have discussed about introduction to MOBATSim from a Cyber Physical system perspective and discuss about fault injection method used to generate data in Simulink-Based Model of MOBATSim . This chapter also discusses details about the Deep Learning techniques and algorithms used for fault detection and classification.

3.1 MOBATSim: Model-based Autonomous Traffic Simulation

MOBATSim is a framework based on MATLAB and Simulink that allows users to develop automated driving algorithms and assess their safety and performance. With the help of this, the safety of the implemented component or algorithm can be measured on both the vehicle level and the traffic level. Since, it is a type of cyber physical systems (CPS) that is defined as systems of collaborating computational entities that are in thorough connection with the neighboring physical world as well as its ongoing processes, thus enabling data processing services [28]. A general CPS system consists of five intersecting components 3.1 as listed below:

- **Physical Space**: The physical space contains physical components of CPSs, e.g., engines, tanks, wheels [15].
- **Actuators**: Actuators receive control commands from control systems and change the running parameters of physical devices [15].
- **Sensors**: Sensors measure the running status of devices and report to the control systems [15].
- **Control System**:Control systems obtain sensor measurement and send control commands to actuators, which follows the predefined control logic [15].
- **Supervisory Control and Data Acquisition (SCADA)**: SCADA systems are used to gather data from control systems and monitor the running status of CPSs for users [15].

Cyber-physical systems (CPSs) are the integration of computation, communication, and control that achieve the desired performance of physical processes [28]. In this thesis, we work with the Traffic Model of MOBATSim and focus on new research efforts that detect anomalies in this type of CPS with the help of emerging deep learning methods.

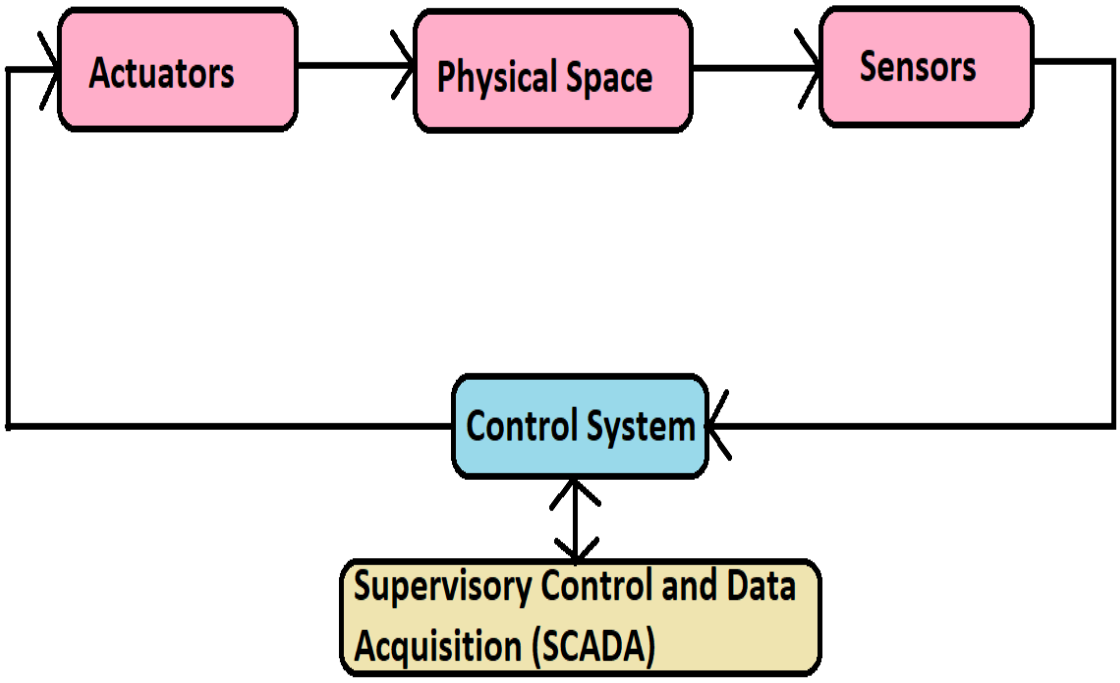

Figure 3.1: CPS block diagram [15]

One of the main features of MOBATSim is that it acts as a solution reflect different control and decision algorithms for autonomous vehicles and autonomous traffic management. It helps in Error propagation analysis for the derivation of fault-error-failure chain relations by simulation-based fault injection method and also enables Scenario-based testing for hazard and risk analysis. MOBATSim provides Simulink 3D Animations for visualization and works 100% Simulink. It allows PC-based simulation with fast, cheap and reproducible tests. In this model, customization of any parameter regarding any vehicle, the driving scenario, or the traffic model is possible. It also permits benchmarking of control and decision algorithms in terms of safety and performance on different abstraction levels such as component level, vehicle level, and traffic level as well as clear structure and Input/Output interface allow the user to inject any fault type(such as sensor, hardware or network faults), thus making simulation-based fault injection a useful method for safety assessment [2].

The automotive applications widely uses Model-based design and allows the modeling of complex systems in a modular way to speed up the procedures of design, integration, and testing [12]. MOBATSim is a common environment for model-based system design and the simulation framework consists of three main parts: the simulation environment, the traffic model, and the autonomous vehicle model. The simulation environment consists of animation using Simulink 3D Animation having non-intractable objects such as buildings and intractable objects such as vehicles and roads. These objects are designed using V-Realm Builder and SolidWorks. The Traffic Model contains a city road map is stored in an m-file as a digraph. The map file is generated using Solid Works 2D Sketch and serves as an input to MOBATSim. The traffic flow of the roads is defined by edges as one-way routes between the nodes of the digraph, which depict checkpoints on the map. Some of these nodes are used

Figure 3.2: Traffic Model of MOBATSim

as starting and destination points for vehicles. The autonomous vehicle model consists of four main interfaces; Perception, Decision-making, Trajectory Planner and Communication Output and allows the user to easily access and customize the low-level components, their inputs/outputs, and the implemented decision and control algorithms by clear interfaces between these components [2].

3.2 Faults in MOBATSim

The Cyber Physical Systems generate unexpected faults because of complexity of systems and heterogeneity of devices [15]. CPS shows faults in two layers:

- **Sensor layer:** One of the most common fault in sensor layer is false sensor value. Physical damage or flaw lead sensors to report inaccurate sensor values. Also, previously unseen circumstances may cause sensors to work beyond their abilities.
- **Control system:** Since there are always situations that may not be covered during the system design stage, CPSs typically hold the dynamic running characteristic. Different orders and timings of events can cause object collisions.

According to previous research [3], it is evident that, MOBATSim has three types of faults i.e. hardware faults (single and multiple bit-flips), sensor faults (offsets, stuck-at faults, and noise), and network faults (delays and packet drops) [29]. In our thesis, we only inject sensor faults but an in-depth analysis of various fault scenarios and their mitigation policies should be studied.

3.3 Fault-Injection

In this research work, Simulink's FI block [4] is used for the injection of the sensor faults discussed above. The FI block has two input ports (Idata and Iflag) and two output ports (FData and Fflag). The FIBlock injects different types of fault with a certain stochastic method and duration where the user is able to turn on and off specific FI Block, in case when the fault injection on a certain block is not necessary.

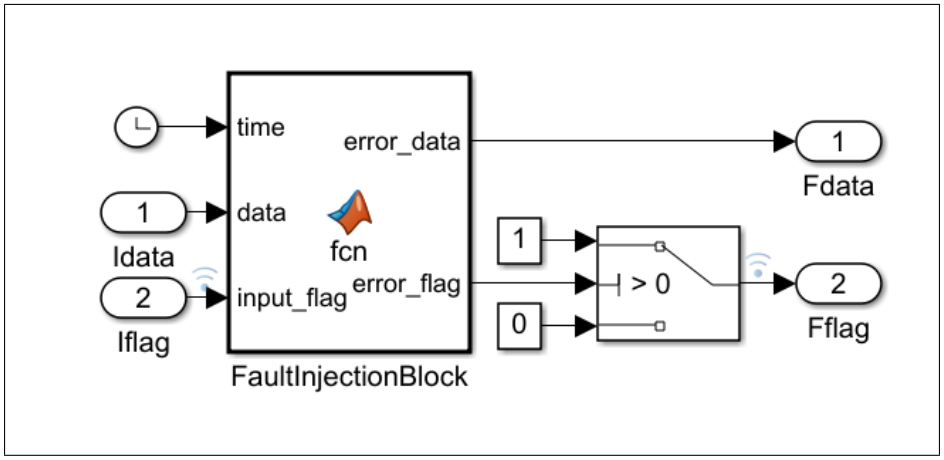

Figure 3.3: Structure of FI block [4]

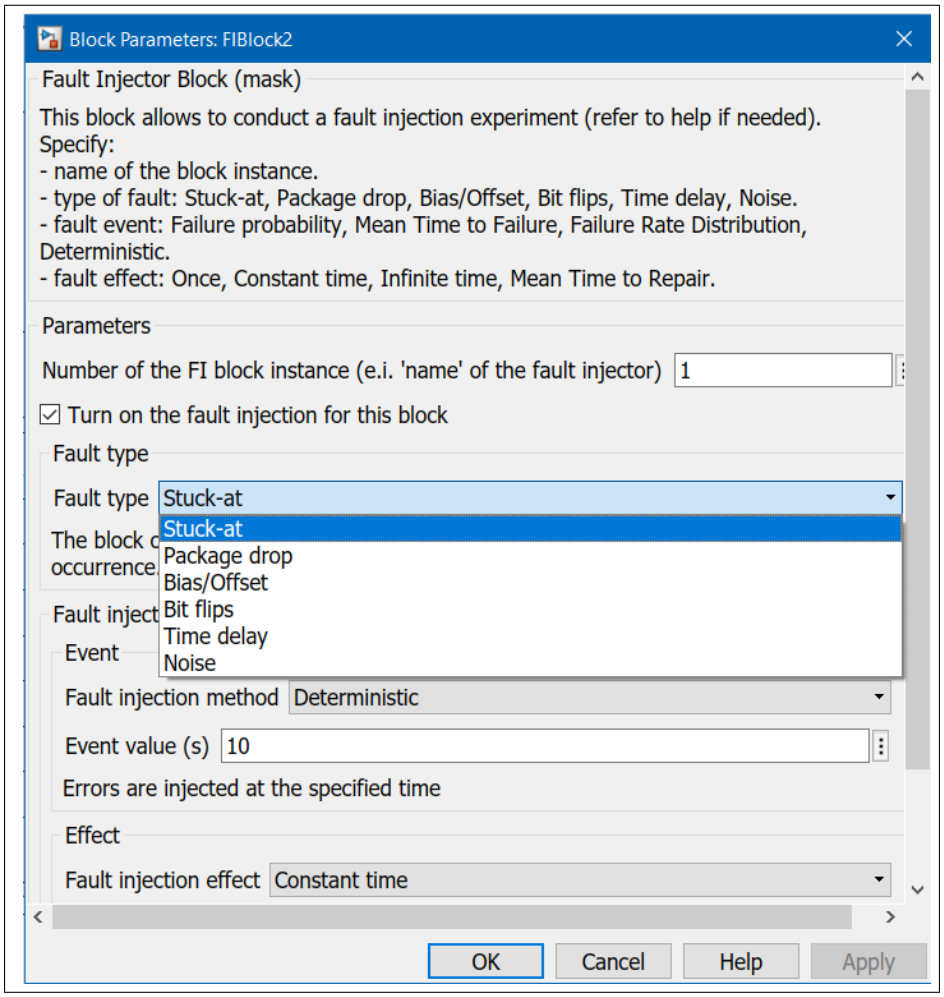

Figure 3.4: Fault Injection Block Features

This block allows to conduct a fault injection experiment on the Simulink Model by specifying four things [30]:

- **Number of the block:** It specifies the 'name' of the fault injector and the given name has to be unique.
- **Types of fault:**
  1. **Stuck-at:** The block output stays constant, preserving the latest correct value before the error occurrence.
  2. **Package drop:** The correct output is replaced by the specified Value, emulating a package drop.
  3. **Bias/Offset:** The defined positive or negative Bias value is added to the block output.
  4. **Noise:** A random noise value is added where the boundaries are defined as the percentage of the correct value.
  5. **Time delay:** A delay is introduced into the signal where during the delay the value is zero.
  6. **Bit flips:** The defined number of bits are inverted in the binary representation of the correct value.
- **Fault events:** In Failure probability: errors are injected based on the constant failure probability for each execution of the block function and in Mean Time to Failure: errors are injected according to the specified MTTF, normal distribution whereas in Deterministic: errors are injected at specific time step.
- **Fault effect:** When Once is selected: an error appears only one time during a simulation; when Constant time is selected: the block produces erroneous output during the specified time period; when Infinite time is selected: the block produces erroneous output until the end of the simulation; and when Mean Time to Repair is selected: normally distributed MTTR regulates the time of the error effect.

Fault injection methods help to evaluate system fault tolerance. In this work, noise, stuck-at and offset/bias faults are injected to the speed and distance sensor data. The fault-injection is started by controlling the Iflag port using a random pulse generator.

3.4 Deep Learning Algorithms

Deep learning algorithms is known to be a sophisticated and mathematically complex evolution of machine learning algorithms [31] shown in figure 3.5. Deep learning describes algorithms helps to analyze data with a logic structure to draw conclusions and works for both through supervised and unsupervised learning [32]. To achieve this, deep learning applications use a layered structure of algorithms called an artificial neural network (ANN) [33] [34]. While deep learning algorithms feature self-learning representations, they depend upon ANNs that mirror the way the brain computes information. During the training process, algorithms use unknown elements in the input distribution to extract features, group objects, and discover useful data patterns [35]. Deep learning models make use of several algorithms like Convolutional Neural Networks (CNNs) [36], Recurrent Neural Networks (RNNs) [37], Autoencoders [38], etc.

In this thesis, Deep Learning Models used are Simple RNN, GRU-Simple RNN, Bi-directional GRU-Bi-directional Simple RNN, Autoencoder, Transformer based classifier and LSTM-autoencoder. Figure 3.6 shows the work flow architecture of Deep learning Anomaly Detection (DLAD).

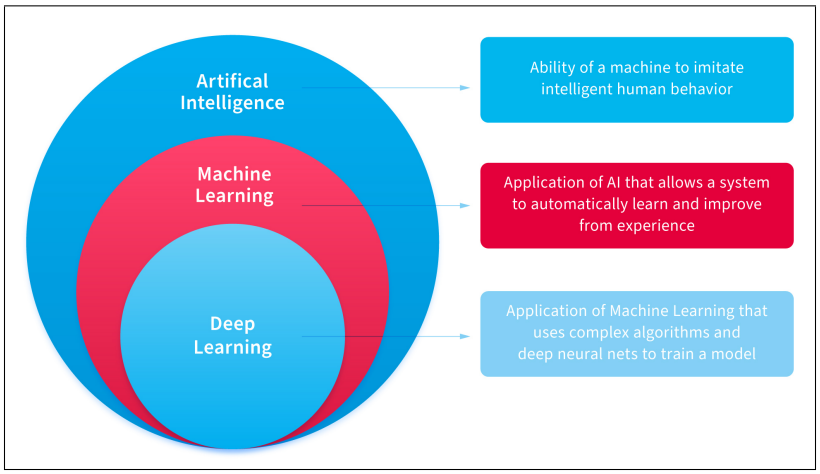

Figure 3.5: DL vs ML vs AI [31]

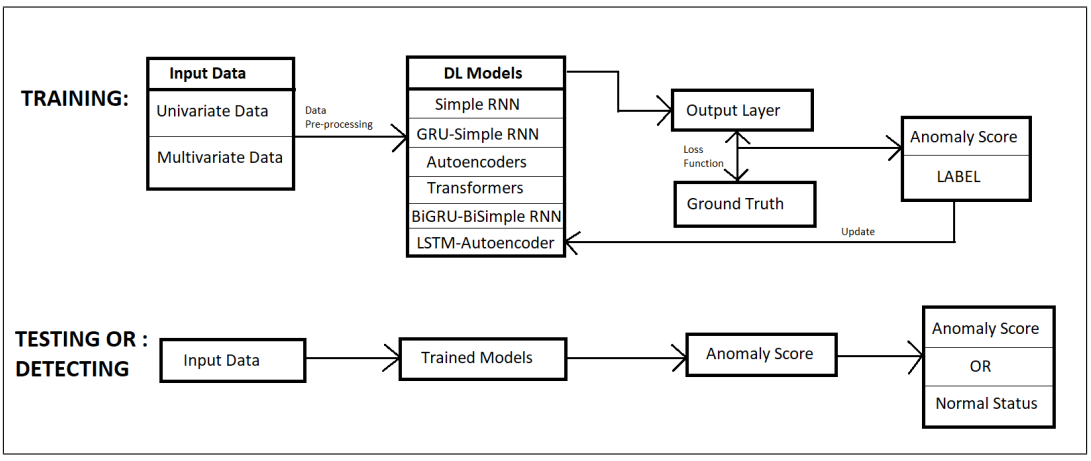

Figure 3.6: Workflow of DLAD method [15]

Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) differ from standard neural networks by allowing the output of hidden layer neurons to feedback and serve as inputs to the neurons. RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer [39] [40]. Feed-forward neural network has issues like it cannot handle sequential data, it only considers the current input and it cannot memorize previous inputs and a solution to all these issues is RNN as it compensates for all the mentioned issues [41] [42].

Bidirectional recurrent neural networks (BRNN) helps in connection of two hidden layers of opposite directions to the same output and helps the output layer can get information from past (backwards) and future (forward) states simultaneously [43]. The input sequence is fed in normal time order for one network, and in reverse time order for another. Applications of BRNNs are : Speech Recognition, Translation, Handwritten Recognition, Protein Structure, Prediction, Part-of-speech tagging, Dependency Parsing, and Entity Extraction.

There are three built-in RNN layers : Simple RNN, GRU and LSTM.

- **Simple Recurrent Neural Networks (RNN):** Simple RNN is a fully-connected RNN where the output from previous time step is to be fed to next time step [44]. Built-in RNNs supports many features like Recurrent dropout, Ability to process an input sequence in reverse, and Loop unrolling. The output of SimpleRNN will be a 2D tensor of the mentioned shape. RNNs are by definition capable of handling multivariate sequential input data without any modifications and treat it as time series data. In this model, the it consists of embedding layer where sentences will be represented as max length by embedding dim vectors with a simple RNN layer and then the dense layers [45].

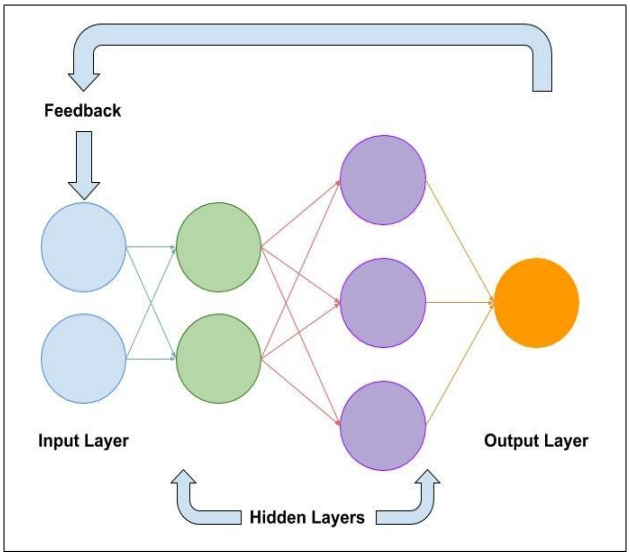

Figure 3.7: Simple RNN Architecture [44]

- **Long Short-Term Memory (LSTM):** LSTM networks are very well-suited to classifying, processing and making predictions based on time series data [46].

LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. LSTMs have a slightly more complex structure where at each time step, the LSTM cell takes in 3 different pieces of information – the current input data, the short-term memory from the previous cell and lastly the long-term memory showing that LSTM cell has 3 gates, named as input gate, forget gate, and output gate as shown in the Fig. The short-term memory is referred to as the hidden state, and the long-term memory is known as the cell state. LSTMs also have this chain like structure, but instead of having a single neural network layer, there are four, interacting in a very special way [47] [48].

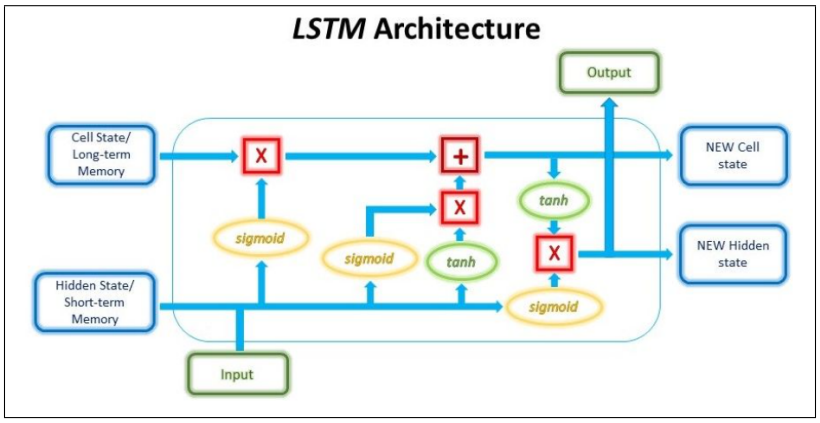

Figure 3.8: LSTM Architecture [46]

- **Gated Recurrent Units (GRU):** Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks. This is an improved version of the RNN model and is more efficient than Simple RNN models. It has two gates: reset and update. GRU's performance on certain tasks of polyphonic music modeling, speech signal modeling and natural language processing was found to be good [49]. GRUs have been shown to exhibit better performance on certain smaller and less frequent datasets [50]. GRU is efficient that LSTM by preserving the advantages of LSTM's ability to learn the context of the context, and shortens the training time because the structure is smaller than the LSTM [51].

In this thesis, algorithms like Simple RNN and hybrid algorithm of GRU-Simple RNN is used for anomaly detection [52] [53]. All these three layers also have bi-directional algorithms that have been developed for various of applications. Generally, Bi-LSTM is used for anomaly detection [48] [54], so here we have considered using hybrid algorithm of Bi-Simple RNN and Bi-GRU [43] to understand their performance and classification techniques for anomaly detection [55].

Autoencoder Based Classifier

Autoencoder is a type of neural network that is referred to learn a compressed representation of raw data as it is composed of an encoder and a decoder sub-models [38]. The encoder compresses the input and the decoder attempts to recreate the input from the compressed version provided by the encoder. After training the model, the encoder is saved and the decoder is deleted. The encoder can then

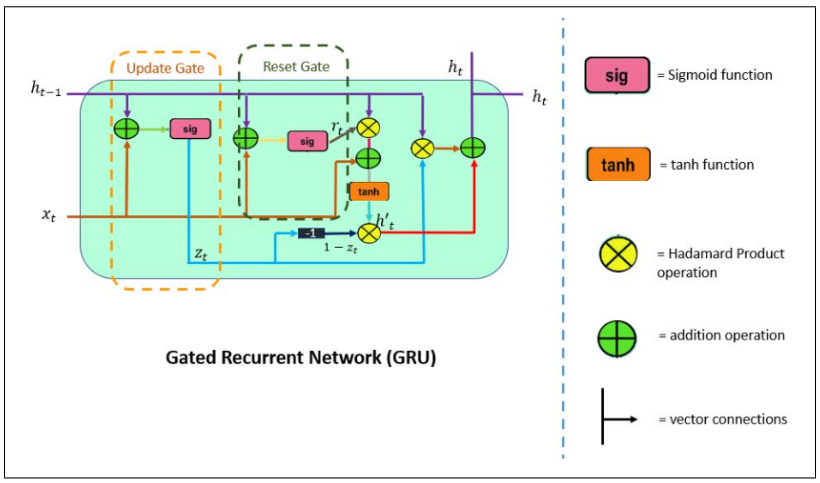

Figure 3.9: GRU Architecture [49]

be used as a data preparation technique in order to be able to detect and classify anomalies in the data [56].

The design of the autoencoder model consists of an architecture consisting of a bottleneck at the midpoint of the model, from which the reconstruction of the input data is performed. There are many types of autoencoders, and their use varies, but perhaps the more common use is as a learned or automatic feature extraction models [57] .

The autoencoder techniques can perform non-linear transformations with their non-linear activation function and multiple layers [58].If the number of neurons in the hidden layers is more than those of the input layers, the neural network will be given too much capacity to learn the data, therefore, in an autoencoder model, the hidden layers are supposed to have fewer dimensions than those of the input or output layers.

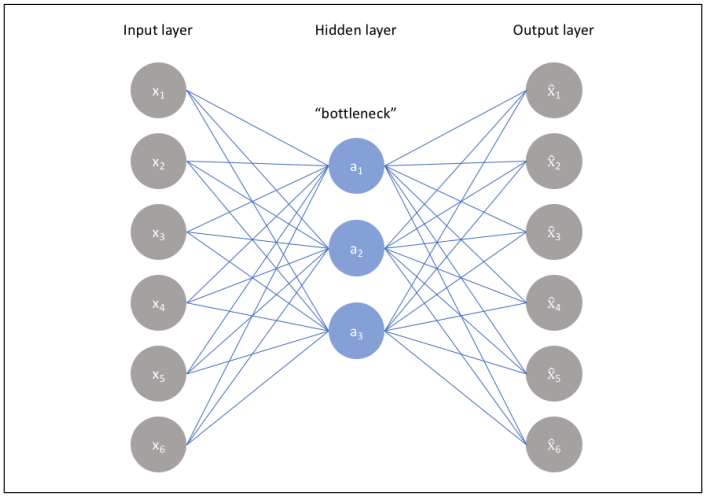

Figure 3.10: Autoencoder Based Classifier [56]

Transformer Based Classifier

A transformer model is a neural network that learns context and thus meaning by tracking relationships in sequential data. Transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways even distant data elements in a series influence and depend on each other.Transformers are popular deep learning models that have been used in various natural language and vision processing tasks [59].

Transformers have demonstrated great results in sequential data processing, such as natural language processing, audio processing and computer vision. Benefiting from the advantage of the self-attention mechanism, Transformers are used to discover the reliable long-range temporal dependencies, thus, helping in building temporal modeling and the reconstruction criterion for anomaly detection [60]. Anomaly Transformer renovates the self-attention mechanism to the Anomaly-Attention based on associated discrepancy [61].

The transformer neural network receives an input and converts it into two sequences: a sequence of word vector embeddings, and a sequence of positional encodings as shown in Fig 3.11. The transformer adds the word vector embeddings and positional encodings together and passes the result through a series of encoders, followed by a series of decoders. Encoders convert their input into encodings and the decoders do the reverse. Both encoder and decoder contains a component called the attention mechanism, extracting relevant information from every input. The ability to use multi-head attention mechanism is one advantage of transformers over LSTMs and RNNs [62]. Transformer neural networks are useful for many sequence-related deep learning tasks, such as machine translation, information retrieval, text classification, document summarization, image captioning, and genome analysis [59].

Hybrid LSTM-Autoencoder Architecture

LSTM-Autoencoder is an encoder that makes use of LSTM encoder-decoder architecture to compress data using an encoder and decode it to retain original structure using a decoder. The Autoencoder is a kind of unsupervised learning [63], which can be used as feature extractor of data and the extracted features are put into the prediction network shown in the Fig 3.12. The AutoEncoder is divided into two parts, which are the encoder and the decoder where the encoder, as a data feature extractor, is part of the LSTM-AE prediction model and the decoder is used to verify the validity of the extracted characteristics, and is discarded after the training. LSTM model after the encoder forms an efficient LSTM-AE prediction model for better f1 accuracy for anomaly detection [64].

In conclusion, LSTM-Autoencoder is configured to read the input sequence, encode it, decode it, and recreate it and the performance of the model is evaluated based on the model's ability to recreate the input sequence [65]. LSTM-Autoencoders can learn a compressed representation of sequence data and have been used on video, text, audio, and time series sequence data.

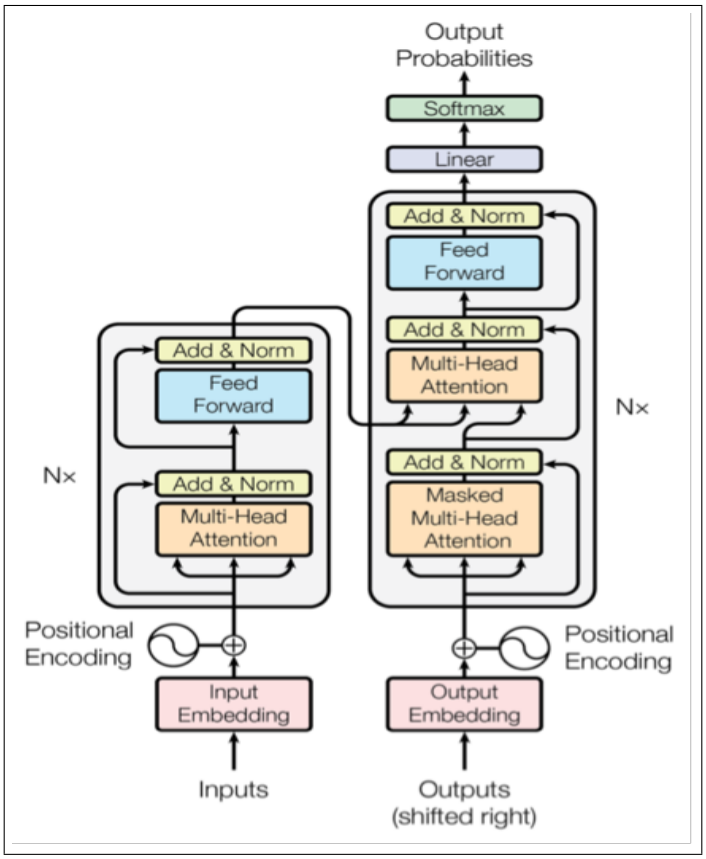

Figure 3.11: Transformer Neural Network Architecture [59]

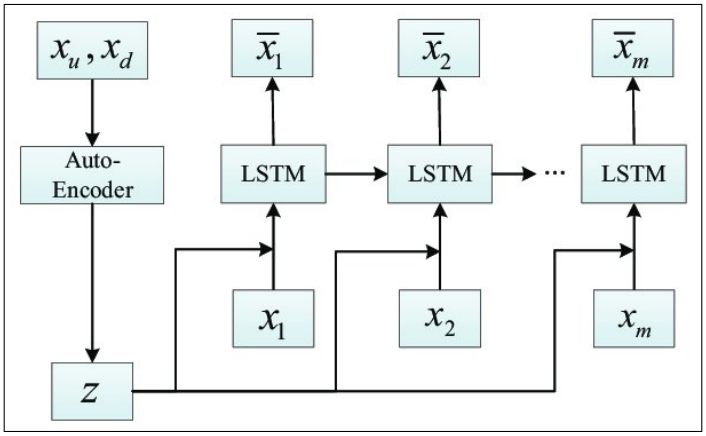

Figure 3.12: LSTM-Autoencoder Architecture [65]

Chapter 4

Model-based Autonomous Traffic Simulation Framework

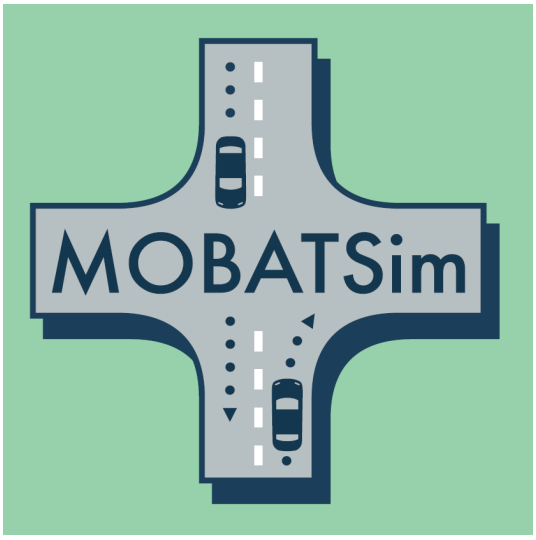

Figure 4.1: MOBATSim [2]

MOBATSim [2] [3], entirely based on MATLAB Simulink, allows the user to customize decision and control algorithms for the modeled autonomous vehicles as discussed in the previous chapter and analyze their effects on the overall safety of the high-level urban traffic environment. The user can define safety goals, derive functional safety requirements, describe driving scenarios, and verify if these goals are met for certain autonomous vehicle functions that are tested. The simulation-based fault injection is used to perform the tests in the presence of various types of random or predetermined faults of low-level vehicle components such as sensors and communication modules. Automatically generated reports allow the user to save time and costs in the early design phase through comprehensive testing and the data can be logged during the simulations and later be used by Simulink 3D Animation for visual investigations.

4.1 The Workflow Diagram

MOBATSim contains multiple m-files, MATLAB classes, and Simulink models. The initialization of the simulation framework starts with an m-file which prepares the

required workspace variables, generates the map and the vehicles on a 2D plot, and opens the main Simulink model src/Simulink Models/MOBATSim.slx. By running the Simulink model, the workspace variables that contain instances of the Vehicle Class, their parameters, intersection manager, and constant parameters required by the model are loaded. The main workflow of MOBATSim is shown abstractly in the Fig 4.2

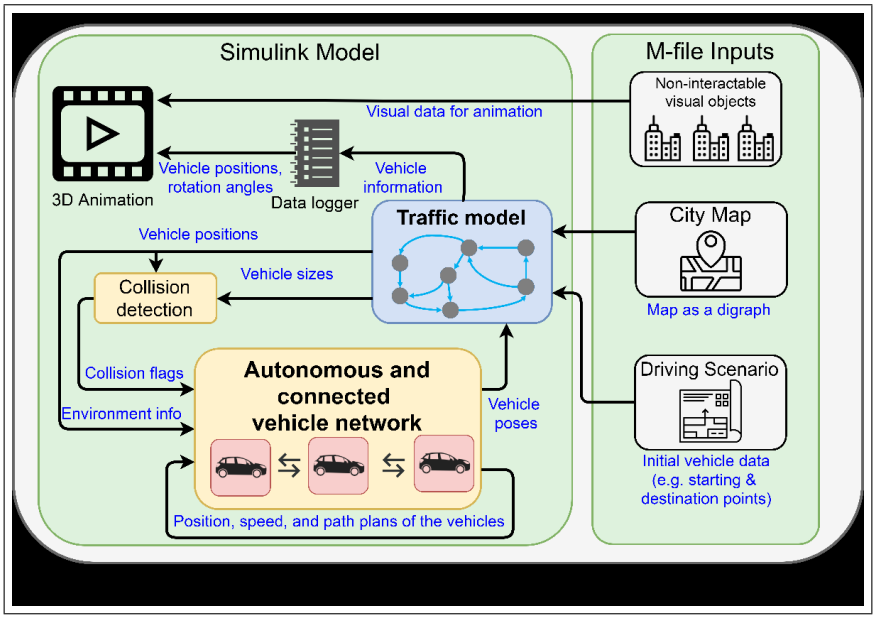

Figure 4.2: MOBATSim Workflow Diagram

The initial parameters regarding the map and the driving scenario that will be simulated are stored in MATLAB m-files. These files contain information regarding the structure of the map and the initial parameters regarding the vehicles. Then this data is passed to constructor methods of the relevant classes to create an instance of the map and an array containing the instances of each vehicle to be simulated and the functional structure of the vehicle is contained in the Simulink model as shown in Figure. Once the initial traffic model is set up along with the vehicles, the Simulink model is opened and run to start the simulation. When the simulation starts, the MATLAB System Blocks found in the main Simulink model access the base workspace to use the Map and the Vehicle Objects by reference. The Simulink model consists of both continuous models as well as discrete MATLAB System Blocks.

4.2 The Main Simulink Model

The elements of the top-level Simulink model [2] shown in the Fig 4.3 are:

- **Autonomous Vehicle Models - Masked Subsystem Blocks:** The structure of an autonomous vehicle in MOBATSim consists of different abstraction levels in order to allow the user to easily access and customize the low-level components, their inputs/outputs, and the implemented decision and control algorithms by clear communication between four main interfaces: Perception, Decision-making, Control, and Communication Output as shown in Fig 4.7.

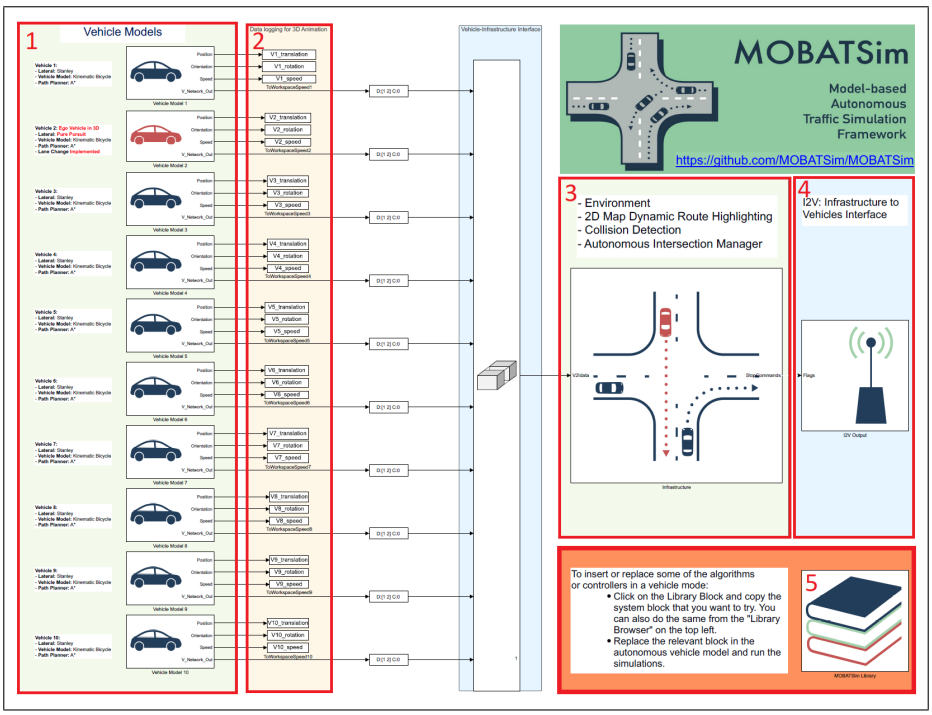

Figure 4.3: Main Simulink Model

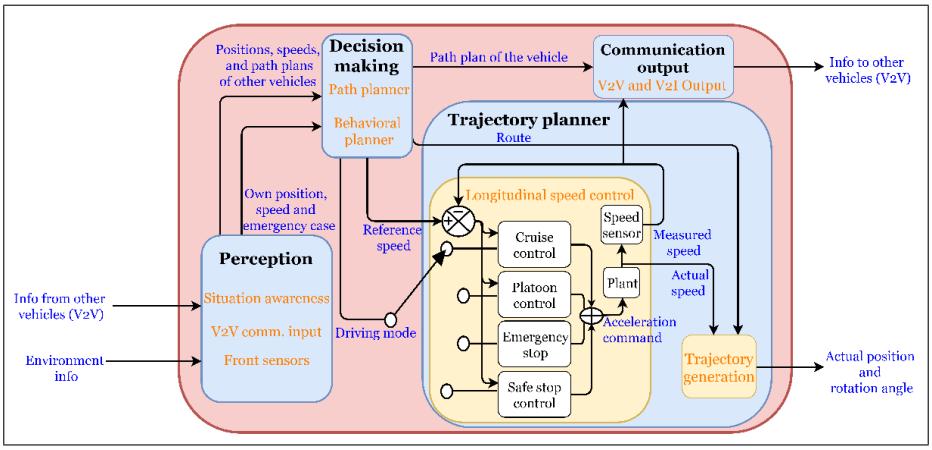

Figure 4.4: Structure of Autonomous Vehicle Model

When you click on the small arrow on the lower-left corner of a Vehicle block, a subsystem vehicle model as shown in the Fig appears.Various procedures can be performed inside the vehicle model and the behaviour of the vehicle during run time is observed.

- **Data Logging for later 3D visualization:** The data is logged using To Workspace blocks and is logged as vehicle position, orientation, and speed data as they are required for further visualization.

- **Infrastructure (Environment) MATLAB System Block:** The whole infrastructure and the environment are implemented as a single MATLAB System Block which takes the Map, Vehicles, and the Intersection Manager objects as arguments. It is divided into Dynamic Map update, Collision detection and Autonomous intersection management.

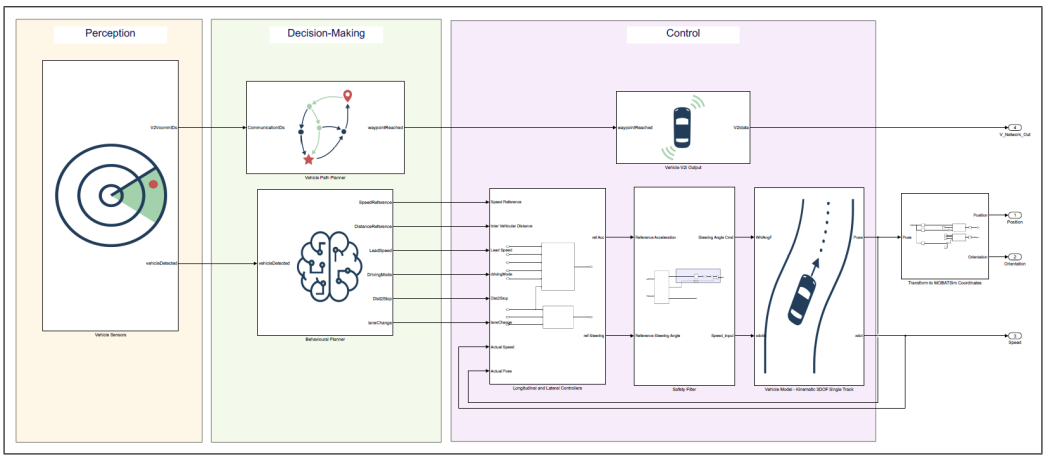

Figure 4.5: Vehicle Model

- **I2V Interface from the Autonomous Intersection Manager to Vehicles:** The array of pass/stop signals coming from the intersection manager is then taken by the I2V block as an input. The only task of this block is to assign the commands that come from the Infrastructure block to relevant Vehicles.

- **MOBATSim Simulink Block Library:** By clicking on the Simulink Library Browser from the top pane of the main Simulink model, the MOBATSim Library can be seen and would be most helpful in testing out new algorithms or controllers that serve a similar purpose. For example, a vehicle's Lateral Controller, Path Planner, or Platoon Controller can be changed with another type of block available in the library.

4.3 Simulation Process

- **Initialization:** The script called **prepare simulator.m** contains all the necessary variables for the workspace that are simulation parameters used by the model. When the script is run, it generates the map and vehicle objects as well as the initial 2D plot.

- **Driving Scenario and the Vehicles:** The next step is to load the driving scenario and initialize the Vehicles. There are many driving scenarios available in this model as shown in the Figure . In this thesis, we consider only three of them i.e. Road Merge Collision, Platoon Control and Urban City Traffic in order to observe faulty behaviour of vehicles in different scenarios. After setting all the necessary initial values and instances, the Simulink model is ready to start the simulation. Right after this, the Simulink model is opened, a 2D map plot according to the selected driving scenario will appear in a new window as shown in Figure 4.6

- **The Traffic Model and 2D Visualization:** The traffic model of MOBATSim used to be an m-file that contained checkpoints and the routes between these points. The structure was converted into a digraph consisting of nodes and edges. The nodes are the points that are then used by the vehicles as starting and ending points. The edges are the routes between these nodes and are defined either as straight or curved roads.

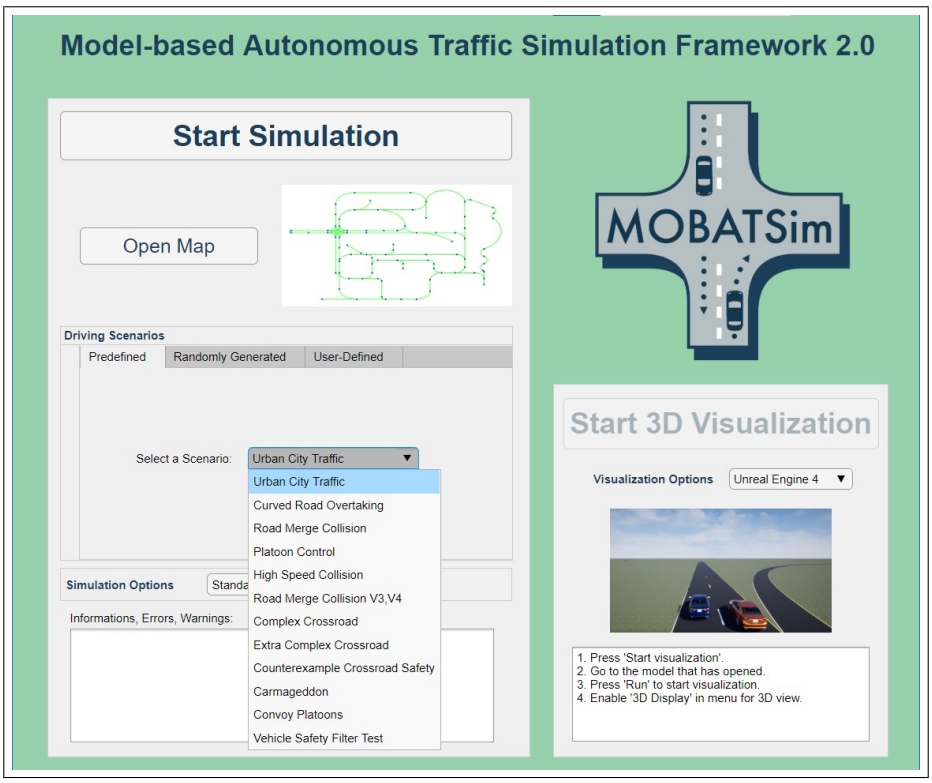

Figure 4.6: Simulation Window

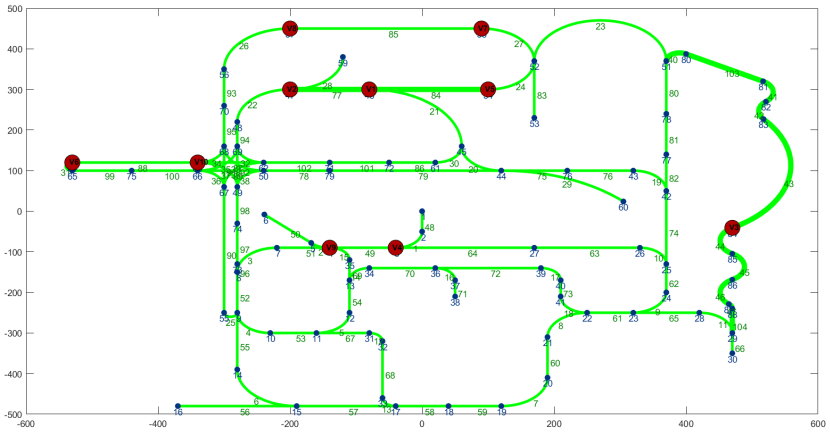

Figure 4.7: 2D Plot Road Map

The Path Planner component of the vehicle uses its path finding algorithms to derive a path, which is a set of nodes and edges to reach the destination node. After determining the Path, the nodes in the Path array are used by the Local Trajectory Planner component of a vehicle to determine their desired trajectory and to track their planned route. The Mobatkent map plotted as a digraph, shown in the 2D plot figure, displays the traffic simulation during the simulation runtime allowing faster simulation in the first place for the data logging. Once the simulation stops and the data is logged, more sophisticated 3D visualizations such as Bird's Eye View or Unreal Engine 4 support can be used.

4.4 Data Features

The data for each vehicle is consists of three features :

- *Speed:* Generated as a time-series univariate data
- *Rotation:* Generated as a time-series multivariate data with 4 columns.
- *Translation:* Generated as a time-series multivariate data with 3 columns.

The final multivariate data set collected format consists of 4 columns that vary according to time named Speed, Rotation, Position, and Translation as plotted in the Figure 4.8. The final uni-variate data set consists of just Speed and corresponding time shown in Figure 4.9 plotted along with driving mode. Further details about data collection is mentioned in Chapter 6

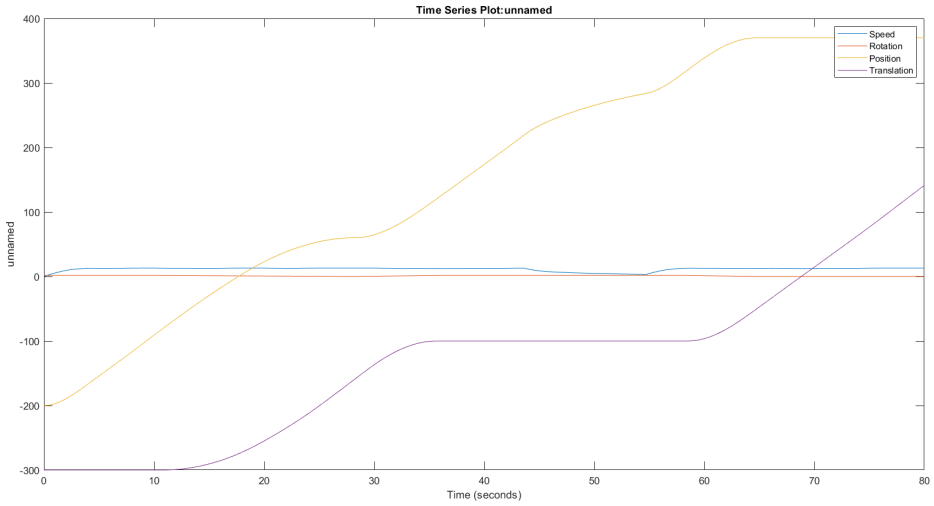

Figure 4.8: Multivariate Data Features

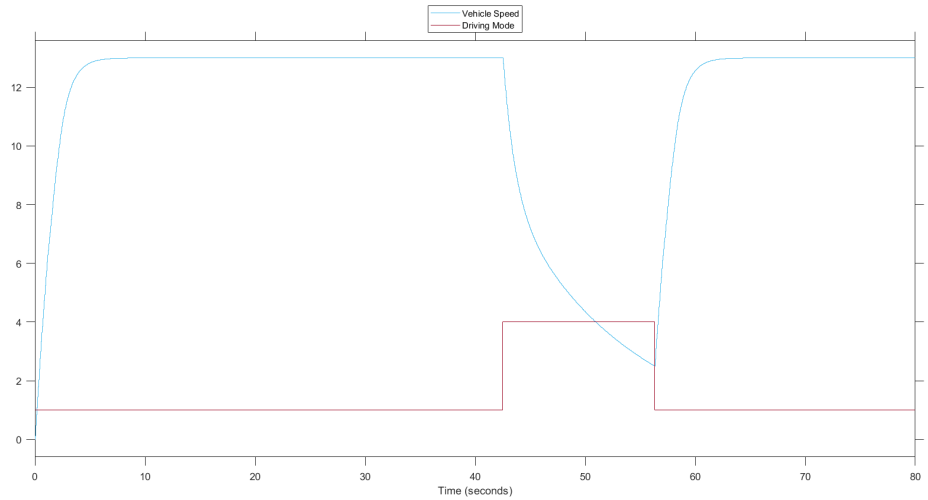

Figure 4.9: Uni-variate Data with driving mode

Chapter 5

Thesis Workflow

The goals of this thesis are listed below:

- Getting familiar with the Simulink model of MOBATSim
- Building scenarios of more than one faulty vehicles with fault injection in different driving scenarios in order to also observe that a faulty vehicle that is overtaking with fault injection.
- Collection of Faulty and Fault-Free Data with different types of sensor faults.
- Train different deep learning models for univariate and multivariate anomaly detection and protocol the evaluation metrics.
- Comparison between different DL models with obtained results through F1 score.

To achieve the above mentioned tasks, the thesis workflow is shown in Figure 5.1 and these parts are briefed below:

1. The Simulink Traffic model of MOBATSim is adopted with three different driving scenarios. In order to contribute to previous work [3] on fault injection, in this thesis we inject fault in more than one vehicle.
2. After preparing the model, an automated data collection process is implemented to collect uni-variate and multivariate fault-free data for different driving scenarios. Three different types of faults are introduced to the model's speed and distance sensor units by using fault injection technique and all the simulation data (uni-variate and multivariate) is stored.
3. Once the data is collected, both fault-free and faulty data are analyzed, preprocessed and trained with different Deep Learning (Simple RNN, GRU-Simple RNN, Autoencoders, Transformer based classifier, BiGRU-BiSimpleRNN and LSTM-AE) algorithms.In this part, Python is used as programming language and DL trainings are performed in Google Colaboratory and Jupyter Notebook.
4. Finally, a comparison between all the DL models is shown in a tabular form based on F1 score for both univariate and multivariate data.

In the subsequent chapters, all these parts will be discussed in detail.

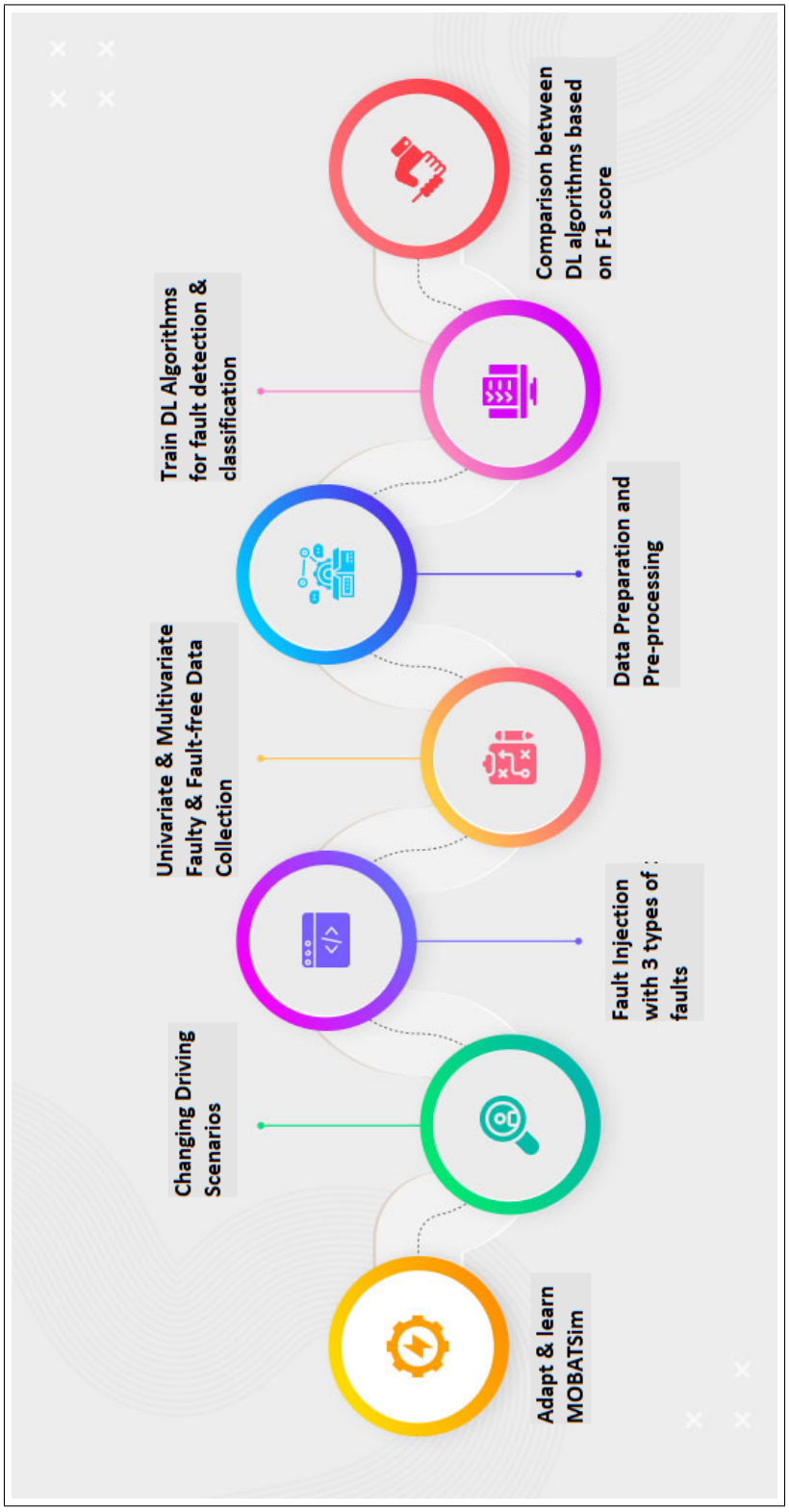

Figure 5.1: Thesis Workflow

Chapter 6

Data Generation

The previous chapters mention all types of theoretical background knowledge about the model, related work and its fault types. In this chapter, a detailed process for data generated from the MOBATSim model is shown. The topics discussed here are the types of data that have been collected and how the data has been generated for faulty and fault-free scenarios.

6.1 Types of Data

In this thesis, two types of data sets have been generated i.e. Uni-variate and Multivariate. As discussed in the previous chapters, since speed is one the major feature that is affected due to fault injection, the uni-variate data set shown in Table 6.1 only consists of the speed feature collected by injecting 3 different types of fault at random time intervals. Another type is the multivariate data set, shown in Table 6.2, that consists of three more attributes namely Rotation, Position and Translation. So this data set contains in total 4 features including speed collected by injecting faults at random time intervals.

6.2 Data Generation for Fault-free Case

As discussed in the previous chapters about the three driving scenarios being used, fault-free data is generated for all three scenarios named 'Urban City Traffic', 'Platoon Control', and 'Road Merge Collision'. The initial data set generated to the workspace is collected in three categories named V1 _ speed (1 data column), V1 _ rotation (4 data columns), and V1 _ translation (3 data columns) where V1 is the denoted format to represent vehicle number. Each generated data set for a particular vehicle now consists of 8 data columns out of which 4 are always a constant number as the input of those 4 ports are constants. Therefore, the final multivariate data set collected format consists of 4 columns that vary based on time named Speed,

Time	**Speed**
0 to 80 secs	0 to Max Speed
Time Interval 0.02 secs	Max speed defined in simulation

Table 6.1: Uni-variate Data

Time	Speed	Rotation	Position	Translation
0 to 80 secs	0 to Max Speed	-3 to 3	-400 to 400	-500 to 300
Time Interval 0.02 secs	Max speed defined in simulation	values defined in simulation	values defined in simulation	values defined in simulation

Table 6.2: Multivariate Data

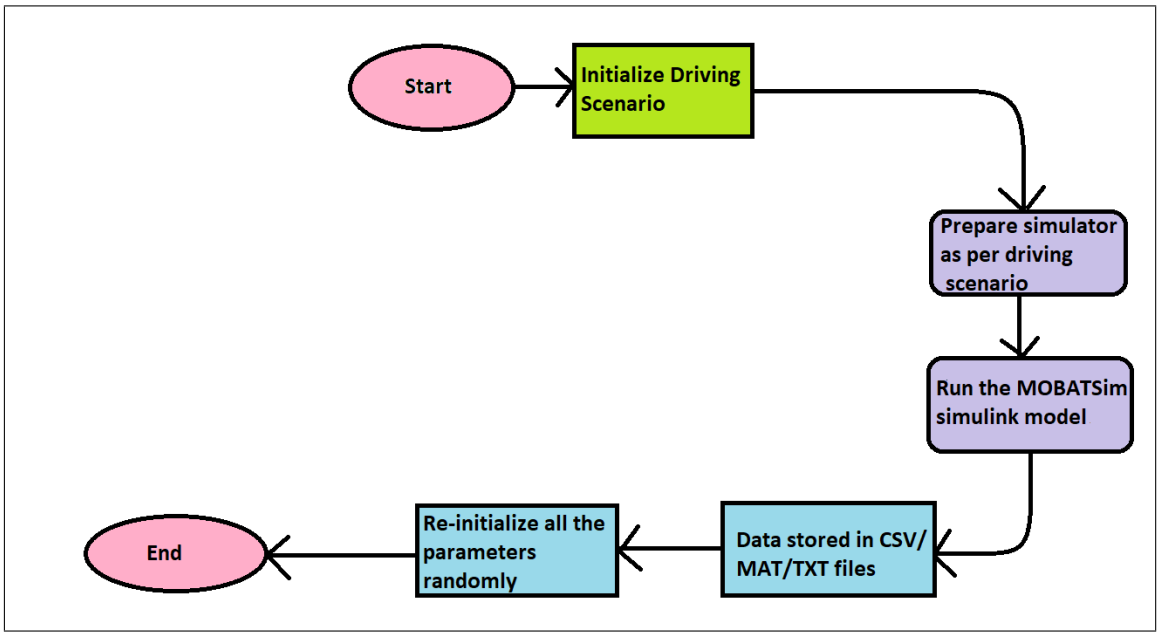

Figure 6.1: Fault Free Data Collection Process

Rotation, Position, and Translation, in a time-series format as shown in Table 6.2.

The complete fault free data collection process is shown in Figure 6.1. For each scenario, 30 simulation files in time-series format (3 as specified above for each vehicle) are generated, thus resulting in a total of 90 simulation fault-free data files for 10 vehicles. These 30 simulation files for each scenario are then concatenated, reformed, and confined into 10 simulation files (1 for each vehicle as specified above) with mentioned features in time-series format. Therefore, eventually, 30 fault-free multivariate data sets are formed. The duration of the simulation is 80 seconds, and the data is re-sampled from 0.005 seconds to 0.02 seconds for data compression. Therefore, each data set generated for one simulation of 80 secs has 4000 data points (entries/rows). These data sets are then stored in MAT/TXT file format and eventually evaluated and reformed and finally stored in CSV formats with the features reflected as column names.

The same scenario is repeated for generation of uni-variate data set as shown in Table 6.1, thus generating 30 fault-free uni-variate data sets, where just the speed column and timestamp are saved as a data set in the MAT/TXT/CSV formats.

6.3 Data Generation for Faulty Case

In this thesis, out of the 10 vehicles in the MOBATSim Model, we consider injecting fault in two vehicles. To demonstrate the fault classification mechanism, sensor level faults are injected at both distance as well speed sensor. A total of 3 fault types are considered. These are:

1. Noise Fault,
2. Stuck-at Fault,
3. Bias/Offset Fault,

Injecting Package drop faults and bit flip faults did not affect the features at all as compared to the fault free case, so they are not further considered here.

In order to make the data collection strategy look more related to real world scenario, we assume that fault could be introduced at any point of time during the simulation and fault could last for any duration of time during the simulation. The fault values are also chosen at random for Bias and Noise faults but from the given values in the Table 6.3.

Simulink's FI Block [4] is used for injecting these faults and introduced at the speed and distance sensors in the MOBATSim Model. Figure 6.2 and Figure 6.3 shows the integration of fault injection sub-module with both the vehicle models at the targeted variables for speed and distance sensor. The FI block's constant flag generator is replaced with non-uniform pulse generators, assuming that fault could be introduced at any point of time. Hence, during the simulation, these non-uniform pulses can be randomly chosen and the fault is then injected based on the pulse value. These pulse generators are shown in Figure 6.4 for Vehicle 2 and in Figure 6.5 for Vehicle 6. The initial time delay and duration of fault occurrence are set randomly, from a predefined range of values in the table. All the chosen values are listed in Table 6.3

Fault Category	**Fault Value**	**Fault Delay**	**Fault Duration**
Noise	20%, 30%, 40%, 50%	1, 2, 3, 4, 5	1, 2, 3, 4, 5
Stuck-at	Stuck to last value	1, 3, 5, 7, 9, 11, 13, 15	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Bias/Offset	-1, -2, -3	1, 2, 3, 4, 5	2, 3, 4, 5, 6, 7

Table 6.3: Values, delay and duration of different faults

All these faults, i.e. noise, stuck-at, and bias/offset fault are randomly with a sequential fault value are injected in Vehicle 2 and Vehicle 6 for both speed and distance sensor at targeted variables called Speed Reference and Inter-vehicular Distance, as shown in Figure 6.3, during the MOBATSim simulation. The faulty data collection process is simulated for all 3 scenarios as mentioned in the start. Many combinations of different fault values, delays, and fault duration, and for each simulation, these values are randomly chosen as per table 6.3. The non-uniform pulse is generated differently for both vehicle 2 and vehicle 6 as shown in Figure 6.4 and Figure 6.5.

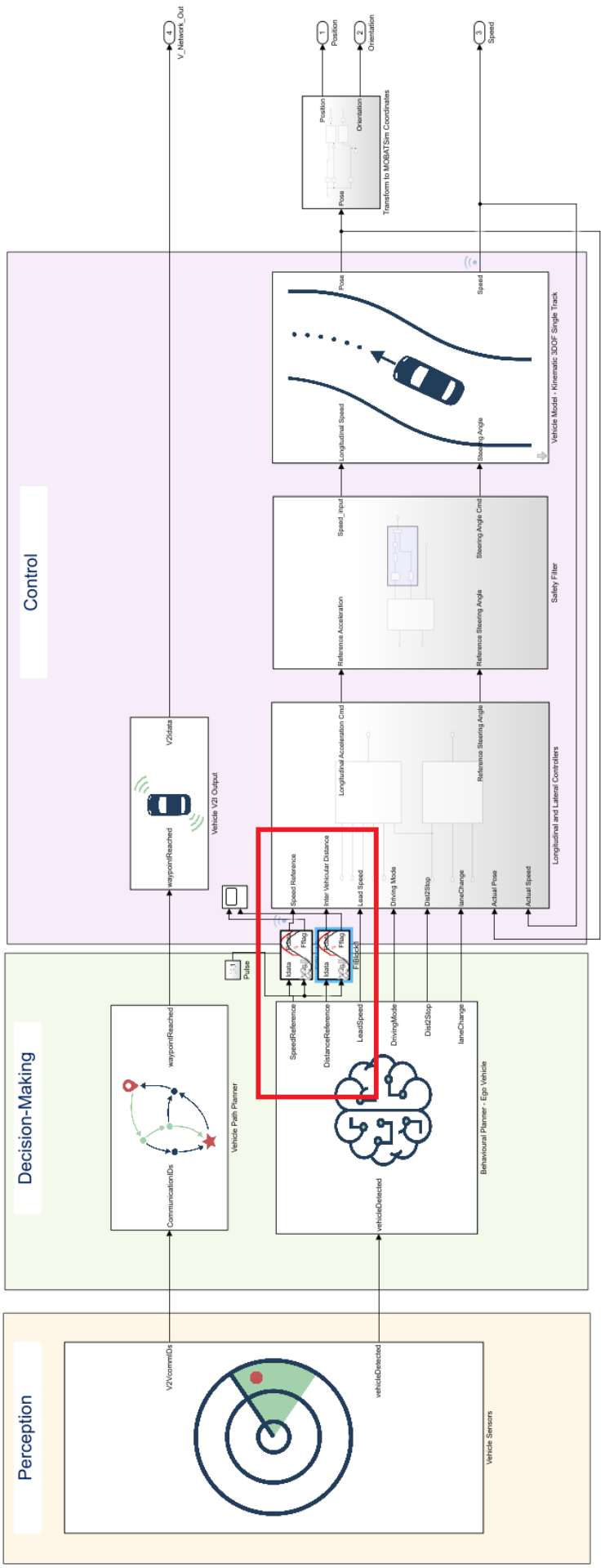

Figure 6.2: Integration of Fault Injection to MOBATSim model

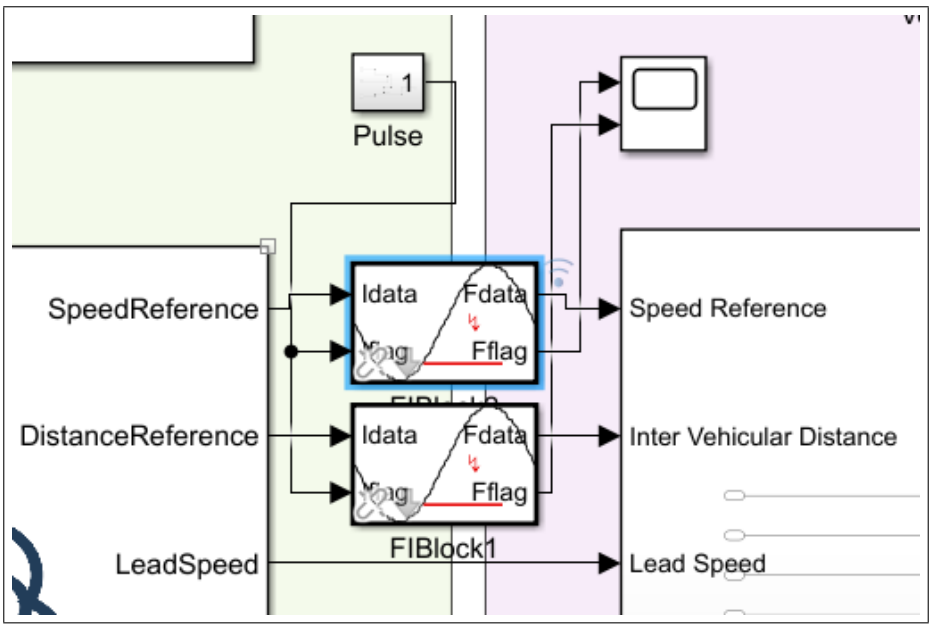

Figure 6.3: Fault Injection Architecture

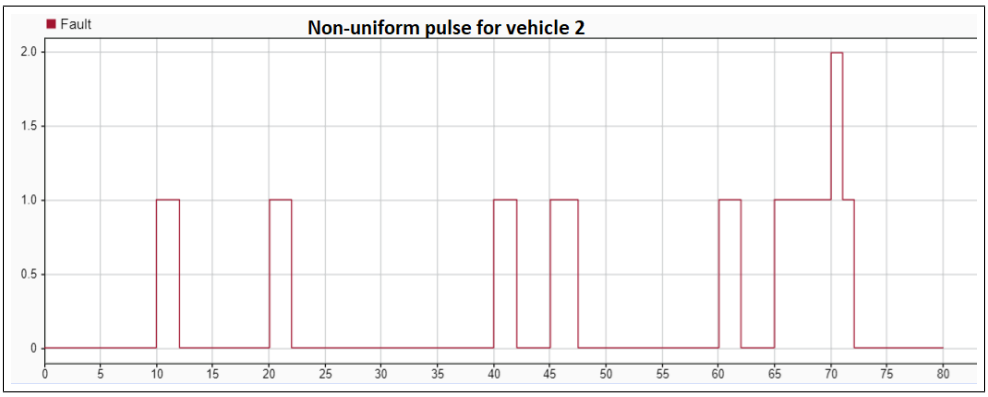

Figure 6.4: Non-uniform Pulse for vehicle 2

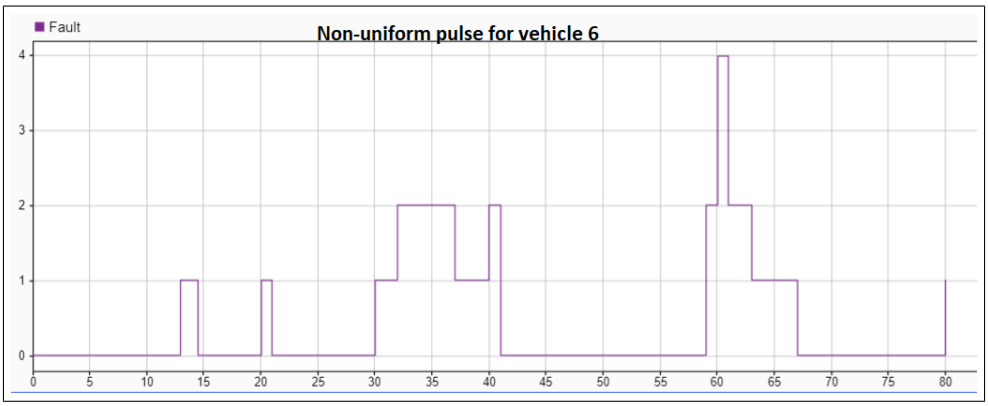

Figure 6.5: Non-uniform Pulse for vehicle 6

For each value of fault duration, data is collected for every scenario considering each fault consecutively. For example, at respective fault values, data is collected for random fault delays and fault duration of Noise, Stuck-at and Offset/Bias for both vehicle 2 and vehicle 6 for all three scenarios. Therefore, for each fault duration, 18 files are generated for each uni-variate and multivariate data. Therefore, at the end a total of more than 200 data files are collected.

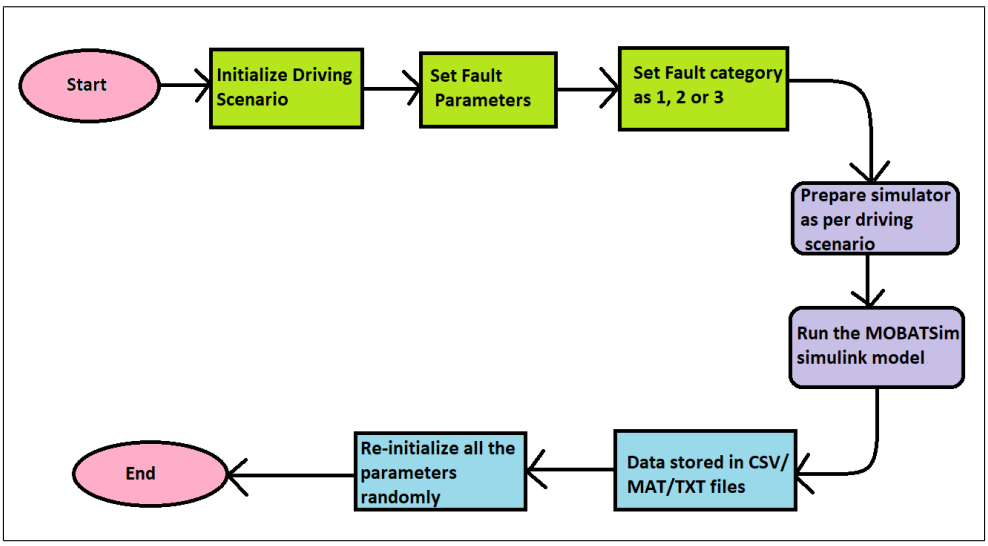

Figure 6.6: Faulty Data Collection Process

The complete process for faulty data generation is shown in Figure 6.6. The duration of each simulation is 80 seconds with 4000 data points each. Similar to the fault-free data collection, the collected data is in both multivariate and uni-variate formats and is then converted to time-series and then re-sampled to 0.02 seconds. Finally, the data is saved as MAT/CSV/TXT files.

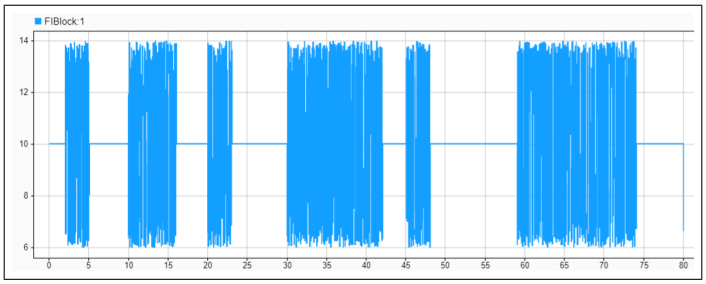

Figure 6.7: Noise Fault : Sampling rate 0.02 sec

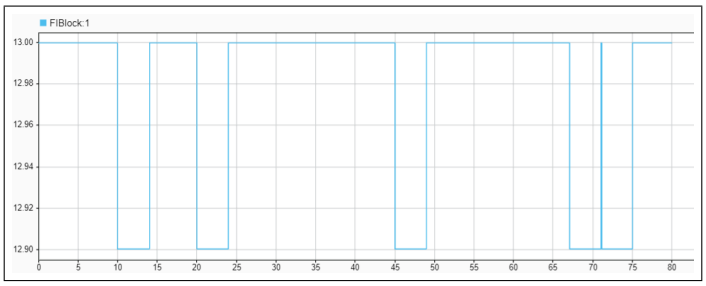

Figure 6.8: Stuck-at Fault : Sampling rate 0.02 sec

Above Figures 6.7, 6.8 and 6.9, representing Noise fault, Stuck-at fault and offset fault respectively , shows just an example of the way, time intervals and intensity of different types of fault injected with a sampling rate 0f 0.02 secs. It is also observed from the figures that noise have far more intense fault injection that offset and stuck-at. Also, as shown in the Table 6.3, negative offset values have been used as the

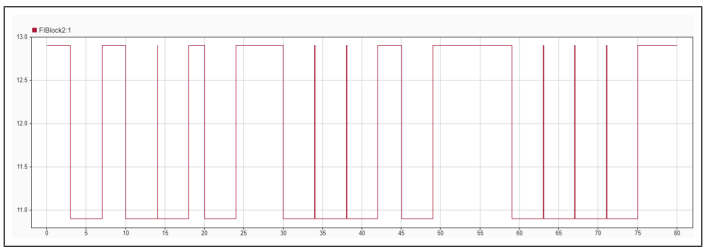

Figure 6.9: Offset Fault : Sampling rate 0.02 sec

vehicles are already travelling at maximum speed assigned to them and therefore do not show any distortions for positive offset values.

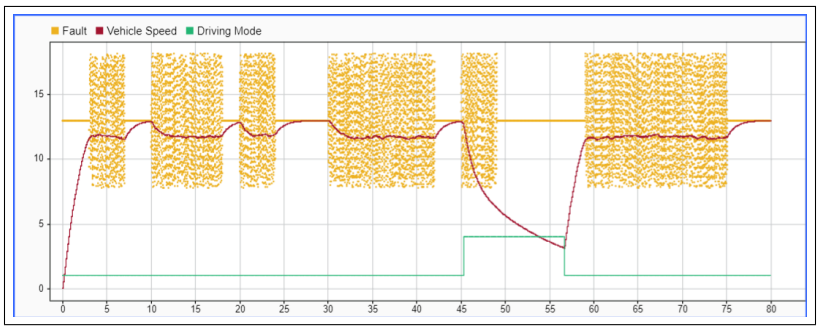

Figure 6.10: Noise Fault Injection on vehicle 2

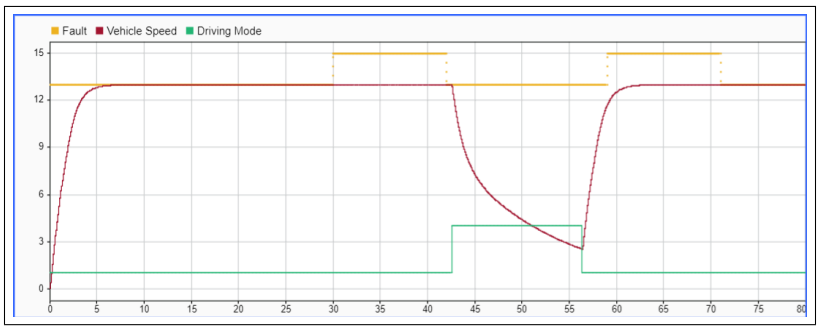

Figure 6.11: Stuck-at Fault Injection on vehicle 2

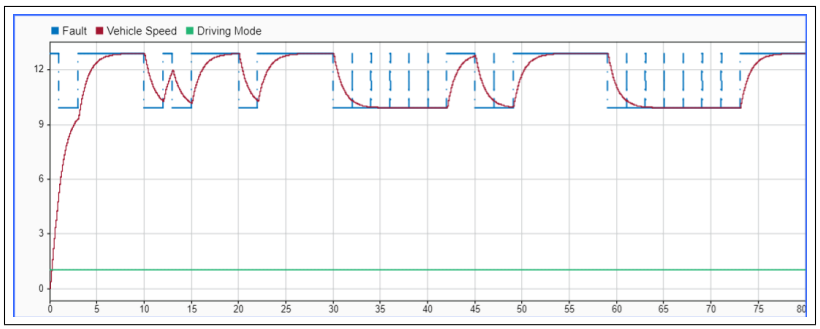

Figure 6.12: Offset Fault Injection on vehicle 2

Above Figures 6.10, 6.11 and 6.12 represents how the speed attribute show variations with respect to time when different categories of fault is injected in vehicle

2. These above figures just represent a sample data for any randomly picked fault value and driving scenario.

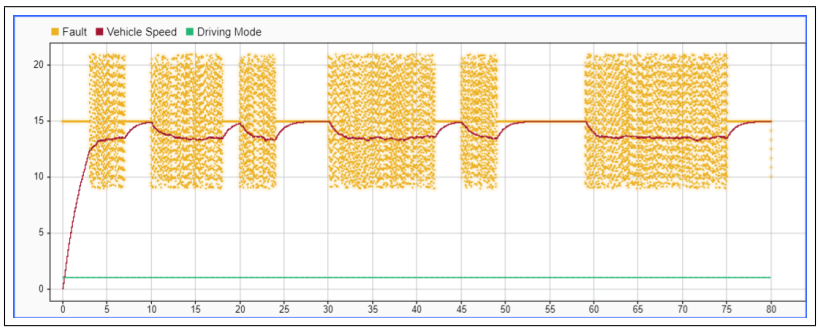

Figure 6.13: Noise Fault Injection on vehicle 6

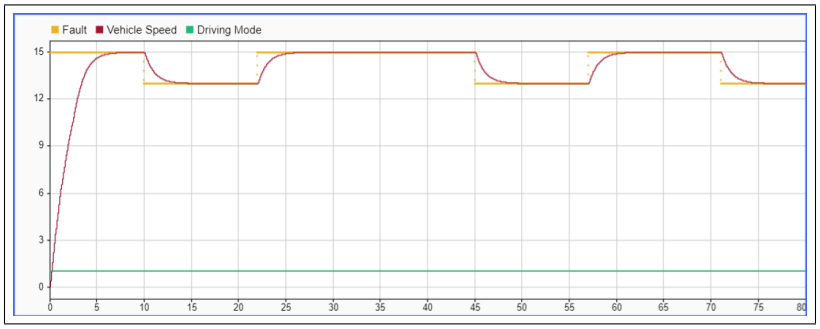

Figure 6.14: Stuck-at Fault Injection on vehicle 6

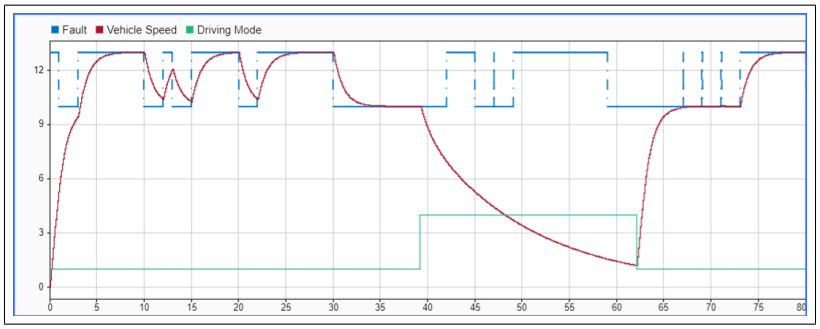

Figure 6.15: Offset Fault Injection on vehicle 6

Above Figures 6.13, 6.14 and 6.15 represents how the speed attribute show variations with respect to time when different categories of fault is injected in vehicle 6. These above figures just represent a sample data for any randomly picked fault value and driving scenario.

It is observed that, in case of noise fault, the speed graph shows proper distortion in case of a fault injection. In case of offset fault, it does not show any changes when a positive fault value is applied whereas when a negative fault value is applies, the speed distortion is directly proportional to the value of offset/bias fault. Lastly, in case of stuck-at fault, vehicle 6 shows distortion with respect to speed when fault is injected whereas vehicle 2 shows very minimal, almost negligible, distortions when fault is injected.

Chapter 7

Data Pre-processing

As discussed n the previous chapter, we generated both uni-variate and multivariate time series data and we mainly operate for data preparation, pre-processing and training by using the CSV files. In order to build an efficient Deep Learning model, we need to prepare data in different manners. In this chapter, we discuss a generic manner used for all the Deep Learning models built for this thesis.

7.1 Steps of Data Pre-processing

Data pre-processing refers to manipulation or dropping of data before it is used in order to ensure or enhance performance [66]. The following steps are performed in order to prepare the data:

1. **Assigning column names:** The generated data file according to the previous section have data column names as Data:1, Data:2 and so on as a time series data in Matlab. The columns are then assigned names in the CSV files as Time and Speed for uni-variate data as shown in figure and Time, Speed, Rotation, Position and Translation for multi-variate data as shown in figure.

2. **Ground Truth Labeling:** The data files generated for both uni-variate and multivariate data is now introduced with a new column called Label. It consists of values 0, 1, 2 and 3 based on the type of fault injected. The values 0 denote fault-free scenario, 1 for noise fault, 2 for stuck-at and 3 for offset.

3. **Data Shuffling:** The total amount of entries for both uni-variate and multivariate data is approximately 300K each. This data set is made up by concatenating all the generated simulation files in a sequential manner. Therefore, in order to build a better DL model, we shuffle the data set for our further use.

4. **Train-Test Data Split:** Since, we already shuffled the data, we can directly perform the splitting of data into test and train data with a random state of 42 and the test data will still be unknown to the Deep Learning model. The data is then split randomly with 0.33 part being test data and 0.67 being the train data. The test data is also further split into half for assigning value to validation data.

5. **Categorical Conversion:** In this thesis, since multiple fault categories are considered, we also perform the conversion of class vector(integers) to binary class matrix for the train data set in order to use with categorical cross-entropy.

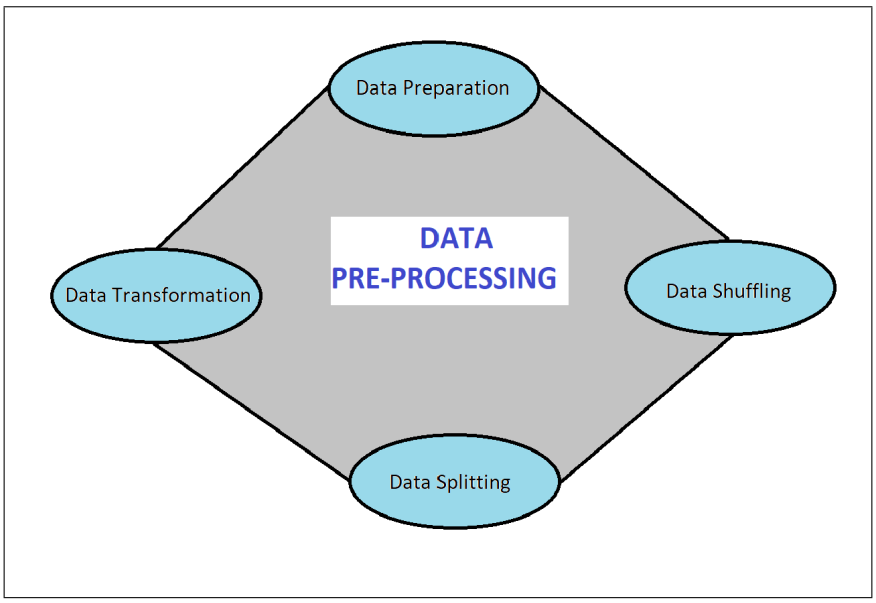

Figure 7.1: Steps for Data Pre-processing

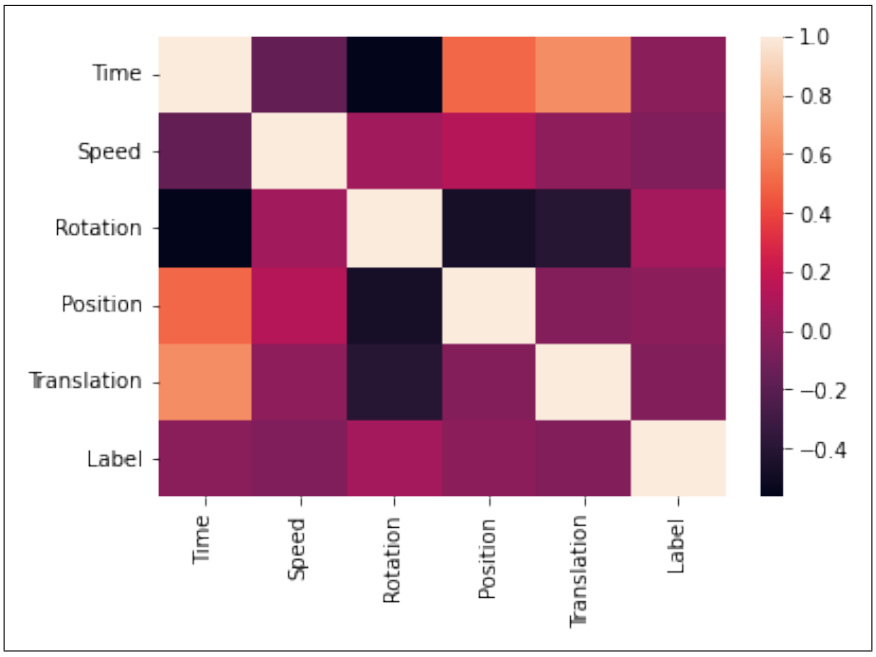

Figure 7.2: Correlation between Features and the Label in Multivariate Dataset

The figure shows correlation between different attributes considered for generation of uni-variate and multivariate data set with the label column.

Other pre-processing procedures like feature scaling could not be performed on the multivariate data set as it would not generate better accuracy, since feature scaling is useful only if it has many features to compare with and our multivariate data set only consists of 4 attributes as well as DL algorithms learn the hidden features through the training process, hence do not require feature scaling as such. Also, the generated data is just enough to train the data with DL model, so down sampling resulted in inefficient training and was eventually only able to detect 0 and 1 labels.

Chapter 8

Fault Classification Training of Deep learning Models

The complete data generation and collection procedures and techniques have been discussed in previous chapters along with the procedure of pre-processing of both uni-variate and multivariate data. In this Chapter, training of different DL models for fault detection and classification are shown. In this thesis, the architectures used for training the models are Simple RNN, Autoencoder based Classifier, Transformer based Classifier, GRU-Simple RNN, Bidirectional GRU-Bidirectional RNN and LSTM-AE. These algorithms are selected based on the previous related work discussed in chapter.

This section focuses on detailed DL model architectures, hyper-parameters used, and presents the training performance of the DL-based candidate models with both uni-variate and multivariate data sets. The models have been generalized and build in such a way that they work without any changes for both types of data.

8.1 Simple RNN

In order to test the performance, initially we use a simple Recurrent Neural Network (RNN) architecture [67] ,shown in Figure 8.1, built by using a Simple RNN layer followed by four dense layers), as shown in Figure. The activation function used in Simple RNN layer and first three dense layers is *Relu*. After RNN layer and dense layers, batch normalization is used in order to make neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. Also pooling function is introduced as pooling provides the ability to learn invariant features and also reduces the problem of overfitting. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both uni-variate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.1 represents all the hyper-parameters used to train the model.

Figure 8.2 shows the performance of the model during training process with univariate data. It is evident that the validation accuracy reached to 60%, and the validation loss dropped from 1.4 to 0.7. Figure 8.3 shows the performance of the model during training process with multivariate data. It is evident that the validation accuracy reached to 66%, and the validation loss dropped from 1.3 to 0.7.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	1024
No. of Epochs	50

Table 8.1: Hyper-parameters values for Simple RNN

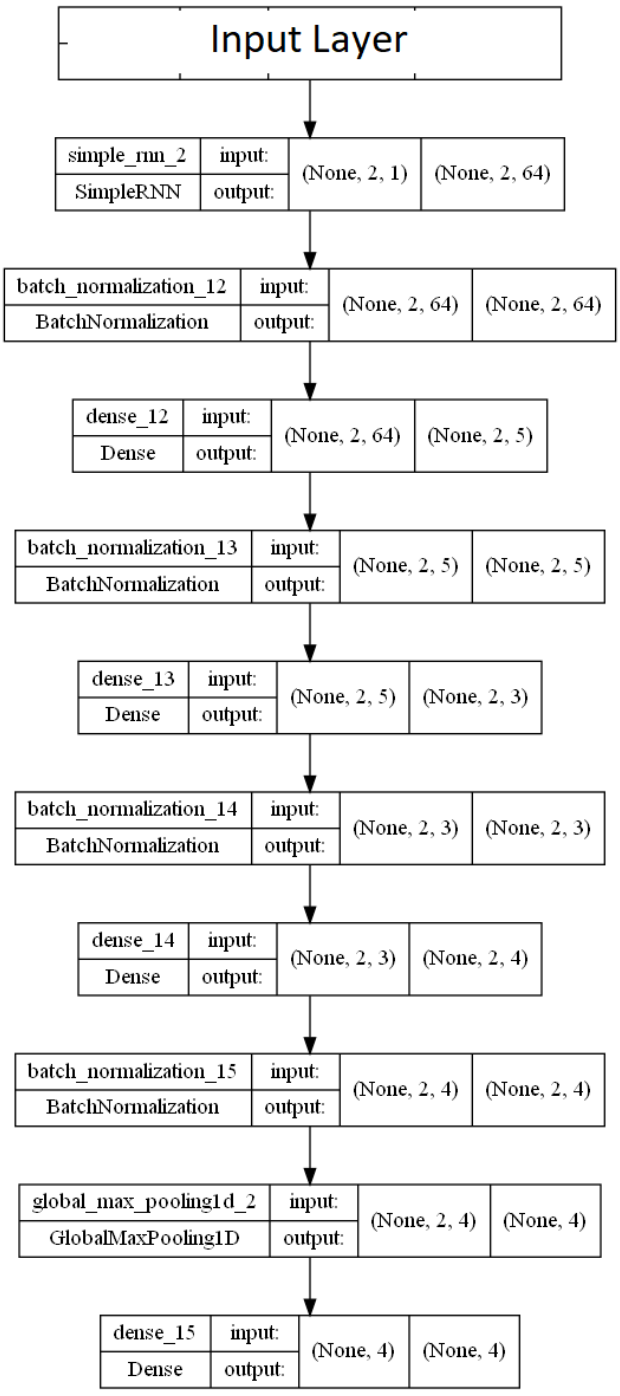

Figure 8.1: Simple RNN Model

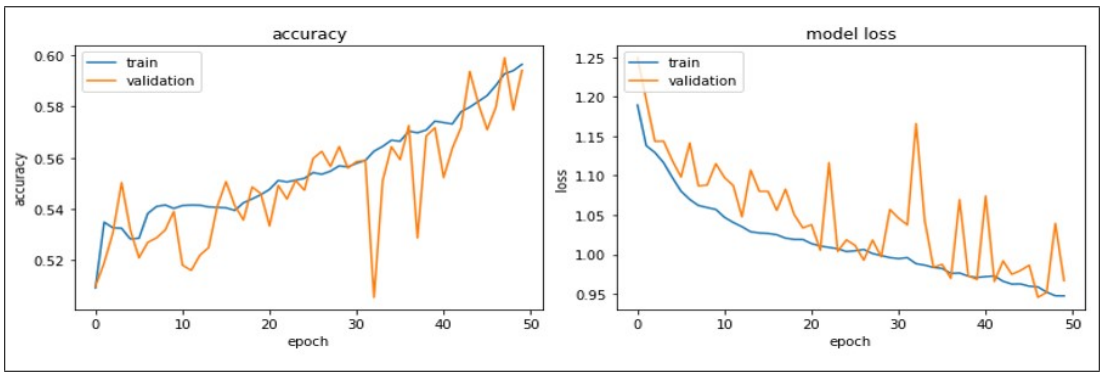

Figure 8.2: Training Accuracy and Loss for Univariate Data for Simple RNN architecture

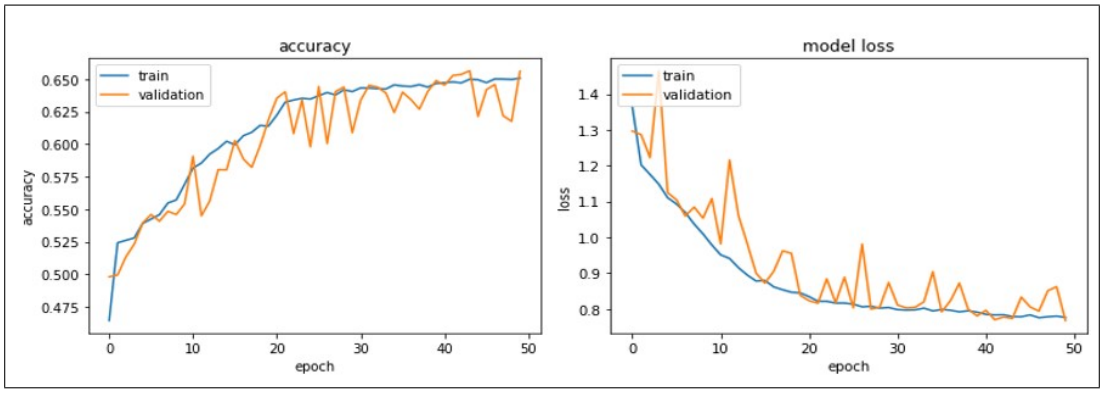

Figure 8.3: Training Accuracy and Loss for Multivariate Data for Simple RNN architecture

8.2 Hybrid GRU-Simple RNN

Figure 8.4 shows the architecture for a hybrid model of GRU-Simple RNN [68]. This architecture is built by introducing GRU layer followed by simple RNN layer with *Relu* activation function. This is followed by 3 dense layers with *Relu* activation function and 4 Batch Normalization functions in order to make neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. Also max pooling function is also used in order to enable the model to learn invariant features and reduce overfitting. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both univariate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.2 represents all the hyper-parameters used to train the model.

Figure 8.5 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 60%, and the validation loss is across 0.9. Figure 8.6 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 76%, and the validation loss dropped from 1.4 to 0.5.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	2048
No. of Epochs	50

Table 8.2: Hyper-parameters values for GRU-Simple RNN

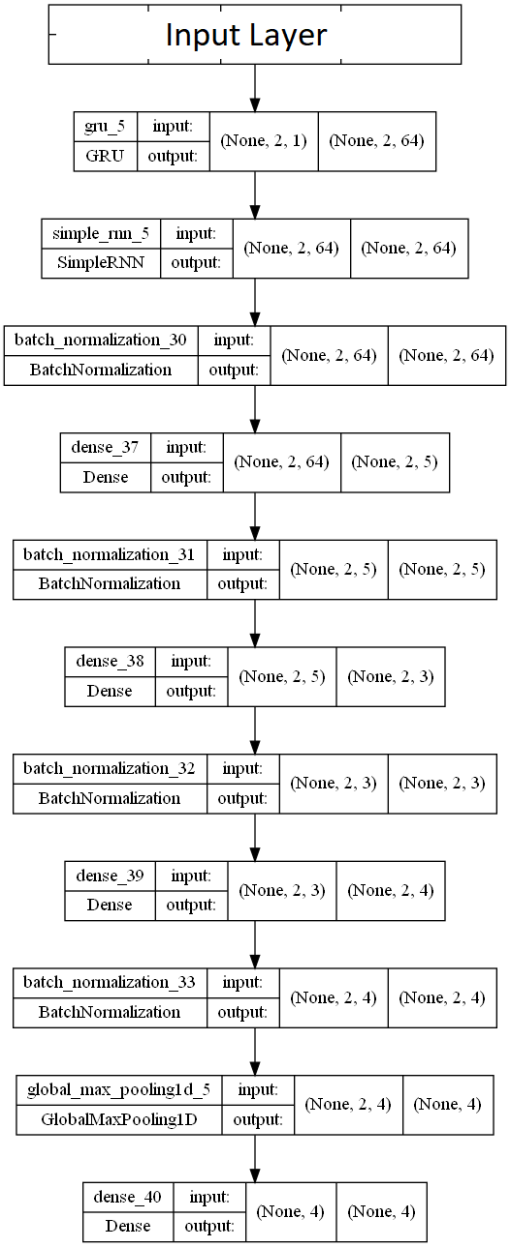

Figure 8.4: Hybrid GRU-Simple RNN Model

8.3 Bidirectional GRU-Bidirectional Simple RNN

Figure 8.7 shows the architecture for a hybrid model of Bidirectional GRU- Bidirectional RNN [69]. This architecture is built by introducing bidirectional GRU layer followed by bidirectional simple RNN layer with *Relu* activation function. This is

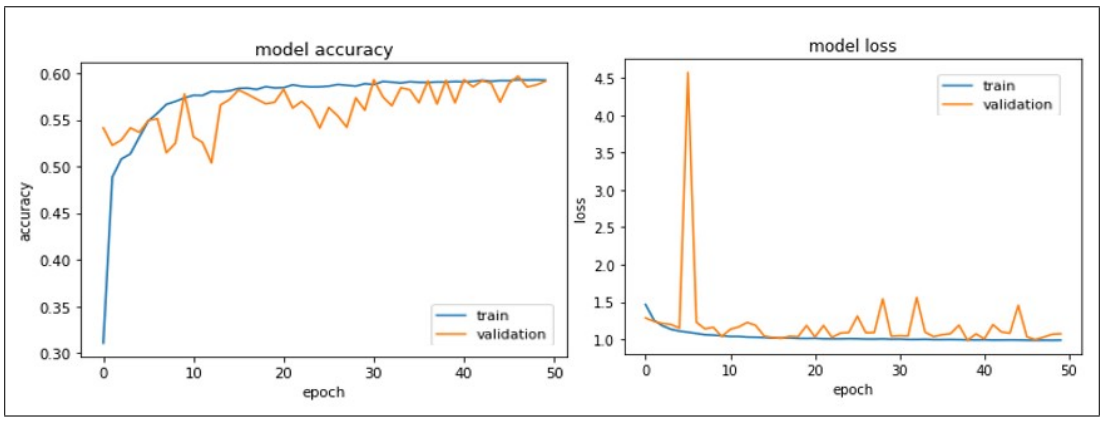

Figure 8.5: Training Accuracy and Loss for Univariate Data for Hybrid GRU-Simple RNN architecture

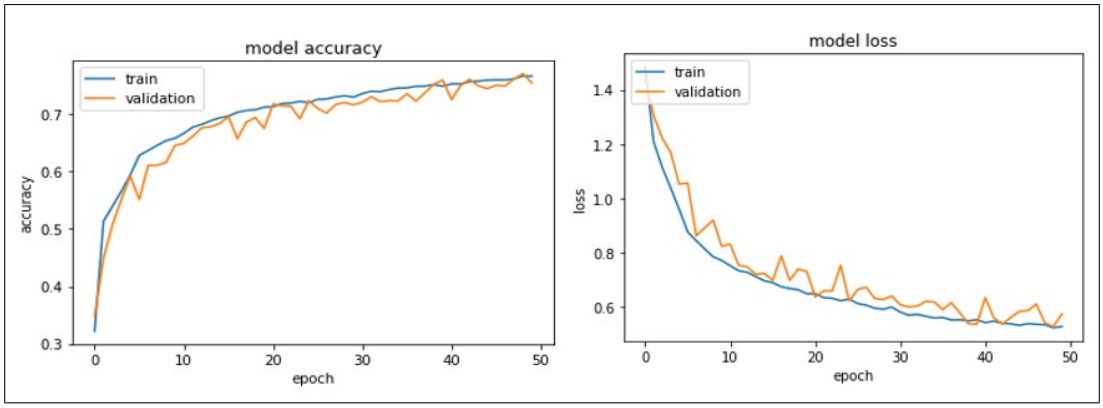

Figure 8.6: Training Accuracy and Loss for Multivariate Data for Hybrid GRU-Simple RNN architecture

followed by 3 dense layers with *Relu* activation function and 4 Batch Normalization functions in order to make neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. Also max pooling function is also used in order to enable the model to learn invariant features and reduce overfitting. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both uni-variate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.3 represents all the hyper-parameters used to train the model.

Figure 8.8 shows the performance of the model during training process. It is evident that the validation accuracy reached to 72%, and the validation loss dropped from 1.4 to 0.7. Figure 8.9 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 81%, and the validation loss dropped from 1.3 to 0.4.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	2048
No. of Epochs	50

Table 8.3: Hyper-parameters values for Bidirectional GRU-Bidirectional Simple RNN

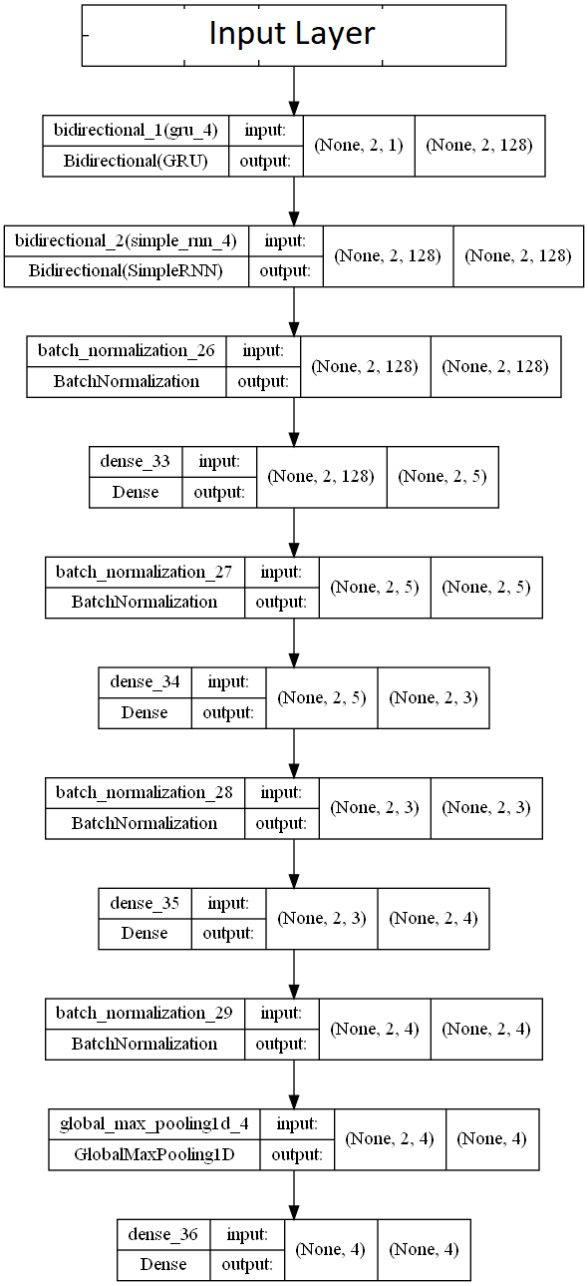

Figure 8.7: Bidirectional GRU-Bidirectional Simple RNN Model

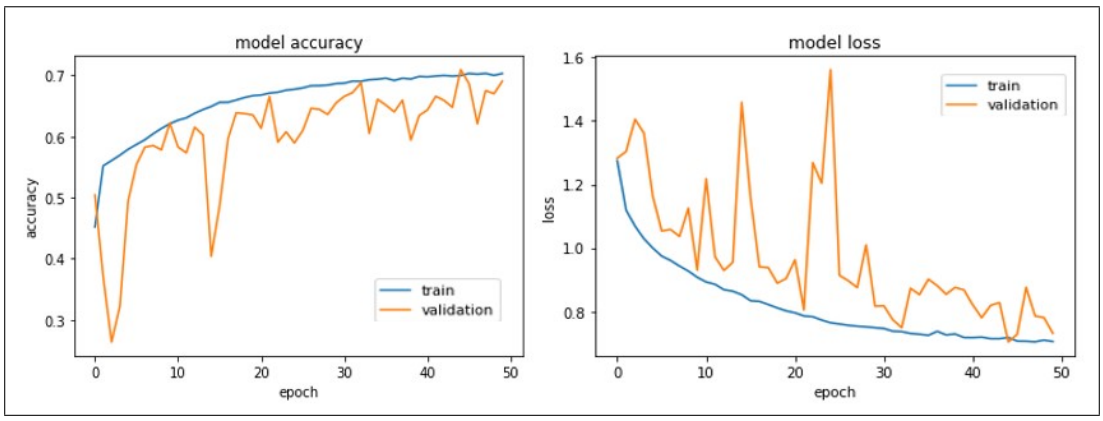

Figure 8.8: Training Accuracy and Loss for Univariate Data for Bidirectional GRU-Bidirectional Simple RNN architecture

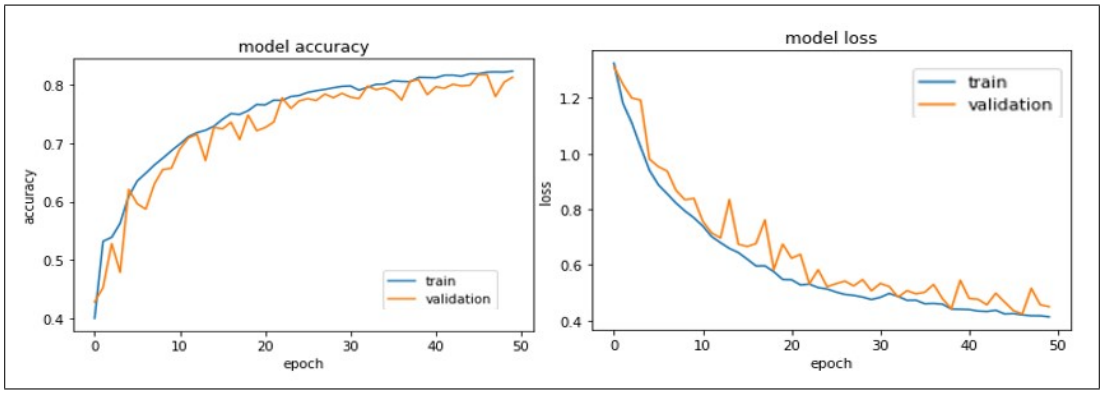

Figure 8.9: Training Accuracy and Loss for Multivariate Data for Bidirectional GRU-Bidirectional Simple RNN architecture

8.4 Autoencoder based classifier

Figure 8.10 shows the architecture for Autoencoder based Classifier [70]. A simple Autoencoder architecture is built by first dividing the model into encoder and decoder section. In the encoder network, we used 4 dense layers with batch normalization in the encoded version whereas in the decoder network, we use them in the decoded version. The activation function used for all these dense layers is *Relu*. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both uni-variate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.4 represents all the hyper-parameters used to train the model.

Figure 8.11 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 62%, and the validation loss dropped from 0.9 to 0.8. Figure 8.12 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 62%, and the validation loss dropped from 1.2 to 0.9.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	512
No. of Epochs	50

Table 8.4: Hyper-parameters values for Autoencoder based Classifier

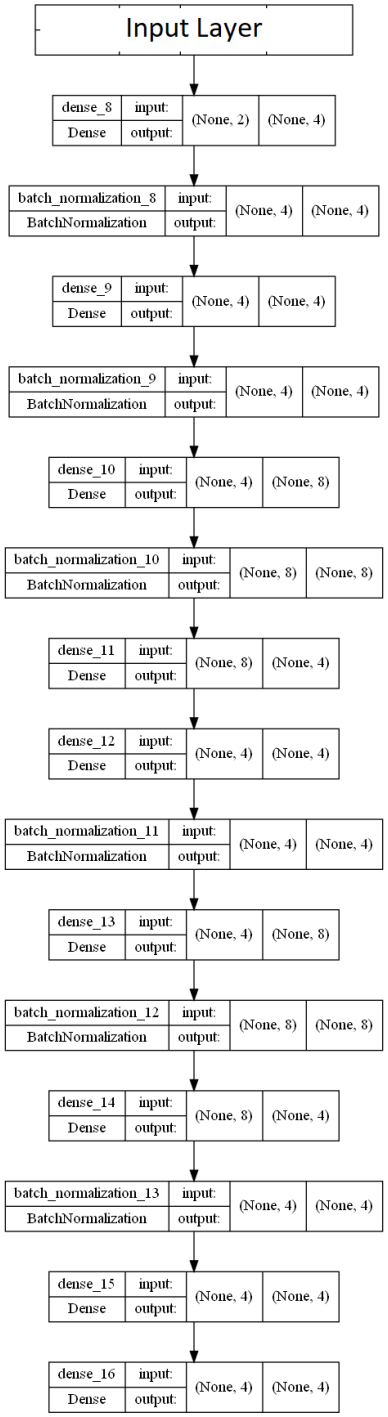

Figure 8.10: Autoencoder based Classifier Model

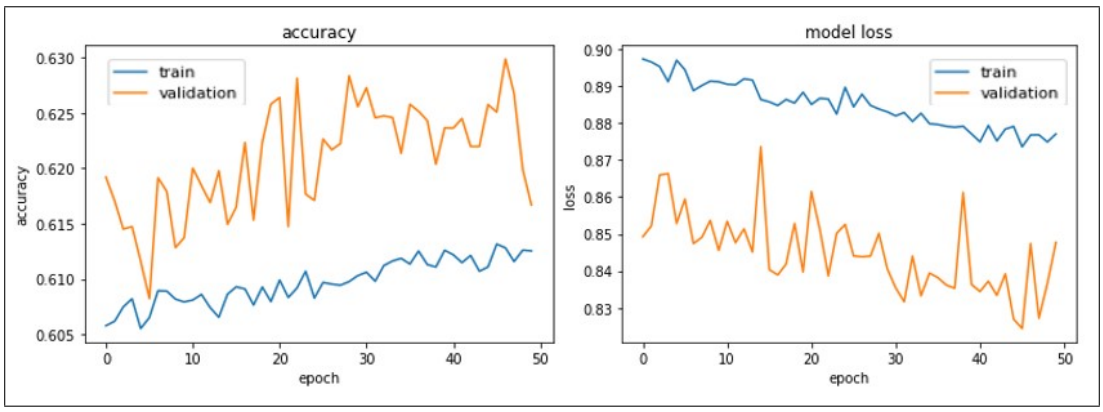

Figure 8.11: Training Accuracy and Loss for Univariate Data for Autoencoder architecture

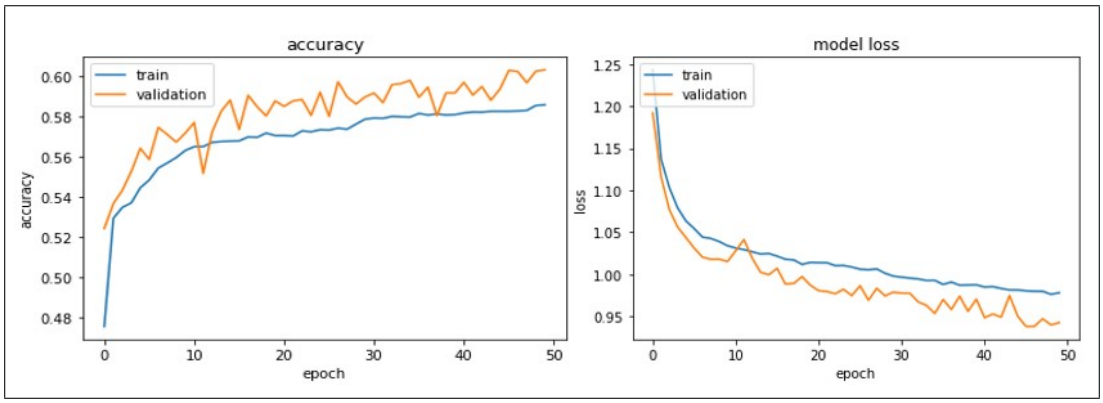

Figure 8.12: Training Accuracy and Loss for Multivariate Data for Autoencoder architecture

8.5 Transformer based classifier

Figure 8.13 shows the architecture for Transformer based Classifier [71]. This classifier is built by first defining two important classes called 'Transformer Block' and 'Token And Position Embedding'. Building the model is started by using a dense layer with *Relu* activation function followed by 'TokenAndPositionEmbedding' Layer and Transformer Block layer with embedding size defined as 40, number of attention heads as 5 and Hidden layer size in feed forward network inside transformer as 6. This is followed by Average pooling layer, dense layer with *Relu* activation function and dropout layers for optimization of the model. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both uni-variate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.5 represents all the hyper-parameters used to train the model.

Figure 8.14 shows the performance of the model during training process. It is evident that the validation accuracy reached to 71%, and the validation loss dropped from 1.3 to 0.7. Figure 8.15 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 57%, and the validation loss dropped from 1.2 to 1.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	2048
No. of Epochs	50
Embedding size	40
Attention heads	5
Hidden Layer size	6

Table 8.5: Hyper-parameters values for Transformer based Classifier

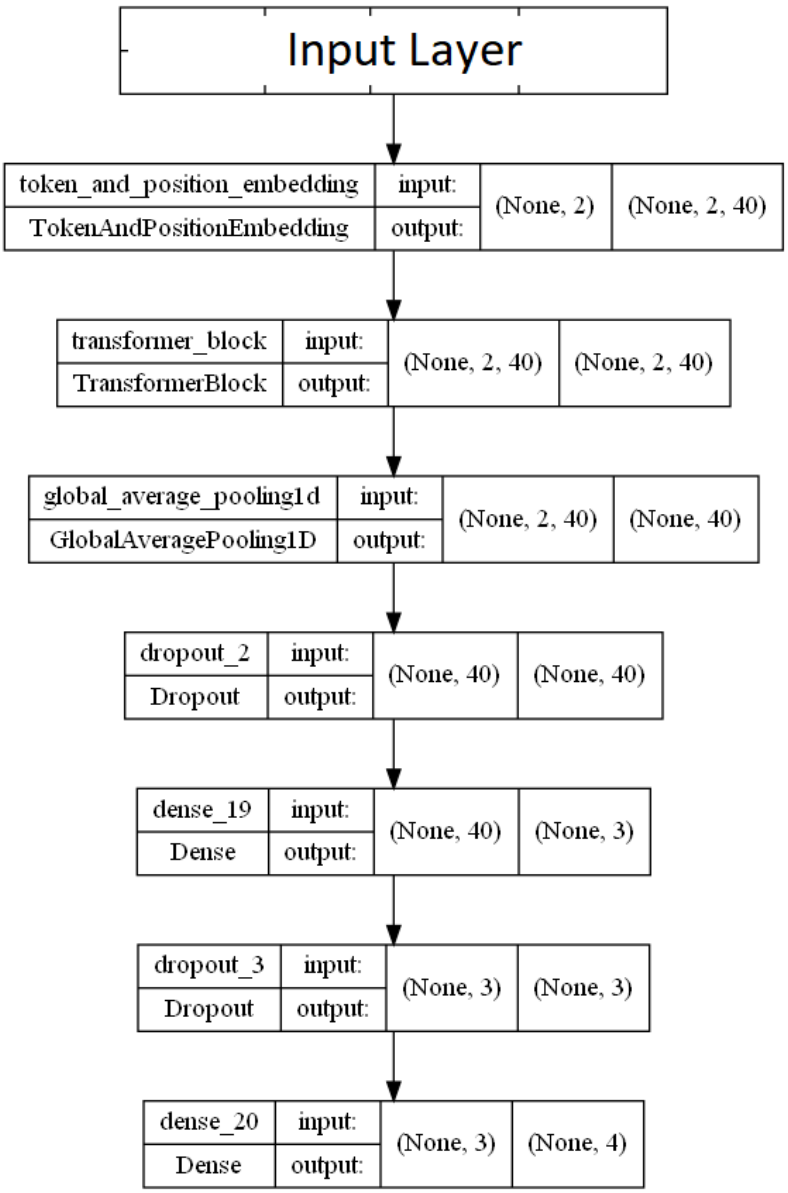

Figure 8.13: Transformer based Classifier Model

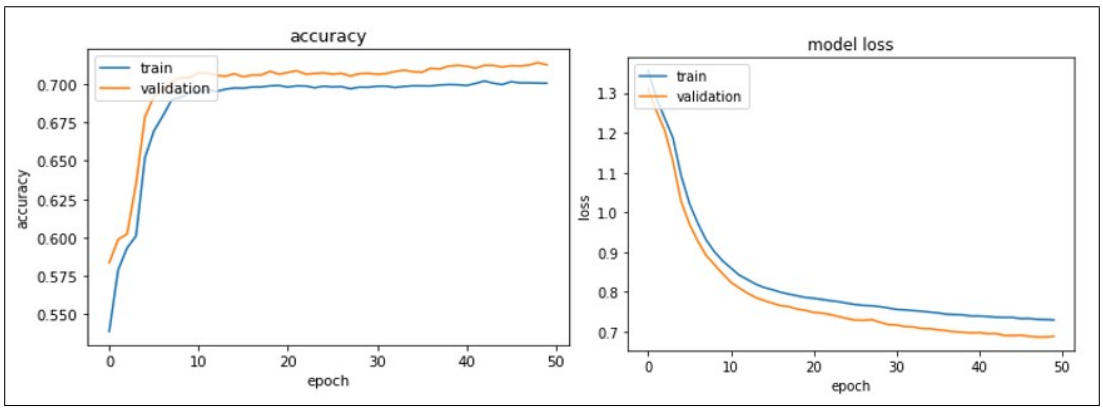

Figure 8.14: Training Accuracy and Loss for Univariate Data for Transformer architecture

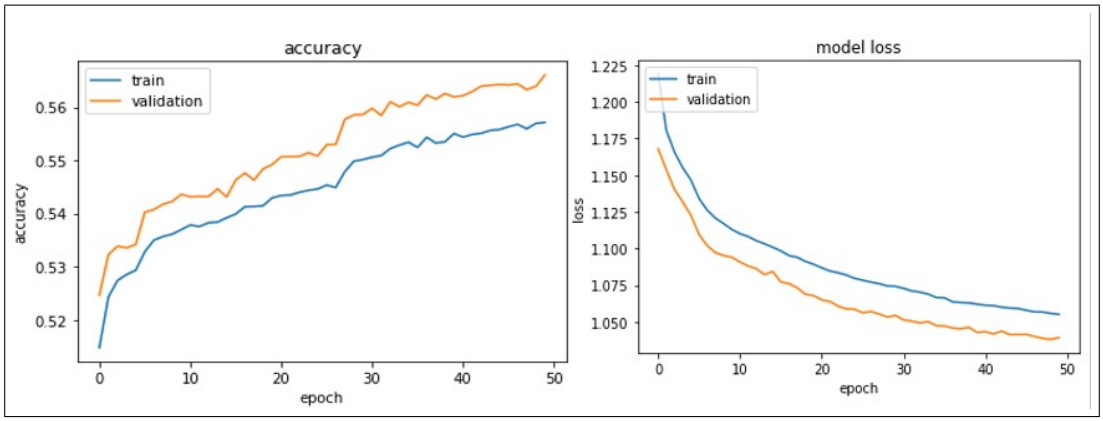

Figure 8.15: Training Accuracy and Loss for Multivariate Data for Transformer architecture

8.6 Hybrid LSTM-Autoencoder

Figure 8.16 shows the architecture for a hybrid model of LSTM - Autoencoder [72]. This architecture is built by introducing LSTM layer followed by a drop out layer and a RepeatVector layer that acts as a bridge between the encoder and decoder modules and prepares the array input for the first LSTM layer in Decoder designed to unfold the encoding. This is followed by another LSTM layer with drop out layer and TimeDistributed layer allowing to apply a layer to every temporal slice of an input. Also average pooling function is also used in order to enable the model to learn invariant features and reduce overfitting. In the last Dense layer, *Softmax* is used as an activation function to enable multi-class classification. Both uni-variate and multivariate data are given as input separately and eventually enables in classifying the data label as as one of the fault classes (fault-free, noise, stuck-at and offset). Table 8.6 represents all the hyper-parameters used to train the model.

Figure 8.17 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 77%, and the validation loss dropped from 1.2 to 0.5. Figure 8.18 shows the performance of the model during training process. It is evident that the validation accuracy reached to only 83%, and the validation loss dropped from 1.2 to 0.4.

Hyper Parameters Used	Value
Optimizer	Adam Optimizer
Learning Rate	0.001
Type	Sequential
Loss	Categorical Crossentropy
Batch size	2048
No. of Epochs	50

Table 8.6: Hyper-parameters values for LSTM-Autoencoder

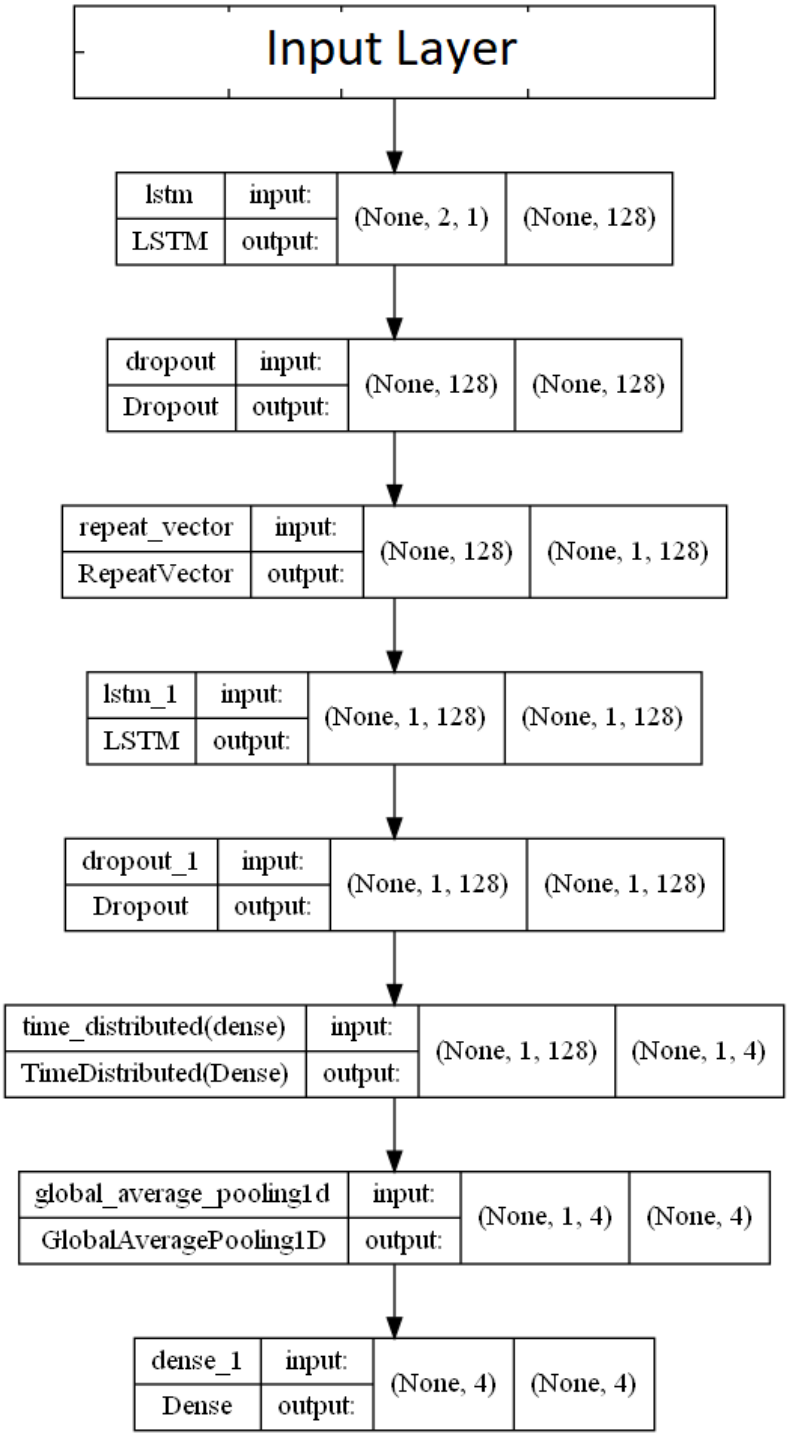

Figure 8.16: LSTM-Autoencoder Model

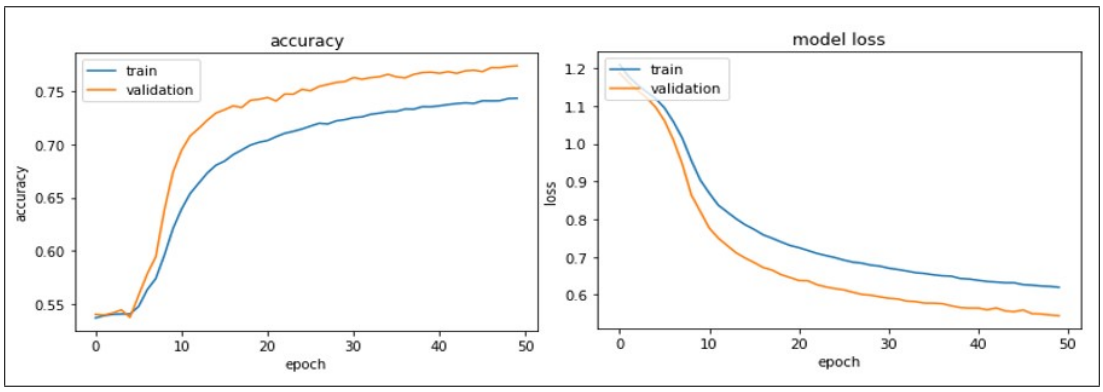

Figure 8.17: Training Accuracy and Loss for Univariate Data for LSTM-Autoencoder architecture

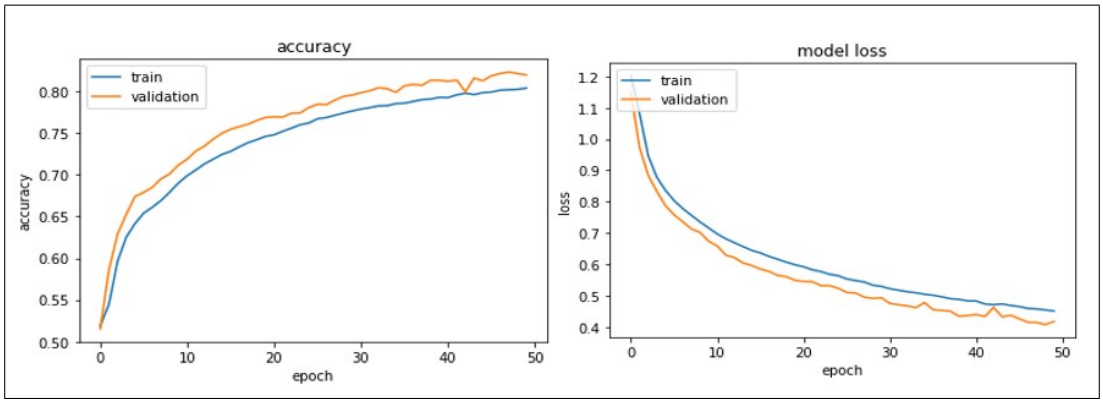

Figure 8.18: Training Accuracy and Loss for Multivariate Data for LSTM-Autoencoder architecture

The results produced by these models for test data in the form of precision, recall, F1 score and accuracy are shown and compared in the next chapter for both types of data.

Chapter 9

Results

In Chapter 8, details, architecture and performances of all the training models used in this thesis are explained. In this chapter, all the results obtained from these models are noted and compared separately for both univariate and multivariate data.

9.1 Deep Learning Model Performance for Univariate Data

Once the training of the data set is performed, all the DL models are tested with the Test Data, which was obtained after pre-processing the data and splitting it as shown in Chapter 7.

Table 9.1 shows the performance of diff DL models like Simple RNN, GRU-Simple RNN, Bidirectional GRU-Bidirectional Simple RNN, Autoencoders, Transformer based classifier and LSTM-Autoencoders on the test data.

Architecture	**Precision**	**Recall**	**F1 score**	**Accuracy**
Simple RNN	57%	60%	51%	60%
GRU-Simple RNN	48%	59%	49%	59%
BiGRU-BiSimple RNN	67%	69%	65%	69%
Autoencoder	57%	61%	55%	61%
Transformer	68%	72%	68%	72%
LSTM-Autoencoder	**76%**	**77%**	**75%**	**77%**

Table 9.1: Performance of Deep Learning Models based on Univariate Test Dataset

It is evident that using Bi-directional hybrid structure for GRU-Simple RNN provides better results than the basic GRU-Simple RNN structure. Transformers behave rather fairly compared to other models and are able to acheive 72% accuracy. Also, it is quite clear that Autoencoders, when used purely, do not show good results. However, when used a Hybrid LSTM-Autoencoder architecture,it shows a drastic improvement and provides the best result among the used DL models with **77%** accuracy, and **75%** F1 score.

To showcase the class-wise performance of the Hybrid LSTM-Autoencoder model, the classification report is shown in Table 9.2. The model is able to identify fault-

Class Name	Precision	Recall	F1 Score	Support
Fault Free	84%	94%	88%	25573
Noise Fault	63%	76%	69%	8366
Stuck-at Fault	75%	62%	68%	6508
Offset Fault	66%	33%	44%	7062

Table 9.2: Classification Report of LSTM-Autoencoder model based on Univariate Test Dataset

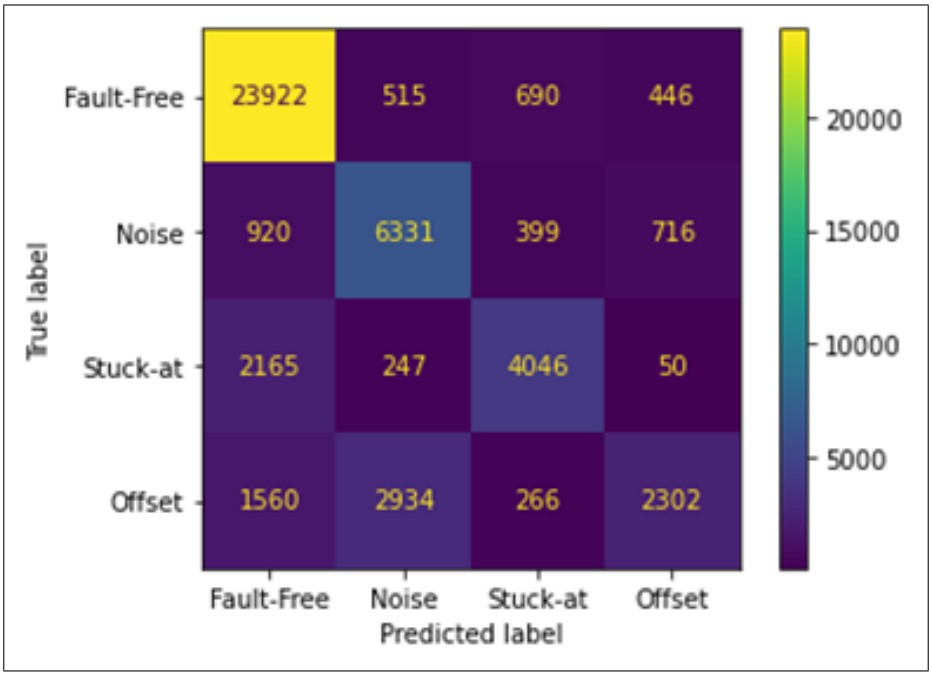

Figure 9.1: Confusion Matrix of LSTM-Autoencoder model based on Univariate Test Dataset

free cases in a better way than the faults ones. Also, it shows better classification for Noise and stuck-at fault than offset faults.

One of the major reasons for the model's inability to identify faulty cases perfectly is because of the use of just one parameter to study the behaviour. Although, speed is the major factor affected by fault injection, there are also time periods where the vehicle reduces its speed because of obstruction or another vehicle in the path instead of fault injection. So, this decrease in speed due to other reasons are also studied as distortion due to fault injection by the DL models. Taking other features like Rotation, Position and Translation along with speed, might not solve the problem of classification completely, but will still be able to help improve the performance as shown further.

9.2 Deep Learning Model Performance for Multi-Variate Data

Similar to the last section, the DL models are now tested with multivariate data and the results with different DL models are compared. The data for both univariate and multivariate scenarios are pre-processed, shuffled and split in the same way as discussed in 7.

Table 9.1 shows the performance of diff DL models like Simple RNN, GRU-Simple RNN, Bidirectional GRU-Bidirectional Simple RNN, Autoencoders, Transformer based classifier and LSTM-Autoencoders on multivariate test data.

Architecture	**Precision**	**Recall**	**F1 score**	**Accuracy**
Simple RNN	65%	66%	63%	66%
GRU-Simple RNN	75%	76%	75%	76%
BiGRU-BiSimple RNN	80%	80%	80%	80%
Autoencoder	58%	61%	55%	61%
Transformer	56%	57%	48%	57%
LSTM-Autoencoder	**83%**	**83%**	**82%**	**83%**

Table 9.3: Performance of Deep Learning Models based on Multivariate Test Dataset

As we can see, using multivariate data increases accuracy for all models except auto-encoders and transformers. It is evident here as well that using Bi-directional hybrid structure for GRU-Simple RNN provides better results than the basic GRU-Simple RNN structure. Transformers behave rather badly in this case compared to other models, showing 57% accuracy. Also, similar to above section, it is quite clear that Autoencoders, when used purely, do not show good results. However, when used a Hybrid LSTM-Autoencoder architecture,it shows a drastic improvement and provides the best result among the used DL models with **83%** accuracy, and **82%** F1 score.

To understand the class-wise performance of the LSTM-Autoencoders, the classification report is shown in Table 9.4. It can be seen that identification of Fault-free cases are rather done quite fairly as compared to faulty cases. In this case as well, Noise and stuck-at fault are classified and detected better than Offset fault. Overall, the performance and classification turns out to be better than Univariate data.

Class Name	**Precision**	**Recall**	**F1 Score**	**Support**
Fault Free	88%	93%	91%	24860
Noise Fault	69%	86%	77%	9148
Stuck-at Fault	84%	76%	80%	6490
Offset fault	84%	49%	62%	7011

Table 9.4: Classification Report of LSTM-Autoencoder model based on Multivariate Test Dataset

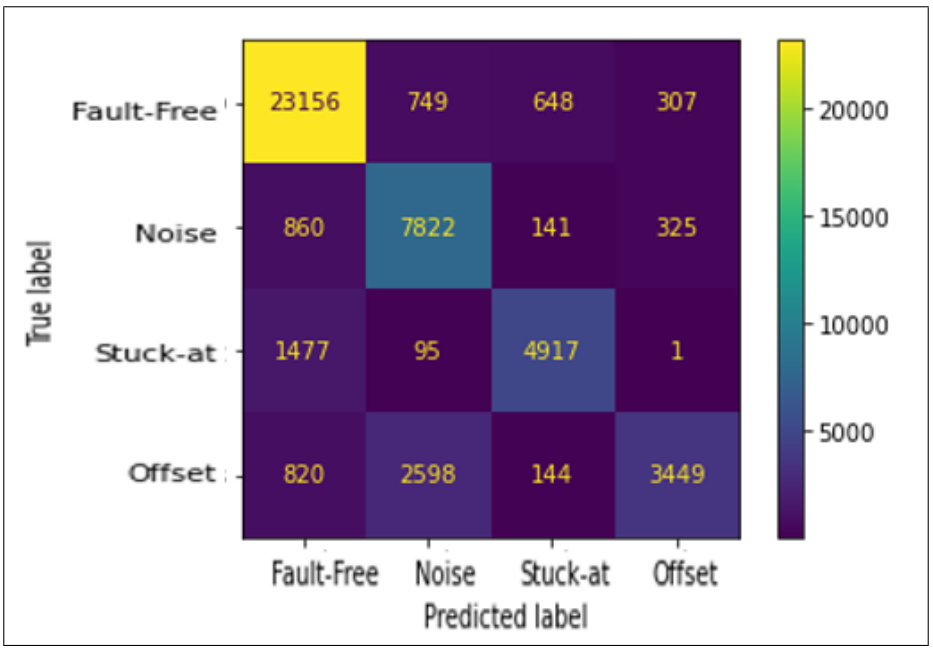

Figure 9.2: Confusion Matrix of LSTM-Autoencoder model based on Multivariate Test Dataset

As mentioned in the previous section, we are able to improve the results because of consideration of different features along with speed. Since, we consider a traffic scenario, other features also get disturbed due to obstructions or other vehicles in the path instead of fault injection, which results errors during classification of the faults. Also, it has been observed that Deep Learning Based Anomaly Detection methods are sensitive to point and contextual anomalies [15] i.e. mainly respond to sudden fault injections and may fail when there are collective anomalies or faults persisting for a long time.

Chapter 10

Conclusion and Future Scopes

10.1 Conclusion

This Master thesis is able to successfully monitor DL based fault injection, detection and classification for urban traffic in MOBATSim Model. Two different types of datasets are considered and evaluated with these Models i.e. univariate and multivariate data for both faulty and fault-free scenarios.

In the pre-processing method, datasets are compressed from 8 columns to 4 columns for multivariate dataset and assigned column names and then proceeded for ground truth labeling based on time at which fault and the type of fault was injected. It is then followed by data shuffling, inorder to get efficient training of models and then split in test and training datasets. Lastly, categorical conversion is used with these data in order to perform class vector conversion to binary class matrix.

After pre-processing both types of data sets, Fault investigation and classification process is performed separately on both types of data sets to observe the performance. For this purpose six different DL model based architectures are implemented i.e. Simple RNN, Simple RNN, Autoencoder based Classifier, Transformer based Classifier, GRU-Simple RNN, Bidirectional GRU-Bidirectional RNN and LSTM-AE on the pre-processed data. It is observed that, **LSTM-Autoencoders** performed best out of all the implemented DL models in this thesis for both uni-variate and multivariate Data, thus providing an accuracy of **77% in case of uni-variate data** and **83% in case of multivariate data**. Also, it is evidently observed that not all DL models perform better in case of multivariate data. Some models like Transformers and Autoencoders provide better accuracy for fault classification in case of uni-variate data. Moreover, Hybrid architecture of Bidirectional GRU- Bidirectional Simple RNN were able to outperform basic architecture of GRU-Simple RNN in both cases.

The reasons behind not having achieved perfect accuracy for fault classification using these Deep learning models can be referred to as lack of understanding, lack of high-quality data and inability to detect fault persisting for a long time. In the study of fault detection and classification, it is highly important to reduce false alarms of fault injection as well as missing the faulty events as it can result in compromising the safety of driving system.

10.2 Future Scopes

- In this thesis, data was collected by changing driving scenarios in-order to provide better evaluation in different traffic models. Further changes in vehicle model especially based on controller block can be implemented and evaluated further.

- Other fault types like hardware and network faults can be implemented as in this thesis, only 3 classes of faults were studied. Since injecting Package drop faults and bit flip faults did not affect the features at all as compared to the fault free case, so they were not further considered here. But further studies to make these fault affect vehicles parameters can be implemented.

- Improvements on the adapted DL models can be performed to acheive perfect accuracy for fault classification. Also, other DL and ML models can be adapted for better fault classification.

- Since, in this thesis DL models were supposed to be built to train both uni-variate and multivariate data, not many pre-processing steps on multi-variate data could be observed as uni-variate data performed poorly with those processes. Therefore, different pre-processing methods can be used in case of multi-variate data in order to improve the performance for different archtectures.

List of Figures

List of Tables

Bibliography

[1] Andrzej Wardziński. Safety assurance strategies for autonomous vehicles.

[2] Mobatsim, url: https://mobatsim.com/ github: https://github.com/MOBATSim/MOBATSim.

[3] Mustafa Saraoglu, Andrey Morozov, and Klaus Janschek. MOBATSim: MOdel-based autonomous traffic simulation framework for fault-error-failure chain analysis. *IFAC-PapersOnLine*, 52(8):239 – 244, 2019. 10th IFAC Symposium on Intelligent Autonomous Vehicles IAV 2019.

[4] Tagir fabarisov (2021). fault injection block (fiblock), github. retrieved august 25, 2021.), url: https://github.com/Flatag/FIBlock/releases/tag/v1.0.3.

[5] Traffic modeling and simulation, url: https://liu.se/en/research/traffic-modelling-and-simulation/.

[6] Muhammad Hanif Tunio, Imran Memon, Ghulam Ali Mallah, Noor Ahmed Shaikh, Riaz Ahmed Shaikh, and Yumna Magsi. Automation of traffic control system using image morphological operations. In *2020 International Conference on Information Science and Communication Technology (ICISCT)*, pages 1–4, 2020.

[7] Peter Kafka. The automotive standard iso 26262, the innovative driver for enhanced safety assessment technology for motor cars. 2012.

[8] Iso 26262-1:2018, url: https://www.iso.org/standard/68383.html.

[9] Ricky Henry Rawung and Aji Gautama Putrada. Cyber physical system: Paper survey. In *2014 International Conference on ICT For Smart Society (ICISS)*, pages 273–278, 2014.

[10] Carl Bergenhem, Erik Hedin, and Daniel Skarin. Vehicle-to-vehicle communication for a platooning system. *Procedia - Social and Behavioral Sciences*, 48:1222–1233, 2012. Transport Research Arena 2012.

[11] Margarita Martínez-Díaz and Francesc Soriguera. Autonomous vehicles: theoretical and practical challenges. *Transportation Research Procedia*, 33:275–282, 2018. XIII Conference on Transport Engineering, CIT2018.

[12] Christina Diakaki, Markos Papageorgiou, Ioannis Papamichail, and Ioannis Nikolos. Overview and analysis of vehicle automation and communication systems from a motorway traffic management perspective. *Transportation Research Part A: Policy and Practice*, 75:147–165, 2015.

[13] Franco Van Wyk, Yiyang Wang, Anahita Khojandi, and Neda Masoud. Real-time sensor anomaly detection and identification in automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1264–1276, 2019.

[14] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[15] Yuan Luo, Ya Xiao, Long Cheng, Guojun Peng, and Danfeng Yao. Deep learning-based anomaly detection in cyber-physical systems: Progress and opportunities. *ACM Computing Surveys (CSUR)*, 54(5):1–36, 2021.

[16] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, et al. Towards automated safety vetting of plc code in real-world plants. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 522–538. IEEE, 2019.

[17] Seongmin Heo and Jay H. Lee. Fault detection and classification using artificial neural networks. *IFAC-PapersOnLine*, 51(18):470–475, 2018. 10th IFAC Symposium on Advanced Control of Chemical Processes ADCHEM 2018.

[18] Qiong Liu and Ying Wu. Supervised learning. 01 2012.

[19] Phillip Wenig and Sebastian Schmidl. Anomaly detection in time series: A comprehensive evaluation. 2021.

[20] Mohammad Braei and Sebastian Wagner. Anomaly detection in univariate time-series: A survey on the state-of-the-art. *CoRR*, abs/2004.00433, 2020.

[21] Kozitsin Vyacheslav Olegovich, Iurii Katser, and Waico team. Time series anomaly detection. 2020.

[22] Astha Garg, Wenyu Zhang, Jules Samaran, Ramasamy Savitha, and Chuan-Sheng Foo. An evaluation of anomaly detection and diagnosis in multivariate time series. *IEEE Transactions on Neural Networks and Learning Systems*, 33(6):2508–2517, 2022.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[24] Taghi M Khoshgoftaar Naeem Seliya Randall Wald Edin Muharemagic Maryam M Najafabadi, Flavio Villanustre. Deep learning applications and challenges in big data analytics. *Journal of Big Data 2*, 2015.

[25] Zhihan Li, Youjian Zhao, Jiaqi Han, Ya Su, Rui Jiao, Xidao Wen, and Dan Pei. Multivariate time series anomaly detection and interpretation using hierarchical inter-metric and temporal embedding. 2021.

[26] Hang Zhao, Yujing Wang, Juanyong Duan, Congrui Huang, Defu Cao, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. Multivariate time-series anomaly detection via graph attention network. *CoRR*, abs/2009.02040, 2020.

[27] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, Mar 2019.

[28] László Monostori. *Cyber-Physical Systems*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[29] Mustafa Saraoğlu, Andrey Morozov, Mehmet Turan Söylemez, and Klaus Janschek. Errorsim: A tool for error propagation analysis of simulink models. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 245–254, Cham, 2017. Springer International Publishing.

[30] 'fault injection block', url: https://flatag.tech/fiblock.html,.

[31] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *CoRR*, abs/2104.05314, 2021.

[32] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[33] Arne Wolfewicz. 'deep learning vs. machine learning – what's the difference?'.

[34] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[35] Ajay Shrestha and Ausif Mahmood. Review of deep learning algorithms and architectures. *IEEE Access*, 7:53040–53065, 2019.

[36] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[37] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, Mar 2020.

[38] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *CoRR*, abs/2003.05991, 2020.

[39] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, abs/1912.05911, 2019.

[40] Anvardh Nanduri and Lance Sherry. Anomaly detection in aircraft data using recurrent neural networks (rnn). In *2016 Integrated Communications Navigation and Surveillance (ICNS)*, pages 5C2–1–5C2–8, 2016.

[41] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, April 1998.

[42] Jeff Donahue, Lisa Anne Hendricks, Marcus Rohrbach, Subhashini Venugopalan, Sergio Guadarrama, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description, 2016.

[43] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[44] Yulius Denny, Harco Leslie Hendric Spits Warnars, Widodo Budiharto, Achmad Imam Kistijantoro, Yaya Heryadi, and Lukas Lukas. Lstm and simple rnn comparison in the problem of sequence to sequence on conversation data using bahasa indonesia. pages 51–56, 09 2018.

[45] C. Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. *Advances in Neural Information Processing Systems*, 26, 01 2013.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9(8):1735–1780, November 1997.

[47] Steven Elsworth and Stefan Güttel. Time series forecasting using lstm networks: A symbolic approach. 2020.

[48] Arvind Mohan and Datta Gaitonde. A deep learning based approach to reduced order modeling for turbulent flow control using lstm neural networks. 04 2018.

[49] Guizhu Shen, Qingping Tan, Zhang Haoyu, Ping Zeng, and Jianjun Xu. Deep learning with gated recurrent unit networks for financial sequence predictions. *Procedia Computer Science*, 131:895–903, 01 2018.

[50] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[51] B.C. Mateus, M. Mendes, J.T. Farinha, R. Assis, and A.M. Cardoso. Comparing lstm and gru models to predict the condition of a pulp paper press.

[52] Zhaowei Qu, Lun Su, Xiaoru Wang, Shuqiang Zheng, Xiaomin Song, and Xiaohui Song. A unsupervised learning method of anomaly detection using gru. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 685–688, 2018.

[53] X Xu, F. Qin, and W. Zhao. Anomaly detection with gru based bi-autoencoder for industrial multimode process. *Int. J. Control Autom. Syst. 20, 1827–1840(2022.*

[54] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 273–278, 2013.

[55] Okwudili M. Ezeme, Qusay H. Mahmoud, and Akramul Azim. Dream: Deep recursive attentive model for anomaly detection in kernel events. *IEEE Access*, 7:18860–18870, 2019.

[56] G. E. HINTON and R. R. SALAKHUTDINOV. Reducing the dimensionality of data with neural networks, 2006.

[57] Bang Xiang Yong and Alexandra Brintrup. Do autoencoders need a bottleneck for anomaly detection?, 2022.

[58] Thorben Finke, Michael Krämer, Alessandro Morandini, Alexander Mück, and Ivan Oleksiyuk. Autoencoders for unsupervised anomaly detection in high energy physics. *Journal of High Energy Physics*, 2021(6), jun 2021.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[60] Hongcheng Guo, Xingyu Lin, Jian Yang, Yi Zhuang, Jiaqi Bai, Tieqiao Zheng, Bo Zhang, and Zhoujun Li. Translog: A unified transformer-based framework for log anomaly detection. *CoRR*, abs/2201.00016, 2022.

[61] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. Tranad: Deep transformer networks for anomaly detection in multivariate time series data. *CoRR*, abs/2201.07284, 2022.

[62] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

[63] Hoang Duy Trinh, Engin Zeydan, L. Giupponi, and Paolo Dini. Detecting mobile traffic anomalies through physical control channel fingerprinting: A deep semi-supervised approach. *IEEE Access*, PP:1–1, 10 2019.

[64] Hajar Homayouni, Sudipto Ghosh, Indrakshi Ray, Shlok Gondalia, Jerry Duggan, and Michael G. Kahn. An autocorrelation-based lstm-autoencoder for anomaly detection on time-series data.

[65] Yuanyuan Wei, Julian Jang-Jaccard, Wen Xu, Fariza Sabrina, Seyit Camtepe, and Mikael Boulic. Lstm-autoencoder based anomaly detection for indoor air quality time series data, 2022.

[66] Nazri Mohd Nawi, Walid Atomi, and Syed Muhammad Rehman Gillani. The effect of data pre-processing on optimized training of artificial neural networks. volume 11, 06 2013.

[67] Build a simple recurrent neural network with keras, url: https://pythonalgos.com/build-simple-recurrent-neural-network-with-keras/.

[68] Implementation of simplernn, gru, and lstm models in keras and tensorflow for an nlp project, url: https://regenerativetoday.com/implementation-of-simplernn-gru-and-lstm-keras-and-tensorflow/.

[69] Bidirectional layer, url: https://keras.io/api/layers/recurrent_layers/bidirectional/.

[70] Anomaly detection with autoencoders made easy, url: https://towardsdatascience.com/anomaly-detection-with-autoencoder-b4cdce4866a6.

[71] Transformer explained, url: https://paperswithcode.com/method/transformer.

[72] A gentle introduction to lstm autoencoders, url: https://machinelearningmastery.com/lstm-autoencoders/.

Declaration

Herewith, I declare that I have developed and written the enclosed thesis entirely by myself and I have declared all the sources or means used .

The report is being submitted for the fulfillment of Master Thesis in Electrical Engineering, to the University of Stuttgart, Germany.

This thesis has not been submitted to any other authority for achieving an academic grade and has not been published elsewhere.

Stuttgart, 13th June 2022 Nikita Jhawar