Technische Universität München

The present work was submitted to the Chair of Space Mobility and Propulsion

presented by

Miguel Barros Marques

Student ID No.: 03793021

Master Thesis

Deep Learning for Fault Detection and Identification of a Hopper Propulsion System

Munich, May 30, 2025

Supervising professor: Prof. Dr.-Ing. Chiara Manfletti
Assistant supervisor: Felix Ebert, M.Sc.

1st examiner: Prof. Dr.-Ing. Chiara Manfletti

Statutory Declaration

I hereby declare that the present work was written independently by me, and that no other aids than those specified were used. The sections of the work that have been taken from other works, either verbatim or in essence, are clearly marked as such in each individual case, with an indication of the source. This declaration also extends to graphics, drawings, map sketches, and pictorial representations contained in the work.

I also agree that my Master's paper may be made accessible by the department to professionally interested persons upon request, including through a library, and that the results contained therein, as well as the developments and programs created during this work, may be used without restriction by the Chair of Space Mobility and Propulsion. (Rights to any programs and inventions that may arise must be clarified in advance.)

Name: Miguel Marques

Matr. Nr.: 03793021

Place, Date 30/05/2025

Signature Miguel Marques

Abstract

This thesis investigates the application of deep learning methods for fault detection and identification (FDI) in a simulated rocket propulsion system for a small hopper vehicle. Two neural network architectures, a feedforward neural network (FNN) and a long short-term memory (LSTM) network, are developed and trained on synthetic time-series data generated via *EcosimPro*. The aim is to assess whether such models can meet the reliability and latency requirements of real-time fault management, while retaining adaptability to system design changes.

Both architectures demonstrate the ability to learn meaningful fault representations and generalize across a diverse set of failure modes. The LSTM outperforms the FNN in key macro-averaged metrics, precision (0.850 vs. 0.745), recall (0.821 vs. 0.695), and F1 score (0.826 vs. 0.706). These improvements, however, come at the cost of increased model complexity.

A central finding of this study is the pivotal role of labeling strategy with misalignments between fault flags and observable signal deviations leading to degraded detection performance, particularly for delayed valve response and sensor freeze faults. Furthermore, this work emphasizes the importance of evaluation metrics that isolate fault detection capabilities, explicitly excluding correct "Normal" classifications to avoid inflated performance estimates.

The study demonstrates that deep-learning FDI can potentially satisfy rocket propulsion latency and accuracy targets. The results establish a foundation for data-driven FDI in aerospace propulsion systems and identify several directions for future work, including hybrid architectures, confidence-based thresholding, and anticipatory fault prediction mechanisms.

Keywords Fault Detection and Identification (FDI), Rocket Propulsion, Hopper, Deep Learning, LSTM.

Contents

List of Figures

List of Tables

List of Symbols

General Symbols

d_{hidden}	[-]	Number of neurons in hidden layer
L_{orig}	[-]	Original model loss
L_{reg}	[-]	Regularized model loss
m_{prop}	[kg]	Initial propellant mass
p	[-]	Dropout probability
T_{asc}	[N]	Ascent thrust
Y_{targ}	[m]	Target height
$\mathbf{b}$	[-]	Neuron bias (Vector)
b	[-]	Neuron bias (Scalar)
T/W	[-]	Thrust-to-weight ratio
$\mathbf{W}$	[-]	Neuron weight (Vector)
w	[-]	Neuron weight (Scalar)
$\mathbf{x}$	[-]	Input Vector
$\mathbf{y}$	[-]	Output Vector

Greek Symbols

η	[-]	Learning rate
γ	[-]	Focal loss focus parameter
λ	[-]	L2 regularization coefficient
σ	[-]	Activation function

Abbreviations

AE	Autoencoder

AI	Artificial Intelligence
ANN	Artificial Neural Network
ATA	Adaptive Threshold Algorithms
CEL	Cross-entropy loss
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-separated values
CUDA	Compute Unified Device Architecture
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DL	Deep Learning
FDI	Fault Detection Identification
FNN	Feed Forward Neural Network
FN	False Negatives
FPR	False Positive Rate
FP	False Positives
GA-LSSVR	Genetic Algorithm-Optimized Least Squares Support Vector Regression
GB	Gigabyte
GPU	Graphics Processing Unit
HMS	Health Monitoring System
kNN	k-Nearest Neighbors
LRE	Liquid-Propellant Rocket Engines
LSTM	Long Short-Term Memory
LUMEN	Liquid Upper Stage Demonstrator Engine
ML	Machine Learning
NB	Naïve Bayes
NN	Neural Network
NPY	NumPy
PCA	Principal Component Analysis
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SSME	Space Shuttle Main Engine
SVM	Support Vector Machine
THAES	Titan Health Assessment Expert System
TN	True Negatives
TPR	True Positive Rate
TP	True Positives
TUM	Technical University of Munich

1 Introduction

Rocket propulsion technology stands as the backbone of space exploration since the inception of the Space Age, inaugurated by the launch of Sputnik 1 on 4 October 1957. This milestone marked the beginning of a global technological race that, over the decades, has transformed space activity into a strategic pillar of modern society.

In recent years, particularly since the 2010s, the space sector has entered an era of accelerated expansion and diversification, driven by the dual momentum of public investment and the emergence of private-led initiatives under the so-called *NewSpace* paradigm. By 2023, the global space economy reached revenues of approximately US$570 billion, reflecting a 7.4% increase over the previous year and nearly doubling that of the past decade [1].

Expansion in launch capacity has paralleled a diversification of space mission objectives. Rocket propulsion now serves as a foundational technology for a variety of domains. In the scientific realm, it enables interplanetary missions, Earth observation, and astronomical research. For commercial applications, launch vehicles support rapidly growing satellite constellations for telecommunications, navigation, and real-time telemetry. Finally, as an emerging class, the space tourism sector provides recreational offers such as microgravity experimentation and private crewed flights, further underscoring the need for safe space vehicles.

The increasing attention devoted to fault detection and identification (FDI) in launcher systems stems from the critical impact of propulsion failures, which can cause substantial financial losses, mission aborts, or irreversible harm to on-orbit assets. As the aerospace sector continues to prioritize reliability, cost-efficiency, and fault tolerance, the ability to automatically detect anomalies and infer their origin has become an essential component of modern launch vehicle design and operation.

The present thesis investigates the design of an FDI framework for a space hopper's propulsion system using deep learning techniques. In this chapter, the context and motivation for the work are outlined, including an overview of the sector's growth and its applications, the specific challenges of propulsion system fault management, and the objectives and scope of the research.

1.1 Motivation and Objectives

The paradigm shift led by the *NewSpace* movement has ushered in a novel era defined by the proliferation of different actors, including emerging national programs, private aerospace companies, and startups. This democratization of access to space has driven up launch rates significantly. Between 2016 and 2021 alone, the annual number of orbital launch attempts increased by approximately 70%, as recorded by global spaceflight statistics [2]. The market is now shaped by a dynamic interplay between public and

private actors, both of which are pushing the development of increasingly cost-effective and flexible launch systems. The result is a competitive environment that demands shorter development cycles, modular architectures, and recoverable systems. Premises for which robust and fault-tolerant performance is key.

Analysis of historical launch data shows that propulsion-related malfunctions are the single largest contributors to launch vehicle failures, amounting to close to 50% of all major failures [2, 3]. The propulsion subsystem performs the essential task of delivering thrust under extreme conditions, and its complexity makes it prone to a variety of failure modes. As a result, ensuring the health and fault-free operation of rocket engines is paramount for mission success and safety. An undetected fault in an engine could lead to mission failure or even catastrophic accidents, with severe financial and safety implications. Conversely, early detection and identification of such faults can enable mitigative actions. The goal shall be enabling system reconfiguration or backup switchover to ensure mission completion and, in more critical situations, controlled shutdown and mission abortion safeguards equipment and prevents catastrophic outcomes, including life-endangering scenarios.

FDI frameworks are therefore indispensable in modern rocket launch systems. Broadly speaking, the concept refers to monitoring techniques that automatically recognize when a component or subsystem is deviating from normal behavior and determine the nature and source of the fault. Such solutions have long been studied in other domains. For instance, in robotics and industrial automation, they are used to improve safety and autonomy. Over the years, FDI methods have evolved from classical approaches into sophisticated and intelligence-driven tools. In the robotics field, early FDI and fault-tolerant control schemes were largely model-based or rule-based (i.e., analytically defined), but recent advances integrate artificial intelligence (AI) to enhance fault detection capabilities. This evolution highlights how contemporary trends like machine learning (ML) and digital twin technology are transforming fault diagnosis in complex systems. The success of these approaches in robotics underscores the potential of data-driven FDI techniques to handle system complexities that are out of reach for purely analytical models [4].

The development of heritage rocket engines requires dozens of test firings and multiple design and component iterations in order to reach stable full-power operation, illustrating the difficulty of predicting and diagnosing issues early on. Thus, traditional analytical FDI methods may become cumbersome and unfeasible solutions when the system starts to deviate from the initial assumptions or unmodeled dynamics are found. This complexity motivates the use of more flexible approaches, such as data-driven solutions [5, 6]. By leveraging data (either from real-world testing or high-fidelity simulations), ML methods can learn the subtle signatures of faults directly from the system's behavior, potentially detecting incipient anomalies that would be missed by coarse analytical models while remaining flexible to changes in the underlying design.

Another motivator is the push for early and proactive fault management. As further discussed in section 4.3, some faults have a small mitigation window, allowing for less than 50 ms decision periods. Besides this, even in more lax scenarios, the overhanging complexity of FDI systems can postpone the inference process just enough to deem the system unsuitable for real-time applications. With these premises in mind, the system's accurate fault identification is no longer enough, and solutions that are able to predict anomaly onset are becoming increasingly important. For instance, Yu et al. [3] developed an engine health management scheme that uses a Long Short-Term Memory (LSTM) neural network to forecast fault evolution in real-time, combined with a fuzzy-logic control law to mitigate performance degradation when a fault is detected. This illustrates the trend toward intelligent fault management

systems in rocketry, where forecasting, detection, and mitigation are integrated to handle off-nominal conditions autonomously.

In light of the above considerations, the present thesis proposes to harness deep learning (DL) for FDI in a hopper's propulsion system. ML based FDI has the potential to handle the complexity of engine data and detect subtle anomalies without the burden of an exact physics model. By learning directly from data, this solution aims to generalize across the engine's operating envelope and identify faults rapidly. The ultimate motivation is to achieve a solution that can be deployed in real-time, providing early warning of faults and identifying their source so that appropriate action can be taken.

The objectives defined within the present thesis are enumerated below:

1. **Develop an FDI framework that is scalable and adaptable** – The solution shall be easily adaptable to design or system configuration changes. Given that the propulsion system may evolve, the FDI approach is expected to be robust against modifications and require minimal re-tuning. This calls for a generalizable model structure and a training process that can leverage new data as it becomes available.
2. **Achieve high fault detection accuracy with low detection latency** – The proposed system shall reliably detect the occurrence of faults and correctly identify their source. Important performance metrics include high accuracy and a fast detection time. Ideally, the system should recognize a fault shortly after its onset, within a time window that allows intervention before the fault leads to mission failure. Concrete quantitative references are proposed in section 4.7.
3. **Incorporate forecasting capabilities** – Beyond just detecting faults after they occur, the framework shall, ideally, provide a form of prognosis, anticipating or predicting developing faults.
4. **Ensure real-time deployability** – The system's integration and deployment within the hopper's hardware shall deliver the performance achieved during development.

1.2 Scope and Limitations

The scope of the present work is focused on demonstrating the feasibility of the proposed FDI approach on a specific rocket propulsion system, and within a controlled set of fault scenarios. The testbed for this study is a bi-liquid pressure-fed engine developed at the Technical University of Munich (TUM) for a hopper vehicle. Further details on the hopper's architecture are explored in section 4.2. All experiments and data in this work are based on this particular configuration. Consequently, the findings are tailored to the characteristics of this system, and direct generalization to considerably different setups should be approached with caution. Nevertheless, the methodology is formulated to be adaptable, as one of the goals is to handle changes in hardware. An important consideration given that the hopper design is not yet finalized and may undergo revisions.

As will be discussed in chapter 4, another limitation is in the data used. Given that the hopper build has yet to be finalized, synthetic data is instead generated from a high-fidelity analytical engine replica using *EcosimPro*. The use of synthetic data for model training under controlled conditions. However, it also introduces some limitations: the models are only as good as the fidelity of the simulator, and certain real-world effects (sensor noise, unmodeled physics, unexpected fault modalities) may not be fully

captured in the generated data. Thus, one shall be cautious in extrapolating the achieved performance directly to real-world operation. The present thesis discusses these limitations, and the FDI system is designed with flexibility to be retrained or fine-tuned when real data becomes available.

It should also be noted that the scope of this work is limited to the detection and identification of faults. In other words, once a fault is detected and classified, the thesis does not delve into the subsequent fault-tolerant control actions or engine reconfiguration that would ideally follow in an operational system. Handling the system's response to faults (such as throttling down a failing engine or switching to backups) is a separate challenge beyond this work's scope.

1.3 Thesis Outline

This thesis is organized into eight chapters, each building progressively toward the demonstration of deep learning-based fault detection and identification (FDI) for a hopper propulsion system.

- **Introduction:** Introduces the context and motivation for the study, outlines the problem domain, and presents the research objectives and scope.
- **Fundamentals:** Reviews the theoretical background, covering classical machine learning, neural networks, time-series modeling, and fault detection strategies, with emphasis on their relevance to aerospace systems.
- **FDI Applications in Rocket Propulsion:** Literature review on FDI solutions in the context of rocket propulsion.
- **Methodology:** Presents the simulation setup and dataset generation based on an *EcosimPro*-modeled propulsion system, followed by the methodological framework used for model development, including data preprocessing, model architecture design, and strategies to address overfitting, class imbalance, and deployment constraints.
- **FNN:** Presents the feedforward neural networt experimental results, including hyperparameter tuning, model training behavior, and performance comparisons across fault classes.
- **LSTM:** Provides the experimental evaluation of the long short-term memory model, covering hyperparameter optimization, training dynamics, and fault-wise performance analysis.
- **Discussion** Provides an in-depth discussion of the results, highlighting strengths and limitations of each model, interpreting class-specific behaviors, and addressing deployment-time considerations.
- **Conclusion and Future Work:** Concludes the thesis with a summary of key findings and a discussion of limitations. It closes with suggestions for future work and possible avenues for improving early fault identification.

2 Fundamentals

The present chapter establishes the theoretical and conceptual foundation of the thesis. It introduces essential definitions and principles related to fault detection and identification, with a particular focus on learning-based approaches. The covered content provides the necessary context to understand the techniques and results discussed throughout the thesis.

2.1 FDI: What is it?

FDI is a critical element in the maintenance and safe operation of complex engineering systems such as rocket propulsion. As systems increase in complexity and autonomy, particularly in safety-critical domains, the ability to continuously monitor, diagnose, and respond to faults becomes a central requirement. This section presents the key concepts underpinning FDI, specifically the definition behind detection, isolation, and identification, and situates them within the broader framework of a health monitoring system (HMS). The general sequence encompassing fault detection, isolation, and identification is illustrated in Figure 2.1. This progression aims to acquire comprehensive diagnostic information systematically, thereby establishing a solid foundation for subsequent fault mitigation strategies, such as the implementation of reconfigurable control schemes or for limiting the adverse impact of detected faults.

Fault Detection Consists of the process of determining whether a fault, defined as any unpermitted deviation of at least one characteristic property or parameter of the system from acceptable behavior, has occurred within a system. At this higher level, the main goal is a binary classification, that is, to determine if the system is functioning correctly or if there is evidence of abnormal operation. This step is foundational, as undetected faults can spread throughout the system and cause performance degradation, reduced safety margins, or catastrophic failure [7].

Fault Isolation Follows detection and concerns pinpointing the specific subsystem or component in which the fault has manifested. In complex systems equipped with multiple sensors and actuators, fault isolation is essential to narrow down the set of candidate components responsible for the detected anomaly. Effective isolation reduces the time required for subsequent troubleshooting, repair, or reconfiguration and is a prerequisite for mitigation actions [7].

Fault Identification Refers to the process of characterizing or classifying the fault after it has been detected and isolated. This involves determining not only the presence and location of a fault but also its type (e.g., drift, spike, freeze) and possibly its magnitude or severity. Fault identification allows for a nuanced assessment of the system's remaining functional capabilities and informs subsequent

decision-making processes, such as reconfiguration or mission adaptation. Accurate fault identification is particularly valuable in systems where different fault types necessitate distinct responses, for example, distinguishing between a minor drift and a catastrophic malfunction in a propulsion control sensor. [7]

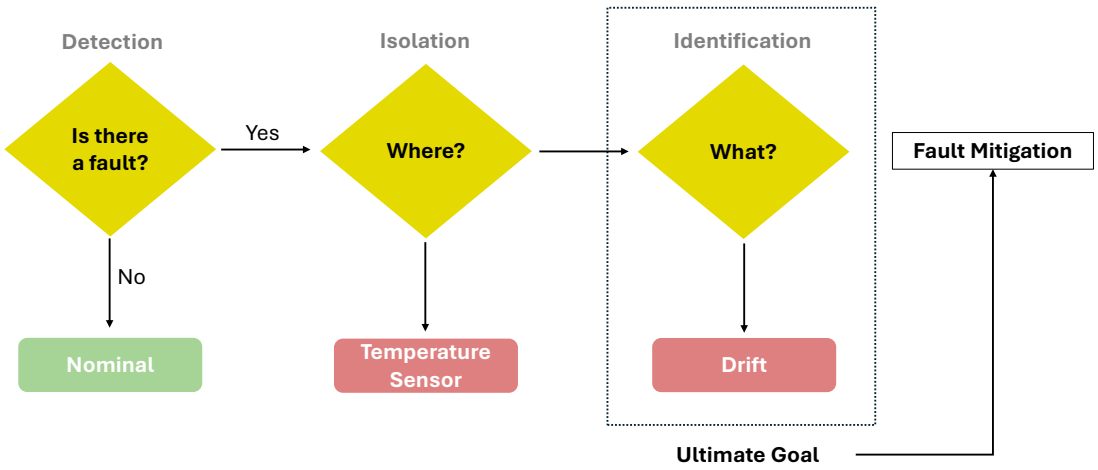


Figure 2.1: Fault detection, isolation, and identification sequence.

2.2 Evaluation Metrics for FDI

Performance metrics are quantities used to evaluate an FDI system's ability to accurately identify faulty and non-faulty conditions. A key element in understanding these metrics is the confusion matrix, shown in Figure 2.2, which categorizes classification outcomes into four groups: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Using these values, different metrics can be computed which, collectively, provide a comprehensive framework for evaluating the performance of FDI systems, particularly in the presence of class imbalance and varying cost implications for different types of classification errors.

2.2.1 Accuracy

Accuracy is a general metric defined as the ratio of correctly classified samples to the total number of samples, as defined in Equation 2.1.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

Although widely used, accuracy can be misleading when dealing with imbalanced datasets. For example, in a case where only 5% of the data contains faults, a model that predicts every sample as "no fault" achieves an accuracy of 95%, despite completely failing to detect any faults.

Confusion Matrix		Predicted	
		Fault	No Fault
Test	Fault	TP	FN (Missed Fault)
	No Fault	FP (False Alarm)	TN

Figure 2.2: Confusion Matrix.

2.2.2 Precision

A more informative metric in such scenarios is the precision, defined in Equation 2.2.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}$$

Precision measures the proportion of correctly predicted faulty instances among all instances classified as faulty. It is useful in situations where false positives incur a high cost. High precision implies a low rate of FPs. However, precision does not account for false negatives.

2.2.3 Recall

Recall, also known as sensitivity or true positive rate (TPR), is given by Equation 2.3.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}$$

It quantifies the proportion of actual faults that are correctly identified. This metric is particularly important when the cost of missed faults is high. Nevertheless, recall alone does not capture the false alarm rate. A classifier labeling all instances as faults achieves a recall of 100% but lacks practical value.

2.2.4 False Positive Rate

The False Positive Rate (FPR) reflects the proportion of non-faulty instances incorrectly classified as faulty and is defined in Equation 2.4.

$$\text{FPR} = \frac{FP}{FP + TN} \tag{2.4}$$

Together with TPR, FPR is useful in Receiver Operating Characteristic (ROC) curve analysis, which visualizes the trade-off between detecting actual faults and avoiding false alarms.

2.2.5 F1 Score

To address the limitations of using precision or recall in isolation, the F1 Score is employed. It is the harmonic mean of precision and recall, as shown in Equation 2.5

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.5}$$

The F1 Score penalizes large discrepancies between precision and recall, making it more robust in cases where the two metrics diverge significantly.

2.3 AI: Field History and Overview

Artificial Intelligence (AI) is a broad domain within computer science concerned with developing machines capable of performing tasks that traditionally require human intelligence, such as reasoning, learning, perception, and decision-making. These systems are designed to emulate human cognitive functions: they analyze data, recognize patterns, and autonomously generate decisions or predictions.

The conceptual roots of AI trace back to the mid-20th century. Alan Turing, whose seminal contributions to cryptanalysis during World War II, most notably through the cracking of the Enigma machine, marked a turning point in computational theory, was among the first to formally propose that machines could exhibit intelligent behavior. His 1950 paper, *Computing Machinery and Intelligence*, laid the theoretical foundation for what would eventually evolve into the field of artificial intelligence.

The formal establishment of the field occurred in 1956, when a group of pioneering researchers, led by John McCarthy, Marvin Minsky, Claude Shannon, and others, convened the Dartmouth Summer Research Project on AI. It was during this seminal workshop that the term *artificial intelligence* was first coined, marking the inception of AI as a formal academic discipline and initiating a trajectory of research and innovation that would span decades [8].

2.3.1 Historical Evolution and Major Breakthroughs

The evolution of Artificial Intelligence spans more than seven decades, shaped by alternating periods of rapid progress and stagnation. Early research efforts in the 1950s and 1960s focused on symbolic reasoning, basic problem-solving algorithms, and game-playing systems. Notable early milestones include the first artificial neural network (NN) in 1951 and Arthur Samuel's self-learning checkers program in 1952, which demonstrated that machines could improve performance through experience. In 1966, the world saw the first steps toward natural language processing, exemplified by Weizenbaum's *ELIZA* chatbot, which simulated conversational dialogue. However, the trajectory of AI was far from linear. The field experienced cycles of inflated expectations followed by disillusionment, leading to long periods of progress stagnation [8].

A resurgence occurred in the 1980s with more capable expert systems and the introduction of Bayesian networks. These consist of graph-based models that encode dependencies between variables using conditional probabilities. Unlike rule-based systems, which rely on deterministic logic, Bayesian networks allow for reasoning under uncertainty, as they quantify how likely a hypothesis is, given partial or noisy

evidence. This probabilistic approach enabled more robust and interpretable decision-making in uncertain or variable environments. In parallel, the backpropagation algorithm provided a practical means of training multi-layer neural networks, opening the path toward deeper architectures (under the name of deep learning) [8].

The late 1990s and early 2000s marked a shift from symbolic reasoning to data-driven learning. Milestone achievements include IBM's *Deep Blue*, which defeated world chess champion Garry Kasparov in 1997 and showcased AI's ability to tackle complex decision-making problems. These advances were underpinned by increasing computational power and algorithmic refinements. Another major turning point came in 2012, when a deep convolutional neural network (*AlexNet*) significantly outperformed traditional methods in image classification tasks. This event is widely recognized as the beginning of the deep learning revolution, initiating a rapid expansion in the adoption of multi-layer neural networks across vision, language, and speech domains. In 2016, DeepMind's *AlphaGo* defeated Go world champion Lee Sedol, showcasing the potential for reinforcement learning in high-dimensional strategic environments [8].

Between 2020 and 2022, AI entered a new era with the emergence of Generative AI, a class of models capable not only of recognizing or classifying patterns, but also of creating entirely new content that resembles the data they were trained on. Unlike earlier AI systems, which primarily operated within the confines of rule-based logic or predictive classification, generative models learn the underlying statistical structure of a domain and use this knowledge to synthesize novel outputs. *OpenAI*'s *ChatGPT* exemplified this shift, showcasing AI's ability to autonomously produce human-like text, code, images, and even artistic content, blurring the boundary between computational pattern recognition and creative expression.

These landmark developments collectively illustrate the field's transition from rule-based systems to learning-based models and ultimately to today's data-centric, deep learning paradigm. The increasing capabilities of AI systems have been driven primarily by algorithmic advances, greater availability of high-quality data, and the exponential growth in computational resources.

2.3.2 Subfields

AI is an umbrella term encompassing various subfields and approaches. Within this broad domain, Machine Learning (ML) has emerged as a foundational discipline, providing data-driven techniques that allow systems to learn from experience rather than relying on hard-coded rules.

ML itself can be divided into two main branches: classical ML, which includes well-established algorithms such as decision trees and support vector machines, and neural networks, inspired by the structure of biological neurons and capable of modeling more complex patterns.

These relationships are inherently hierarchical: ML is a subset of AI; neural networks are a specialized branch within ML; deep learning refers to neural networks with multiple layers capable of memorizing high-level representations; and, at the most complex and deep level, generative AI builds on deep learning to create new content. This hierarchical structure is illustrated in Figure 2.3, which provides a visual representation of the relationships between the key subfields of AI.

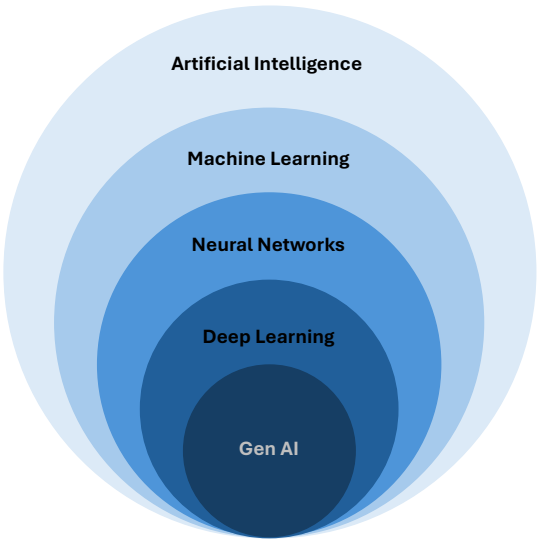


Figure 2.3: Simplified conceptual hierarchy of AI subfields.

2.3.3 Key Concepts and Definitions

Before delving into the specifics of ML applications, it is essential to establish a clear understanding of the key concepts that underpin the development and training process. These foundational principles provide the necessary context for the methodological choices and model design decisions discussed in the subsequent sections.

Parameters an Hyperparameters A parameter is an internal value learned during training (e.g., weights and biases in a neural network). In contrast, a hyperparameter is predefined and governs the learning process or model structure (e.g., learning rate, batch size, number of layers). Unlike parameters, hyperparameters must be set before the learning process begins and are often tuned experimentally to achieve optimal performance. Automated tools such as *Optuna* and *Ray Tune* are increasingly employed to efficiently search for optimal hyperparameter values, thereby improving model performance with minimal manual intervention [9].

Architecture vs. Model The term architecture refers to the structural design of a machine learning system, the number and type of layers, connectivity patterns, activation functions, etc. A model, by contrast, denotes a specific instantiation of an architecture with a fixed set of trained parameters. Multiple models may share the same architecture but differ in performance due to different initializations, training durations, or training settings [10].

Sample A sample refers to a single observation or data point drawn from a dataset. It represents one instance of the input-output relationship that the model aims to learn. Each sample typically consists of a vector of input features such as sensor readings, system states, or time-stamped measurements, and, in supervised learning (subsubsection 2.3.4), an associated label indicating the correct output.

Datasets Two main types of datasets can be distinguished, training and validation. The training set is used to fit the model's parameters, while the validation set is used to monitor generalization and guide hyperparameter tuning. The two sets must be disjoint to ensure unbiased evaluation. Validation performance typically informs decisions such as model selection and architecture refinement. A test set, if used, is held out for final evaluation after model selection [10].

Epoch Denotes one complete pass of the training algorithm over the entire training dataset. During each epoch, all training samples are presented to the model.

Overfitting Overfitting arises when a model captures idiosyncrasies of the training data rather than the underlying distribution, leading to high training accuracy but poor generalization to unseen samples, i.e., the model memorizes the training set [11]. It is usually diagnosed through a widening performance gap between training and validation curves after a certain number of epochs. Pronounced overfitting is common in deep neural networks due to their large capacity.

Regularization Regularization encompasses techniques designed to mitigate overfitting by penalizing model complexity. In neural networks, for example, regularization includes techniques such as dropout (random deactivation of neurons during training). This reduces model variance and promotes generalization without requiring additional data [11].

Inference Inference denotes forward-propagating a network to obtain outputs for the given inputs. Only deterministic computations run. Computational latency and memory footprint (i.e., temporal and spatial complexity) during inference are critical for real-time applications.

Types of Classification Tasks

Given that FDI entails determining the current operational state of the system, this task can be framed as a classification problem within the context of machine learning. Depending on the nature and complexity of the faults, different classification paradigms can be employed, as outlined below:

- **Binary classification**: exactly two mutually exclusive classes (e.g., fault vs. normal).
- **Multiclass classification:** deals with more than two classes, where each sample belongs to exactly one class (e.g., identifying the type of fault among several options).
- **Multilabel classification:** allows each input to be assigned multiple labels simultaneously, suitable for concurrent or compound faults (e.g., attributing more than one fault per sample).

Under this topic it is important to distinguish between the terms class and label, which are often used interchangeably but refer to different concepts in supervised learning. A class represents one of the predefined categories in a classification task, forming the set of possible outcomes that a model can predict. A label, by contrast, is the actual class assigned to a given data sample, indicating its ground truth category. For example, in a multiclass classification problem with classes such as {Cat, Dog, Bird}, each individual sample, such as an image, would carry a label corresponding to its true class, e.g., "Dog".

While classes define the prediction space, labels denote how specific samples are categorized within that space.

Vanishing and Exploding Gradient The vanishing and exploding gradient problem refers to numerical instabilities that can occur during training when gradients become either too small or too large as they are propagated through the network. Vanishing gradients hinder learning by causing earlier layers to update slowly or not at all, while exploding gradients lead to unstable updates and divergence. These issues are common in deep networks and can impair model performance [12].

Labeling Method Since most learning algorithms require numerical input, categorical class labels must be transformed into a numerical format prior to training. Two common encoding strategies are label encoding and one-hot encoding. Label encoding assigns a distinct integer to each class, which may inadvertently introduce an ordinal relationship between categories that is not inherently meaningful. This can lead to unintended biases in models that interpret numerical proximity as semantic similarity. In contrast, one-hot encoding represents each class as a binary vector in which only one element is active (set to one) and all others are zero. This approach maintains categorical neutrality by treating all classes as equidistant. Despite its higher memory footprint, one-hot encoding is widely adopted in neural network-based classification tasks due to its effectiveness in preserving the discrete nature of categorical labels [13]. Figure 2.4 provides a visual interpretation of the two described approaches.

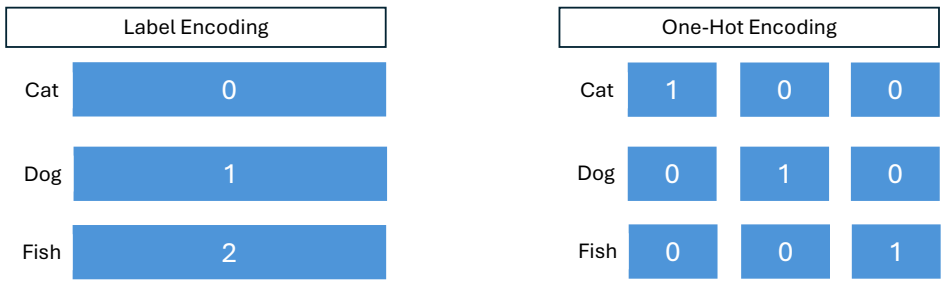


Figure 2.4: Data labeling approaches.

2.3.4 Machine Learning Overview

ML refers to algorithms and statistical models that allow computers to improve their performance on a task by learning from data. Rather than being explicitly programmed with a rigid set of rules, AI systems are trained on example data. They identify patterns and relationships within that data, and use those patterns to make predictions or decisions on new, unseen inputs.

As mentioned, ML techniques are categorized into two main classes: classical methods and neural networks. Classical methods typically establish a direct mapping between manually defined input features and output predictions. In contrast, neural networks are built upon an internal layered architecture. This architecture enables them to iteratively transform raw input data into increasingly abstract representations, a process known as hierarchical feature extraction.

This structural distinction gives rise to two principal differences between these approaches. Classical ML methods' performance is highly dependent on manual feature engineering, requiring domain expertise to select or construct informative features that the algorithm can use. Neural networks, on the other hand, learn relevant features automatically during training, thus reducing reliance on handcrafted representations and human input. Secondly, classical methods often struggle to model complex, nonlinear relationships within the data due to their lack of abstraction capability. Neural networks overcome this limitation by leveraging their depth to capture intricate patterns and interactions that are typically inaccessible to classical models without substantial prior transformation of the input space [14].

Learning Paradigms

Parallel to the distinction between classical ML methods and neural networks, ML algorithms as a whole are also commonly categorized according to how they learn from data. The primary paradigms are supervised learning, unsupervised learning, and semi-supervised learning. These approaches differ according to the type of data utilized and the nature of feedback available during training. Furthermore, there is reinforcement learning RL, a fundamentally different paradigm that does not fit within this taxonomy and that will be addressed later in this section. Table 2.1 provides a summary of the working principle behind each of these methods, as well as their main advantages and disadvantages.

Supervised Learning In supervised learning, the algorithm is trained on labeled data, that is, each training example comes with an input and a desired output (label). The goal is for the model to learn a mapping from inputs to outputs so that it can predict the correct output for new, unseen inputs. For example, an input might be an image, and the label is the category ("cat" or "dog"), or the input could be a past financial record, and the label is a future stock price. The learning algorithm is guided by explicit feedback on performance. Classification (predicting discrete labels) and regression (predicting continuous values) are two common types of supervised tasks. Supervised learning is powerful because the feedback (in the form of labels) guides the model to learn the exact desired behavior. On the other hand, it requires a comprehensive labeled dataset, which can be labor-intensive to obtain [15].

Unsupervised Learning Unsupervised learning, by contrast, explores patterns within unlabeled data, uncovering clusters or structures without direct supervision.

In unsupervised learning, the data has no labels. The algorithm's task is to discover structure in the data on its own. This could mean finding natural groupings (clustering), detecting anomalies, or learning lower-dimensional representations of the data. For example, an unsupervised algorithm could ingest thousands of customer transactions and automatically segment customers into groups with similar behaviors, without being told what those groups are in advance. Unsupervised learning is often exploratory, i.e., it can reveal patterns humans are not aware of. It is especially useful when labeling data is impractical, too expensive, or simply not available [16].

Semi-Supervised Learning This approach is a hybrid of the above two. In semi-supervised learning, the algorithm is given a small amount of labeled data and a large amount of unlabeled data. The idea is to leverage the plentiful unlabeled examples to help structure the learning, while the labeled examples guide

the model with ground truth on a subset of data. Many real-world datasets use this approach. Obtaining labels is expensive (requiring expert human annotation), but gathering raw data is cheap (sensors, web scraping, etc.). Semi-supervised methods employ techniques such as first using unlabeled data to learn underlying patterns or clusters, then fine-tuning the model with the labeled data. This paradigm has gained importance, especially in domains like image recognition or medicine, where labels are scarce but unlabeled data abounds [17].

Reinforcement Learning In RL, an agent interacts dynamically with its environment, learning optimal behavior through a process of trial and error. Rather than receiving fixed labeled examples, the agent receives reward signals, i.e., numerical feedback that may be immediate or delayed, based on the consequences of its actions. In essence, instead of receiving the correct output and comparing its own inference against the model receives instead a goal and is positively or negatively rewarded if it gets closer or further away from said goal. The objective is to learn a policy that maximizes cumulative reward over time, which often requires making a sequence of interdependent decisions and reasoning about the long-term effects of actions. The core idea is learning through trial and error. The agent takes an action in a given state of the environment, the environment transitions to a new (observable) state and provides a scalar reward signal, and the agent updates its strategy (policy) to favor actions that lead to higher cumulative rewards. Figure 2.5 provides a visual, schematic interpretation of the described process. A classic example of RL is the case of an autonomous car navigating a maze. Here, the agent (the car) must learn through repeated trial and error how to reach the goal efficiently. After each action (such as moving forward or turning), it receives feedback in the form of rewards: for instance, a positive reward for reaching the exit, a negative reward for colliding with obstacles, or a reward proportional to the reduction in distance to the goal. Over time, the car independently discovers a strategy (policy) that optimizes its route through the maze, relying solely on these reward signals rather than explicit instructions or labeled examples. This demonstrates how reinforcement learning enables agents to solve complex, sequential decision-making tasks in dynamic environments. Unlike supervised learning, where the correct output is provided for each input, RL agents must independently discover effective strategies, contending with delayed feedback and the so-called "credit assignment problem", determining which actions were responsible for observed outcomes. In summary, reinforcement learning stands apart from the classical supervised, unsupervised, and semi-supervised paradigms by its reliance on interaction-driven, reward-based feedback rather than static labeled or unlabeled datasets. It is especially well-suited for problems requiring adaptive, sequential decision-making and can be complementary to other learning approaches [14].

Classical Machine Learning

Classical ML encompasses a suite of well-established algorithms developed primarily before the deep learning revolution of the 2010s. These models are characterized by their reliance on explicit feature engineering, wherein domain experts define informative attributes from raw data, which are then utilized by relatively shallow models to infer patterns and make predictions [18]. They remain popular in diagnostics applications due to their interpretability, lower data requirements, and well-established theoretical foundations [19]. The discussion is organized by method category, covering linear models, tree-based models, kernel methods, instance-based learning, probabilistic models, clustering techniques,

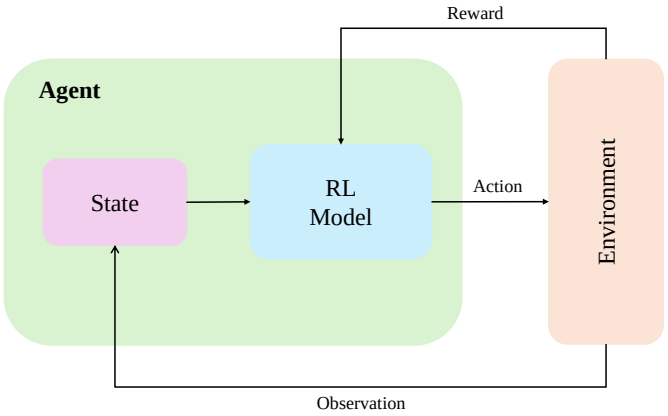


Figure 2.5: Learning framework of an RL model.

Table 2.1: Learning paradigms summary.

Method	Learning Principle	Advantages	Disadvantages
Supervised Learning	Learns from labeled data by mapping inputs to known outputs. Uses explicit feedback.	High accuracy, clear performance evaluation, widely used in classification and regression.	Requires large labeled datasets, which are time-consuming and expensive to collect.
Unsupervised Learning	Identifies hidden patterns in unlabeled data.	No need for labeled data, useful for clustering and anomaly detection.	Results may be difficult to interpret, no ground truth to validate findings.
Semi-Supervised Learning	Small amount of labeled data combined with a large quantity of unlabeled data.	Efficient use of limited labeled data, practical in domains with scarce annotations.	More complex to implement.
Reinforcement Learning	Learns through trial and error by interacting with an environment and receiving reward signals.	Ideal for sequential decision-making, adapts to dynamic environments, does not require labeled data.	Computationally expensive, slow learning process; suffers from delayed feedback.

dimensionality reduction, and gradient boosting. Given the thesis's topic, a contextualization within FDI is also provided when appropriate.

Linear Models Linear models make predictions or classifications using a linear combination of input features. Linear regression models the relationship between input features and a continuous output by fitting a linear function. It is widely used for forecasting and trend analysis. On the other hand, logistic regression extends the linear framework to binary classification (e.g., normal vs. faulty condition). Logistic regression fits a linear decision boundary in the features space by estimating coefficients via maximum likelihood, using a logistic sigmoid to output a class probability [11]. Linear models are valued

for their speed, interpretability, and effectiveness when patterns are approximately linear, making them robust and useful as benchmarks in FDI tasks. They provide clear insight into the contribution of each feature to the prediction process and generally perform well when their underlying assumptions are satisfied. However, their utility is limited when faced with complex or nonlinear patterns, as their accuracy may decline without careful feature engineering. Additionally, while logistic regression must be adapted to handle multiclass problems, linear models overall may be less effective if the real-world data deviates from their assumptions or if signatures are subtle and difficult to capture through linear combinations of features.

Tree-Based Methods Tree-based methods classify or predict outcomes by learning a hierarchy of "if-then" rules that recursively partition the feature space into distinct regions. The fundamental approach is the decision tree, which builds flowchart-like structures to classify data by sequentially splitting it based on thresholds, with the goal of maximizing class purity in the resulting subsets. Figure 2.6 provides an illustrative example of a decision tree. It starts with a root node that encompasses all data samples. From this starting point, the tree recursively splits the data at each internal node (represented in green), based on the value of a selected feature and an associated threshold. Each branch represents a possible outcome of the split, partitioning the data into increasingly homogeneous subsets. This process continues until the data reaches a leaf node, which serves as a terminal node where a final decision or class label is assigned. The resulting flowchart structure provides an interpretable mapping from input features to predicted outcomes, making decision trees both intuitive and transparent. The allowed depth, i.e., number of node layers (e.g., two layers in Figure 2.6), must be carefully set in order to avoid overfitting. To improve performance and avoid overfitting, ensemble tree methods have become a prominent implementation. These consist of a combination of many shallow decision trees. Random forests are an example of such implementation, a model trained on bootstrap-resampled data subsets and feature subsets (i.e., each model in an ensemble is trained on its own randomly generated subset of the data), which then averages their outputs (majority vote for classification). This ensemble approach greatly enhances robustness and accuracy by reducing variance [20, 21]. Tree-based methods, including ensemble approaches, are well-regarded for their interpretability, flexibility in handling both continuous and categorical data, and their capacity to capture nonlinear interactions among features. Ensemble techniques, such as random forests, typically provide high accuracy, robust resistance to overfitting, and intrinsic estimates of feature importance, all while demanding relatively minimal parameter tuning. Despite these strengths, individual decision trees are susceptible to overfitting if not carefully pruned. Besides, ensemble methods, while more accurate, often sacrifice some degree of interpretability and can be computationally intensive. Moreover, tree-based models may be less effective at representing linear combinations of features, and both single trees and ensembles are vulnerable to overfitting in the presence of noise or outliers unless appropriate regularization and cross-validation strategies are employed.

Kernel-Based Methods Kernel methods refer to algorithms that rely on measuring similarities (inner products) between data points in potentially high-dimensional feature spaces. The most prominent example is the support vector machine (SVM). SVMs perform classification by finding the optimal hyperplane that separates data points of different classes with the largest margin. The hyperplane boundary consists of a so-called "support vector", hence the name. Using kernel functions (such as polynomial or radial-basis function kernels), SVMs can efficiently construct a nonlinear decision boundary in the original input

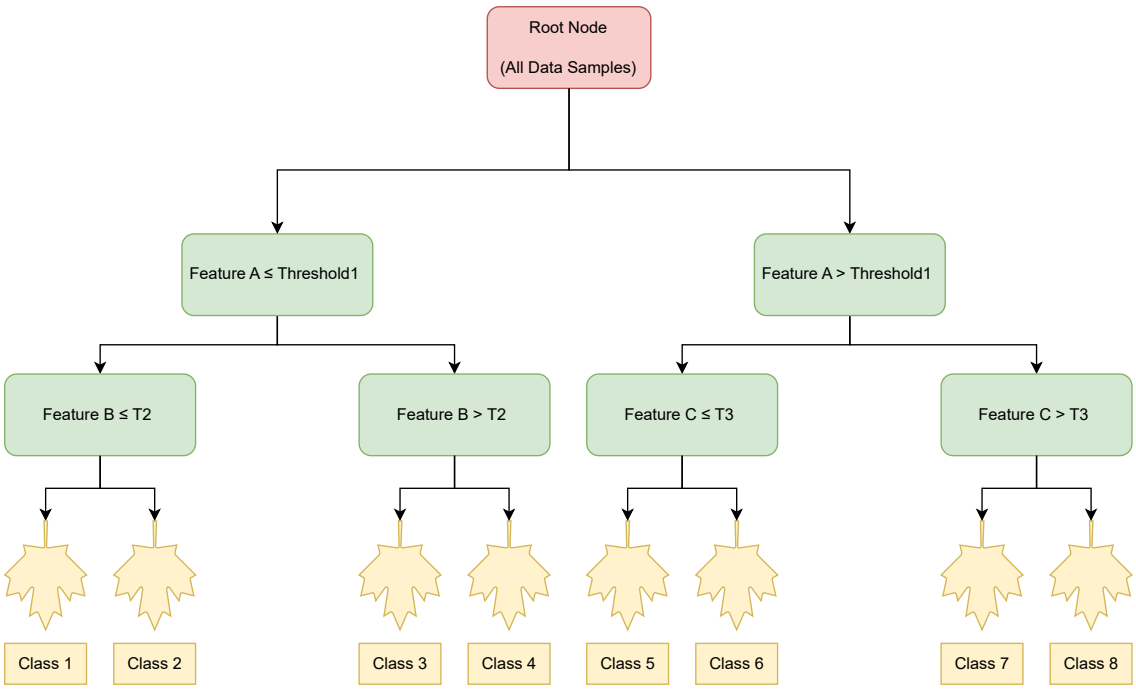


Figure 2.6: Decision tree illustration. T2 and T3 stand for threshold 2 and threshold 3, respectively.

space by operating in an implicit higher-dimensional space. This ability to handle nonlinear separations while avoiding the curse of dimensionality makes SVMs a powerful tool in the context of FDI, where fault symptoms manifest as nonlinear patterns in sensor data [20]. Kernel-based methods offer high accuracy and strong generalization, particularly in high-dimensional feature spaces, by maximizing the margin between classes. Their use of kernel functions allows these models to capture complex, nonlinear relationships, while regularization techniques help to mitigate overfitting, even when only limited data is available. However, these methods can be computationally intensive and require substantial memory when applied to large datasets, and their performance is often sensitive to the choice of kernel and associated parameters. Compared to tree-based approaches, kernel methods generally lack interpretability and, natively, only address binary classification tasks, needing additional strategies for multiclass problems. Furthermore, retraining kernel-based models with new data is resource-intensive, which limits their practicality for online or adaptive learning scenarios.

Instance-Based Learning Instance-based learning methods make predictions for new data by comparing them directly to training instances, rather than learning explicit global models. The quintessential method is k-Nearest Neighbors (kNN). kNN implementations store the historical examples of each class. To classify a new condition, it finds the k most similar past instances (typically via Euclidean distance in feature space) and lets them vote on the label (class). Essentially, the decision boundary is determined by the distribution of the stored samples [20]. Instance-based methods are intuitive and simple to implement, requiring no explicit training phase and adapting easily to new data by updating the stored instance base. They naturally accommodate multiclass classification problems, operate without making distributional assumptions about the data, and are capable of modeling complex decision boundaries when sufficient representative examples and appropriate distance metrics are available. Nonetheless, these methods can

become computationally expensive for large datasets due to the need for a full dataset scan at inference time, and their performance tends to deteriorate in high-dimensional feature spaces. Additionally, their effectiveness is highly contingent on the size and diversity of the dataset, which can limit their scalability and robustness in more complex applications.

Probabilistic Models Probabilistic methods use probability theory to model the relationships between features and the output, computing the likelihood of an outcome given the observed data. These methods are appealing for their ability to handle uncertainty and incorporate prior knowledge, and are especially suited for classification tasks. A basic probabilistic classifier is the Naïve Bayes (NB) classifier. NB applies Bayes' Theorem to compute the probability of a class given the input features, making predictions by selecting the class with the highest posterior probability. Bayes' Theorem is a fundamental principle in probability theory that describes how to update the probability of a hypothesis as more evidence becomes available. Mathematically, it states that the probability of a class C given observed features x is proportional to the product of the prior probability of the class and the likelihood of the observed features given that class:

$$P(C|x) = \frac{P(x|C)P(C)}{P(x)} \tag{2.6}$$

The "naïve" aspect refers to the simplifying assumption that all features are conditionally independent given the class label, which greatly simplifies the computation of the likelihood $P(x|C)$ as the product of the individual feature likelihoods. Despite this strong assumption, Naïve Bayes often performs surprisingly well in practice, especially for high-dimensional data, and is valued for its simplicity, computational efficiency, and ability to handle small datasets. However, feature independence is rarely met in practice, potentially reducing accuracy [22].

Clustering Techniques Clustering algorithms are unsupervised learning methods that group data points based on similarity, without the use of class labels. This approach is used to discover natural groupings in operational data, for instance, to differentiate normal operation clusters from anomalous operation, or to cluster data into distinct categorical groups when labels are not initially known. K-Means clustering is a classic algorithm that partitions data into K clusters by minimizing the intra-cluster variance (assigning each data point to the nearest cluster centroid). For example, in the context of FDI, it has been applied to process monitoring by clustering historical data to establish a normal operation cluster and then flagging points that lie far from this cluster as potential faults. Clustering methods are a valuable proposition for analyzing unlabeled data and uncovering its inherent structure. This is particularly useful when labeled data is scarce or expensive to obtain. However, because clustering is inherently unsupervised, its results require interpretation and may not align with actual classes, often necessitating expert mapping of clusters to real-world conditions. The effectiveness of clustering depends on choices such as the number of clusters and the distance metric, both of which can significantly impact the resulting groupings and may introduce ambiguity or misclassification, particularly in high-dimensional spaces. Furthermore, while clustering can highlight patterns and organize complex data, interpreting these groupings and understanding their physical relevance can be challenging, making clustering most effective as an exploratory or supplementary tool rather than a standalone solution.

Dimensionality Reduction Dimensionality reduction methods serve to transform high-dimensional data into a more compact, lower-dimensional representation that retains the most salient information and are typically employed as unsupervised learning techniques. For example, in the context of FDI, these techniques serve multiple purposes: reducing noise and redundancy in data, visualizing complex data structures, and compressing data to a few features that can be monitored or fed into simpler fault classifiers. The most established technique is principal component analysis (PCA). PCA finds a set of orthogonal principal components (linear combinations of the original variables) that capture the maximum variance in the data. By projecting data onto the top principal components, we obtain a low-dimensional subspace that contains most of the variance in the system. Dimensionality reduction techniques simplify data and models by condensing many variables into a few informative components, improving computational efficiency, reducing noise, and aiding data visualization. However, they may miss important nonlinear or subtle patterns, may be difficult to interpret physically, and introduce extra setting choices and complexity. Thus, careful application and validation are needed to ensure critical information is not lost [20].

Gradient Boosting Gradient boosting belongs to the family of ensemble methods, where multiple models (usually decision trees) are combined to produce a stronger overall predictor than any single model alone. This approach builds a model in a stage-wise fashion by sequentially adding weak learners to correct the errors of the combined ensemble. The term "boosting" refers to the combination of multiple models, while the term "gradient" is used given that the training process relies on gradient descent, with each new model added in a way that most reduces the overall prediction error. A popular implementation within this scope is XGBoost. Overall, gradient boosting offers high inference accuracy and considerable flexibility, making this category effective for modeling complex relationships across a variety of data types and loss functions. However, these approaches are computationally intensive, sensitive to hyperparameter choices, and can be prone to overfitting [23].

Classical machine learning methods play a vital role in a myriad of engineering applications, including FDI. Each category offers a unique balance of theoretical rigor, interpretability, and practical performance. Linear models provide simplicity and transparency, tree ensembles yield powerful nonlinear decision tools with robust performance, and kernel methods like SVM offer excellent accuracy for complex fault patterns. Instance-based approaches exemplify simplicity and adaptability, while probabilistic methods contribute with the ability to reason under uncertainty and integrate expert knowledge. Finally, within the realm of unsupervised learning, clustering and dimensionality reduction techniques enable the interpretation of large, unlabeled datasets by extracting meaningful patterns and simplifying data.

Overall, these learning models are highly valuable for structured datasets, moderate sample sizes, and tasks where interpretability and computational efficiency are important. Their primary limitations lie in their dependence on effective feature engineering and their relative inability to automatically discover abstract representations in complex, high-dimensional data, an area where deep learning models excel.

Neural Networks and Deep Learning

Neural networks (also known as artificial neural networks, ANNs) are a class of machine-learning models that draw inspiration from the human brain's neural circuitry. The basic unit is an artificial "neuron" (or

node) which receives multiple inputs, combines them through a weighted linear operation, and emits a single output signal. Neurons are arranged in consecutive layers: an input layer, one or more hidden layers, and an output layer, forming a directed computational graph. After the linear combination, each neuron applies a nonlinear activation function to its intermediate result before forwarding the transformed value to the next layer [10, 14]. A detailed exposition of neural network operation is provided in section 2.4.

What sets NNs apart is not merely the capacity for nonlinearity itself, but also their ability to automatically and hierarchically learn highly intricate nonlinear features directly from raw data without explicit human guidance.

Autoencoders Autoencoders (AEs) are a special type of neural network designed to learn compact, informative representations of input data by training the network to reconstruct the original input from a compressed latent code. Structurally, an AE comprises an encoder that compresses the input into a latent space and a decoder that reconstructs the input from this representation. In the FDI context, when trained on normal, fault-free data (unsupervised learning), the autoencoder learns to model the nominal system behavior. As a result, it struggles to accurately reconstruct inputs containing anomalous patterns, leading to high reconstruction errors that can serve as an effective anomaly score. In this sense, autoencoders can be viewed as nonlinear extensions of PCA, with the added benefit of modeling complex relationships among inputs. Within fault detection, this unsupervised learning paradigm is particularly advantageous when labeled data is scarce, which is often the case in propulsion testing scenarios. Their ability to generalize and isolate multivariate anomalies makes them highly effective in fault detection frameworks, but requires the effort of defining a robust threshold mechanism for the reconstruction error [10, 24].

FNN Feedforward neural networks (FNNs) constitute the most fundamental and straightforward class of artificial NNs. In this architecture, information flows unidirectionally from the input layer to the output layer through one or more hidden layers, hence the term feedforward. The layers are fully connected, i.e., each neuron in a layer connects to all the neurons in the following layers (refer to section 2.4 for a better understanding of the concept). Crucially, there are no recurrent connections or feedback loops, and as such, no temporal dependencies are modeled intrinsically. The term deep learning is often used to describe neural networks with multiple hidden layers, typically three or more. While some FNNs consist of only one or two hidden layers and therefore do not meet this threshold, deeper variants with several hidden layers do qualify as deep learning models. In such cases, they are capable of learning increasingly abstract representations of input data across successive layers, a property that becomes especially valuable for complex classification tasks. This architectural simplicity renders FNNs particularly suitable for static input-output mapping tasks. Within the context of FDI, an FNN may be employed to classify the system state (e.g., healthy or faulty) based on a snapshot of sensor readings. In practical implementations, fixed-size input vectors are often extracted from sliding windows of time-series sensor data and used to train the model. However, a fundamental limitation of standard FNNs is their inability to capture temporal dependencies across inputs. Each input vector is processed independently, without any form of memory or state retention. As a result, any temporal patterns must be manually encoded into the input, typically through feature engineering or windowing techniques. This introduces trade-offs: if the window is too short, important long-term fault precursors may be missed; if too long, the input dimensionality increases,

potentially complicating training and reducing generalization performance. In essence, while FNNs can effectively model instantaneous relationships between inputs and outputs, they are inherently limited in tasks where the sequence and timing of events are critical. For such applications, models capable of learning temporal dependencies, such as recurrent neural networks, may offer significant advantages [10, 12].

CNN Convolutional neural networks (CNNs) represent a class of neural networks originally developed for visual data processing but have found broader applications in domains involving structured spatial or local dependencies. Their hallmark is the use of convolutional layers, where small learnable filters (or kernels) are applied across the input to detect local features (such as edges, textures, or shapes in an image). In contrast to fully connected layers, convolutional layers exploit spatial locality and parameter sharing, resulting in significantly fewer trainable parameters and improved generalization. This architectural principle is particularly well-suited for image classification, object recognition, and similar tasks where spatial structure carries essential meaning [12].

RNN Recurrent neural networks (RNNs) consist of a deep learning architecture that possesses feedback connections allowing information from previous time steps to influence the network's output at the current time step. In principle, a basic RNN can maintain a hidden state that acts as a memory of past inputs. They introduce recurrent connections that feed the output of a neuron at one time step back into the network as input for the next time step. In effect, an RNN maintains an internal hidden state vector, h_t, that is updated at each time step, t, based on both the new input, x_t, and the previous state, h_{t-1}. For a simple RNN, this can be written as in Equation 2.7. Where W_{xh} and W_{hh} are weight matrices for input-to-hidden and hidden-to-hidden connections (with bias b_h) and η is a nonlinear activation function (such as tanh).

$$h_t = \eta(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2.7}$$

The hidden state, h_t, thus acts as a memory element, allowing the network to remember past inputs. In contrast to an FNN's fixed-size input/output paradigm, an RNN can, in principle, process sequences of arbitrary length. This makes RNNs naturally suited for sequence data like time-series sensor measurements, text, or audio. Indeed, an RNN can learn to capture temporal patterns and long-term dependencies that are essential for analyzing system behavior over time [25]. Figure 2.7 provides an illustration of an unfolded RNN, over consecutive time steps. The left side shows its compact representation, where the hidden state h is updated based on the current input x and the previous hidden state. The right side illustrates the equivalent unfolded view, where each RNN cell is shown at a different time step t. Here, x_t denotes the input at time step t, h_t represents the hidden state, and o_t is the output. U, W, and V denote the parameters associated with the input-to-hidden, hidden-to-output, and hidden-to-hidden connections, respectively.

Nonetheless, training standard RNNs on long sequences is difficult due to vanishing and exploding gradient problems. As gradients propagate across many time steps, they tend to either decay to near-zero or blow up, impeding the learning of long-range dependencies. In practice, this meant vanilla RNNs struggled to remember information from the distant past in a sequence. This limitation motivated more

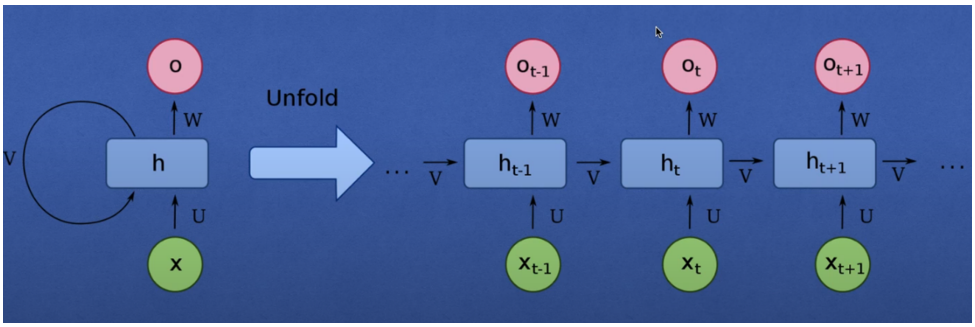


Figure 2.7: Unfolded, layered structure of an RNN.

advanced RNN architectures designed to learn what to remember. The most influential of these is the long short-term memory (LSTM) network, introduced by Hochreiter and Schmidhuber in 1997 [26].

LSTM An LSTM incorporates memory cells with gating units that control the addition and removal of information. The LSTM neuron (commonly referred to as a cell) includes an input gate, forget gate, and output gate that modulate the cell's internal state. Essentially, these gates learn to decide how much of the past to remember and when to update the cell's memory with new information. As a result, LSTMs can capture both short-term and long-term dependencies in sequenced data. Compared to feedforward or even classical RNN approaches, LSTMs have proven superior in many fault diagnosis tasks precisely because they better exploit the temporal information. They mitigate the vanishing gradient problem, enabling training on sequences with hundreds of time steps if needed. Despite their power, LSTMs are not without challenges. They are more complex and computationally expensive than FNNs, with more parameters and a more involved training process, and they can overfit, especially on small datasets, due to their depth [26]. Beyond classification tasks, LSTM networks can be leveraged for predictive modeling and temporal forecasting. In the FDI context, an LSTM can be trained to anticipate the future trajectory of selected sensor variables based on their historical behavior. This forecasting capability enables a shift from mere fault detection toward fault prediction, allowing the system to flag anomalies based on divergences between predicted and observed signals.

2.4 Understanding Neural Networks

The implementations derived through the remainder of the present thesis rely on neural networks, for which this section provides a basic introduction in order to facilitate the understanding of the concept. Artificial NNs consist of many interconnected processing nodes (neurons) organized into layers. Taking a feedforward network (FNN) as a starting example, it consists of an input layer, one or more hidden layers, and an output layer. Each neuron in the input layer receives one feature of the input data (e.g., sensor reading or image pixel) and passes its value to neurons in the next layer (the first hidden layer). Every neuron in a hidden or output layer takes inputs from all neurons of the previous layer, i.e., it is a fully connected layer. The input information is propagated through the network via a sequence of mathematical computations between the neurons. Each connection between neurons carries a trainable parameter called a weight, which scales the signal from the source neuron. In addition to and apart from the input neurons, each neuron also has an associated bias, which is a constant added to the weighted sum

of its inputs. Intuitively, the weight, w, determines how much the neuron in the next layer "listens" to the neuron in the previous layer, while the bias, b, shifts the neuron's overall input up or down, allowing the network to fit data not centered at zero. The weights and biases are the learnable parameters of the network, adjusted during training to minimize errors. Modern deep NN can contain millions or even billions of such parameters [10, 27].

Figure 2.8 illustrates a simple five-layer network. The input layer stems from three individual features and is represented in green. The information is then passed through three hidden layers (blue) and finally converges to a single output in the output layer (red). The parameters are represented with w for the weights and b for the biases. In total, this example contains 39 parameters, of which 30 are weights and 9 are biases.

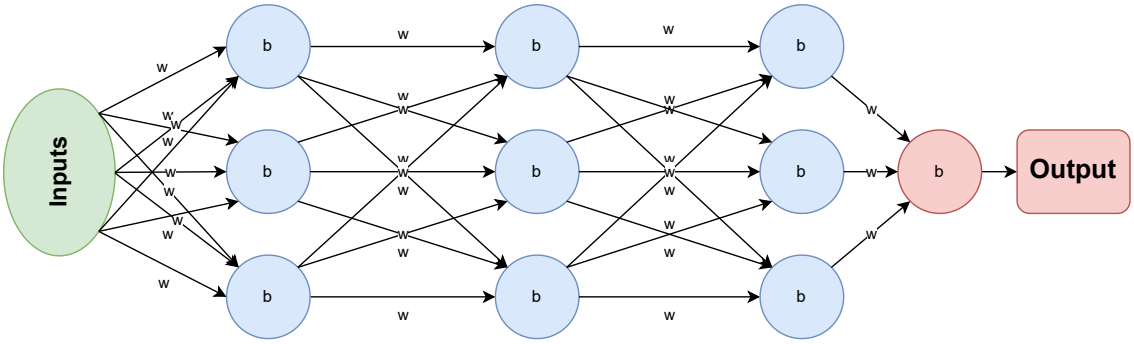


Figure 2.8: Schematic representation of a neural network.

2.4.1 Forward Pass Computation

Neural network layers perform a simple mathematical operation on their inputs. For a layer with input vector, $\mathbf{x}$, output vector, $\mathbf{y}$, weight matrix, W, and bias vector, $\mathbf{b}$, the computation is represented in Equation 2.8. Where σ is an activation function applied element-wise. In words, each neuron computes a weighted sum of its inputs (using the weights and biases) and then applies a nonlinear activation function, σ, to produce its output. The activation function is responsible for introducing nonlinearity. Without it, stacked layers would collapse into an equivalent single linear model. Nonlinear activations enable neural networks to approximate complex, nonlinear relationships in data [10].

$$\mathbf{y} = \sigma\left(W\mathbf{x} + \mathbf{b}\right), \tag{2.8}$$

Several activation functions are commonly employed in neural networks, each exhibiting distinct characteristics and serving different purposes. The most representative and frequently used examples are listed below. Figure 2.9 illustrates these functions.

- **Sigmoid:** Outputs a value between 0 and 1 for any real input (S-shaped curve). Historically popular and is still used for output neurons in binary classification (to produce a probability). However, at extreme inputs, the output saturates, which might limit the neural network's learning process through a process called vanishing gradients. Mathematically: $\sigma(x) = \frac{1}{1+e^{-x}}$.
- **ReLU:** The rectified linear unit has become one of the most popular activation functions in modern

deep networks due to its effectiveness in practice. Outputs 0 for any negative input and linear (identity) for any positive input, i.e., $f(x) = \max(0, x)$. ReLU is simple and computationally efficient, and it mitigates the vanishing gradient problem by providing a gradient of 1 for positive inputs. A drawback is that neurons can sometimes "die" (output zero permanently) if weights update in a way that drives inputs negative, as such neurons stop learning.

- **Leaky ReLU:** A variant of ReLU that allows a small, non-zero slope for negative inputs. Instead of outputting 0 for $x < 0$, a Leaky ReLU outputs a small factor (e.g., 0.01 * x). This ensures that neurons never completely die. Instead, they continue to get a small gradient even when inactive. Leaky ReLU thus helps mitigate the problem of ReLU neurons becoming stuck at zero (a form of vanishing gradient).

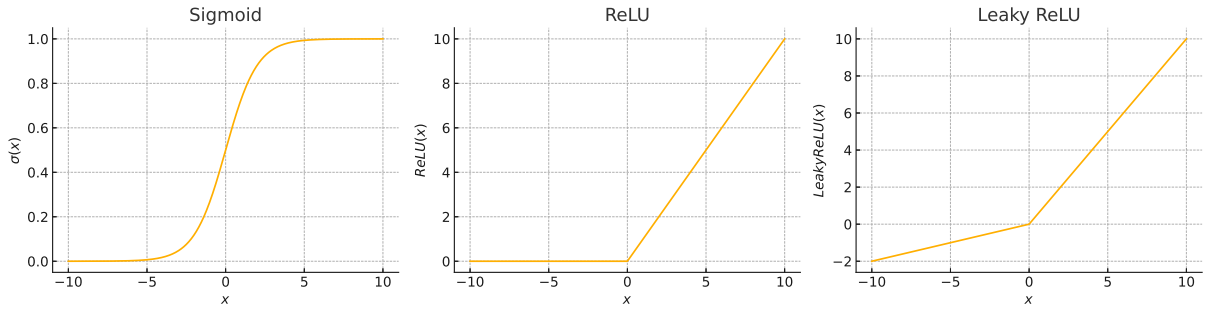


Figure 2.9: Illustration of the sigmoid, ReLU, and Leaky ReLU activation functions. The horizontal axis represents the input x, and the vertical axis shows the output of each activation function.

2.4.2 Learning Process: Loss, Gradient Descent and Backpropagation

Training a NN means adjusting its weights and biases so that the network's outputs match the desired targets for the given inputs. This is achieved through an iterative learning process consisting of: (1) defining a loss function to quantify prediction error, and (2) using gradient descent and backpropagation to minimize that loss by updating parameters.

Loss Function

The loss function quantifies the discrepancy between the network's predicted outputs and the true target values. For multiclass classification problems (i.e., multiple outputs but where only one is correct, such as classifying a dog's breed based on a dog's picture), the standard choice is the categorical cross-entropy loss. This loss compares the predicted probability distribution over all possible classes with the true class label and penalizes the model when a low probability is assigned to the correct class. For a dataset with N samples and C classes, the categorical cross-entropy loss is defined as in Equation 2.9, where $y_{i,c}$ is an indicator variable equal to 1 if sample i belongs to class c (and 0 otherwise), and $\hat{y}_{i,c}$ is the predicted probability that sample i belongs to class c. Selecting an appropriate loss function is fundamental, as it directly shapes the learning process by guiding how the network parameters are updated to improve classification performance [12, 28].

$$L_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log \hat{y}_{i,c} \tag{2.9}$$

Gradient Descent

Neural networks learn by minimizing the loss function. Gradient descent is an algorithm that finds the weight updates that reduce the loss. In essence, it computes the gradient of the loss, L, with respect to each weight, w, and bias, b, and then adjusts each parameter a small amount in the opposite direction of the gradient (hence descending the gradient). For a weight, w, the update can be written as in Equation 2.10 where η is a small constant called the learning rate, i.e., the step size. This update rule means we subtract the gradient, $\partial L/\partial w$, scaled by η. Intuitively, if changing w a little causes L to increase, the gradient is positive and the weight is decreased. If L were to decrease, the gradient is negative, and w is increased. By repeatedly applying such updates across all parameters, the network's predictions should gradually become more accurate, and the loss should decrease. The learning rate, η, controls the size of these weight steps. A higher η speeds up learning but risks overshooting the minimum of the loss function (causing divergence or oscillation), while too low of an η makes training very slow or can get the model stuck in a suboptimal state (e.g., a local minimum) [12].

$$w_{\text{new}} = w_{\text{old}} - \eta\frac{\partial L}{\partial w} \tag{2.10}$$

Backpropagation

In a multi-layer network, determining the gradient of each weight efficiently is non-trivial because each weight indirectly influences the final loss through downstream layers. Backpropagation is the algorithm that computes these gradients systematically for all layers using the chain rule of calculus. It works by propagating the error of the network backward from the output layer to earlier layers. Concretely, after a forward pass computes the outputs and loss, backpropagation computes the partial derivative of L with respect to each weight and bias by moving layer by layer in reverse (from output back to input), applying the chain rule to accumulate gradients. This gives the "direction" each weight should be adjusted to reduce error. The combination of backpropagation to compute gradients and gradient descent to update the parameters is what enables the network to learn from data. Each training iteration (also called a training step) consists of a forward pass, which computes a prediction through inference and the respective loss, followed by backpropagation and a gradient descent weight update. Over many iterations, the network's parameters ideally converge to values that produce low loss on the training data [27].

Figure 2.10 consists of an illustration of the learning process that has been presented and described under this section.

2.4.3 Batch Processing and Training Epochs

Neural network training typically occurs over multiple epochs, where an epoch is defined as a complete pass through the entire training dataset. Updating weights only after traversing the entire dataset (i.e.,

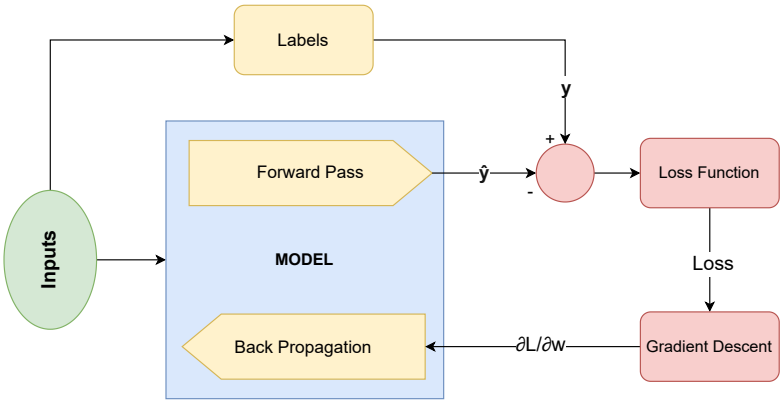


Figure 2.10: Illustration of the learning process for a neural network under supervised learning. $\hat{y}$ is the inference output, y is the ground-truth output, and $\frac{\partial L}{\partial w}$ represents the gradient.

full-batch training) is often inefficient for large datasets and may increase the risk of the algorithm becoming trapped in local minima. To address this, the training data is commonly divided into smaller subsets known as batches. The batch size refers to the number of training samples processed before updating the model parameters. This batching strategy not only improves the statistical properties of gradient estimation but also enables efficient utilization of modern graphics processing units (GPUs) for parallel computation. GPUs, which feature thousands of cores, are particularly well-suited for the simultaneous processing of multiple samples. *NVIDIA*'s CUDA (compute unified device architecture) platform is integral to modern deep learning, as it allows the exploitation of GPU parallelism for computationally intensive operations such as matrix multiplication and backpropagation. This capability has substantially accelerated the training process compared to traditional CPU-based computation. Consequently, contemporary deep learning frameworks, including *TensorFlow* and *PyTorch*, are optimized to leverage CUDA-capable GPUs, thereby enabling high-performance model development and large-scale experimentation for both research and industrial applications [10, 29].

Regarding the training strategy, this typically involves one of three main strategies: batch gradient descent, stochastic gradient descent, or mini-batch gradient descent. Each differs in how much data is used to compute weight updates during learning. Batch gradient descent computes the average loss and corresponding gradients over the entire training set before performing a single weight update per epoch. This results in stable and smooth parameter updates, but requires significant memory and can be inefficient for large datasets. In contrast, stochastic gradient descent updates the weights after computing the loss for every individual sample, which causes the parameters to change rapidly and frequently, often "jumping" around the loss landscape within each epoch. While this can help escape local minima, it introduces high variance and can result in a noisy convergence process. Mini-batch gradient descent, the standard in deep learning, represents a practical compromise by updating the weights after processing small batches of samples. For example, if there are 1 000 training samples and the mini-batch size is 100, then one epoch comprises 10 sequential mini-batches, and the model parameters are updated 10 times per epoch. This approach benefits from the computational efficiency of batch operations and the regularization effect of noise introduced by smaller batches, often resulting in faster and more robust convergence than either of the extreme cases. [12].

2.4.4 Summary

A neural network is a function composed of layers of simple neuron units that perform weighted sum-and-activation computations. Through the forward pass, the network transforms inputs to outputs; through the learning process (driven by a loss function, gradient descent, and backpropagation), it iteratively tunes its weights and biases to improve accuracy. Key hyperparameters such as the learning rate and batch size influence how efficiently and reliably the network converges to a good solution.

3 FDI Applications in Rocket Propulsion

FDI strategies can be broadly classified into three categories:

- Signal-based;
- Model-based;
  - Analytical;
  - Qualitative.
- Data-driven.

Each category employs a different scheme of information sources and methodology to detect anomalies, with their inherent and distinct advantages and limitations. This chapter presents a literature review on FDI applications within the rocket propulsion realm. It surveys the principal detection and identification strategies employed across these categories, highlighting their theoretical underpinnings, operational suitability, and domain-specific challenges. Special emphasis is placed on data-driven methods, particularly those leveraging neural network architectures, as they constitute the methodological foundation of the present work.

3.1 Signal-Based

Signal-based techniques rely directly on processing and analyzing measurable outputs (sensor signals) to infer the presence of faults. Common methods include time-domain analysis and frequency-domain transforms (e.g., Fourier, Wavelet) that, through threshold-based rules, allow for flagging anomalous behavior. Time-domain features consist of mean and standard deviation, whereas frequency-domain analysis aims at identifying frequencies related to faults [4]. Simplicity comes from not requiring an explicit model of the system, instead the signal itself provides the fault indicator (e.g., vibration amplitude, temperature spikes, etc.) [24].

A classic example is the redline algorithm, where a fault is flagged if a signal exceeds predefined limits. Under this approach, the decision algorithm can be either time-independent (threshold-based method) or use historical data to define a nominal, time-dependent, operating window (envelope-based method). Although simple and straightforward to implement, these solutions are heavily dependent on a fixed parameter (the threshold) and can yield poor sensitivity, either triggering false alarms when too strict or missing faults when too lenient. To overcome such disadvantages, adaptive threshold algorithms (ATA) have been developed where the threshold is no longer fixed but adapted given the system's operating conditions. Implementations of the latter described strategy have demonstrated higher sensitivity and fewer false alarms during real-time monitoring [24].

Despite improvements through ATA, signal-based FDI remains inherently limited in its ability to identify the root cause of faults. A pressure drop, for instance, could arise from a sensor issue, a flow restriction, or a leaking valve. Fixed time-dependent redline thresholds often fail to capture evolving system dynamics and are especially inadequate during transient engine phases.

In summary, signal-based FDI, although of low complexity and computationally light, suffers from robustness issues, low identification resolution, and doesn't offer forecasting capabilities. Moreover, the design of such architectures is a highly time-intensive process and may still fail to ensure protection across all fault scenarios.

3.2 Model-Based

Model-based approaches utilize mathematical models that represent the nominal behavior of the system. Faults are inferred by comparing the real-time measurements from the physical system with the expected outputs generated by the model, and analyzing the resulting residuals (deltas). In the context of aerospace propulsion, such techniques are well-established due to their capacity for precise fault identification and their integration into health monitoring frameworks.

Historical research emphasizes the prominence of this method in fault diagnosis for liquid-propellant rocket engines (LRE), where model-based techniques encompass one of the three main approaches, alongside signal-based and AI-driven methods [24].

A classic application dates back to the Space Shuttle Main Engine (SSME), where model-based FDI was employed for real-time detection of actuator faults. A dedicated system utilized observer-based fault detection schemes that could identify actuation faults by estimating physical parameters and comparing them with their nominal values. The use of analytical redundancy, as opposed to physical sensor redundancy, was central to this implementation and allowed for high-fidelity fault detection in real-time. [30]

In another implementation, a model-based fault detection is integrated into an Intelligent Control System for reusable rocket engines. This work focused on identifying frozen valve faults by estimating deviations in valve position based on residual dynamics. The results demonstrated real-time fault accommodation capabilities, highlighting the role of model-based schemes in safety-critical operations [31].

Model-based FDI can be further categorized into analytical and qualitative approaches, each defined by the nature of the model and inference mechanisms employed.

3.2.1 Analytical Model-Based FDI

Analytical model-based methods rely on mathematically derived models that quantitatively describe the system's nominal behavior. These models are constructed from first principles or system identification techniques and are embedded in control-theoretic frameworks. Fault detection is performed by monitoring the residual, which is defined as the discrepancy between the observed output and the model-predicted value [24, 32]. As for residual generation, Frank [33] proposes the following approaches:

- **Parity Space Approach:** Upon system identification, built-in mathematical equations (denominated parity equations) can be derived under which the relationship between input and output

must consistently hold true. Under abnormal circumstances, the result from this parity check is no longer close to zero but increases substantially, signalling a potential fault.

- **Observer-Based Methods:** The residual is generated through an output observer using a digital twin philosophy, where a virtual model runs in parallel with the real system, and the outputs of both are continuously monitored. If this difference stays small, the system is assumed to be healthy. If the difference grows beyond a certain threshold, it means that the real system is behaving differently than expected, possibly due to a fault.
- **Parameter Estimation Methods:** Here, the focus is not on outputs, but on the internal parameters of the system, such as sensor gains, valve coefficients, or motor efficiencies. These methods continuously estimate those parameters in real-time.

  An advantage of parameter estimation is that it provides an explicit quantitative assessment of fault severity. By continuously estimating key system parameters and comparing them to nominal values, these methods allow the size of the deviation to be measured directly in physical terms, thereby offering insight into the magnitude of the fault, something that the former residual approaches do not inherently provide.

  However, a notable limitation is that these techniques require the system to be sufficiently excited during operation, since they rely on identifying how inputs affect system outputs to identify parameter values. Therefore, parameter estimation methods face performance challenges under steady state, where the estimates can become unreliable or ambiguous.

A paradigmatic difference within model-based analytical methods consists of the difference between state estimation and parameter estimation. Whereas observers rely on state estimation, which is inherently fast and has a very short time delay, parameter estimation is slower but, in turn, more suitable for subsequent control reconfiguration [34].

The use of observers in the SSME for real-time fault detection is a prominent example of this class [30]. More recently, in 2019, Tsutsumi et al.[35] developed a model-based FDI algorithm for LRE electromechanical actuator diagnosis, and, in 2020, Ye et al.[36] used an extended Kalman filter to detect thrust anomalies. In 2024, Kurudzija et al.[37] proposed a parity-equation-based analytical model FDI technique for sensor fault detection in the liquid upper stage demonstrator engine (LUMEN), an LRE testbed developed at the Deutsches Zentrum für Luft- und Raumfahrt (DLR), which focuses on advancing technologies for a reusable, storable bipropellant upper-stage engine. The method enabled the identification of sensor faults with high precision, achieving a detection delay of less than one second.

3.2.2 Qualitative Model-Based FDI

Also referred to as soft model-based or knowledge-based, these methods employ symbolic reasoning, logical inference, and heuristic rules rather than explicit mathematical models. They are particularly suitable when the system's physical model is either too complex to formulate or partially unknown. Under modeling uncertainty, where robust analytical models are impractical or unreliable, these methods become particularly useful. Rather than relying solely on continuous numerical output signals, such as precise temperature or pressure values that depend heavily on accurate modelling and data, qualitative methods evaluate a broader set of observable symptoms. These can include events such as logical states

(e.g., “Valve failed to close.”) and time-dependent trends (e.g., “Pressure increasing unusually fast.”). By focusing on such high-level indicators, which are often easier to define and less sensitive to uncertainties in the system model, qualitative methods are able to improve robustness [33].

These techniques include approaches such as fault trees, qualitative simulation, and reasoning based on causal or functional dependencies. Usually deployed in expert systems, they aim at leveraging a priori knowledge, linguistic rules, and historical process behavior to infer faults. In such systems, fault detection and identification are achieved through structured rule-based reasoning, such as "IF–THEN" logic [24, 32, 33].

The Titan Health Assessment Expert System (THAES), developed by *Aerojet*, consisted of an "IF-THEN" logic framework comprised of 90 rules that was employed in Titan rocket engines [32].

Although qualitative approaches lack the numerical precision of their analytical counterparts, they offer notable advantages in interpretability and flexibility, particularly in systems where expert insight is available but detailed analytical models are not. However, a key limitation of these methods lies in knowledge acquisition, the process of formally encoding expert reasoning, process behavior, or heuristic rules into the system. Traditionally reliant on human input, this step has increasingly been augmented by machine learning techniques, which can automatically extract diagnostic patterns and relationships from large volumes of operational data. This shift enables the identification of complex fault signatures that might otherwise be difficult or time-consuming for human experts to define explicitly [32, 33].

3.2.3 Pros and Cons

Model-based FDI remains a cornerstone in rocket engine diagnostics due to its ability not only to detect anomalies and localize their source but also to estimate the magnitude of faults. In particular, parameter estimation techniques offer significant value for fault mitigation, as the updated parameters can be directly employed for model reconfiguration and adaptive control. Additionally, model-based schemes are capable of detecting incipient faults, subtle degradations that may precede critical failures, by identifying small but systematic trends in residuals, often beyond the sensitivity of simple thresholding techniques.

Nonetheless, the effectiveness of model-based FDI is strongly dependent on the fidelity of the system model. Incomplete or inaccurate modeling can introduce residual errors unrelated to actual faults, resulting in missed detections or false alarms. Moreover, these methods tend to lack flexibility as even minor hardware modifications may necessitate a complete redesign or revalidation of the diagnostic model. As a result, while model-based approaches are powerful and have formed the backbone of many FDI systems, their practical applicability is often constrained by the availability of high-quality models and the cost of maintaining them in evolving system architectures.

3.3 Data-Driven

Effective FDI in rocket propulsion systems has evolved beyond traditional model-based and expert-driven methods to incorporate data-driven approaches powered by ML. The inherent complexity and nonlinear dynamics of rocket engines, compounded by extreme operational conditions, pose challenges for constructing accurate analytical models. In response, ML techniques that infer fault patterns directly

from sensor data have gained increasing relevance. This section surveys key contributions from both classical algorithms (such as SVMs) and deep learning architectures (including autoencoders and LSTMs), with a focus on their effectiveness in fault detection within the aerospace propulsion domain [38].

3.3.1 Classical Machine Learning Methods

With the rise of statistical learning, support vector machines became a popular choice in the 1990s for rocket fault detection due to their ability to handle scarce and nonlinear data. Researchers have widely applied SVM variants to rocket engine subsystems like turbopumps. In 2022, Huang et al. [39] proposed a real-time fault detection method for liquid hydrogen/oxygen rocket engines based on a genetic algorithm-optimized least squares support vector regression (GA-LSSVR) model. The approach leverages the genetic algorithm to automatically tune the model's hyperparameters for improved performance. Sensor data from engine tests is normalized prior to training, and the model is validated on experimental data. Results demonstrated that the GA-LSSVR method was capable of effectively detecting faults in real-time. The detection is immediate once the residual begins to drift and the model achieves a FPR of zero, highlighting its potential for practical application in propulsion system health monitoring.

Although classical ML FDI methods are scarcely reported in the context of rocket propulsion, in the context of general machinery fault diagnosis, SVMs and random forest classifiers often achieve high fault classification accuracy (>95%) by aggregating multiple decision trees, suggesting potential for application to rocket engine datasets [38].

3.3.2 Neural Networks

With the increase in computational power and data availability, neural networks and deep learning have gained traction in rocket propulsion FDI, often yielding improved accuracy and earlier fault detection than classical methods. One of the earliest neural network applications in this domain was by Flora et al. [40], proposing a multilayer NN to detect, isolate and even replace faulty sensor signals in a rocket engine, with the model being capable of replacing faulty sensor data with high accuracy, ensuring that the substituted values differ by no more than 0.7% from the true measurement.

Modern feed-forward networks continue this line of work: for example, Yu and Wang [41] designed a backpropagation NN optimized via an adaptive genetic algorithm for real-time engine fault detection. Using historical sensor data from a liquid hydrogen/oxygen engine, the model produced more sensitive and robust fault alarms than a standard network. The improved convergence from the adaptive genetic algorithm yielded higher detection reliability in the simulations.

Given the scarcity of failure examples, unsupervised learning on nominal data has been especially valuable for rocket fault detection. AE networks, which learn a compressed representation of normal engine behavior, are widely used to flag anomalies as reconstruction errors. Zhu et al. [42] developed a convolutional autoencoder (i.e., an AE with convolutional layers), combined with a one-class SVM, to monitor steady-state phases of a liquid rocket engine. This method learns the nominal steady-state sensor patterns and, whenever a fault occurs, the model can no longer reconstruct the sensor data properly and declares an anomaly. This approach proved effective in detecting subtle deviations without requiring any labeled fault samples. In another study, researchers proposed a memory-augmented deep autoencoder to

detect rocket engine anomalies, using an external memory module to better remember nominal dynamics [43]. Such autoencoder-based systems have shown potential in test scenarios, often identifying faults that traditional threshold checks miss.

In 2023, Dresia et al. [44] presented a comprehensive application of machine learning for fault detection in rocket engine test facilities, specifically focusing on the liquid oxygen feed-line system at DLR's P5 test bench. The fault detection model is a semi-supervised FNN trained on 15 000 synthetic test runs (12 million datapoints), generated using a high-fidelity *EcosimPro* model. The network forecasts the values of critical parameters such as tank pressure, pump inlet pressure, and the control valve position, one second into the future, using both current and historical measurements as input. Anomalies are detected by thresholding the mean squared prediction error, calibrated to yield a FPR below 1%, on nominal validation data. The model successfully detects various sensor faults (offsets, drifts, freezes) and system-level anomalies (leakages and altered valve dynamics), achieving a TPR between 0.94 and 1.0 for sensor faults and 0.98 for leakages. However, it performs less reliably for dynamic faults such as increased valve time constants, with a TPR of only 0.21 and detection times between 6 and 8 seconds, which is attributed to the transient-dependent nature of such anomalies. Nonetheless, this forecasting-based approach shows promise in replacing static redline solutions, enabling more adaptive and context-aware fault monitoring.

Building on Dresia et al.'s developments, Lemke [45] presents an unsupervised, reconstruction-based anomaly detection approach for the same DLR's P5 test bench. The proposed method employs a feedforward autoencoder architecture (i.e., an autoencoder with fully connected layers) trained exclusively on nominal time-series data, leveraging, once again, the reconstruction error, quantified by the mean squared error, as the anomaly score. Anomalies are detected by applying both static and adaptive thresholding mechanisms, which are calibrated to maintain a strict FPR of 1%, as demanded by operational safety constraints. In essence, the model is trained to identify and recreate the behavior of the test bench, effectively creating a digital twin of it. This approach demonstrates strong detection capabilities across a range of simulated fault scenarios, including sensor drift, valve misbehavior, and leakages, achieving a maximum F1 score of 0.90. While promising, the study acknowledges limitations in the detection of more complex or subtle fault types, underscoring the need for further refinement before deployment in live test campaigns.

RNNs have further advanced FDI by capturing temporal dynamics of engine operation. LSTM networks, in particular, have been applied to rocket engine start-up and firing data to detect incipient faults. Soon-Young et al. [46] presented a deep learning FDI method for a liquid-propellant engine's startup transient using a combination of LSTM and CNN models. By fusing convolutional layers (to extract spatial features from multiple sensor signals) with LSTM layers (to track time-series trends), this system is able to recognize abnormal startup signatures that unfold in milliseconds. This deep LSTM–CNN approach successfully detects engine faults during the highly nonlinear ignition ramp-up, providing earlier warnings than static threshold limits. More recently, in 2023, Zhang et al. [47] proposed an interpretable LSTM-based diagnosis model for a simulated 200-ton class liquid hydrogen/oxygen engine. The model employs a 1-D CNN to preprocess multi-sensor sequences and a bidirectional LSTM to model temporal patterns. The model achieved a 97.39% fault recognition accuracy on various startup and steady-state fault scenarios, which was the highest among the compared approaches (CNN alone, CNN+SVM, and a standard CNN-LSTM). This demonstrates that deep sequence models can attain very high precision in differentiating fault modes, while also offering some explainability for the decision process.

While data-driven approaches, particularly those leveraging neural network architectures, have demonstrated considerable promise in the detection of faults within rocket propulsion systems, several limitations remain. Chief among these is the lack of comprehensive validation across diverse operational conditions and fault types. Existing studies often focus on a limited subset of predefined anomalies, thereby constraining their generalizability. Moreover, the vast majority of these models are designed solely for fault detection, without addressing the equally critical task of fault identification, which is a crucial step in providing information for mitigation. This gap highlights a pressing need for further research aimed at developing models that not only detect but also reliably classify and interpret a broader spectrum of fault conditions in complex aerospace propulsion environments.

4 Methodology

This chapter outlines the methodology adopted for developing a deep learning-based FDI system for the hopper's propulsion system. It covers the vehicle's architecture, synthetic data generation using a high-fidelity simulation model, preprocessing procedures, and the implementation of FNN and LSTM models. Emphasis is placed on strategies to enhance model robustness, including regularization, class imbalance mitigation, and optimization of hyperparameters through Bayesian search.

4.1 Workflow

The complete methodology workflow adopted for the present thesis is illustrated in Figure 4.1, covering all stages from system modeling to results generation. At the top, the hopper serves as the reference propulsion system, whose physical behavior is simulated using an *EcosimPro* model. This high-fidelity simulation environment enables the generation of synthetic data under both normal and faulty conditions.

Within the data generation block, two distinct fault categories are considered: system faults, which are injected directly into the *EcosimPro* model, and sensor faults, which are introduced during the post-data-generation phase, to simulate realistic measurement anomalies.

The preprocessed data then feeds into the model training stage, where two deep learning architectures, an FNN and an LSTM, are independently trained for multiclass fault classification. The training process is enhanced by means of hyperparameter optimization, initially performed using *Optuna*, to conduct an efficient search across the parameter space, followed by manual refinement based on the most promising configurations.

In parallel, the framework allows for data influence studies, where the impact of different dataset configurations (e.g., complete dataset or faulty-only subset) on model performance is assessed. Trained models are then subjected to performance evaluation, where metrics such as accuracy, F1 score, and fault detection delays are computed to quantitatively assess FDI capabilities.

The pipeline culminates in the results, which consolidate and interpret the performance of each model, providing insights into the relative strengths and offering guidance for future improvements.

4.2 Hopper Architecture

The hopper developed by the Chair of Space Mobility and Propulsion at TUM serves as the reference propulsion platform for the present study. This vehicle is vertically staged and comprises discrete modules dedicated to avionics, payload, and propulsion systems. It uses a bi-liquid pressure-fed engine with liquid

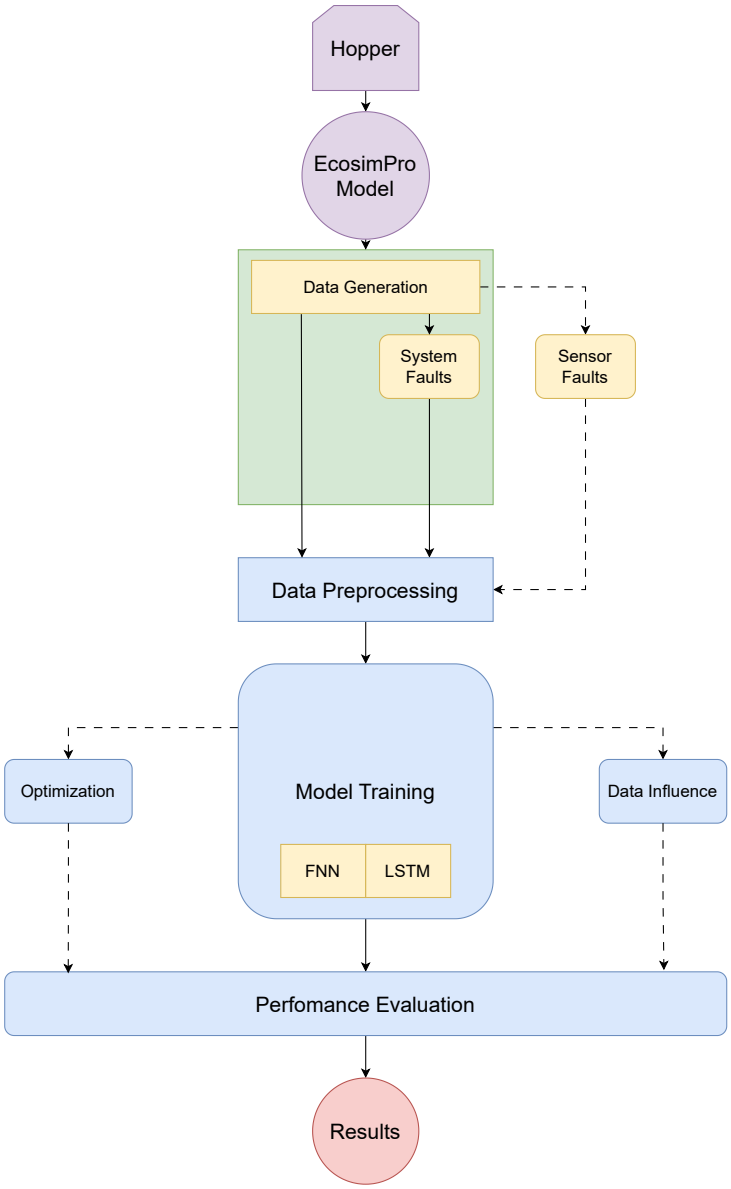


Figure 4.1: Workflow overview.

ethanol as the fuel and liquid oxygen as the oxidizer. When the ethanol is pressurized to 40 bar, and the oxygen to 32 bar, combustion occurs at a chamber pressure of 20 bar, achieving an efficiency of 91.2%.

The thrust-to-weight ratio (T/W) lies between 0.8 and 1.2. The dry mass of the hopper is 147 kg, and its wet mass, including payload, reaches 241 kg. The engine supports a throttling capability of up to 40% of the maximum thrust, achieved with 47% of the maximum chamber pressure. This corresponds to a thrust output range from 1 150 N to 2 835 N, with a maximum mass flow rate limited to 1.38 kg/s.

These specifications constitute the foundation for the *EcosimPro* model used for data generation throughout this work.

4.3 Data Generation

The development of a digital twin of the hopper serves a dual purpose. First, it enables the modeling of a propulsion system for which the physical hardware is not yet available, achieved by relying on design specifications. Second, it provides a flexible and controlled environment capable of generating large volumes of synthetic data, including both normal and fault scenarios. This capability is essential for training deep learning models, particularly given the practical limitations of collecting sufficient real-world data, especially under fault conditions. The digital twin, implemented in *EcosimPro*, thus constitutes a critical component of the proposed methodology, supporting both system representation and data generation.

This section presents a structured overview of the data generation pipeline, detailing the rationale behind the selection of fault types and the development of the *EcosimPro* simulation model, originally implemented by Singh [48], to represent the propulsion system and enable fault injection. The simulation model is executed via a *Python*-based automation script that randomizes trajectory parameters and fault occurrences. While system-level faults are directly injected within the simulation environment, sensor faults are introduced in post-simulation, as these faults are not intrinsic to the system dynamics but instead emulate measurement anomalies.

4.3.1 Fault Selection

For selecting the faults to be simulated within this work, an approach based upon three key criteria was employed: probability of occurrence, reaction feasibility, and simulation capability. Initially, a comprehensive assessment of historical launch vehicle failures was conducted to identify subsystems most susceptible to faults. According to Fernández et al. [2], propulsion-related anomalies accounted for approximately 54% of all observed launch vehicle failures between 2006 and 2022. Within propulsion subsystems, further granularity revealed that approximately 65% of propulsion faults were associated explicitly with the feeding system, pipes, and valves [2]. This quantitative assessment underscores the high likelihood and practical relevance of targeting propulsion subsystem faults, specifically focusing on pipe leakages and blockages, and valve malfunctions.

Additionally, the selection of these specific faults aligns with common practices observed in contemporary rocket propulsion test facilities. Dresia et al. [44] highlight that faults related to the feed-line system and associated components, including pipe leaks and valve failures, are prevalent due to the complex operating conditions and inherent vulnerability of these components to anomalies during operation. The frequent occurrence of these faults under operational conditions further supports their inclusion in the simulation framework.

Sensor faults were also selected due to their critical impact on overall system reliability and operational safety. According to Kurudzija et al. [37], sensor faults exhibit a higher probability of occurrence compared to system faults, significantly influencing rocket propulsion operations by potentially leading to incorrect system responses or mission aborts. Similarly, Dresia et al. [44] emphasize that sensor anomalies, such as malfunctions or erroneous readings, can critically affect operational integrity. Simulating sensor faults, therefore, provides an essential dimension for training robust fault detection and identification models capable of recognizing and mitigating such anomalies.

Considering the practical relevance discussed and simulation feasibility, the faults chosen for simulation include pipe blockages and leakages, valve malfunctions, and sensor anomalies. Pipe-related faults, namely blockages and leakages, are introduced directly via the *EcosimPro* model implementation. The same applies to valve failures, which are represented by increasing their time constant, this being the feasible fault injection method supported by the *EcosimPro* model. Sensor anomalies, in contrast, are simulated through post-processing, emulating bias, drift, and freeze in the readings.

The selected faults are further classified according to the *A-B-C* fault taxonomy defined by Wu [49]. In the context of liquid-propellant rocket engine health monitoring, faults are typically categorized by duration and the urgency required for their detection and response: A-class faults are extremely short-duration events (less than 0.05 seconds), B-class faults have intermediate durations (approximately 0.05 seconds), and C-class faults correspond to long-duration or slowly developing anomalies. Table 4.1 summarizes the fault classification system, providing examples and indicating how the selected faults are categorized. In this context, no faults are excluded on the basis of reaction feasibility, as all selected cases are assumed to permit a minimum reaction time of 50 ms. Indeed, this constraint shall be leveraged in model development, with the identification time considered as a critical metric for assessing real-time application feasibility.

Table 4.1: Fault taxonomy.

Category	Duration	Examples	Selected Faults
A-Class	< 0.05 s	Turbo-pump seizure, rapid ablation or rupture of engine components.	Sensors: freeze and critical measurement (e.g., chamber pressure). Pipe blockage.
B-Class	≈ 0.05 s	Short-duration valve malfunctions, brief pressure spikes, transient sensor dropouts.	Sensors; Valves (increased time constant); Pipe blockage.
C-Class	> 0.05 s	Gradual wear, small leaks, slow sensor drifts.	Sensors: bias and drift; Valves (increased time constant); Pipe leakage; Pipe blockage.

4.3.2 EcosimPro Model and Data Generation Framework

The *EcosimPro* model of the hopper is provided by Singh [48]. *EcosimPro* is a commercial simulation environment widely used for the dynamic modeling and analysis of complex engineering systems, particularly in the aerospace and propulsion domains. It enables the creation of modular and reusable component models using a graphical interface and an object-oriented modeling language. In this context, *EcosimPro* facilitates the simulation of detailed physical processes, subsystem interactions, and fault scenarios, thereby serving as an effective platform for generating representative datasets required for the development and validation of fault detection and identification algorithms.

In Table 4.2, a summary of the available model features and their description is presented.

To facilitate the automatic generation of simulation data based on the *EcosimPro* model, a dedicated *Python* script, `data_gen.py`, was developed. This script interfaces with the model to produce randomized

simulation runs by varying key trajectory parameters:

- T_{asc}**:** thrust limit during the ascent phase, ranging from 2.4 to 2.9 kN, with 80 discretized values.
- Y_{targ}**:** target height, from 1 to 100 m, with 99 values.
- m_{prop}**:** between 66 and 88 kg, with 110 values. Notably, 66 kg constitutes the minimum mass required to achieve the full 100 m trajectory.

These parameter combinations result in a design space comprising 871 200 distinct trajectories. For each configuration, the control inputs are computed using the auxiliary module `TrajectoryCalc_Functions.py`.

The simulation framework supports the generation of both normal and faulty scenarios. In total, five simulation categories are implemented: one normal and four fault-induced cases. Fault injection follows a two-stage approach, where each simulation begins with a normal phase, after which faults are introduced at randomized time instants. This ensures that models are exposed to transitions from normal to faulty conditions, better supporting the training of temporal fault detection algorithms.

Among the simulated fault scenarios, a primary focus is placed on pipe blockages, combined blockage and leakage events, and valve actuator degradation. Pipe blockages are implemented with varying degrees of severity and temporal profiles, designed to capture the diversity of plausible failure modes. Three distinct blockage scenarios are modeled: (i) severe blockage followed by recovery, (ii) severe blockage followed by recovery over an alternative timeframe, and (iii) severe blockage followed by a sustained partial blockage for the remainder of the simulation. Severe blockages correspond to flow area restrictions in the range of 79% to 100%, whereas partial blockages are implemented with obstructions between 40% and 60%, allowing reduced but non-normal flow rates.

In the case of combined blockage and leakage, an initial blockage leads to the development of a downstream leak, simulating progressive system degradation (i.e., *normal* $\rightarrow$ *blockage* $\rightarrow$ *blockage + leak* $\rightarrow$ *leak*).

Valve actuator faults are simulated by increasing the system's response time. Under normal conditions, the valve time constant is set to 0.5 seconds. In fault scenarios, however, this parameter is randomly assigned a value between 3 and 5 seconds, thereby replicating actuator degradation effects that cause significant delays in tracking control commands. As with all fault types, valve faults are injected at randomized times within each simulation. A specific characteristic of this fault mode is that, if the injection occurs during steady-state operation, when no significant valve actuation is required, the fault may remain latent until a control transient occurs. Consequently, although labeled as a fault from the injection time onward, its effects become measurable only once the system departs from steady-state conditions.

In addition to physical system faults, the simulation framework includes a dedicated post-processing stage for the injection of sensor faults. This is performed using the auxiliary module `sensor_label.py`, which modifies normal simulation data to emulate faults such as sensor bias, drift, and freeze. The sensors affected can be configured through the *sensor_faults* dictionary located in the `feature_fault_dict.py` configuration file. The full list of sensors considered in the present work is summarized in Table 4.3.

Table 4.3: Sensors selected for fault injection.

Sensors	*P_ch_1*
	T_ch_1
	CC_oxy_m
	CC_red_m
	PLB_11_Pin
	PLB_12_Pout
	PLB_21_Pin
	PLB_22_Pout
	PLB_11_Tin
	PLB_12_Tout
	PLB_21_Tin
	PLB_22_Tout
	PLB_11_m_in
	PLB_21_m_in
	Valve_1_apos
	Valve_2_apos

All simulation outputs are logged at a fixed sampling frequency of 100 Hz and stored in comma-separated value (CSV) format using scientific notation with a precision of 10 significant digits. The corresponding fault labels are one-hot encoded to ensure compatibility with supervised learning models. The complete framework described above is illustrated in Figure 4.2.

A comprehensive enumeration of all classes used for labeling purposes is presented in Table 4.4. The classification scheme comprises a total of 63 distinct labels, including 14 associated with system-level faults, 48 corresponding to sensor-related anomalies, and a single "Normal" label used to denote normal system operation.

Table 4.4: Classes considered in sample labeling grouped by type.

Class	Type	Description
Normal	Normal	Nominal operation
PLB_11_Block	System Fault	Blockage in oxidizer before valve.
PLB_11_Leak		Leak in oxidizer before valve.
PLB_11_Block_Leak		Blockage + leak in oxidizer before valve.
PLB_12_Block		Blockage in oxidizer after valve.
PLB_12_Leak		Leak in oxidizer after valve.
PLB_12_Block_Leak		Blockage + leak in oxidizer after valve.
PLB_21_Block		Blockage in fuel before valve.
PLB_21_Leak		Leak in fuel before valve.

Continued on next page

Table 4.4 (continued)

Class	Type	Description
PLB_21_Block_Leak	System Fault	Blockage + leak in fuel before valve.
PLB_22_Block		Blockage in fuel after valve.
PLB_22_Leak		Leak in fuel after valve.
PLB_22_Block_Leak		Blockage + leak in fuel after valve.
Valve_1_Status		Slow oxidizer valve.
Valve_2_Status		Slow fuel valve.
P_ch_1_freeze	Sensor Fault	Freeze in combustion-chamber pressure.
P_ch_1_bias		Bias in combustion-chamber pressure.
P_ch_1_drift		Drift in combustion-chamber pressure.
T_ch_1_freeze		Freeze in combustion-chamber temperature.
T_ch_1_bias		Bias in combustion-chamber temperature.
T_ch_1_drift		Drift in combustion-chamber temperature.
CC_oxy_m_freeze		Freeze in oxidizer mass-flow rate.
CC_oxy_m_bias		Bias in oxidizer mass-flow rate.
CC_oxy_m_drift		Drift in oxidizer mass-flow rate.
CC_red_m_freeze		Freeze in fuel mass-flow rate.
CC_red_m_bias		Bias in fuel mass-flow rate.
CC_red_m_drift		Drift in fuel mass-flow rate.
PLB_11_Pin_freeze		Freeze in oxidizer before valve, inlet pressure.
PLB_11_Pin_bias		Bias in oxidizer before valve, inlet pressure.
PLB_11_Pin_drift		Drift in oxidizer before valve, inlet pressure.
PLB_12_Pout_freeze		Freeze in oxidizer after valve, outlet pressure.
PLB_12_Pout_bias		Bias in oxidizer after valve, outlet pressure.
PLB_12_Pout_drift		Drift in oxidizer after valve, outlet pressure.
PLB_21_Pin_freeze		Freeze in fuel before valve, inlet pressure.
PLB_21_Pin_bias		Bias in fuel before valve, inlet pressure.
PLB_21_Pin_drift		Drift in fuel before valve, inlet pressure.
PLB_22_Pout_freeze		Freeze in fuel after valve, outlet pressure.
PLB_22_Pout_bias		Bias in fuel after valve, outlet pressure.
PLB_22_Pout_drift		Drift in fuel after valve, outlet pressure.
PLB_11_Tin_freeze		Freeze in oxidizer before valve, inlet temperature.
PLB_11_Tin_bias		Bias in oxidizer before valve, inlet temperature.
PLB_11_Tin_drift		Drift in oxidizer before valve, inlet temperature.

Continued on next page

Table 4.4 (continued)

Class	Type	Description
PLB_12_Tout_freeze		Freeze in oxidizer after valve, outlet temperature.
PLB_12_Tout_bias		Bias in oxidizer after valve, outlet temperature.
PLB_12_Tout_drift		Drift in oxidizer after valve, outlet temperature.
PLB_21_Tin_freeze		Freeze in fuel before valve, inlet temperature.
PLB_21_Tin_bias		Bias in fuel before valve, inlet temperature.
PLB_21_Tin_drift		Drift in fuel before valve, inlet temperature.
PLB_22_Tout_freeze		Freeze in fuel after valve, outlet temperature.
PLB_22_Tout_bias		Bias in fuel after valve, outlet temperature.
PLB_22_Tout_drift		Drift in fuel after valve, outlet temperature.
PLB_11_m_in_freeze		Freeze in oxidizer before valve, mass-flow rate.
PLB_11_m_in_bias	Sensor Fault	Bias in oxidizer before valve, mass-flow rate.
PLB_11_m_in_drift		Drift in oxidizer before valve, mass-flow rate.
PLB_21_m_in_freeze		Freeze in fuel before valve, mass-flow rate.
PLB_21_m_in_bias		Bias in fuel before valve, mass-flow rate.
PLB_21_m_in_drift		Drift in fuel before valve, mass-flow rate.
Valve_1_apos_freeze		Freeze in oxidizer-valve position.
Valve_1_apos_bias		Bias in oxidizer-valve position.
Valve_1_apos_drift		Drift in oxidizer-valve position.
Valve_2_apos_freeze		Freeze in fuel-valve position.
Valve_2_apos_bias		Bias in fuel-valve position.
Valve_2_apos_drift		Drift in fuel-valve position.

4.3.3 Dataset Description

The construction of the dataset was guided by two primary objectives: ensuring a balanced distribution of simulations across the five data categories and guaranteeing that each class is represented at least twice in the validation set following the training–validation split. To this end, a total of 4 823 simulation runs were performed, distributed as follows: Normal operation (990 simulations), Valve faults (835), Pipe Blockages (823), Combined Blockage and Leak (832), and Sensor Faults (1 343). This corresponds to a total of 12 698 652 samples, which at 100 Hz yields $\approx$ 35 hours of simulation data. Although the initial aim was to produce approximately 800 to 1 000 simulations per category, additional sensor fault simulations were generated to ensure sufficient representation of all 48 sensor-related classes. From a simulation-wise perspective, the dataset is relatively well balanced, with each category accounting for between 17.1% and 27.9% of the total simulations, as detailed in Table 4.5.

Additional details regarding data splitting and preprocessing procedures are presented in section 4.4.

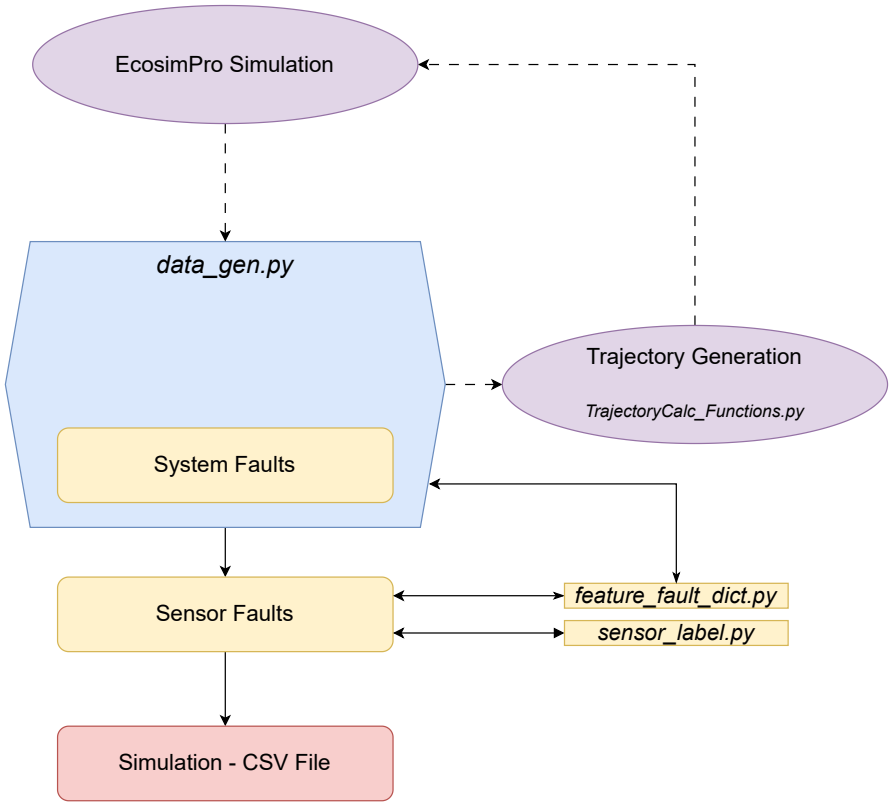


Figure 4.2: Data generation framework.

Table 4.5: Dataset composition and category distribution, simulation-wise.

Category	Simulations	Share [%]
Normal operation	990	20.5
Slow valve	835	17.3
Pipe blockage	823	17.1
Blockage + Leak	832	17.3
Sensor fault	1 343	27.9
Total	**4 823**	**100**

However, when shifting focus from the number of simulations to the distribution of individual data samples, a clear imbalance emerges. After splitting, the training dataset comprises 9 523 989 samples, of which 5 812 292 (61.0%) correspond to normal system operation. This overwhelming proportion of normal data substantially outweighs the combined representation of all fault categories, which together account for only 39.0% of the dataset (3 711 697 samples). As highlighted in Table 4.6, this imbalance can pose a fundamental challenge for supervised learning, as the model may become biased toward detecting normal operation.

Table 4.6: Training dataset composition and category distribution by simulation and sample count.

Category	Samples	Share [%]	Sims. per Class
Normal operation	5 812 292	61.0	Present in all
Slow valve	956 602	10.0	313 ± 12
Pipe blockage	326 183	3.4	305 ± 20
Blockage + Leak	444 979	4.7	131 ± 42
Sensor fault	1 983 933	20.8	20 ± 10
Total	**9 523 989**	**100**	–

Further inspection of the fault data reveals additional nuances. At first glance, sensor faults appear to dominate the fault portion of the dataset, representing 20.8% of all samples, more than four times the contribution of pipe blockage data (4.7%). However, this comparison is misleading when considering the number of individual classes within each fault category. Sensor faults are distributed across 48 distinct classes, each with relatively few examples, typically 20 simulations per fault type, as can be seen in Table 4.6. By contrast, pipe blockages are divided into just four core classes, each represented by over 300 simulations on average.

A summary of the validation dataset distribution by simulation count is provided in Table 4.7.

Table 4.7: Validation dataset composition and category distribution by simulation count.

Category	Simulations	Share[%]	Sims. per Class
Normal operation	1 207	48.1	Present in all
Slow valve	209	8.3	105 ± 2
Pipe blockage	412	16.4	102 ± 6
Blockage + Leak	347	13.8	41 ± 15
Sensor fault	336	13.4	7 ± 5
Total	**2 511**	**100**	–

This discrepancy between category-level and class-level distribution introduces an additional layer of complexity for model training. Although the dataset appears balanced at the category level, the fine-grained class distribution reveals substantial variation in sample frequency, particularly for sensor fault modes. This imbalance reinforces the necessity of incorporating class-aware training strategies and performance metrics that are sensitive to rare class detection, topics that are discussed in section 4.5.1.

4.4 Data Preprocessing

The `preprocessing.py` script constitutes the first stage of the data preprocessing pipeline and is responsible for converting the raw simulation outputs, originally stored CSV files, into *NumPy* (NPY) binary format (`.npy`). This conversion is motivated by the need to reduce both the spatial and temporal complexity of the dataset, thereby enabling efficient model training under constrained computational resources. Specifically, the transformation significantly reduces memory consumption by leveraging compact data

storage formats: input features are stored as 64-bit floating point numbers, which offer a precision of approximately 16 significant digits, sufficient for the recorded sensor data, which contains no more than 10. Labels, which are one-hot encoded, are cast to 8-bit integers, reducing their memory footprint to one byte per label entry. While 32-bit floating point (`fp32`) representations would further halve the memory requirements, their 7-digit precision is insufficient for preserving the necessary signal resolution.

This optimization is critical given the dataset's scale. The training data comprises over 9 million time-stamped samples, and the initial preprocessed dataset, prior to format conversion, exceeded 200 GB in size once in training format, far surpassing the available 64 GB of system random access memory (RAM). The `.npy` conversion reduced the dataset to approximately one-quarter of its original size, shrinking from 9.3 GB in CSV format to just 2.5 GB, thus enabling feasible in-memory operations.

In addition to memory savings, the use of `.npy` files substantially improves data loading performance. NPY's binary format allows direct memory-mapped access and avoids the parsing overhead associated with CSV files. Empirical benchmarks report that reading data from `.npy` files is up to 250 times faster than from CSV counterparts [50]. Given that training routines involve frequent random access to individual samples, this improvement in data access latency translates directly to faster training epochs and reduced computational overhead.

Following the initial conversion of raw CSV data into compact `.npy` format, subsequent preprocessing steps are carried out using the modules `npy_processing.py` and `sliding_window.py`, both of which are part of the `Preprocessing_Pkg` package.

The `npy_preprocessing.py` module defines the *NPYDataset* class, which serves as the central component for structuring and preparing the raw data. Its primary function is to partition the available simulation data into training and validation datasets according to a 75% - 25% split. This is accomplished by randomly selecting an equal number of simulations from each category defined in the *data_types* list (e.g., `['Normal', 'Valve', 'Block', 'Block_Leak', 'Sensor_Fault']`). After partitioning, the module verifies that both subsets include examples of all label classes listed in Table 4.4, thereby ensuring sufficient class coverage for supervised learning.

To facilitate high-throughput access during model training, the contents of the partitioned datasets are loaded and cached into memory. The class further computes global feature statistics, specifically, the mean, standard deviation, minimum, and maximum, exclusively from the training dataset. These statistics are later used to normalize input features on demand during window generation by the *SlidingWindow* class, whose description follows below. Additionally, the module includes functionality for quantifying class distributions across both the number of files and the number of individual samples for each class. To address class imbalance, it calculates class weights based on the inverse of the number of instances per class. In this approach, classes with fewer instances are assigned higher weights, thereby increasing their influence during model training. If enabled (through the *normalize_weights* flag), these weights are subsequently normalized by dividing each weight by the sum of all class weights. This normalization ensures that the weights collectively sum to one, preserving their relative proportions while maintaining a consistent overall scale. By incorporating these weights into the loss function, the model is encouraged to treat all classes more equitably, which is particularly important when dealing with imbalanced datasets.

The final stage of the preprocessing pipeline is handled by the `sliding_window.py` module, also within `Preprocessing_Pkg`. This module implements the *SlidingWindow* class, which dynamically generates

fixed-length temporal segments (sliding windows) from the cached data. The length of said temporal segment can be chosen with the *window_size* setting. Each sliding window is computed independently within a single simulation file to preserve temporal boundaries, and the assigned label corresponds to that of the final sample of the window, thereby maintaining causal consistency in the training data. The class supports both *zscore* and *min–max* normalization schemes (chosen with the *scaler_type* setting), which are essential for standardizing input features prior to model training. In *zscore* normalization, each feature is rescaled by subtracting its global mean and dividing by its global standard deviation. This transformation centers the data around zero and scales it to have unit variance, thereby ensuring that features with larger absolute values do not dominate the learning process. In contrast, *min–max* normalization rescales each feature to a fixed range, typically between zero and one, by subtracting the global minimum and dividing by the range (i.e., the difference between the maximum and minimum). In both cases, the normalization relies on global statistics computed from the training set to ensure consistency and prevent data leakage.

The configuration setting *flatten_window* provides flexibility for adapting the input structure to different model architectures. In particular, it enables the transformation of 2D windowed input arrays into 1D vectors, which is required for FNNs, while preserving the 2D structure for recurrent models such as an LSTM, where the temporal order of observations must be maintained.

To further improve computational efficiency during training, where sliding windows are sampled at random, the class internally maps global sample indices to the corresponding simulation files using precomputed cumulative indices. This mapping is resolved via binary search, significantly reducing the temporal complexity of the data access operation. As a result, the *SlidingWindow* class supports high-throughput batch sampling and efficient integration with *PyTorch*'s data loading utilities, ensuring both scalability and responsiveness during model training.

The full preprocessing workflow is illustrated in Figure 4.3. Together, the described implementations enable a modular, reproducible, and computationally efficient preprocessing pipeline, ensuring that the resulting datasets are well-structured and ready for neural network training.

4.5 Model Implementation

The implementation of the neural network models is conducted using the *PyTorch* [51] deep learning library. *PyTorch* is an open-source framework developed by *Meta AI* and widely adopted in both academia and industry for deep learning applications. Its popularity stems from its intuitive interface and *Python*-native integration, which provides enhanced flexibility for research workflows and rapid model prototyping.

Another widely used deep learning framework is *TensorFlow* [52]. Developed by *Google*, this framework is known for its robust production capabilities and scalability, but is based on a static graph computation paradigm that can complicate debugging and iterative model design.

The decision to adopt *PyTorch* is guided not only by its technical advantages but also by its growing dominance within the machine learning research community. Recent analyses indicate that *PyTorch* is currently the preferred framework for academic research, with over 75% of new deep learning publications

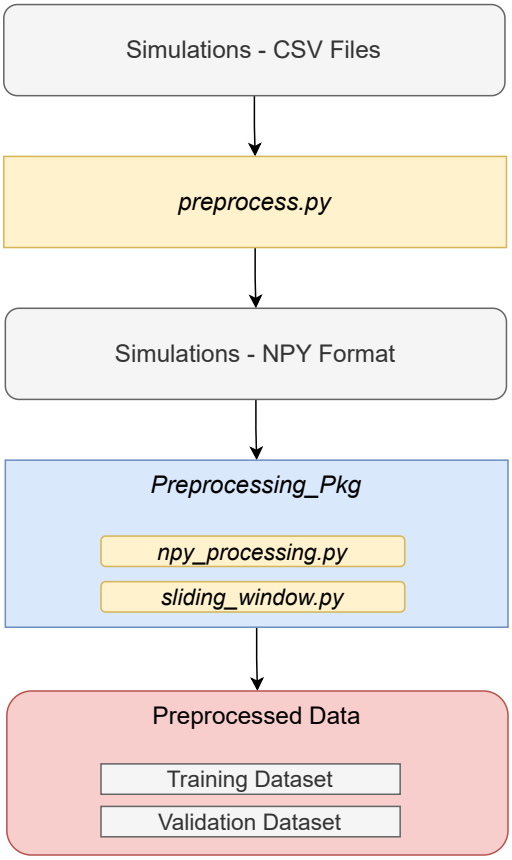


Figure 4.3: Data preprocessing framework.

reportedly using it [53]. This trend highlights a broader shift toward frameworks that emphasize flexibility and user-centered design, particularly in research-driven settings such as the one pursued in this thesis.

Model training is performed on an *NVIDIA* GPU using CUDA acceleration, which enables parallelized computation and significantly reduces training time compared to CPU-based execution. Leveraging *PyTorch*'s native support for CUDA devices facilitates efficient matrix operations and large-batch processing, which are critical for the rapid development and tuning of deep learning models.

4.5.1 Building a Resilient NN

In this section, the architectural and methodological choices aimed at enhancing model robustness in the context of FDI are presented. The discussion is structured around three key areas: *Preventing Overfitting*, *Counteracting Data Imbalance*, and *Improving Model Stability*. Each topic addresses a specific challenge encountered during the training of neural networks and outlines the strategies adopted to ensure reliable generalization and resilient performance under real-world conditions.

Preventing Overfitting

To enhance the generalization capabilities of pursued implementations, three complementary techniques are integrated into the training procedure: L2 regularization, dropout layers, and early stopping. Each strategy addresses overfitting from a distinct perspective, constraining model complexity, introducing stochasticity during training, and adaptively limiting training duration based on validation performance.

L2 Regularization (Weight Decay) L2 regularization penalizes large weight magnitudes by augmenting the loss function with a term proportional to the squared norm of the model parameters. The regularized loss takes the form:

$$L_{\text{reg}} \;=\; L_{\text{orig}} \;+\; \lambda \sum_{i} w_i^2, \tag{4.1}$$

where λ is the regularization coefficient, w_i are the model weights, L_{orig} is the original loss (e.g., cross-entropy) and L_{reg} the regularized loss. This formulation implicitly discourages overly complex models by promoting smaller weight values, resulting in smoother and more generalizable mappings. In practice, L2 regularization is implemented via the *decay* hyperparameter in *PyTorch*. Typical values of λ explored during training range from 10^{-5} to 10^{-3} [54].

Dropout Layers Dropout is a stochastic regularization technique that mitigates overfitting by randomly dropping out a fraction of neurons during training. Srivastava et al. (2014) [55] introduced dropout as "randomly dropping units (along with their connections) from the neural network during training". By removing neurons at random on each training batch, the network is prevented from co-adapting too strongly to particular features. In other words, each neuron cannot rely on the presence of specific other neurons, forcing the model to learn redundant, distributed representations that generalize better. This process can be seen as sampling from an ensemble of many smaller networks during training and implicitly averaging them at inference time, which reduces overfitting compared to a single large network [55].

During training, each dropout layer randomly deactivates a fraction p of neuron outputs, while at inference, all activations are used and scaled by $(1 - p)$ to preserve output consistency. In the implemented architecture, dropout is applied to fully connected layers with probabilities ranging from 0.2 to 0.5. This hyperparameter takes the name of *drop_prob*, in the provided implementation.

Early Stopping To prevent overfitting and enhance generalization, an early stopping mechanism is incorporated during training. This technique halts the training process once validation performance ceases to improve, thereby avoiding unnecessary epochs that may lead to overfitting and degradation in inference quality. The implementation adopted in this work, contained in the `early_stop.py` module of the `Custom_NN_Pkg` package, monitors a user-defined validation metric (e.g., loss or accuracy) and terminates training when no substantial improvement is observed within a specified patience window. The settings of the implementation are as follows: *stop_patience* (int) defines the number of epochs to wait for improvement; *stop_min_delta* (float) specifies the minimum relative change required to count as an improvement; and *stop_mode* (str) sets whether lower (`'min'`) or higher (`'max'`) values indicate better performance. The employed configuration specifically monitors the validation loss and stops training if

the loss fails to decrease over a defined number of consecutive epochs. This strategy provides a simple yet effective form of regularization that preserves the model's ability to generalize without modifying its architecture.

Counteracting Data Imbalance

Machine learning datasets often exhibit class imbalance, where certain classes have far more samples than others. In neural network training, a severe imbalance can skew the model's learning towards the majority class, resulting in biased predictions favoring that class. The model tends to minimize overall error by mostly predicting the prevalent class, which causes minority classes (with few samples) to be misclassified or ignored. For example, in the dataset used for this work, the *Normal* operating class appears approximately 140 times more frequently than the *PLB_12_Leak* fault class. This illustrates the need for techniques to counteract data imbalance so that minority fault classes can be learned effectively and the model does not simply default to the majority class.

Weighted Cross-Entropy Loss One straightforward strategy to address class imbalance is to use weighted cross-entropy loss, which modifies the standard cross-entropy by assigning larger weights to errors on minority classes. In essence, misclassifying a rare class incurs a higher penalty than misclassifying a common class. In the present work, two weighting scheme options are provided:

- **Inverse-frequency weighting:** Each class is weighted in inverse proportion to its occurrence in the training set, a process already explained in section 4.4. This is known as a balanced cross-entropy approach, since classes with fewer samples get a larger weight [56]. By amplifying the loss contribution of the rare fault classes, the network is encouraged to learn those minority patterns despite the class imbalance.
- **Fixed weighting (fault vs. normal):** As an alternative heuristic, a fixed weighting scheme is used where fault classes are simply given a higher weight than the normal class. Under this scheme, misclassification of any fault class is penalized at double the rate of the normal class (i.e., weight of 2 for fault classes vs. 1 for normal). This approach uniformly boosts the importance of all fault classes, on the assumption that detecting any fault is more critical than an error on a normal sample. The boolean setting *use_manual_weights* provides this functionality.

Focal Loss While class weights address imbalance at the class level, they do not distinguish between easy and hard examples within a class. Focal loss is a technique designed to focus learning on the harder, misclassified examples and reduce the influence of those that are already well-classified, doing it so dynamically over the training epochs. Originally developed for object detection with extreme class imbalance, focal loss reshapes the cross-entropy loss by adding a modulating factor that down-weights easy examples. The focal loss for a single example is defined as in Equation 4.2, where p_t is the predicted probability of the true class for that sample and γ is the focusing parameter (set by the *gamma_fl* hyperparameter). Intuitively, p_t close to 1 means the sample is correctly and confidently classified; in that case, the factor $(1 - p_t)^\gamma$ is near zero, thus down-scaling the loss. This mechanism significantly reduces the loss contribution from easy, well-classified instances and emphasizes those samples the model is missing. Setting $\gamma > 0$ thus "focuses" training on misclassified examples, as γ increases, the effect is

more aggressive in lowering the loss for correctly predicted samples, so the model concentrates on the harder cases. In practice, the focusing parameter γ is typically set in a moderate range (often $\gamma \approx 1$ to 2) to effectively focus on difficult examples without making training unstable. Note that $\gamma = 0$ yields the ordinary cross-entropy loss as a special case (no focusing effect). [57]

$$L_{\mathrm{FL}} = -(1 - p_t)^{\gamma} \log(p_t) \tag{4.2}$$

Macro-Averaging of Metrics In addition to modifying the loss function, it is important to use evaluation metrics that reflect performance on all classes fairly. For imbalanced classification, macro-averaging is adopted for key metrics (such as precision, recall, and F1 score). Macro-averaging means computing the metric for each class independently and then taking an unweighted average across all classes. This approach treats all classes equally, regardless of their support (number of samples). In the context of this work, macro-averaged metrics ensure that the performance on rare fault classes is given the same importance across the board when reporting overall model effectiveness.

To further clarify the concept, a simplified example involving a three-class confusion matrix is provided. In this scenario, the "Normal" class overwhelmingly dominates the data distribution, with 95 true positives and 5 false positives, whereas the minority classes "Fault 1" and "Fault 2" are underrepresented, yielding 6 true positives and 6 false positives, and 4 true positives and 2 false positives, respectively. The class-wise precision for each class:

$$P_c = \frac{\mathrm{TP}_c}{\mathrm{TP}_c + \mathrm{FP}_c}, \tag{4.3}$$

which results in

$$P_{\mathrm{Normal}} = \frac{95}{95+5} = 0.95, \quad P_{\mathrm{Fault\,1}} = \frac{6}{6+6} = 0.50, \quad P_{\mathrm{Fault\,2}} = \frac{4}{4+2} = 0.67. \tag{4.4}$$

The macro-averaged precision treats all classes equally, regardless of their frequency, and is computed as

$$P_{\mathrm{macro}} = \frac{1}{3}(0.95 + 0.50 + 0.67) = 0.71, \tag{4.5}$$

indicating that the lower precision scores of the minority classes reduce the overall metric.

In contrast, the micro-averaged precision aggregates all true and false positives across classes before computing the precision:

$$\mathrm{TP}_{\mathrm{tot}} = 95 + 6 + 4 = 105, \quad \mathrm{FP}_{\mathrm{tot}} = 5 + 6 + 2 = 13, \tag{4.6}$$

leading to

$$P_{\mathrm{micro}} = \frac{105}{105+13} = 0.89. \tag{4.7}$$

Due to the overwhelming number of true positives in the "Normal" class, the micro-averaged precision reflects a deceptively high performance. This discrepancy underscores the importance of using macro-averaged metrics in imbalanced fault classification tasks, as they offer a more representative assessment

of classifier performance across all classes.

Improving Model Stability

Ensuring stability during neural network training is critical to achieving both convergence and generalization. Instabilities, manifested as oscillating loss curves or degraded performance, arise from poorly conditioned gradients or erratic parameter updates. This section discusses strategies implemented to improve training stability. The focus lies on layer and batch normalization and mini-batch size.

Layer and Batch Normalization Layer normalization standardizes the summed inputs to a neuron across the feature dimension of each individual sample, thereby stabilizing the distribution of layer activations independently. This approach is particularly well-suited for recurrent architectures such as the proposed LSTM, and empirical studies confirm that layer normalization significantly improves convergence and mitigates vanishing or exploding gradients in LSTM-based models [58]. In contrast, batch normalization computes statistics across the mini-batch dimension and is therefore more appropriate for an FNN, where it effectively accelerates convergence and regularizes training [59].

It is important to distinguish between these forms of internal normalization and the external data normalization applied during preprocessing. The latter standardizes input features globally and while data normalization ensures numerical stability and consistent feature scaling at the input level, layer and batch normalization operate within the model to stabilize the intermediate representations and gradients.

Mini-batch Size The selection of mini-batch size exerts a non-negligible influence on training dynamics. Smaller mini-batch sizes introduce stochasticity in gradient updates, which can help escape local minima but may result in noisy convergence. Conversely, larger mini-batches yield smoother gradient estimates, contributing to more stable updates at the expense of potentially slower generalization [60]. In this work, batch size is treated as a tunable hyperparameter (*batch_size*) and is adjusted to balance memory constraints and the stability of training dynamics. To ensure efficient utilization of GPU resources and reduce overall training time, a mini-batch size of 512 is adopted.

4.5.2 FNN

To establish a baseline for supervised classification, a Feedforward Neural Network (FNN) architecture is implemented. The motivation for this choice lies in the simplicity, transparency, and ease of interpretability associated with FNNs, qualities that make them well-suited for initial experimentation and benchmarking in structured data scenarios, serving as a robust reference point before evaluating more complex temporal models such as LSTMs. Operating on fixed-length feature vectors, the FNN performs multiclass classification over the defined label space. Architecturally, it comprises four fully connected layers (thus falling under the category of deep learning architectures), with dropout and batch normalization applied to enhance generalization and improve training stability.

The network accepts as input a vector of dimensionality *input_dim*, corresponding to the number of preprocessed features. It comprises three hidden layers, each with d_{hidden} number of neurons (configured via the hyperparameter *hidden_dim*), and concludes with an output layer of size *output_dim* representing

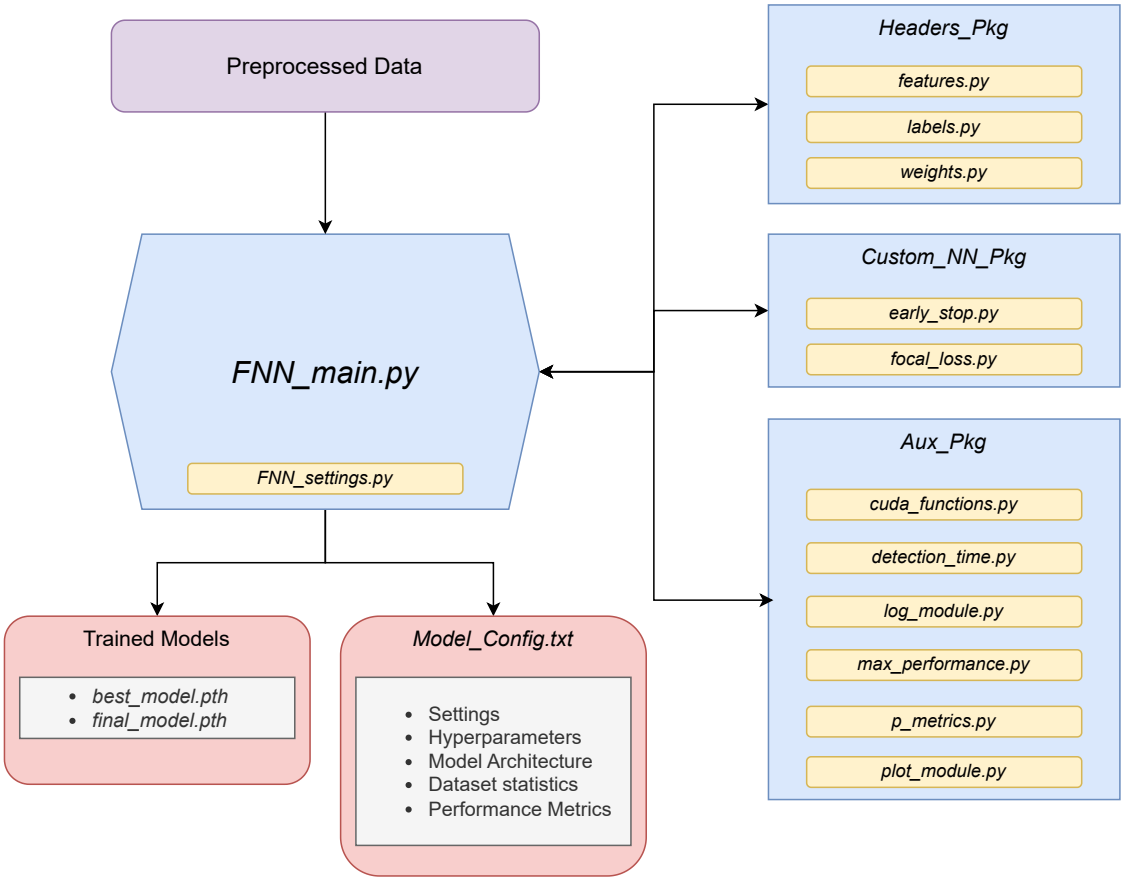


Figure 4.4: FNN framework.

the number of possible classes (Table 4.4). Each hidden layer consists of a linear transformation followed by batch normalization, a Leaky ReLU activation function, and a dropout layer. This combination is intended to mitigate vanishing gradient issues, prevent overfitting, and accelerate convergence. The final layer is a linear transformation projecting to the output label space.

A schematic of the architecture follows below:

- **Input layer:** Fully connected linear layer with input dimension d_{in}
- **Hidden layer 1:** Linear → BatchNorm → Leaky ReLU → Dropout
- **Hidden layer 2:** Linear → BatchNorm → Leaky ReLU → Dropout
- **Hidden layer 3:** Linear → BatchNorm → Leaky ReLU → Dropout
- **Output layer:** Linear transformation to output dimension d_{out}

Formally, the network can be expressed as:

$$y = \texttt{fc4} \circ \texttt{Dropout} \circ \texttt{LeakyReLU} \circ \texttt{BatchNorm} \circ \texttt{fc3} \circ \cdots \circ \texttt{fc1}(x) \tag{4.8}$$

where $x \in \mathbb{R}^{input\_dim}$ is the input feature vector, `fc*` represent the fully connected layers, and $y \in$

$\mathbb{R}^{output\_dim}$ is the unnormalized logit vector output by the network (i.e., raw scores before being converted to probabilities and, finally, the predicted class).

Dropout regularization with probability p is applied after each hidden layer to improve generalization. Leaky ReLU is preferred over the conventional ReLU to prevent vanishing gradients in early training stages. The final output is passed to a *softmax* (converts logits into a probability distribution over the multiple classes) function during training for use with a classification loss. At inference time, the raw logits are converted into predicted class labels by selecting the index of the highest value in the output vector using an `argmax` operation, effectively assigning each input sample to the class with the highest predicted confidence.

Two different loss functions are supported in the script: the conventional cross-entropy loss CEL and the focal loss, with the choice between them determined by the boolean flag *use_focal_loss*. When using CEL, class weighting can optionally be applied to mitigate class imbalance, controlled by the *apply_class_weights* setting. This design allows for flexibility in handling a variety of imbalance scenarios during training.

Model training is conducted over a fixed number of epochs (defined by the hyperparameter *num_epochs*), during which network parameters are iteratively updated using backpropagation and the *Adam* optimizer. The initial learning rate is controlled via the *learning_rate* hyperparameter. To improve training stability and convergence, a learning rate scheduler is employed, which automatically reduces the learning rate when the monitored performance metric, typically the validation loss, ceases to improve. This dynamic adjustment helps the optimizer escape shallow local minima and converge more effectively as training progresses. The behavior of the scheduler is controlled by three main settings. The *scheduler_mode* determines whether improvement is interpreted as a decrease (e.g., for loss functions) or an increase (e.g., for metrics such as accuracy). The *scheduler_patience* defines the number of consecutive epochs without significant improvement before the learning rate is adjusted. Finally, the *scheduler_factor* specifies the multiplicative factor by which the learning rate is reduced once the scheduler is triggered. These settings allow for adaptive control of the training process, reducing the likelihood of overshooting the optimal parameter region and improving overall generalization.

At the end of each epoch, model validation is conducted using `model.eval()`, which disables stochastic elements such as dropout, fixes batch normalization statistics, and halts parameter updates. During validation, gradient computation is also disabled via `torch.no_grad()`, which reduces memory consumption and computational overhead. Performance metrics, including validation loss and macro-averaged F1 score, are tracked throughout training. The model state achieving the best validation performance is saved, and early stopping is applied to halt training once the convergence criteria are met. This comprehensive training procedure ensures both efficiency and robustness in learning from an imbalanced multiclass dataset.

Figure 4.4 illustrates the complete workflow implemented for training and evaluating the FNN model. The central module `FNN_main.py` orchestrates the training process, interfacing with preprocessed data, model configuration settings, and auxiliary components. It retrieves dataset-specific headers from the `Headers_Pkg`, which define the desired features, labels, and manual weights. Loss function selection and early stopping logic are encapsulated within the `Custom_NN_Pkg`. Meanwhile, the `Aux_Pkg` provides supporting utilities for logging, performance tracking, plotting, CUDA setup, and fault detection timing. The training outputs include the best and final (last) models, along with a structured `Model_Config.txt`

file containing all metadata necessary for result reproducibility and summarized performance metrics. A list of the main available hyperparameters and settings (configured in `FNN_settings.py`) is provided in Table 4.8, clarification on the *filter_normal* setting is provided in section 4.6.

4.5.3 LSTM

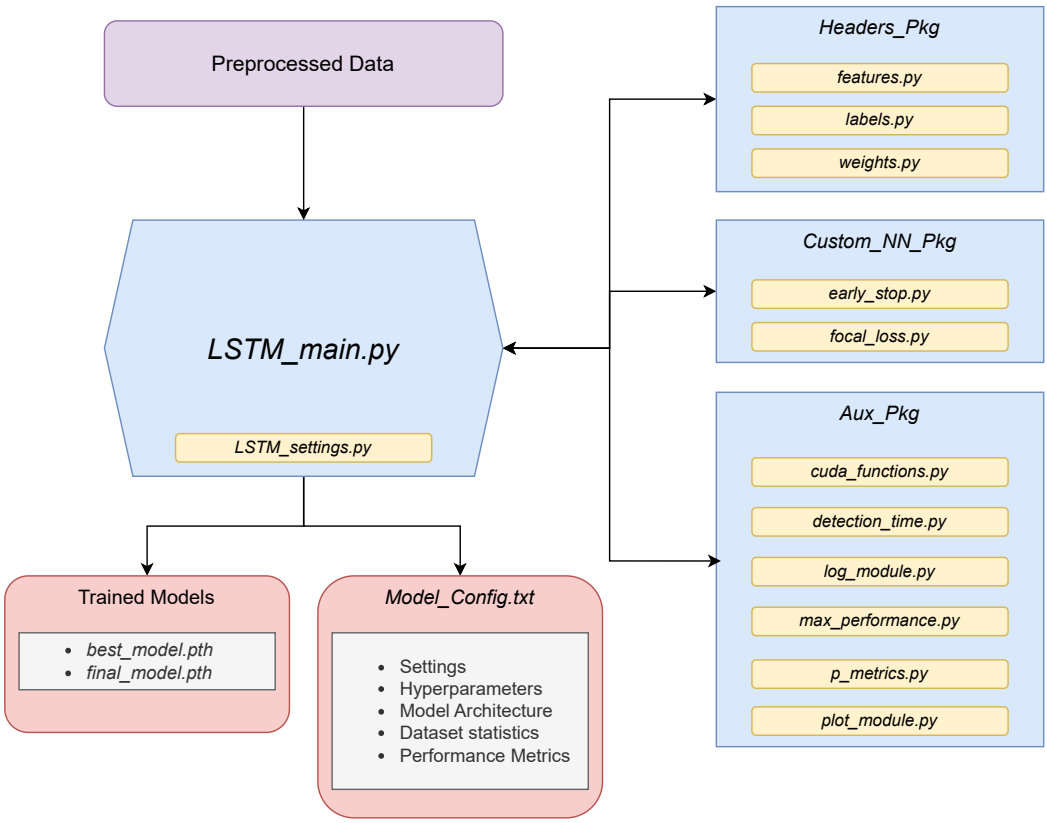


Figure 4.5: LSTM framework.

A Long Short-Term Memory (LSTM) architecture is implemented to address the limitations of the FNN in capturing the temporal dependencies inherent to time-series data. Unlike the FNN, which treats each input independently, the LSTM is specifically designed to model sequential patterns by maintaining a memory of past inputs over time. This makes it particularly suitable for learning temporal fault signatures and evolving failure dynamics, which are central to the task of fault identification in propulsion systems.

The implemented model is composed of a stack of LSTM layers followed by a fully connected linear projection to the output class space. Dropout regularization is applied after the LSTM layers to mitigate overfitting, and optional bidirectionality (set through the *bidirectional* boolean flag) can be enabled to allow the network to access both past and future context.

The input to the model is a window of preprocessed features of shape (*window_size, input_dim*), where *window_size* is the number of time steps in the sliding window and *input_dim* is the number of features. This input is passed through *num_layers* stacked LSTM layers, each with d_{hidden} hidden units (configurable via the *hidden_dim* hyperparameter). The final hidden state of the last LSTM layer is projected

through a linear transformation to the output vector $y \in \mathbb{R}^{output\_dim}$, which represents unnormalized class scores.

A schematic of the architecture follows below:

- **Input:** 2D tensor of shape (*window_size, input_dim*) representing a temporal window of *window_size* feature vectors
- **LSTM layers:** *num_layers* stacked LSTM layers, each with d_{hidden} hidden neurons
- **Dropout:** Applied to the output of the final LSTM layer
- **Output layer:** Linear projection to class scores of dimension *output_dim*

The training and evaluation procedure for the model largely mirrors the pipeline described in subsection 4.5.2. The model is optimized using the Adam optimizer, with learning rate scheduling and early stopping applied to improve convergence and generalization. The scheduler is configured via three main settings: *scheduler_mode*, which defines the direction of improvement; *scheduler_patience*, the number of epochs to wait before reducing the learning rate; and *scheduler_factor*, the multiplicative factor by which the learning rate is reduced.

The model supports both cross-entropy and focal loss functions, selectable via *use_focal_loss*, with optional class weighting enabled via *apply_class_weights* when using CEL. Validation is performed using `model.eval()` and `torch.no_grad()`, and predictions are obtained by applying a *softmax* activation to the output logits followed by an `argmax` operation.

Figure 4.5 illustrates the overall workflow used for training and evaluating the LSTM model. Similar to what is valid for the FNN and described in subsection 4.5.2, the `LSTM_main.py` script orchestrates the training logic, drawing configuration parameters from `LSTM_settings.py`, and invoking auxiliary modules for dataset loading, performance tracking, and checkpoint management. The configuration settings table is provided in Table 4.9.

4.6 Performance Metrics

The performance of the proposed models is assessed using a comprehensive suite of metrics that reflect both general classification accuracy and task-specific fault identification capabilities. Each metric provides insights into different aspects of model behavior in the context of time-series fault identification. A summary of the performance metrics used in this work, whose detailed description follows, is provided in Table 4.10.

Loss

The loss function guides the learning process by quantifying the discrepancy between the model's predictions and the true labels. During training, this loss is minimized via backpropagation. Both training and validation losses are computed at the end of each epoch and recorded. Tracking these losses is the primary method for monitoring convergence: when the validation loss approaches the training loss, it suggests that the model is learning in a generalizable manner. Conversely, if the training loss continues to decrease while the validation loss increases, this signals potential overfitting to the training data.

Accuracy

Accuracy measures the proportion of all correctly classified samples over the total number of samples (Equation 4.9). It is calculated over the full label space and includes both normal and faulty samples. As a general-purpose metric, it provides a high-level view of the model's overall correctness, but can be misleading in the presence of class imbalance. Like the loss, accuracy is recorded for both training and validation sets at each epoch. When considered together, loss and accuracy offer a more nuanced view of model performance. For instance, if validation loss increases while accuracy also increases, it may indicate that regularization techniques are working effectively to reduce overfitting. However, this is only a positive indication if the loss subsequently begins to decrease while accuracy continues to rise. If the loss continues to grow unchecked, the model is likely diverging, often due to an excessively high learning rate.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{4.9}$$

Precision, Recall, and F1 Score (Macro-Averaged)

To evaluate fault identification performance more rigorously, macro-averaged precision, recall, and F1 score are employed on the validation step. To focus exclusively on fault identification capabilities, the metrics are computed after excluding the "Normal" class using the *filter_normal* setting. This ensures that the evaluation reflects how well the model distinguishes among fault types, rather than being biased by the overrepresented normal class. Additionally, per-class precision and recall are computed to offer insights into which specific faults are better captured by the model.

Identification Delay

Also referred to as identification time, this metric quantifies the average number of time steps between the actual onset of a fault and its first correct identification by the model. A lower identification delay is critical for timely fault mitigation in real-time applications. For example, if the model detects a fault on average five time steps after its onset and the data sampling rate is 100 Hz, this corresponds to a detection latency of 50 ms. This metric is evaluated through the calculated average and class-wise identification time.

Identification Accuracy

In a multiclass classification setup, it is possible for the model to correctly detect that a fault has occurred while misidentifying the specific fault class. Identification accuracy measures the percentage of simulations of a given class for which the first predicted fault label is correct. If the model first predicts the wrong fault and only later corrects itself, the instance is not counted as a success in this metric.

Identification Confidence

This metric evaluates the model's confidence at the moment it identifies a fault. After computing the class probabilities using the *softmax* function, the predicted class is the one with the highest probability (obtained via `argmax`). The value of that probability is recorded as the confidence level. Low confidence

may indicate that the model is undecided between multiple classes, which can lead to incorrect early predictions. This metric can be useful for implementing future thresholds, wherein the model would only assert a fault prediction if its confidence exceeds a given value.

Together, these metrics provide a comprehensive evaluation framework, capturing not only the classification performance but also the timeliness, accuracy, and reliability of the model's fault identification capability.

Table 4.10: Overview of the used performance metrics.

Metric	What it measures	When	Key note
Loss	Pred.–truth gap	Each epoch	Convergence / overfit cue
Accuracy	Correct / total	Each epoch	Skewed by imbalance
Precision (macro)	Correct faults	Validation	Flags false alarms
Recall (macro)	Caught faults	Validation	Flags missed faults
F1 (macro)	Precision–recall trade-off	Validation	Single quality score
Ident. Delay	Time to first hit	Inference	Lower = faster response
Ident. Accuracy	First fault right?	Inference	Penalizes misclassification
Ident. Confidence	*Softmax* score	Inference	Threshold for yielding a fault

4.7 Performance Goals

Defining explicit performance requirements in this work is challenging due to the lack of directly comparable studies. However, precedent exists in the work of Dresia et al. [44] and Lemke [45], both of whom implemented neural network-based fault detection frameworks using synthetic data generated via *EcosimPro*. Their efforts focused solely on fault detection and reported a maximum F1 score of 0.90. The present thesis extends beyond detection to address fault identification, aiming to meet or exceed this benchmark by correctly classifying the specific type of fault as early and accurately as possible.

In addition to overall classification performance, the real-time applicability of the model introduces more granular objectives related to the timing and reliability of fault identification. These targets are defined in alignment with the fault taxonomy and class importance discussed in subsection 4.3.1.

For high-priority **A-Class** faults, such as sensor freezes, rapid identification is critical. Therefore, the objective is to achieve correct identification with a detection delay within 50 ms. For **C-Class** faults, such as leakages or valve degradations, which have less critical safety implications, no strict timing constraint is enforced. Faults in **Class B** or borderline cases, such as moderate blockages or certain sensor degradations, may have intermediate urgency. In such cases, although no hard constraint is imposed, early detection remains desirable.

By establishing both quantitative performance goals (e.g., F1 > 0.90) and class-dependent timing expectations, this framework aims to rigorously assess whether the trained models satisfy not only statistical classification accuracy but also practical operational requirements essential for reliable FDI.

4.8 Model Optimization

To achieve the best-performing models for both the FNN and LSTM architectures, a two-stage optimization strategy is employed. The first stage uses hyperparameter optimization, while the second stage investigates model robustness under different training data configurations.

4.8.1 Hyperparameter Optimization

The initial phase of optimization uses *Optuna* [9], a hyperparameter optimization framework based on Bayesian search techniques. Rather than performing an exhaustive grid search, *Optuna* leverages a probabilistic model to guide the selection of promising hyperparameters and setting configurations, improving efficiency and convergence toward high-performing models. Under this specific implementation, the optimization objective (objective function) is a composite score based on the macro-averaged F1 score and overall accuracy.

For each architecture, two separate optimization runs are performed: one using CEL and another using the focal loss. These scripts that provide the FNN and LSTM optimization frameworks are `FNN_Optuna.py` and `LSTM_Optuna.py`, respectively. Each optimization run is constrained to a maximum wall time of 48 hours. To allow for a large number of trials per optimization, training is shortened to 20 epochs per trial, in order to accelerate this process. This comes under the rationale that strong configurations will already show promising trends within this interval.

For the FNN, the following hyperparameters and settings are explored:

- *hidden_dim*: 32 to 256
- *drop_prob*: 0.0 to 0.5
- *learning_rate*: $1 \cdot 10^{-5}$ to $1 \cdot 10^{-2}$
- *decay*: $1 \cdot 10^{-7}$ to $1 \cdot 10^{-3}$
- *window_size*: 1 to 50 time steps
- *gamma_fl*: 0.0 to 3.0 (only for focal loss)
- *apply_class_weights*: [True, False]
- *use_manual_weights*: [True, False] (only sampled if class weights are applied)

For the LSTM, the optimization includes the two following additional architectural hyperparameters:

- *num_layers*: 1 to 3
- *bidirectional*: [True, False]

For each optimization run, *Optuna* logs the resulting metrics and ranks them according to the composite objective. It also generates importance metrics identifying which hyperparameters most strongly influence performance. The configuration achieving the highest combined F1 and accuracy is selected for further model optimization.

Following *Optuna*-based optimization, a full training session of 60 epochs is conducted using the best configuration. In this second stage, manual tuning is applied iteratively to refine the model further.

Adjustments are guided by empirical observations on validation performance, regularization effectiveness, convergence stability, and class-specific precision or recall. Hyperparameter refinement continues until further changes result in negligible or negative gains, at which point the model is deemed final.

4.8.2 Data-Based Optimization

Beyond hyperparameter tuning, additional performance gains are sought by altering the training data composition. Three distinct configurations are tested:

1. **Full Dataset Training:** The default configuration uses the complete training set, including both normal and faulty simulations.
2. **Fault-Only Simulations:** The model is trained only on simulations that include faults. Note that even these sequences include an initial phase of normal behavior, ensuring that the model is still exposed to normal operating conditions.
3. **Fault-Only with Clipped Duration:** This configuration uses the same fault-only dataset, but truncates the fault segment of each simulation to a maximum of 2 seconds (200 time steps). The rationale is to focus the training process on fault onset characteristics rather than long-term effects, improving the model's ability to respond quickly when a fault begins.

These data-based strategies are evaluated for their impact on detection timing, fault-specific classification accuracy, and identification confidence.

Table 4.2: Features provided by the *EcosimPro* model.

Feature	Description	Unit
Thrust	Engine Thrust	N
P_ch_1	Combustion-chamber pressure in the first segment	Pa
T_ch_1	Combustion-chamber temperature in the first segment	K
CC_oxy_m	Combustion-chamber oxidizer mass-flow rate	kg/s
CC_red_m	Combustion-chamber fuel mass-flow rate	kg/s
Inj_oxy_dp	Oxidizer-injector pressure drop	Pa
Inj_red_dp	Fuel-injector pressure drop	Pa
J11_dp	Oxidizer tank-junction pressure drop	Pa
J12_dp	Oxidizer combustion-chamber-junction pressure drop	Pa
J21_dp	Fuel tank-junction pressure drop	Pa
J22_dp	Fuel combustion-chamber-junction pressure drop	Pa
Ox_Source_P	Oxidizer-source pressure	Pa
Red_Source_P	Fuel-source pressure	Pa
Ox_Source_T	Oxidizer-source temperature	K
Red_Source_T	Fuel-source temperature	K
PLB_11_Pin	Oxidizer before valve, inlet pressure	Pa
PLB_11_Pout	Oxidizer before valve, outlet pressure	Pa
PLB_12_Pin	Oxidizer after valve, inlet pressure	Pa
PLB_12_Pout	Oxidizer after valve, outlet pressure	Pa
PLB_21_Pin	Fuel before valve, inlet pressure	Pa
PLB_21_Pout	Fuel before valve, outlet pressure	Pa
PLB_22_Pin	Fuel after valve, inlet pressure	Pa
PLB_22_Pout	Fuel after valve, outlet pressure	Pa
PLB_11_Tin	Oxidizer before valve, inlet temperature	K
PLB_11_Tout	Oxidizer before valve, outlet temperature	K
PLB_12_Tin	Oxidizer after valve, inlet temperature	K
PLB_12_Tout	Oxidizer after valve, outlet temperature	K
PLB_21_Tin	Fuel before valve, inlet temperature	K
PLB_21_Tout	Fuel before valve, outlet temperature	K
PLB_22_Tin	Fuel after valve, inlet temperature	K
PLB_22_Tout	Fuel after valve, outlet temperature	K
PLB_11_m_in	Oxidizer before valve, inlet mass-flow rate	kg/s
PLB_12_m_in	Oxidizer after valve, inlet mass-flow rate	kg/s
PLB_21_m_in	Fuel before valve, inlet mass-flow rate	kg/s
PLB_22_m_in	Fuel after valve, inlet mass-flow rate	kg/s
Valve_1_cpos	Oxidizer valve commanded position	%
Valve_1_apos	Oxidizer valve actual position	%
Valve_1_dp	Oxidizer valve pressure drop	Pa
Valve_2_cpos	Fuel valve command position	%
Valve_2_apos	Fuel valve actual position	%
Valve_2_dp	Fuel valve pressure drop	Pa

Table 4.8: FNN – hyperparameters and main configuration settings (`FNN_settings.py`).

	Name	Description	Type
Hyperparameters	*hidden_dim*	Number of neurons per hidden layer	int
	drop_prob	Dropout probability	float
	learning_rate	Learning rate	float
	batch_size	Mini-batch size	int
	num_epochs	Number of training epochs	int
	decay	L2 regularization coefficient	float
	gamma_fl	Focal loss focus parameter	float
Settings	*data_types*	Data categories to include in training	list[str]
	normalize_weights	Normalize computed class weights	bool
	use_manual_weights	Load externally defined class weights	bool
	apply_class_weights	Include class weights in loss computation	bool
	window_size	Length of each sliding window in time steps	int
	scaler_type	Normalization method (*zscore* or *min-max*)	str
	flatten_window	Flatten window or keep 2D	bool
	use_focal_loss	Use focal or cross-entropy loss	bool
	scheduler_mode	Learning rate scheduler mode	str
	scheduler_patience	Learning rate scheduler epoch patience	int
	scheduler_factor	Learning rate scheduler factor	float
	stop_patience	Early stopping epoch patience	int
	stop_min_delta	Early stopping minimum delta	float
	stop_mode	Early stopping mode	str
	filter_normal	Exclude “Normal” class from metrics	bool

Table 4.9: LSTM – hyperparameters and main configuration settings (`LSTM_settings.py`).

	Name	Description	Type
Hyperparameters	*hidden_dim*	Number of neurons per hidden layer	int
	num_layers	Number of hidden layers	int
	drop_prob	Dropout probability	float
	learning_rate	Learning rate	float
	batch_size	Mini-batch size	int
	num_epochs	Number of training epochs	int
	decay	L2 regularization coefficient	float
	gamma_fl	Focal loss focus parameter	float
Settings	*bidirectional*	Use bidirectional LSTM	bool
	data_types	Data categories to include in training	list[str]
	normalize_weights	Normalize computed class weights	bool
	use_manual_weights	Load externally defined class weights	bool
	apply_class_weights	Include class weights in loss computation	bool
	window_size	Length of each sliding window in time steps	int
	scaler_type	Normalization method (*zscore* or *min-max*)	str
	flatten_window	Flatten window or keep 2D	bool
	use_focal_loss	Use focal or cross-entropy loss	bool
	scheduler_mode	Learning rate scheduler mode	str
	scheduler_patience	Learning rate scheduler epoch patience	int
	scheduler_factor	Learning rate scheduler factor	float
	stop_patience	Early stopping epoch patience	int
	stop_min_delta	Early stopping minimum delta	float
	stop_mode	Early stopping mode	str
	filter_normal	Exclude “Normal” class from metrics	bool

5 FNN

This chapter presents the results obtained using the feedforward neural network (FNN) model for fault detection and identification. It includes a detailed discussion of hyperparameter optimization using *Optuna*, followed by the manual tuning process and its implications. The final section examines how the composition and extent of the training dataset influence model performance, and presents detailed evaluation metrics for the selected final FNN configuration.

5.1 Optuna Results

The *Optuna* optimization framework is employed to identify the most effective configuration for the FNN architecture under two loss functions: CEL and focal loss. In total, 79 optimization trials are done for the CEL configuration, and 73 for the focal loss setup. For both runs, the objective is to maximize a composite score combining macro-averaged F1 and overall accuracy. The best-performing trials for each case are summarized in Table 5.1 and Table 5.2, respectively.

Table 5.1: FNN with CEL - Best Optuna trial.

Settings		**Results**	
Hyperparameter	Value	Metric	Value
apply_weights	False	*val_loss*	1.235
window_size	30	*val_accuracy*	0.696
decay	0.000137	*val_precision*	0.729
hidden_dim	204	*val_recall*	0.703
drop_prob	0.182363	*val_f1*	0.614
learning_rate	0.000101	*avg_identification_time*	509.83 ms

In the CEL optimization, trial 55 emerged as the top performer, yielding a validation loss of 1.235, accuracy of 0.696, and F1 score of 0.614. Precision and recall are also strong at 0.729 and 0.703, respectively. The average fault identification delay achieved is 509.83 ms.

In contrast, the best focal loss configuration, trial 44, achieves a lower validation loss of 0.545 but a higher accuracy of 0.813. However, it underperforms on fault-specific metrics: macro-averaged F1 drops to 0.533, and recall falls to 0.581. The average identification delay is also slightly higher at 526.30 ms. These results underscore that although the focal loss configuration exhibits better general accuracy, it is less effective at fault identification. This discrepancy reinforces the importance of using targeted fault identification metrics when evaluating model performance in imbalanced, multiclass settings.

It is important to note that the validation losses for the two trials are not directly comparable due to

Table 5.2: FNN with focal loss – Best Optuna trial.

Settings		**Results**	
Hyperparameter	Value	Metric	Value
apply_weights	False	*val_loss*	0.545
window_size	31	*val_accuracy*	0.813
gamma_fl	2.731260	*val_precision*	0.677
decay	0.000007	*val_recall*	0.581
hidden_dim	159	*val_f1*	0.533
drop_prob	0.320507	*avg_identification_time*	526.30 ms
learning_rate	0.000237		

their distinct loss formulations. The lower numerical value observed for the focal loss trial does not imply superior performance overall, as can be confirmed by the poorer fault identification performance.

Based on this comparative analysis, the configuration derived from the CEL-based optimization is selected as the foundation for further training and manual fine-tuning.

The hyperparameter importance plots shown in Figure 5.1a and Figure 5.1b summarize the influence of each tuning parameter on model performance, as determined by *Optuna* during the CEL optimization run for the FNN. These results are derived from *Optuna's* built-in importance estimation, which performs an analysis of variance to estimate the relative importance of each hyperparameter in explaining the variation in the objective function. This technique quantifies how much each hyperparameter contributes to performance differences across trials, enabling identification of the most influential settings.

(a) F1 score. **(b)** Accuracy.

Figure 5.1: FNN with CEL – Hyperparameter importance, Optuna.

Figure 5.1a presents the relative importance of each hyperparameter in terms of its impact on the macro-averaged F1 score. The most influential parameter is clearly the dropout probability (*drop_prob*), accounting for 60% of the observed variance in F1 performance. This highlights the central role of regularization in improving class-wise fault identification. The number of neurons per hidden layer (*hidden_dim*) follows with a contribution of 18%, while L2 regularization (*decay*), learning rate (*learning_rate*), and window size (*window_size*) each contribute modestly (between 7–8%). The application of class weights (*apply_class_weights*) has negligible influence in this configuration, contributing less than 1%. It is worth noting that the optimization process tends to favor the use of manually specified weights

(i.e., *use_manual_weights* = True) when class weighting is enabled. This preference suggests that, although the overall impact on performance was limited, the manually defined weighting scheme provides a more stable contribution compared to automatically computed weights (i.e., inverse of class frequency).

In contrast, Figure 5.1b shows the impact of the same hyperparameters on overall accuracy. Here, the most dominant factor is the sliding window size (*window_size*), which alone accounts for 37% of the performance variability. This suggests that the temporal resolution of input data plays a key role in improving overall classification. The learning rate (*learning_rate*) also exhibits strong influence (29%), followed by *drop_prob* (14%) and *decay* (10%). Again, the use of class weights has a minor effect.

These findings confirm that while regularization (e.g., dropout probability) is pivotal for improving class-discriminative performance (i.e., F1 score), input structure (sliding window size) and learning rate tuning are critical for achieving strong overall accuracy. This dual view reinforces the need for multi-metric optimization in highly imbalanced, multiclass classification settings.

5.2 Model Tuning

Following the selection of the best configuration from the *Optuna* optimization (trial 55 of CEL), the FNN model is trained for a full 60 epochs using the setup defined in Table 5.3. This extended training enables the model to fully converge and is intended to extract the maximum performance from the chosen architecture and settings.

Table 5.3: FNN – Hyperparameters and settings - Trial 55 full training.

	Name	Value
Hyperparameters	*hidden_dim*	204
	drop_prob	0.182363
	learning_rate	0.0001014
	batch_size	512
	num_epochs	60
	decay	0.0001372
	gamma_fl	1.13
Settings	*data_types*	All
	apply_class_weights	False
	window_size	30
	scaler_type	zscore
	flatten_window	True
	use_focal_loss	False
	scheduler_mode	min
	scheduler_patience	4
	scheduler_factor	0.5
	stop_patience	60
	stop_min_delta	0.0
	stop_mode	max
	filter_normal	True

Compared to its initial performance in the 20-epoch *Optuna* evaluation (validation accuracy = 0.696, F1 score = 0.614), the full training significantly improves generalization. As reported in Table 5.4, the final validation accuracy increases to 0.839 and the F1 score to 0.706. All class-discriminative performance metrics improve, confirming the model's capacity to generalize with extended training.

Table 5.4: FNN – Trial 55 - Best validation metrics.

Metric	**Value**
Val Loss	0.8656
Val Accuracy	0.8393
Precision	0.7449
Recall	0.6946
F1 Score	0.7058

To gain deeper insight into the training behavior of trial 55 over the full 60 epochs, Table 5.5 presents the performance changes observed immediately after each learning rate reduction. The scheduler is configured with a *scheduler_factor* of 0.5 and a *scheduler_patience* of 4, meaning the learning rate is halved whenever no improvement is observed over four consecutive epochs.

Table 5.5: Impact of each learning rate reduction.

Learning Rate	**Before**		**After (next epoch)**		**Δ (after − before)**	
	Val Acc.	Val Loss	Val Acc.	Val Loss	ΔAcc.	ΔLoss
$1.01\times10^{-4}\rightarrow5.07\times10^{-5}$	0.819	0.872	0.241	5.280	−0.578	+4.408
$5.07\times10^{-5}\rightarrow2.53\times10^{-5}$	0.839	0.866	0.321	3.609	−0.519	+2.744
$2.53\times10^{-5}\rightarrow1.27\times10^{-5}$	0.348	2.075	0.297	2.800	−0.052	+0.725

The results show that every time the learning rate is halved, model performance suffers a sharp degradation. For instance, the first learning rate reduction leads to a drastic accuracy drop of 57.8 percentage points and a corresponding loss surge of +4.408. A similar trend is observed across subsequent reductions, with each step resulting in significantly worse validation accuracy and increased loss.

This pattern indicates that the model may rely on a relatively high learning rate to escape narrow minima and maintain generalization capacity. Not only do the halving steps provide no long-term benefits, but they also seem to trigger sustained collapses in performance. For example, after the third reduction, the model never again surpasses the 0.84 validation accuracy achieved at higher learning rates. Additionally, this behavior can be due to the stochastic properties of the dropout layers, with neurons being excluded at random epochs. Since the *drop_prob* has a strong impact on performance, modifying this hyperparameter can also yield gains.

Such findings motivate a reassessment of the learning rate schedule, as the described "halve-on-plateau" strategy appears too aggressive and counterproductive. In addition, the stochastic nature of dropout may further destabilize training. To address these concerns, a revised model is trained with a smoother learning rate schedule (*scheduler_factor* = 0.9) and dropout disabled (*drop_prob* = 0.0). In this revised configuration, regularization relies solely on L2 weight decay (*decay*). All other hyperparameters remain unchanged. The performance of this configuration, referred to as trial 55.1, is summarized in Table 5.6.

While this setup improves fault identification metrics, achieving higher recall and F1 score, it comes at

Table 5.6: FNN - Trial 55.1 – Best validation metrics.

Item	Value
Hyperparameter updates	
scheduler_factor	$0.5 \rightarrow 0.9$
drop_prob	$0.182363 \rightarrow 0.0$
Best validation metrics	
Val Loss	6.0218
Val Accuracy	0.2592
Precision	0.7370
Recall	0.7278
F1 Score	0.7166

a significant cost in terms of overall accuracy and validation loss. These results suggest that removing dropout and reducing learning rate decay may enhance fault-specific recall but impair general classification performance. Consequently, the final FNN configuration adopted in this thesis remains that of trial 55 (Table 5.3), which offers a more balanced trade-off between global accuracy and class-wise fault identification.

5.3 Data Influence on Performance

To assess how the nature of training data affects performance, three configurations are evaluated: using all data categories, using only faulty data, and using a clipped subset of faulty data (200 seconds). The corresponding metrics are presented in Table 5.7.

Table 5.7: FNN - Validation metrics according to different data configurations.

Data Configuration	Val Loss	Val Acc.	Precision	Recall	F1 Score
All Categories	0.8656	0.8393	0.7449	0.6946	0.7058
Only Faulty Data	1.1825	0.6764	0.7185	0.6765	0.6794
Faulty & clipped ($\leq$200 s)	0.2544	0.9537	0.4900	0.3417	0.3755

Using the complete dataset yields the most balanced performance, with strong results across all evaluation metrics. In contrast, training solely on faulty data leads to a noticeable drop in validation accuracy and F1 score. Interestingly, clipped faulty data, despite producing high validation accuracy, suffers from severely degraded recall and F1 score. This apparent gain in accuracy is likely driven by an increased proportion of normal samples within the clipped sequences, which biases the model toward correctly predicting the dominant class while neglecting minority faults. The resulting class imbalance skews model behavior, inflating accuracy but undermining generalization and fault sensitivity (recall).

These findings support the inclusion of all data categories in the training process. This configuration, which maximizes exposure to both nominal and faulty behavior, forms the basis for the final model. Specifically, the model trained with the settings of trial 55 (Table 5.3) on the full dataset is adopted as the final FNN model.

5.4 Final Model Performance

Table 5.8 summarizes the overall classification performance of the final FNN model, trained using the configuration of trial 55 on the complete dataset. With a macro-averaged F1 score of 0.7058 and overall accuracy of 0.8393, the model achieves generalization while maintaining balanced class-wise performance. These values fall short of the originally defined target of F1 > 0.90. Nonetheless, they reflect meaningful progress toward fault identification capabilities in a multiclass setting, particularly given the added complexity of classifying fault types rather than merely detecting anomalies.

Table 5.8: FNN – Trial 55 - Overall Performance summary.

Metric	Value
Accuracy	0.8393
Precision (macro)	0.7449
Recall (macro)	0.6946
F1 Score (macro)	0.7058

A more detailed breakdown is provided in Table 5.9, which presents aggregated class-wise performance for each fault type. High-performing categories include *Sensor Drift* and *Pipe Blockage*, with both precision and recall above 0.98, indicating that the model reliably identifies these fault types early and accurately. In contrast, the *Sensor Freeze* class records the lowest recall (0.513) and the highest average identification delay (4650.0 ms), falling short of the 50 ms timing target defined for critical A-Class faults, and meaning that the model is missing a considerable amount of fault samples of this type. Despite a relatively high precision (0.778), the delayed and inconsistent detection of this fault type limits its practical reliability in real-time applications.

Table 5.9: FNN - Trial 55 - Aggregated class-wise performance.

Fault Type	**Precision**	**Recall**	**First Ident. Conf.**	**First Corr. Conf.**	**Ident. Acc.**	**Avg. Ident. Time (ms)**
Blockage	0.998	0.981	0.593	0.653	0.504	48.5
Blockage + Leak	0.847	0.844	0.445	0.736	0.200	35.8
Slow valve	0.967	0.825	0.354	0.433	0.203	2302.3
Sensor freeze	0.778	0.513	0.200	0.475	0.250	4650.0
Sensor bias	0.836	0.801	0.368	0.438	0.841	862.1
Sensor drift	1.000	0.989	0.612	0.549	0.153	198.6

The *Slow Valve* and *Pipe Blockage + Leak* faults also present mixed results. Although precision remains high across both classes, recall values are more modest (0.825 and 0.844, respectively), and identification accuracy is low (< 0.203). These findings suggest that while the model frequently flags the presence of a fault, it may initially misclassify the fault type before later correcting its prediction.

This behavior is further supported by comparing the *First Identification Confidence* and *First Correct Identification Confidence* columns. In most cases, the confidence associated with the first corrected identification is higher than the first prediction. This implies that the model often updates its classification as more data becomes available and the faulty pattern becomes clearer, which benefits long-term detection metrics like recall and F1, but negatively impacts identification accuracy.

As a final remark, out of 959 validation simulations, the model fails to correctly identify the fault in 254 cases.

Taken together, these results confirm that the model performs well across most fault types, with particularly strong performance on blockages and sensor drift. However, its effectiveness varies depending on fault class and temporal characteristics. While identification delay and accuracy metrics for some A-Class and B-Class faults do not yet meet the predefined thresholds, the overall F1 score and class-wise breakdown support the viability of the current architecture as a baseline.

6 LSTM

The present chapter details the development, optimization, and evaluation of the LSTM model. It begins by describing the hyperparameter search using *Optuna*, followed by manual tuning based on a priori and empirical insights. A comparative analysis of training data configurations is conducted to assess their influence on model behavior. The final configuration performance is reported both in terms of aggregate metrics and class-specific outcomes, with emphasis on detection timeliness and classification reliability.

6.1 Optuna Results

The *Optuna* framework is employed to optimize the hyperparameters of the LSTM architecture under both the CEL and focal loss configurations. Given the increased computational complexity and training time associated with the LSTM's deeper architecture, fewer trials are completed within the allocated optimization window. A total of 33 trials are conducted for the CEL configuration and 16 for the focal loss. As in the FNN case described in section 5.1, the objective is to maximize a composite score that balances macro-averaged F1 and overall accuracy.

The best-performing trial for the CEL-based configuration is trial 5, summarized in Table 6.1. This configuration achieves a validation accuracy of 0.866 and an F1 score of 0.806, with precision and recall values of 0.835 and 0.794, respectively. The average identification delay is 385.8 ms, representing a significant improvement in responsiveness over the FNN results discussed previously. The use of weighted CEL is favored (again with *use_manual_weights* = True), and the architecture selected is relatively shallow (one LSTM layer, no bidirectionality), with moderate regularization and a modest hidden dimension of 76.

Table 6.1: LSTM with CEL – Best Optuna trial.

Settings		**Results**	
Hyperparameter	Value	Metric	Value
apply_weights	True	*val_loss*	0.700
window_size	37	*val_accuracy*	0.866
decay	0.000002	*val_precision*	0.835
hidden_dim	76	*val_recall*	0.794
num_layers	1	*val_f1*	0.806
drop_prob	0.187058	*avg_identification_time*	385.8 ms
learning_rate	0.000802		
bidirectional	False		

For the focal loss configuration, trial 16 delivers the best result, as shown in Table 6.2. This setup achieves a slightly lower validation accuracy of 0.861 and an F1 score of 0.781. Precision and recall are recorded at 0.813 and 0.770, respectively. Notably, the identification delay increases to 525.2 ms compared to the CEL-based result. As observed in the FNN results, the focal loss yields a lower validation loss (0.378 vs. 0.700) but weaker fault identification performance, reinforcing the earlier conclusion that validation loss alone is not a sufficient indicator of task-specific efficacy. In contrast to the CEL configuration, this trial does not apply class weights and opts for a deeper, bidirectional model with two LSTM layers and a larger hidden dimension of 254.

Table 6.2: LSTM with focal loss – Best Optuna trial.

Settings		**Results**	
Hyperparameter	Value	Metric	Value
apply_weights	False	*val_loss*	0.378
window_size	35	*val_accuracy*	0.861
gamma_fl	1.346628	*val_precision*	0.813
decay	0.000000	*val_recall*	0.770
hidden_dim	254	*val_f1*	0.781
num_layers	2	*avg_identification_time*	525.2 ms
drop_prob	0.375790		
learning_rate	0.000159		
bidirectional	True		

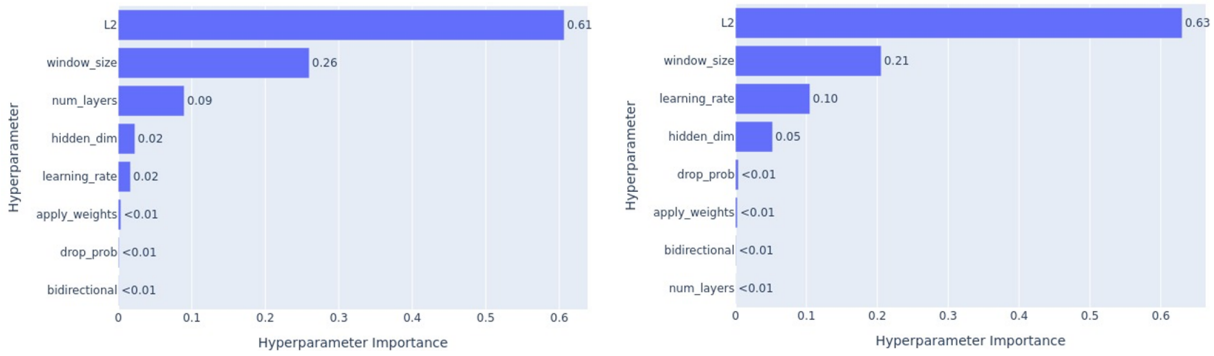
These outcomes align with the broader trends noted in section 5.1: although focal loss can improve general classification accuracy, it often underperforms on fault identification metrics critical in imbalanced, multiclass settings. Furthermore, once again, the optimization process tends to favor the use of manually defined class weights under the CEL configuration, a preference that is similarly observed in the FNN experiments.

The configuration derived from the CEL-based optimization is selected as the foundation for further training and manual model tuning.

The relative influence of hyperparameters on LSTM model performance, as assessed by *Optuna*, is visualized in Figure 6.1a and Figure 6.1b. These plots correspond to the CEL optimization run and are generated using *Optuna*'s internal variance-based attribution method.

For the F1 score (Figure 6.1b), L2 regularization (*decay*) emerges as the dominant factor, accounting for over 60% of the observed variance. This indicates that the generalization ability of the LSTM model relies heavily on controlled weight penalization, rather than on stochastic methods such as dropout. The second most impactful variable is the *window_size* (26%), reaffirming the importance of time-window selection when working with sequential inputs. The number of stacked recurrent layers (*num_layers*) also plays a moderate role (9%), while other parameters, including *hidden_dim*, *learning_rate*, and *apply_weights*, contribute minimally to F1 optimization in this configuration.

A similar trend is observed for overall accuracy, as shown in Figure 6.1a. Here, L2 regularization again holds the highest importance (63%), followed by *window_size* (21%) and *learning_rate* (10%). The neg-



(a) Accuracy. **(b)** F1 score.

Figure 6.1: LSTM with CEL – Hyperparameter importance, Optuna.

ligible influence of dropout and bidirectionality suggests that, within this particular architecture and dataset, such features do not provide significant performance gains. Interestingly, while dropout was a critical regularization mechanism in the FNN case (see section 5.1), it has far less impact in the LSTM setup, where deterministic regularization through weight decay is more effective.

Together, these findings suggest that LSTM-based models benefit most from precise control over weight magnitudes and temporal resolution, rather than from architectural complexity or probabilistic regularization. These insights are consistent with the deeper nature of recurrent networks and inform subsequent model tuning decisions.

6.2 Model Tuning

Once the most promising configuration is selected from the *Optuna* search (Trial 5 of the CEL run), the LSTM model is subjected to an extended training phase spanning 60 epochs. The full training is executed using the hyperparameter and configuration settings listed in Table 6.3. This process aims to allow the network to refine its temporal feature representations and reach a stable optimization state.

When comparing the outcomes of the extended run to the preliminary 20-epoch results used during *Optuna* optimization, no major shifts in metrics are observed. As shown in Table 6.4, the final validation accuracy remains at 0.8663, and the macro-averaged F1 score stabilizes at 0.8062. Precision and recall also remain balanced, suggesting that the model does not overfit and retains strong generalization capabilities across fault types.

This training setup includes manually applied class weights ($use\_manual\_weights$ = True), a conservative regularization regime via L2 penalty ($decay$ = 0.000002), and a smooth learning rate schedule ($scheduler\_factor$ = 0.9). The use of a single-layer, unidirectional LSTM combined with a relatively large window size ($window\_size$ = 37) supports robust performance without excessive complexity.

In addition to the baseline configuration optimized via *Optuna* (Trial 5 of CEL), an alternative LSTM model is trained using a set of hyperparameters selected through a priori knowledge and iterative refinement. This configuration incorporates the focal loss instead, and modified architectural and training settings, designed to improve fault-specific performance, particularly with respect to class imbalance.

Table 6.3: LSTM – Hyperparameters and settings - Trial 5 full training.

	Name	Value
Hyperparameters	*hidden_dim*	76
	drop_prob	0.187058
	learning_rate	0.000802
	batch_size	512
	num_epochs	60
	decay	0.000002
	gamma_fl	1.13
	num_layers	1
	bidirectional	False
Settings	*data_types*	All
	apply_class_weights	True
	window_size	37
	scaler_type	zscore
	flatten_window	False
	use_focal_loss	False
	scheduler_mode	min
	scheduler_patience	6
	scheduler_factor	0.9
	stop_patience	60
	stop_min_delta	0.0
	stop_mode	max
	filter_normal	True

Table 6.4: LSTM – Trial 5 - Best validation metrics.

Metric	Value
Val Loss	0.5939
Val Accuracy	0.8663
Precision	0.8427
Recall	0.7940
F1 Score	0.8062

The comparative effects of the design changes between the baseline model and the focal loss configuration are summarized below. The impact of each adjustment on validation behavior is discussed in relation to the corresponding performance metrics reported in Table 6.4 and Table 6.6.

- **Loss Function:** The baseline model employs a weighted cross-entropy loss, whereas the focal loss configuration adopts focal loss with a γ of 1.13. This modification emphasizes hard-to-classify and minority class samples by down-weighting confident predictions. As a result, recall increases by approximately 1.5 percentage points, and the F1 score improves from 0.8062 to 0.8135. While the validation loss appears higher numerically, this is expected due to the different formulation of focal loss, which is not directly comparable to cross-entropy.

Table 6.5: LSTM – hyperparameters and settings - Focal loss model.

	Name	Value
Hyperparameters	*hidden_dim*	256
	drop_prob	0.24
	learning_rate	0.001
	batch_size	512
	num_epochs	60
	decay	0.0002
	gamma_fl	1.13
	num_layers	2
	bidirectional	False
Settings	*data_types*	All
	apply_class_weights	True
	window_size	11
	scaler_type	zscore
	flatten_window	False
	use_focal_loss	True
	scheduler_mode	min
	scheduler_patience	4
	scheduler_factor	0.5
	stop_patience	60
	stop_min_delta	0.0
	stop_mode	max
	filter_normal	True

- **Window Size:** The input window size is reduced from 37 to 11 time steps. This shorter temporal context reduces irrelevant information and simultaneously increases the number of training samples via overlapping windows. The deeper model architecture compensates for the shorter context, preserving precision and contributing to the observed gains in recall.
- **Model Capacity:** The number of hidden layers is increased from one to two, and the hidden dimension is expanded from 76 to 256 units. This yields approximately 20 times more parameters, enabling the model to represent more complex temporal patterns. These changes are particularly beneficial in the presence of focal loss, which steers the learning process toward underrepresented and difficult fault cases.
- **Regularization:** To mitigate the risk of overfitting introduced by the expanded network capacity, stronger regularization is applied. Dropout probability increases from 0.19 to 0.24, and L2 weight decay rises from 2×10^{-6} to 2×10^{-4}, a 100-fold increase. These adjustments improve generalization and help maintain validation accuracy at 0.866 despite the model's increased complexity.
- **Learning Rate and Scheduler:** The initial learning rate is increased from 8.0×10^{-4} to 1.0×10^{-3}, while the learning rate scheduler is made more aggressive by reducing the patience to 4 epochs and increasing the decay factor to 0.5. This allows faster initial convergence and finer control over the training dynamics. The resulting training behavior leads to a sharper decision boundary and

contributes to the observed increase in precision (+0.6 percentage points).

In summary, the focal loss configuration outperforms the baseline in terms of class-discriminative performance. The integration of focal loss, additional model depth, and enhanced regularization enables better coverage of underrepresented fault modes, justifying the observed improvements in F1 (from 0.806 to 0.814) and recall, without compromising overall classification reliability. It is important to note, however, that the *Optuna* optimization did not identify focal loss as the best-performing configuration during the hyperparameter search. This is likely due to the limited number of trials conducted, which may have constrained the exploration of more complex configurations. Consequently, this outcome highlights the role of a priori knowledge and empirical experimentation as essential complements to optimize a neural network model. The final LSTM configuration adopted in this work is the focal loss model, whose corresponding hyperparameters are summarized in Table 6.5.

Table 6.6: LSTM – Focal Loss model - Best validation metrics.

Metric	Value
Val Loss	0.6569
Val Accuracy	0.8656
Precision	0.8484
Recall	0.8090
F1 Score	0.8135

6.3 Data Influence on Performance

As defined in the methodology, the effect of training data composition on model performance is systematically evaluated. The resulting validation metrics for the three configurations, complete dataset, only faulty data, and clipped faulty sequences, are presented in Table 6.7.

Table 6.7: LSTM - Validation metrics according to different data configurations.

Data Configuration	Val Loss	Val Acc.	Precision	Recall	F1 Score
All Categories	0.6569	0.8656	0.8484	0.8090	0.8135
Only Faulty Data	0.7216	0.8179	0.8497	0.8208	0.8257
Faulty & clipped ($\leq$200 s)	0.2533	0.9412	0.7643	0.5017	0.5681

Contrary to the trend observed with the FNN model, where training on faulty data degraded performance, the LSTM exhibits improved results when trained exclusively on fault examples. In this configuration, both the F1 score and recall increase relative to the baseline that includes nominal data, suggesting that the model is better able to generalize fault patterns without the presence of dominant normal sequences. However, this improvement in fault-centric metrics is accompanied by a slight degradation in overall accuracy and validation loss. This discrepancy likely arises because the model trained on the full dataset benefits from the overrepresentation of "Normal" samples, which artificially inflates accuracy by favoring correct identification of the majority class.

The clipped faulty dataset leads to a different behavior. While validation accuracy rises, this improvement is misleading. As with the FNN, the shorter sequences contain proportionally more "Normal" samples,

which biases the model toward predicting the majority class. This bias results in reduced precision, recall, and F1 score, metrics that are more indicative of fault identification capability.

Based on these findings, the faulty-only dataset configuration is retained for the final LSTM model. This setup provides focused exposure to fault-specific patterns, enhancing the model's capacity to identify and generalize rare fault cases without the confounding influence of overrepresented nominal data.

6.4 Final Model Performance

As justified in the preceding section, the final LSTM model adopts the focal loss configuration trained exclusively on fault data. A summary of the overall classification performance is provided in Table 6.8. The model achieves a macro-averaged F1 score of 0.8257 and a validation accuracy of 0.8497, indicating effective generalization across fault types. While these values remain below the initially defined target of 0.90 for the macro F1 score, they represent a meaningful improvement over the FNN baseline, particularly in terms of fault-class sensitivity.

Table 6.8: LSTM – Focal loss model - Overall Performance summary.

Metric	Value
Accuracy	0.8179
Precision (macro)	0.8497
Recall (macro)	0.8208
F1 Score (macro)	0.8257

Detailed class-wise results are presented in Table 6.9. A notable outcome is the model's consistent performance on all fault classes except two (*Slow valve* for the oxidizer and fuel). For faults such as *Blockage*, *Blockage + Leak*, and *Sensor Drift*, the model exhibits recall close to 1, with identification delays well below the 50 ms threshold. This affirms the model's capacity to rapidly detect faults with strong and early signal patterns. In particular, the *Blockage* class achieves a recall of 0.987 and an average identification time of 34.2 ms, suggesting that the temporal structure of the input data strongly aligns with the LSTM's sequence modeling capabilities.

Table 6.9: LSTM – aggregated class-wise performance.

Fault Type	**Precision**	**Recall**	**First Ident. Conf.**	**First Corr. Conf.**	**Ident. Acc.**	**Avg. Ident. Time (ms)**
Blockage	0.996	0.987	0.651	0.679	0.571	34.2
Blockage + Leak	0.865	0.869	0.711	0.658	0.375	39.5
Slow valve	0.411	0.131	0.350	0.352	0.454	8 144.5
Sensor freeze	0.778	0.665	0.527	0.465	0.434	3 045.5
Sensor bias	0.933	0.963	0.422	0.716	0.247	147.0
Sensor drift	0.999	0.989	0.535	0.576	0.576	89.2

However, fault modes with more subtle temporal dynamics show divergent behavior. For instance, the *Slow valve* class demonstrates both poor recall (0.131) and excessive identification delay (8.1 s), making it the least effectively captured fault in the dataset. Interestingly, despite this poor timing performance, its identification accuracy is moderate (0.454), suggesting that when the model eventually recognizes

the fault, it often does so with correct classification. A similar trend is observed for *Sensor freeze*, with moderately high precision (0.778) but delayed detection (3.0 s), limiting its practical utility in real-time scenarios.

The variation between early and correct identification confidence is also revealing. For classes such as *Sensor bias* and *Blockage + Leak*, the confidence associated with the correct identification exceeds that of the initial guess. This suggests the model refines its internal representation over time, adjusting its decision boundary as new evidence accumulates. For *Sensor bias*, this dynamic results in a recall of 0.963 and precision of 0.933, despite a relatively low initial confidence, indicating high eventual reliability with less certainty at onset.

The strongest performance is recorded for *Sensor drift*, which reaches a precision and recall close to 1, and an average identification delay of under 100 ms. This may be due to the gradually evolving nature of drift faults, which aligns well with the LSTM's memory structure and allows early and stable recognition.

A total of 86 out of 959 validation simulations result in missed fault identifications, underscoring remaining challenges in achieving consistent detection performance across all scenarios. Nonetheless, there is a considerable improvement over the FNN.

Overall, the results affirm that the LSTM architecture, when paired with focal loss and appropriately filtered training data, offers significant advantages for early and robust fault identification, particularly for fault types characterized by clear, temporally correlated patterns. Nevertheless, certain fault modes with delayed onset or subtle transitions remain difficult to capture in a timely and confident manner.

7 Discussion

Following up from the results and the individual discussion of the models, the present chapter provides a holistic comparative assessment of the FNN and LSTM models, analyzing their respective trade-offs in accuracy, fault sensitivity, and computational complexity. It also explores the limitations posed by the current fault labeling strategy, particularly for valve and freeze faults, and concludes with critical deployment-time considerations.

7.1 Model Comparison: FNN vs LSTM

A comparative evaluation of the final FNN and LSTM models is presented in Table 7.1 and Table 7.2, summarizing both global classification metrics and class-wise performance across fault categories. While both models demonstrate competitive results, each exhibits distinct strengths and trade-offs in terms of accuracy, fault sensitivity, computational complexity, and robustness to operational variability.

Table 7.1: Comparison of validation performance metrics for the final FNN and LSTM models.

Model	**Val Acc.**	**Precision**	**Recall**	**F1**
FNN	0.839	0.745	0.695	0.706
LSTM	0.818	0.850	0.821	0.826

Table 7.2: Comparison of class-wise aggregated metrics for the final FNN and LSTM models.

Fault family	**Precision** FNN / LSTM	**Recall** FNN / LSTM	**Ident. Acc.** FNN / LSTM	**Avg. Ident. Time (ms)** FNN / LSTM
Blockage	0.998 / 0.996	0.981 / 0.987	0.504 / 0.571	48.5 / 34.2
Blockage + Leak	0.847 / 0.865	0.844 / 0.869	0.200 / 0.375	35.8 / 39.5
Slow valve	0.967 / 0.411	0.825 / 0.131	0.203 / 0.454	2 302.3 / 8 144.5
Sensor freeze	0.778 / 0.778	0.513 / 0.665	0.250 / 0.434	4 650.0 / 3 045.5
Sensor bias	0.836 / 0.933	0.801 / 0.963	0.841 / 0.247	862.1 / 147.0
Sensor drift	1.000 / 0.999	0.989 / 0.989	0.153 / 0.576	198.6 / 89.2

Table 7.1 shows that the LSTM model achieves higher macro-averaged metrics for precision (0.850 vs. 0.745), recall (0.821 vs. 0.695), and F1 score (0.826 vs. 0.706) compared to the FNN. These improvements suggest a superior ability to identify and correctly classify fault classes, the critical requirement in FDI. The FNN model reports a slightly higher overall accuracy (0.839 vs. 0.818), but this difference is attributable to a key variation in the training datasets: the FNN is trained and validated on a dataset that also includes completely normal simulations, whereas the LSTM is trained exclusively on

fault-containing runs. As a result, the FNN benefits from an inflated accuracy score driven by its ability to correctly classify the dominant "Normal" class, rather than superior performance on fault-specific identification.

Two architectural and training-related factors help explain this performance gap. First, the LSTM benefits from explicit sequence modeling: it processes each 11-step window as a temporal sequence, preserving order and enabling the network to extract dynamic patterns such as gradients, ramps, or oscillations. In contrast, the FNN flattens a longer 30-step window into a static feature vector, discarding temporal structure and treating each time step as independent. This inherently limits the FNN's ability to capture fault dynamics that unfold over time.

Second, the models differ in their loss formulations. The FNN is trained with a standard class-weighted cross-entropy loss, which promotes balanced learning across classes but may still bias the network toward dominant patterns. The LSTM instead leverages a focal loss objective, which dynamically emphasizes hard or ambiguous samples by down-weighting confidently classified examples. This focus improves sensitivity to minority fault types, contributing to the observed recall and F1 improvements.

The class-wise analysis in Table 7.2 provides further nuance. For most fault categories, particularly *Blockage*, *Blockage + Leak*, and *Sensor Drift*, the LSTM matches or surpasses the FNN in recall and identification accuracy. Notably, the LSTM achieves a recall of 0.987 and identification accuracy of 0.571 for the *Blockage* class, with an average identification time of 34.2 ms, outperforming the FNN in all three metrics.

However, performance across all classes should be interpreted with caution. The class-wise metrics exclude missed simulations, those for which the model fails to produce any valid prediction. In this regard, the LSTM demonstrates greater robustness: it misses only 86 out of 959 validation simulations, compared to 254 for the FNN. This substantial improvement highlights the reliability of the recurrent architecture in capturing temporal dependencies critical to fault evolution.

Despite its advantages in detection quality, the LSTM carries a higher computational burden. The model comprises 910 911 parameters, nearly triple the 324 423 parameters of the FNN. This increase translates into longer training durations: the FNN averages roughly 3 minutes per epoch, while the LSTM requires 5.7 minutes. The added complexity stems from the recurrent structure and gating mechanisms of the LSTM, which involve sequential processing along the time dimension.

This computational cost may pose challenges for real-time deployment. Larger memory requirements, higher inference latency, and increased demand for processing resources could limit the feasibility of using the model on embedded devices. Conversely, the simpler and faster FNN may be more suitable for environments where hardware constraints and inference speed are critical, despite its reduced fault sensitivity.

In summary, the LSTM model offers better class-discriminative performance and generalization to fault patterns, especially in fault-heavy scenarios. However, these gains come with increased inference complexity and training cost. The FNN provides a computationally lighter alternative that performs well on dominant classes but underperforms in minority fault detection. The choice between architectures thus depends on the specific requirements of the deployment context, whether it prioritizes classification accuracy across all faults or computational efficiency.

7.2 Why are Valve and Freeze Faults Problematic?

Valve-related faults (*Slow valve*) consistently exhibit the weakest identification performance across both architectures. This behavior can be directly attributed to a misalignment between the fault labeling strategy and the actual manifestation of observable fault symptoms.

Figure 7.1 illustrates the underlying issue with a *Slow valve* fault example. In current simulations, fault labels are activated at the precise moment a valve actuator begins to degrade (denoted by the red dashed line). However, if the commanded position remains unchanged for several seconds after the fault onset, the actual valve position (orange) continues to match the commanded signal (blue), producing no visible discrepancy. As a result, the network is exposed to input sequences that appear identical in terms of observed features, yet are inconsistently labeled as either "Normal" or "Fault." This inconsistency introduces ambiguity into the learning process, resulting in noisy gradients and diminished generalization. The impact is evident, with the LSTM recording a recall of just 0.131 and an average identification delay of 8.1 seconds.

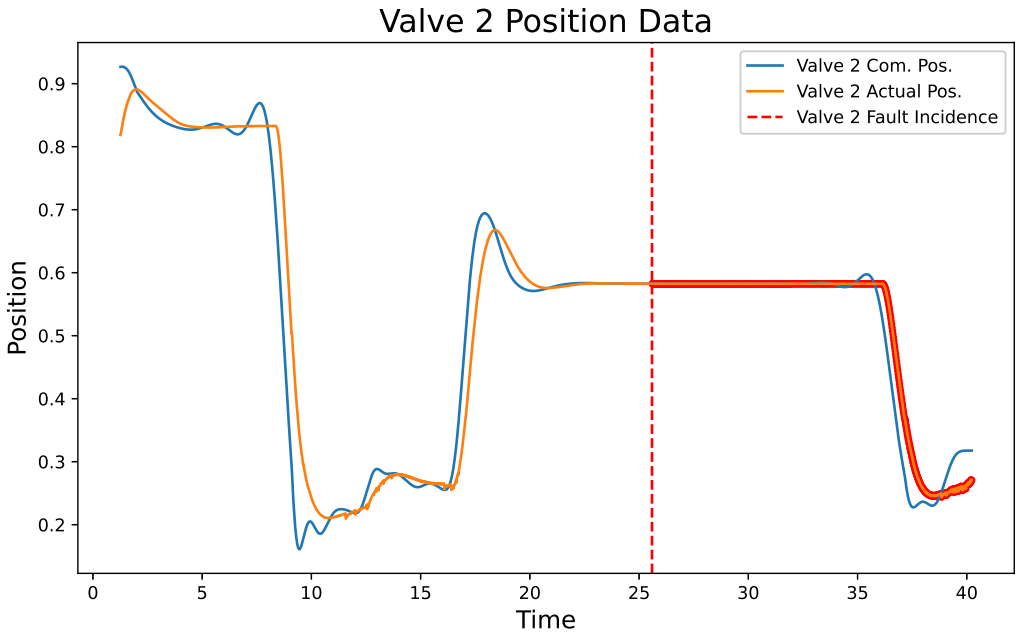


Figure 7.1: Valve fault behavior.

Sensor freeze faults suffer from a related temporal limitation. These faults are only observable once the underlying true signal is expected to evolve. Until that change occurs, a frozen sensor value is indistinguishable from a legitimate steady-state reading. In effect, there is a delay between the moment the freeze is triggered and when it becomes detectable. Both the FNN and LSTM models exhibit this limitation, with average identification delays of 4.6 seconds and 3.0 seconds, respectively.

These findings suggest that the current labeling policy may be suboptimal for faults with delayed or context-dependent manifestations. A more effective approach would involve triggering the valve fault label only when a subsequent command is issued and the actuator fails to respond accordingly, e.g., when the position error exceeds a threshold. Likewise, freeze faults could be labeled after the system leaves

steady-state, rather than at the moment of fault injection. This adjustment would ensure that model updates are driven by meaningful discrepancies, rather than penalizing the network for producing outputs consistent with the observed input.

7.3 Deployment-time Considerations

While the identification delay, defined as the number of time steps required before a correct fault label is first predicted, has been thoroughly evaluated, real-world deployment introduces additional timing and operational constraints. In practice, the total latency from fault onset to actionable output consists of two components: the identification delay (as computed) and the inference delay, which reflects the time required by the deployed model to process the input and produce a prediction.

At the time of writing, the onboard avionics for the hopper are not yet finalized, and no hardware benchmarks have been performed. Nevertheless, hardware limitations must be taken into account when assessing real-time feasibility. The LSTM model, consisting of two stacked layers with 256 hidden neurons each, contains close to one million parameters. This architectural complexity translates to significantly longer inference times, especially on low-power or embedded processors. This highlights the need for future benchmarking on the final embedded platform to verify whether inference delays remain compatible with real-time requirements.

Another key challenge lies in the assumption of consistent input sampling. Both models are trained on data acquired at a uniform 100 Hz sampling rate. However, in a live deployment scenario, this ideal condition may not be guaranteed. Reading loss, clock drift, or timing jitter could result in irregular sequences or effective down-sampling of the data stream. Such disturbances can degrade performance, particularly for the LSTM, which relies heavily on temporal coherence. For example, a reduction to 50 Hz effectively halves the number of distinguishable steps within a transient event, undermining the model's ability to capture fault dynamics. To mitigate this, two possible strategies are recommended:

- **Data augmentation:** Incorporating synthetically decimated sequences into the training process to increase robustness.
- **Preprocessing:** Deploying a lightweight onboard resampling algorithm to reconstruct a uniform 100 Hz time base prior to inference.

Finally, consistent normalization is imperative. During training, all input features were standardized using fixed mean and standard deviation statistics. Any deviation from these constants at inference time, such as recomputing statistics from incoming data, could distort the input distribution, triggering false positives or suppressing valid fault detections. It is therefore essential to freeze and embed the original normalization parameters into the deployment pipeline.

In summary, real-time viability depends not only on the model's predictive accuracy but also on its computational and operational compatibility with the target environment. Model selection, preprocessing reproducibility, and sampling fidelity must all be addressed to ensure reliable fault identification under field conditions.

8 Conclusion and Future Work

This thesis set out to demonstrate that deep-learning–based FDI can meet the tight reliability and latency requirements of a hopper propulsion system, while remaining flexible enough to survive design iterations. Two neural network architectures, namely a feedforward neural network (FNN) and a long short-term memory (LSTM) network, have been implemented and rigorously evaluated, each trained on synthetically generated time-series data derived from simulation runs covering a broad range of normal behavior and fault scenarios.

Both models succeed in learning meaningful fault representations and achieving respectable performance across multiple evaluation metrics. However, the LSTM consistently outperforms the FNN in fault sensitivity and class-wise discrimination. This advantage is attributed to its ability to exploit temporal dependencies inherent in the time-series data, as well as to the adoption of a focal loss function that prioritizes hard-to-classify and minority fault cases. In contrast, the FNN, while computationally lighter and faster to train, proves more susceptible to dominant class bias and performs less robustly in detecting nuanced fault signatures.

A key insight arising from this work is the critical role played by labeling quality. The poor performance observed in valve-related and sensor freeze faults can be directly traced to discrepancies between fault labels and the actual emergence of observable anomalies in the input features. In cases where labels precede detectable symptoms, the network receives ambiguous training signals that compromise its ability to generalize. This limitation manifests in high identification delays and severely degraded recall, particularly for faults that manifest as a delayed valve response.

Moreover, this work underscores the necessity of using evaluation metrics that reflect the true operational capabilities of the models. While overall accuracy is frequently reported, it proves misleading in this context. This work goes a step beyond conventional macro-averaging by also excluding the correct identification of "Normal" samples from the reported fault-centric metrics, thereby ensuring that the evaluation reflects true fault classification performance rather than success on dominant nominal behavior. As shown, a model may achieve high accuracy by merely predicting the "Normal" state correctly, while entirely failing to detect critical fault cases.

Although early prediction of fault onset was originally envisioned as one of the thesis objectives, this goal was not pursued due to time constraints. Both models developed in this work operate reactively, raising alarms only after the fault pattern becomes observable. The predictive dimension of fault management, critical for anticipating failures and enabling preemptive mitigation, remains an open direction for future exploration.

In summary, this work offers a strong foundation for data-driven fault detection and identification in aerospace propulsion systems. Its results affirm the promise of deep learning in this domain, while

also highlighting the practical and methodological challenges that must be addressed to ensure reliable deployment in real-world settings.

8.1 Prospects for Further Development

While this thesis has demonstrated the viability of deep learning for fault detection and identification in a rocket-propulsion FDI context, several avenues remain open for future research and practical enhancement. These span methodological refinements, architectural explorations, and deployment-focused optimizations.

Feature Relevance and Dimensionality Reduction

A targeted evaluation of feature importance can help streamline the input space and reduce model complexity. Techniques such as permutation importance or sparsity-inducing regularization may uncover redundant or low-utility features. Removing these features could not only enhance model interpretability but also reduce inference latency and memory requirements, an important factor for onboard deployment.

Confidence-Based Thresholding

The current formulation treats fault detection as a hard classification task, with binary outputs indicating fault presence or absence. However, the *softmax* confidence scores generated by the models provide a natural opportunity for threshold-based refinement. Rather than relying solely on maximum-class selection, future iterations could introduce confidence-based thresholds to modulate fault activation. Such thresholds would allow predictions to be deferred or suppressed when the network expresses low certainty—mitigating false positives in ambiguous regions of the feature space.

Hybrid and Hierarchical Architectures

Certain fault types—such as sensor freezes or slow valves—pose persistent challenges. Future iterations may benefit from hybrid approaches that combine recurrent and convolutional layers, employ multi-resolution representations, or utilize auxiliary tasks (e.g., signal change detection) to provide additional learning signals. Such architectures may facilitate earlier and more reliable classification under ambiguous conditions.

Scaling and Data-Efficiency Analysis

Although the dataset used in this thesis is extensive, no formal study has been conducted on how model performance scales with dataset size. Investigating this relationship will clarify how much data is necessary to achieve reliable generalization for the current model families. This can help identify the point of diminishing returns or inform minimum data requirements for retraining under updated system configurations.

Forecasting and Early-Warning Capability

The original research plan included a forecasting component, with the goal of enabling proactive fault mitigation. Implementing such capability will require both architectural extensions (e.g., sequence-to-sequence models or temporal attention mechanisms) and curated datasets that contain extended fault ramp-up phases. Introducing this temporal foresight remains an important milestone for future development, particularly in time-critical operational contexts.

Bibliography

[1] PricewaterhouseCoopers. *Space Industry Trends*. Oct. 2023. URL: https://www.pwc.com/us/en/industries/industrial-products/library/space-industry-trends.html (cit. on p. 1).

[2] L. A. Fernández, C. Wiedemann, and V. Braun. "Analysis of Space Launch Vehicle Failures and Post-Mission Disposal Statistics". In: *Aerotecnica Missili & Spazio* 101.3 (2022), pp. 243–259. URL: https://link.springer.com/article/10.1007/s42496-022-00118-5 (cit. on pp. 1, 2, 39).

[3] B. Yu, Y. Peng, J. Huang, and F. Lu. "LSTM-Fuzzy Logic for Fault Forecasting and Mitigation in Propulsion System". In: *IEEE Transactions on Aerospace and Electronic Systems* (2024) (cit. on p. 2).

[4] M. M. Quamar and A. Nasir. "Review on Fault Diagnosis and Fault-Tolerant Control Scheme for Robotic Manipulators: Recent Advances in AI, Machine Learning, and Digital Twin". In: *arXiv preprint* arXiv:2402.02980 (2024). URL: https://arxiv.org/abs/2402.02980 (cit. on pp. 2, 29).

[5] J. Kayser. "Development and Implementation of a Data-Based Adaptive Control System for the Cold Gas Propelled Rocket Hopper". Master's Thesis. Braunschweig, Germany: Technische Universität Braunschweig, Institute of Space Systems, Nov. 2024 (cit. on p. 2).

[6] M. Arps. *System Identification for Control Applications of Pump Fed Rocket Engines*. Bachelor's Thesis. Würzburg, Germany, Oct. 2021 (cit. on p. 2).

[7] L. Biddle and S. Fallah. "A Novel Fault Detection, Identification and Prediction Approach for Autonomous Vehicle Controllers Using SVM". In: *Automotive Innovation* 4.3 (2021), pp. 301–314. URL: https://doi.org/10.1007/s42154-021-00138-0 (cit. on pp. 5, 6).

[8] R. Karjian. *The history of artificial intelligence: Complete AI timeline*. https://www.techtarget.com/whatis/feature/The-history-of-artificial-intelligence-Complete-AI-timeline. 2024 (cit. on pp. 8, 9).

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*. ACM, 2019, pp. 2623–2631 (cit. on pp. 10, 60).

[10] J. Howard and S. Gugger. *Deep Learning for Coders with fastai and PyTorch: AI Applications Without a PhD*. O'Reilly Media, 2020. ISBN: 9781492045526 (cit. on pp. 10, 11, 20, 21, 23, 26).

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer, 2006 (cit. on pp. 11, 15).

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL: https://www.deeplearningbook.org/ (cit. on pp. 12, 21, 24–26).

[13] F. Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *Journal of Machine Learning Research*. Vol. 12. 2011, pp. 2825–2830 (cit. on p. 12).

[14] IBM. *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?* URL: https://www.ibm.com/think/topics/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks (cit. on pp. 13, 14, 20).

[15] IBM. *Supervised Learning*. URL: https://www.ibm.com/think/topics/supervised-learning (cit. on p. 13).

[16] IBM. *Supervised vs. Unsupervised Learning: What's the Difference?* URL: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning (cit. on p. 13).

[17] D. Bergmann. *What Is Semi-Supervised Learning?* URL: https://www.ibm.com/think/topics/semi-supervised-learning (cit. on p. 14).

[18] P. Domingos. “A Few Useful Things to Know About Machine Learning”. In: *Communications of the ACM* 55.10 (2012), pp. 78–87 (cit. on p. 14).

[19] N. Md Nor, C. R. Che Hassan, and M. A. Hussain. “A review of data-driven fault detection and diagnosis methods: Applications in chemical process systems”. In: *Reviews in Chemical Engineering* 36.4 (2019), pp. 513–553 (cit. on p. 14).

[20] [. Name(s)]. “Machine Learning Approaches for Fault Detection in Internal Combustion Engines: A Review and Experimental Investigation”. In: *Informatics* 12.1 (2025), p. 25. URL: https://www.mdpi.com/2227-9709/12/1/25 (cit. on pp. 16, 17, 19).

[21] L. Breiman. “Random Forests”. In: *Machine Learning* 45.1 (2001), pp. 5–32 (cit. on p. 16).

[22] G. Hu, T. Zhou, and Q. Liu. “Data-Driven Machine Learning for Fault Detection and Diagnosis in Nuclear Power Plants: A Review”. In: *Frontiers in Energy Research* 9 (2021), p. 663296. URL: https://www.frontiersin.org/articles/10.3389/fenrg.2021.663296/full (cit. on p. 18).

[23] T. Chen and C. Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794. URL: https://doi.org/10.1145/2939672.2939785 (cit. on p. 19).

[24] T. Wang, L. Ding, and H. Yu. “Research and Development of Fault Diagnosis Methods for Liquid Rocket Engines”. In: *Aerospace* (2022). URL: https://www.mdpi.com/2226-4310/9/9/481 (cit. on pp. 20, 29, 30, 32).

[25] A. Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. Blog post. May 2015. URL: URL:%20%5Curl%7Bhttp://karpathy.github.io/2015/05/21/rnn-effectiveness/%7D (cit. on p. 21).

[26] S.-H. Noh. “Analysis of Gradient Vanishing of RNNs and Performance Comparison”. In: *Information* 12.11 (2021). ISSN: 2078-2489. URL: https://www.mdpi.com/2078-2489/12/11/442 (cit. on p. 22).

[27] D. Bergmann and C. Stryker. *What is Backpropagation?* 2024 (cit. on pp. 23, 25).

[28] O. Elharrouss, Y. Mahmood, Y. Bechqito, M. A. Serhani, E. Badidi, J. Riffi, and H. Tairi. *Loss Functions in Deep Learning: A Comprehensive Review*. 2025. arXiv: 2504.04242 [cs.LG]. URL: https://arxiv.org/abs/2504.04242 (cit. on p. 24).

[29] V. R. Babu. “A Gentle Introduction to Deep Learning Using CUDA”. In: *IEEE Potentials* 36.6 (Nov. 2017), pp. 18–23 (cit. on p. 26).

[30] A. Duyar, T.-H. Guo, W. Merrill, and J. Musgrave. *Implementation of a Model-Based Fault Detection and Diagnosis Technique for Actuation Faults of the Space Shuttle Main Engine*. Tech. rep. NASA-TM-105781. NASA Lewis Research Center, 1991 (cit. on pp. 30, 31).

[31] J. Musgrave, T.-H. Guo, W. Merrill, and A. Duyar. *Real-Time Accommodation of Actuator Faults on a Reusable Rocket Engine*. Tech. rep. NASA Lewis Research Center, 1997 (cit. on p. 30).

[32] J. Wu. “Liquid-Propellant Rocket Engines Health-Monitoring — a survey”. In: *Acta Astronautica* (2004) (cit. on pp. 30, 32).

[33] P. Frank. “Analytical and Qualitative Model-Based Fault Diagnosis – A Survey and Some New Results”. In: *European Journal of Control* (1996) (cit. on pp. 30, 32).

[34] Y. Zhang and J. Jiang. “Bibliographical review on reconfigurable fault-tolerant control systems”. In: *Annual Reviews in Control* (2008). URL: https://doi.org/10.1016/j.arcontrol.2008.03.008 (cit. on p. 31).

[35] K. Kawatsu, S. Tsutsumi, M. Hirabayashi, and D. Sato. “Model-based Fault Diagnostics in an Electromechanical Actuator of Reusable Liquid Rocket Engine”. In: *American Institute of Aeronautics and Astronautics* (2020) (cit. on p. 31).

[36] S. Ye, X. Chen, and C. Xiong. “Thrust Fault Diagnosis of Launch Vehicle Engine Based on Linear-Quadratic Receding Horizon Algorithm”. en. In: *Astronaut* (2020), pp. 29–37 (cit. on p. 31).

[37] E. Kurudzija, K. Dresia, J. Martin, T. Traudt, J. Deeken, and G. Waxenegger-Wilfing. “Virtual Sensing for Fault Detection within the LUMEN Fuel Turbopump Test Campaign”. In: *Space Propulsion Conference 2024* (2024) (cit. on pp. 31, 39).

[38] P. Huang, T. Wang, L. Ding, H. Yu, Y. Tang, and D. Zhou. “Comparative Analysis of Real-Time Fault Detection Methods Based on Certain Artificial Intelligent Algorithms for a Hydrogen–Oxygen Rocket Engine”. In: *Aerospace* 9.10 (2022), p. 582. URL: https://www.mdpi.com/2226-4310/9/10/582 (cit. on p. 33).

[39] P. Huang, H. Yu, and T. Wang. “A Study Using Optimized LSSVR for Real-Time Fault Detection of Liquid Rocket Engine”. In: *Processes* 10.8 (2022), p. 1643. URL: https://www.mdpi.com/2227-9717/10/8/1643 (cit. on p. 33).

[40] J. Flora and D. J. Auxillia. “Sensor failure management in liquid rocket engine using artificial neural network”. In: *NISCAIR-CSIR* (2020) (cit. on p. 33).

[41] H. Yu and T. Wang. “A Method for Real-Time Fault Detection of Liquid Rocket Engine Based on Adaptive Genetic Algorithm Optimizing Back Propagation Neural Network”. In: *Sensors* 21.15 (2021), p. 5026. URL: https://www.mdpi.com/1424-8220/21/15/5026 (cit. on p. 33).

[42] X. Zhu, Y. Cheng, J. Wu, R. Hu, and X. Cui. “Steady-state process fault detection for liquid rocket engines based on convolutional auto-encoder and one-class support vector machine”. In: *Ieee Access* 8 (2019), pp. 3144–3158 (cit. on p. 33).

[43] H. Yan, Z. Liu, J. Chen, Y. Feng, and J. Wang. "Memory-augmented skip-connected autoencoder for unsupervised anomaly detection of rocket engines with multi-source fusion". In: *ISA Transactions* 133 (2023), pp. 53–65. ISSN: 0019-0578. URL: https://www.sciencedirect.com/science/article/pii/S0019057822003688 (cit. on p. 34).

[44] K. Dresia, E. Kurudzija, G. Waxenegger-Wilfing, H. Behler, D. Auer, K. Fröhlke, H. Neumann, A. Frank, J. Laurent, and L. Fabreguettes. "Automation of Testing and Fault Detection for Rocket Engine Test Facilities with Machine Learning". In: *International Journal of Energetic Materials and Chemical Propulsion* 22.6 (2023), pp. 17–33 (cit. on pp. 34, 39, 59).

[45] A. Lemke. *Unsupervised Anomaly Detection for Rocket Engine Test Facilities*. Bachelor thesis, Faculty of Mathematics and Computer Science, in collaboration with the German Aerospace Center (DLR). 2023. URL: https://nbn-resolving.org/urn:nbn:de:bvb:20-opus-325991 (cit. on pp. 34, 59).

[46] S.-Y. Park and J. Ahn. "Deep neural network approach for fault detection and diagnosis during startup transient of liquid-propellant rocket engine". In: *Acta Astronautica* 177 (2020), pp. 714–730. ISSN: 0094-5765. URL: https://www.sciencedirect.com/science/article/pii/S0094576520305166 (cit. on p. 34).

[47] X. Zhang, X. Hua, J. Zhu, and M. Ma. "Intelligent Fault Diagnosis of Liquid Rocket Engine via Interpretable LSTM with Multisensory Data". In: *Sensors* 23.12 (2023). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/23/12/5636 (cit. on p. 34).

[48] S. Singh. *Failure Detection and Identification for a Liquid Rocket Propulsion System Using Machine Learning Algorithms*. Presentation, Technische Universität München — TUM School of Engineering and Design, Chair of Space Mobility and Propulsion. Munich, Germany, 2025 (cit. on pp. 39, 40).

[49] J. Wu. "Liquid-propellant rocket engines health-monitoring—a survey". In: *Acta Astronautica* 56.2 (2005), pp. 347–356 (cit. on p. 40).

[50] K. Sumak. "Data Handling Done Right: Quick Tips for Large CSVs with Pandas". In: *Medium* (2021). URL: https://medium.com/@sumakbn/data-handling-done-right-quick-tips-for-large-csvs-with-pandas-21c25f91380f (cit. on p. 47).

[51] M. AI. *PyTorch*. https://pytorch.org/ (cit. on p. 48).

[52] Google. *TensorFlow*. URL: https://www.tensorflow.org/ (cit. on p. 48).

[53] SoftwareMill. *ML Engineer's Comparison of PyTorch, TensorFlow, JAX, and Flax*. 2024. URL: https://softwaremill.com/ml-engineer-comparison-of-pytorch-tensorflow-jax-and-flax/?utm_source=chatgpt.com (cit. on p. 49).

[54] A. Krogh and J. A. Hertz. "A Simple Weight Decay Can Improve Generalization". In: *Advances in Neural Information Processing Systems* 4 (1992), pp. 950–957 (cit. on p. 50).

[55] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on p. 50).

[56] S. M. Hosseini et al. "Dilated Balanced Cross Entropy Loss for Medical Image Segmentation". In: *arXiv preprint arXiv:2412.06045* (2023) (cit. on p. 51).

[57] A. Yadav. *Implementing Focal Loss in PyTorch for Class Imbalance*. `https://medium.com/data-scientists-diary/implementing-focal-loss-in-pytorch-for-class-imbalance-24d8aa3b59d9`. 2025 (cit. on p. 52).

[58] J. Xu, X. Ren, S. Lin, X. Sun, and W. Ma. "Understanding and Improving Layer Normalization". In: *Advances in Neural Information Processing Systems* 32 (2019) (cit. on p. 53).

[59] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. Vol. 37. PMLR, 2015, pp. 448–456 (cit. on p. 53).

[60] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *International Conference on Learning Representations (ICLR)*. 2017 (cit. on p. 53).