TU Clausthal
Clausthal University of Technology

MASTER THESIS

Explainable AI for Deep Learning-Based Fault Detection and Diagnosis of Automotive Software Systems During the Real-Time Validation Process

Ghannoum, Ehab

Matriculation Number: 539074

MSc Informatik

Erstgutachter:	PD Dr. Christoph Knieke
Zweitgutachter:	Dr. Stefan Wittek
Betreuer:	Dr. Mohammad Abboush

Technische Universität Clausthal - Clausthal University of Technology, Germany

January 15, 2025

Declaration of Authorship

I have read and understood the guidelines of the Clausthal University of Technology. I confirm that, I have prepared this master thesis independently by myself. Any information taken from other sources and being reproduced in this thesis is clearly referenced.

In terms of the general examination regulations, this work has not yet been submitted to any other examination division.

I hereby agree that my master thesis may be exhibited in the institute's or university library and kept for inspection.

Clausthal-Zellerfeld, _____ _____

Location, Date First name and family name

Abstract

The evolution of Automotive Software Systems (ASSs) has benefited from rapid technological advancements, improving functionality while increasing the risk of failures, which raises significant safety concerns. The growing complexity of modern ASSs necessitates rigorous testing to ensure safety and reliability. However, traditional testing methods often fail to address these challenges effectively.

The ISO 26262 standard, a key framework for functional safety in road vehicles, emphasizes Hardware-in-the-Loop (HIL) testing and real-time driving to validate ASSs. HIL testing simulates real-world hardware-software interactions in a controlled environment, facilitating fault identification. However, the vast and complex data generated during real-time driving and HIL testing of physical systems often surpasses the capacity of traditional expert-driven analysis. This highlights the need for data-driven approaches that can efficiently analyze historical test data to accurately detect and classify faults.

Artificial intelligence (AI)-based fault detection and diagnosis offers high performance, but current AI models often operate as "black boxes," making their decision-making processes difficult to interpret. This lack of transparency limits trust and hinders the adoption of AI in critical safety applications. Explainable AI (XAI) addresses this challenge by offering insights into how models reach conclusions, thereby enhancing trust, accountability, and error correction.

This study presents two hybrid models for classifying fault types and locations (single and concurrent) within ASSs. The proposed models combine Convolutional Neural Networks (CNNs) for feature extraction with Gated Recurrent Units (GRUs) to capture temporal dependencies. The Fault Location Model (FLM) achieved an accuracy of 97.40%, with matching recall, precision, and F1 scores of 97.40%. Similarly, the Fault Type Model (FTM) achieved an accuracy of 97.19%, with recall, precision, and F1 scores of 97.21%, 97.22%, and 97.21%, respectively.

XAI techniques—Integrated Gradients (IGs), DeepLIFT, Gradient SHAP, and DeepLIFT SHAP—were used to calculate Global Feature Importance (GFI), Per-Class Feature Importance (PCFI), and Feature Interactions (FIs). GFI was leveraged to reduce the feature sets from 24 to 10 for both the fault type and fault location classification models, resulting in minimal accuracy losses of 2.74% and 1.78%, respectively. This demonstrates the potential of XAI to simplify complex models while maintaining high performance. Additionally, PCFI identified critical features associated with specific fault types and locations, uncovering distinctive patterns and enabling more interpretable, targeted diagnostics. This improved the overall trust and usability of the models. Finally, FIs highlighted the significance of synergistic relationships between features, revealing that fault detection and diagnosis often depend on complex FIs rather than isolated features.

The time consumption of different XAI techniques, as well as baseline models, the originally developed models, and the retrained models, was analyzed. The analysis revealed that DeepLIFT was generally the fastest attribution method, while DeepLIFT SHAP was the slowest. The computation speed of the attribution method increased as the model complexity decreased.

Zusammenfassung

Die Entwicklung von Automobilsoftware-Systemen (ASS) hat von rasanten technologischen Fortschritten profitiert, die die Funktionalität verbessern, während sie gleichzeitig das Risiko von Ausfällen erhöhen, was erhebliche Sicherheitsbedenken aufwirft. Die zunehmende Komplexität moderner ASS erfordert rigorose Tests, um die Sicherheit und Zuverlässigkeit zu gewährleisten. Traditionelle Testmethoden sind jedoch oft nicht in der Lage, diese Herausforderungen effektiv zu bewältigen.
Die ISO 26262-Norm, ein zentrales Rahmenwerk für funktionale Sicherheit in Straßenfahrzeugen, betont Hardware-in-the-Loop (HIL)-Tests und Echtzeitfahrten zur Validierung von ASS. HIL-Tests simulieren die Interaktionen von Hardware und Software in einer kontrollierten Umgebung und erleichtern die Fehleridentifikation. Allerdings übersteigt die riesige und komplexe Datenmenge, die während der Echtzeitfahrten und HIL-Tests von physischen Systemen erzeugt wird, oft die Kapazität der traditionellen, von Experten gesteuerten Analyse. Dies verdeutlicht die Notwendigkeit datengetriebener Ansätze, die historische Testdaten effizient analysieren können, um Fehler genau zu erkennen und zu klassifizieren.
Fehlererkennung und -diagnose auf Basis von Künstlicher Intelligenz (KI) bieten eine hohe Leistung, aber aktuelle KI-Modelle arbeiten oft als „Black Boxes“, wodurch ihre Entscheidungsprozesse schwer verständlich sind. Dieser Mangel an Transparenz schränkt das Vertrauen ein und behindert die Einführung von KI in sicherheitskritischen Anwendungen. Erklärbare KI (XAI) adressiert diese Herausforderung, indem sie Einblicke in die Entscheidungsfindung der Modelle bietet, wodurch Vertrauen, Verantwortung und Fehlerkorrektur gestärkt werden.
Diese Studie stellt zwei hybride Modelle zur Klassifizierung von Fehlerarten und -orten (einzelne und gleichzeitige Fehler) innerhalb von ASS vor. Die vorgeschlagenen Modelle kombinieren Convolutional Neural Networks (CNNs) zur Merkmalsextraktion mit Gated Recurrent Units (GRUs), um zeitliche Abhängigkeiten zu erfassen. Das Fault Location Model (FLM) erzielte eine Genauigkeit von 97,40 %, mit übereinstimmenden Recall-, Präzisions- und F1-Scores von 97,40 %. Ebenso erzielte das Fault Type Model (FTM) eine Genauigkeit von 97,19 %, mit Recall-, Präzisions- und F1-Scores von 97,21 %, 97,22 % und 97,21 %, jeweils.
XAI-Techniken – Integrated Gradients (IGs), DeepLIFT, Gradient SHAP und DeepLIFT SHAP – wurden verwendet, um die Global Feature Importance (GFI), die Per-Class Feature Importance (PCFI) und die Feature Interactions (FIs) zu berechnen. GFI wurde genutzt, um die Merkmalsmengen sowohl für das Modell zur Fehlerartklassifikation als auch für das Modell zur Fehlerortklassifikation von 24 auf 10 zu reduzieren, was zu minimalen Genauigkeitsverlusten von 2,74 % bzw. 1,78 % führte. Dies zeigt das Potenzial von XAI, komplexe Modelle zu vereinfachen und gleichzeitig eine hohe Leistung aufrechtzuerhalten. Darüber hinaus identifizierte PCFI wichtige

Merkmale, die mit bestimmten Fehlerarten und -orten in Verbindung standen, und deckte markante Muster auf, die gezieltere und besser interpretierbare Diagnosen ermöglichten. Dies verbesserte das Vertrauen und die Benutzerfreundlichkeit der Modelle insgesamt. Schließlich hob FIs die Bedeutung synergetischer Beziehungen zwischen den Merkmalen hervor und zeigte, dass Fehlererkennung und -diagnose oft von komplexen Merkmalsinteraktionen abhängen, anstatt von isolierten Merkmalen. Der Zeitaufwand der verschiedenen XAI-Techniken sowie der Baseline-Modelle, der ursprünglich entwickelten Modelle und der nachtrainierten Modelle wurde analysiert. Die Analyse ergab, dass DeepLIFT allgemein die schnellste Attributionsmethode war, während DeepLIFT SHAP die langsamste war. Die Berechnungsgeschwindigkeit der Attributionsmethode nahm zu, je geringer die Komplexität des Modells war.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

The increasing complexity of ASSs in modern vehicles, which depend on interconnected Electronic Control Units (ECUs), necessitates efficient fault detection, diagnosis, and a comprehensive real-time validation process. These systems are prone to faults arising from intricate hardware-software interactions, and accurate diagnostics are crucial to prevent hazardous failures.

Traditional manual methods struggle to keep up with the vast amounts of data generated during validation processes, such as HIL testing with real physical systems, prompting the adoption of AI-driven diagnostic techniques.

XAI addresses transparency issues in AI by making decision-making processes more interpretable and enhancing trust by clarifying the dependencies between features and their impact on fault detection and diagnosis. Additionally, XAI facilitates the analysis of incorrect decisions made by traditional Deep Learning (DL) models.

XAI enhances model transparency, interpretability, and trust by identifying critical features, which guide optimization and improve decision-making. To achieve this, techniques such as IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP were utilized.

These techniques provided insights into feature dynamics: GFI assessed the overall significance of features, highlighting their impact on model performance, PCFI identified fault-specific features, and FIs revealed complex relationships between features.

Integrating XAI into diagnostic systems enables precise fault type classification and accurate fault location identification, both critical for safety-critical systems like Advanced Driver Assistance Systems (ADAS). XAI enhances the robustness, reliability, and transparency of diagnostics, ensuring compliance with safety standards and advancing the safety of next-generation vehicles.

1.2 Research Questions

This master's thesis addresses the following research questions:

- **RQ1:** How can AI algorithms accurately classify both single and concurrent fault types and fault locations in ASSs using real-time data from HIL tests?
- **RQ2:** How can XAI enhance fault detection and diagnosis in ASSs by improving model reliability, interpretability, and efficiency, without compromising performance?
- **RQ3:** What are the trade-offs between model complexity and interpretability in achieving effective fault detection and diagnosis for ASSs?

1.3 Related Work

This chapter reviews research pertinent to this study, emphasizing the increasing integration of XAI techniques with AI models for fault detection and diagnosis across diverse fields.

1.3.1 Scientific Review

Sinha et al. [77] introduce the XAI-LCS method, designed for fault diagnosis in IoT networks using low-cost sensors. By leveraging the eXtreme gradient boosting (XGBoost) algorithm, the method achieves an impressive validation accuracy of 99.8% in detecting four distinct sensor faults. It effectively addresses imbalanced data distributions, preventing biased predictions. To enhance model transparency, SHapley Additive exPlanations (SHAP) is used to interpret XGBoost outcomes, supporting comprehensible and informed decision-making. The approach is demonstrated in real-time sensor health monitoring, emphasizing the importance of early fault detection in ensuring uninterrupted IoT services.

Liang et al. [76] propose a hybrid model designed to distinguish between sensor and system faults, focusing on accuracy and reliability for practical deployment. Their methodology integrates CNNs, XGBoost, and Denoising Autoencoders (DAE) to tackle the challenges of imbalanced data and prediction bias. The model involves data collection, CNN-based feature extraction, XGBoost-assisted fault isolation, DAE-driven fault correction, and a hybrid CNN-XGBoost system for fault detection, with XAI for interpretation. Notable results include the successful detection of sensor and system faults, achieving a 99.77% accuracy rate in system fault detection and a Root Mean Square Error (RMSE) of 0.0576 for sensor fault data reconstruction.

Sinha et al. [75] present a fault detection technique for bearing faults, combining ML with XAI to improve model transparency and trust. The study introduces a gradient boosting-based technique for bearing fault detection and applies SHAP to interpret the model's outcomes. The main challenge highlighted is the "black-box" nature of the CatBoost model, which limits full interpretability. SHAP evaluates feature contributions but emphasizes that feature dependencies are model-specific, not implying direct correlations.

Brito et al. [21] introduce FaultD-XAI, a novel approach for fault diagnosis in rotating machinery that incorporates synthetic data, Transfer Learning, and XAI. This method eliminates the need for labeled data for each fault, enhancing decision-making in fault diagnosis. The approach combines synthetic data generation with Transfer Learning for model training, focusing on predictive maintenance. However, the model's effectiveness is constrained by the need for a few real signals to adjust the data augmentation process and difficulties in modeling synthetic faults based on prior knowledge. FaultD-XAI integrates interpretability to ensure AI model outputs

are understandable, supporting more informed fault diagnosis.

Mey et al. [59] explore the use of XAI for vibration data analysis in rotating machinery fault detection. They create a synthetic dataset to evaluate XAI algorithms, comparing GradCAM, LRP, and modified LIME. The study achieves a 99.66% classification accuracy on real-world imbalance detection datasets. However, limitations include partial success in providing sample-specific explanations and identifying relevant features. The methodology involves deep neural networks, CNNs for classification, and XAI algorithms for interpreting vibration data, shedding light on the saliency values related to rotation speed.

Brusa et al. [11] tackle the challenge of interpreting ML models for industrial fault diagnosis using SHAP for feature selection and interpretation. Their application to bearing fault detection demonstrates that SHAP values are effective in identifying important features for accurate diagnosis. The study highlights four critical features, including skewness and shape factor, for machine fault detection in rotating machinery. SHAP is used to calculate feature importance in both regression and classification problems, providing insights into how each feature influences model predictions.

Hua et al. [53] aim to improve the attack detection performance of intrusion detection systems (IDSs) for IoT environments. They utilize ensemble tree methods, such as Decision Trees (DTs) and Random Forests (RF), achieving 100% accuracy and F1 score on large IoT-based IDSs datasets. SHAP is employed for both global and local model explanations, enhancing the trustworthiness of the predictions. The methodology enhances IDSs performance by improving detection rates and applying SHAP for model interpretation.

Brito et al. [10] propose a novel approach for fault detection in rotating machinery, integrating ML and DL. The methodology includes feature extraction, anomaly detection for fault detection, and fault diagnosis through unsupervised classification or root cause analysis. Their findings show that the model performs well for unsupervised fault detection without labeled training data. However, challenges remain, including variability in computational costs and the trade-off between response time and precision in explanation models. Despite these challenges, the methodology proves effective for unsupervised fault detection and classification.

Hasan et al. [38] introduce an XAI-based fault diagnosis model for bearings, emphasizing interpretability and generalization. The model involves preprocessing using the Fast Discrete Orthogonal Stockwell Transformation (FDOST), followed by feature extraction, selection, filtration, and classification using the K-Nearest Neighbor (kNN) algorithm. Demonstrated on two bearing test datasets, the model achieves 100% classification accuracy, showcasing its robustness. SHAP is used for model interpretation, enhancing explainability and providing insights into varying working conditions, demonstrating the model's potential in generalized fault diagnosis.

1.4 Review Summary

Key insights from the reviewed work include:

1. **Importance of XAI**: Numerous studies underscore the vital role of XAI in improving the trustworthiness and interpretability of ML models, particularly in fault diagnosis. Techniques such as SHAP are noted for their ability to provide clear explanations of model predictions and feature contributions, thereby promoting transparency in AI-driven decision-making.

2. **Hybrid Models for Enhanced Performance**: Hybrid models that integrate techniques such as CNNs, XGBoost, and DAE have shown strong performance in distinguishing between various fault types. These models enhance both accuracy and reliability, rendering them highly suitable for practical deployment in fault diagnosis applications

3. **Addressing Imbalanced Data**: Imbalanced data distributions pose a significant challenge in fault diagnosis. Strategies such as synthetic data generation, Transfer Learning, and ensemble methods are employed to address this issue, ensuring unbiased predictions and strengthening the robustness of fault detection systems.

4. **Practical Applicability**: The reviewed methodologies demonstrate substantial practical applicability in real-world scenarios, including real-time sensor health monitoring and fault detection in rotating machinery. These approaches underscore the critical role of early fault detection in ensuring uninterrupted services and facilitating predictive maintenance.

5. **Model Interpretability**: Model interpretability is essential for understanding the impact of extracted features on fault predictions and supporting more informed decision-making. Techniques such as SHAP provide meaningful insights into feature contributions, assisting in effective feature selection and improving the comprehensibility of ML models for engineers and operators.

The gaps identified in prior research are addressed in this master's thesis through the following approaches:

- Detects fault types and locations (both single and concurrent).
- Utilizes XAI for fault type and location detection and diagnosis.
- Employs four different XAI techniques.
- Relies on real-time data.

1.5 Solution

This study proposes an intelligent fault detection and diagnosis framework for ASSs leveraging supervised learning algorithms. The process begins by collecting comprehensive datasets encompassing both healthy and faulty data, which are critical for training and evaluating the models. These datasets are generated via a real-time fault injection framework simulating diverse fault types and locations within the system.

In a HIL simulation environment, provided by dSPACE, demonstrates the application of the framework. This simulation replicates software-hardware interactions, ensuring its relevance to the automotive industry. A case study employing a gasoline engine within a HIL simulation environment, provided by dSPACE, illustrates the framework's application. This simulation emulates software-hardware interactions, highlighting its relevance to the automotive industry.

The study tackles the challenges of detecting and diagnosing both single and concurrent faults, including the classification of fault types and fault locations critical to safety in automotive applications. To overcome these challenges, the framework incorporates XAI techniques, which enhance fault detection and diagnosis by elucidating the models' decision-making processes

Specifically, the study employs techniques such as IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP to perform GFI, PCFI, and FIs. While interpretability is not explicitly mandated by ISO 26262, XAI can support model verification, regulatory compliance, and bolster system reliability in safety-critical environments.

1.6 Objective

The primary objective of this master's thesis is to develop reliable AI models for fault detection and diagnosis in ASSs, leveraging real-time data from HIL testing. This thesis addresses the challenge of detecting and diagnosing both single and concurrent faults, including identifying fault types and locations in realistic, real-world scenarios.

A key objective is to leverage XAI techniques to enhance the transparency of AI-driven fault detection systems in ASSs. XAI helps engineers gain insights into model behavior, validate decision-making, and identify critical features. By revealing GFI, PCFI, and FIs, XAI improves model clarity and diagnostic capabilities. GFI highlights overall feature relevance, PCFI identifies features specific to fault types and locations, and FIA shows how feature combinations affect fault detection. Although ISO 26262 does not explicitly require interpretability, XAI fosters trust, aids in error correction, and supports compliance with safety standards, ensuring more accurate and reliable fault detection.

The effectiveness of the developed models will be assessed using standard perfor-

mance metrics, including accuracy, precision, recall, and F1 score. This evaluation ensures that the model meets reliability and safety benchmarks. By integrating advanced AI methods with explainability, this thesis contributes to developing trustworthy and efficient fault detection and diagnosis systems, advancing safer ASSs.

1.7 Structure

The master's thesis is structured into six chapters, each of which is briefly described below:

- **Chapter 1: Introduction** This chapter provides an overview of the thesis, starting with the context and motivation. It defines the problem statement and presents the research questions. A review of related work in fault detection and diagnosis, with a specific focus on XAI, highlights recent advances and challenges in integrating explainability into fault diagnosis systems. The chapter then summarizes key findings from the literature. The proposed solution and research objectives are outlined, and the structure of the document is described to assist in navigating the subsequent chapters.

- **Chapter 2: Background** This chapter introduces the key concepts and methodologies used in the thesis. It begins with an overview of Model-Based Development (MBD), followed by a discussion of fault injection techniques. A brief overview of fault detection and diagnosis methods is provided, along with a detailed exploration of ML paradigms, classification and regression techniques, and DL architectures. Furthermore, imbalance mitigation strategies are discussed. Lastly, the chapter introduces XAI and its key techniques.

- **Chapter 3: Methodology** This chapter outlines the methodology employed in the thesis, starting with data collection. It then details the data preprocessing steps, including cleaning, scaling, and time-series processing. The discussion progresses to model development, focusing on both the baseline and proposed models, along with their evaluation and tuning. Finally, the chapter concludes with a description of the integration of XAI techniques.

- **Chapter 4: Implementation** This chapter presents the gasoline engine case study, focusing on simulation models and key engine components. It includes data collection and preprocessing steps. The chapter also details the development and training of DL models, along with hyperparameter tuning. Finally, it explains how feature importance is assessed using the XAI techniques, covering GFI, PCFI, and FIs.

- **Chapter 5: Results and Discussion** This chapter outlines the evaluation metrics and applies them to assess the performance of the optimized DL models

using test data. Additionally, the chapter discusses the results, emphasizing the use of XAI techniques to interpret and validate the findings.

- **Chapter 6: Conclusion and Future Work** This chapter concludes the master's thesis and discusses potential directions for future research.

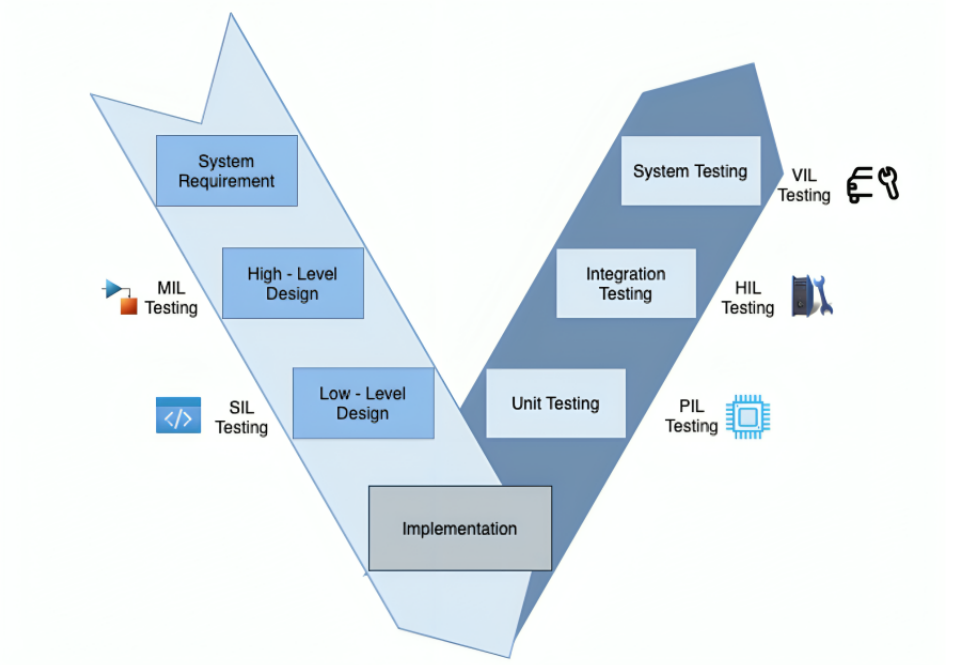
2 Background

This chapter introduces the key concepts and techniques used in this master's thesis, including MBD, simulation approaches, and fault injection methods. It covers ML paradigms, such as supervised and unsupervised learning, along with classification and regression techniques. The chapter also discusses DL architectures, data imbalance strategies, and concludes with an introduction to XAI.

2.1 Model-Based Development and Testing Phases

Model-Based Development (MBD) has become essential in the automotive industry for efficiently designing complex embedded software systems. By using mathematical or visual models, MBD facilitates early design validation and verification [27], as highlighted in numerous studies [6, 7, 66, 47, 70, 46, 64, 3]. A key advantage of MBD is the ability to automate code generation, which reduces human error and accelerates development. The generated code is tested through stages such as Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL), Hardware-in-the-Loop (HIL), and Vehicle-in-the-Loop (VIL), aligning with the V-model's design and verification phases, as shown in 1. This systematic approach enhances the reliability and efficiency of embedded automotive systems, supporting advances in vehicle technology and safety.

Figure 1: V-shaped MBD with Testing [1]



2.1.1 Model-in-the-Loop (MIL)

Model-in-the-Loop (MIL) is a key testing method in the development of embedded software, especially in the automotive and aerospace industries. In MIL, a mathematical or simulation model representing part or all of a system is tested within a virtual test environment to simulate real-world behaviors and responses. This technique ensures that the software functions and performs as expected in a controlled, virtual setting, allowing for early validation of design accuracy and functionality [73].

2.1.2 Software-in-the-Loop (SIL)

Software-in-the-Loop (SIL) is a critical testing method in embedded software development, widely used in the automotive, aerospace, and industrial automation industries. SIL testing allows the software to interact with a simulated model of the hardware it will control, enabling engineers to assess functionality, performance, and integration before deployment. By identifying issues like logic errors and timing problems early, SIL testing enhances reliability and safety while reducing costs associated with late-stage fixes [72].

2.1.3 Processor-in-the-Loop (PIL)

Processor-in-the-Loop (PIL) testing is used in the development of embedded software, particularly in automotive and aerospace industries. In PIL, the software runs on its target hardware processor within a simulated environment, allowing engineers to assess its functional correctness and performance under realistic conditions. PIL testing [73] identifies integration issues early, ensuring the software operates reliably on the actual hardware and enhancing the quality and safety of embedded systems in critical applications.

2.1.4 Hardware-in-the-Loop (HIL)

Hardware-in-the-Loop (HIL) testing is a vital method for developing and validating embedded systems, especially in the automotive, aerospace, and industrial automation industries. In HIL, software is tested alongside physical hardware components that replicate its real-world operational environment, including interactions with sensors, actuators, and controllers. This setup enables developers to assess software performance, functionality, and hardware integration early on, identifying potential issues related to timing and system dynamics. HIL testing ensures the reliability, safety, and robustness of embedded systems, meeting stringent industry standards.

2.1.5 Vehicle-in-the-Loop (VIL)

Vehicle-in-the-Loop (VIL) testing is a specialized method widely used in the automotive industry for developing and validating vehicle systems. In VIL testing, vehicle control software is evaluated within a virtual environment that incorporates a physical vehicle or a high-fidelity simulation model. This setup allows engineers to simulate real-world driving scenarios, including various road and weather conditions, while assessing the software's performance, safety, and functionality. VIL testing enables early detection of potential issues, ensuring reliable and safe operation in actual driving conditions. This approach accelerates development timelines, reduces costs associated with physical prototypes, and helps deliver vehicles that meet strict performance and safety standards.

2.2 Fault Injection Approachs

Fault injection is a validation technique used to assess a system's dependability by analyzing its behavior in the presence of faults [86]. This well-established technique has been in use for a long time and is strongly recommended by ISO 26262 across various stages of the V-model [1, 62]. The primary goal of fault injection is to evaluate how the system reacts when faults occur, which is done by observing the system's outcomes and comparing them with its specifications. To effectively implement fault injection, three key attributes are considered: fault time, fault location, and fault type. Fault time refers to when the fault is introduced, while fault location specifies where it is injected. Both fault types and fault locations are detailed in the following subsections.

2.2.1 Fault Types

2.2.1.1 Noise Fault

The term "noise fault" describes the existence of random fluctuations or interference in the sensor output that results in inaccurate measurements [68]. Electrical interference, environmental conditions, or limitations in sensor technology are all potential causes. Noise faults can induce uncertainty and reduce measurement accuracy [33].

2.2.1.2 Gain Fault

A gain failure occurs when the sensor's output is scaled improperly, deviating from the predicted correlation between the output and the measured quantity.

2.2.1.3 Offset/Bias Fault

An offset or bias fault refers to a bias or deviation in the sensor output [29]. In this type of fault, the output differs from the true values by a fixed amount.

Figure 2 compares the healthy signal with variations due to noise, gain, and offset. Table 1 provides the mathematical representations for each fault type, including the

healthy signal, gain, noise, and offset faults.

Figure 2: Healthy Signal vs. Variations: Noise, Gain, and Offset

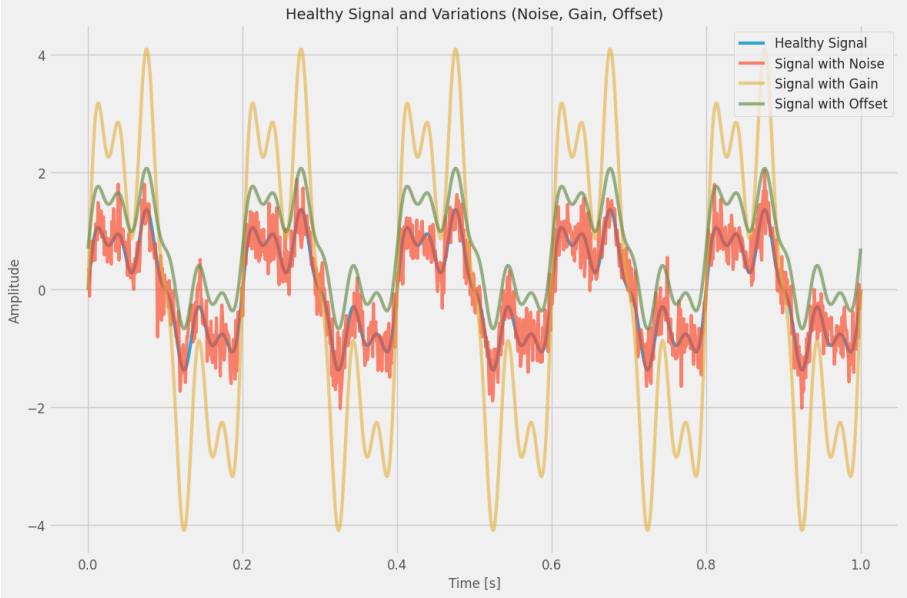


Table 1: Mathematical Representations of Signal Fault Types

Fault Type	Mathematical Representation
Healthy Signal	Healthy Signal(t) = True Value(t)
Gain Fault	Gain Fault Signal(t) = Gain $\times$ True Value(t)
Noise Fault	Noise Fault Signal(t) = True Value(t) + Noise(t)
Offset Fault	Offset Fault Signal(t) = True Value(t) + Offset

2.2.2 Fault Locations

2.2.2.1 Position of Accelerator Pedal

This refers to the position of the accelerator pedal and indicates how much the pedal is being pressed down. A value of 0% typically means the pedal is not pressed at all, while a value of 100% indicates the pedal is fully pressed down.

2.2.2.2 Position of Brake Pedal

This refers to the position of the brake pedal. Similar to the accelerator pedal position, it indicates how much the pedal is being pressed. A value of 0% typically means the pedal is not pressed at all, while a higher value indicates an increasing application of the brake.

2.2.2.3 Engine Speed (RPM)

This refers to the engine speed, measured in revolutions per minute (RPM). It indicates how many times the engine's crankshaft completes a full rotation within one minute. A low RPM value typically corresponds to idle or low engine power output,

while higher RPM values indicate greater engine activity, often during acceleration or high-speed driving.

2.3 Fault Detection and Diagnosis Methods

Fault detection and diagnosis in ASSs involve identifying and analyzing faults within the system. Fault detection refers to recognizing the presence of faults, while fault diagnosis focuses on determining the type and location of these faults. This process relies on data analysis techniques to accurately identify and classify faults, making troubleshooting and resolution more efficient. An intelligent model can assist testers in detecting and diagnosing faults by using methods that consider factors such as data availability and system characteristics.

2.3.1 Signal-based Methods

Signal-based methods analyze fault symptoms by comparing healthy and faulty signals using indicators such as frequency, amplitude, and ripple features [22]. These methods require expert knowledge and may produce similar symptoms for different faults, making them less suitable for classifying multiple fault types. They focus on signal properties like frequency content and noise levels, using techniques such as filtering and time-frequency analysis [34].

2.3.2 Model-based Methods

Model-based methods analyze system behavior and connections using mathematical or system models [22]. These methods include process history, qualitative, and quantitative approaches. While effective in simple systems, model-based methods are challenging for complex systems like ASSs, as developing a redundant mathematical model requires expert knowledge. The method compares the outputs of a real system and its parallel model with the same inputs. However, building such models for complex systems is difficult, making fault classification in ASSs a challenge [31].

2.3.3 Data-based Methods

Data-based methods use historical system data to train AI algorithms or statistical models to discover hidden patterns [32]. These methods provide accurate results without the need for expert knowledge when sufficient data is available. However, handling large datasets requires high computational power and storage capacity. With advancements in computer systems and cloud storage, managing large datasets is no longer a major challenge. Data-driven models, which capture typical behaviors using past sensor data, can detect anomalies by comparing real-time data to trained models. Thus, data-based methods are well-suited for fault detection and diagnosis, especially when large datasets are available.

2.4 Machine Learning Paradigms

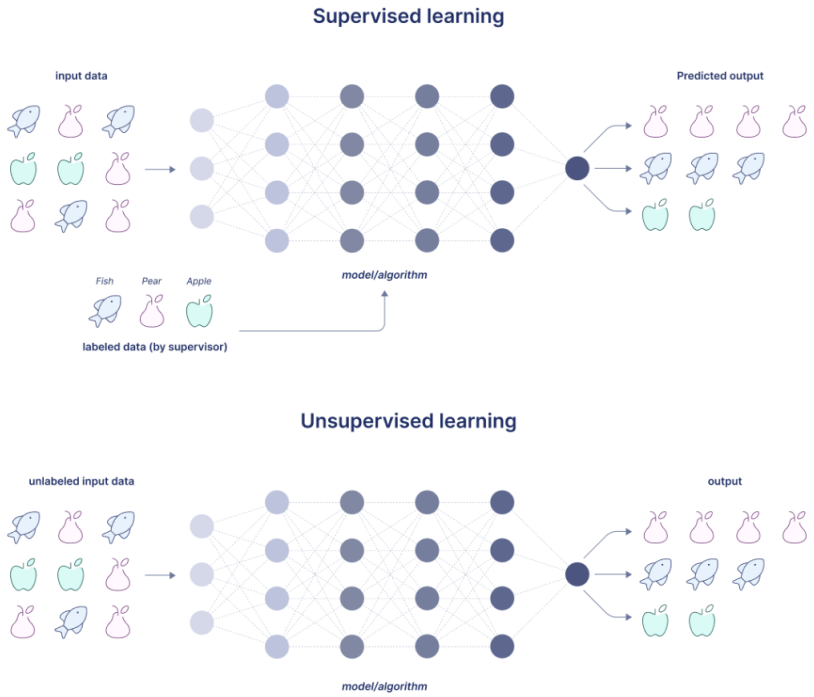
2.4.1 Supervised Learning

Supervised learning, as shown at the top of Figure 3, is a fundamental approach in ML where the model is trained using labeled data, meaning each input is paired with a corresponding output. During the training process, the model repeatedly makes predictions on the input data and compares these predictions to the actual labels. It then adjusts its internal parameters to minimize the error between its predictions and the correct outputs. This iterative process enables the model to learn patterns in the data, allowing it to make accurate predictions on new, unseen data. Common algorithms used in supervised learning include linear regression, logistic regression, DTs, support vector machines (SVMs), and neural networks [9, 63, 81, 24].

2.4.2 Unsupervised Learning

Unsupervised learning, as shown at the bottom of Figure 3, involves training a model on a dataset without labeled outputs. Instead of learning to predict specific outcomes, the model is tasked with discovering hidden patterns, groupings, or structures within the data. Since there are no predefined labels to guide the learning process, unsupervised learning models must rely on the intrinsic relationships within the data to identify meaningful insights. Algorithms commonly used in unsupervised learning include k-means clustering, hierarchical clustering, principal component analysis (PCA), and autoencoders, which are often employed in tasks such as data segmentation, dimensionality reduction, and anomaly detection [9, 63, 81].

Figure 3: Comparison of Supervised and Unsupervised Learning [71]



2.5 Classification and Regression Techniques

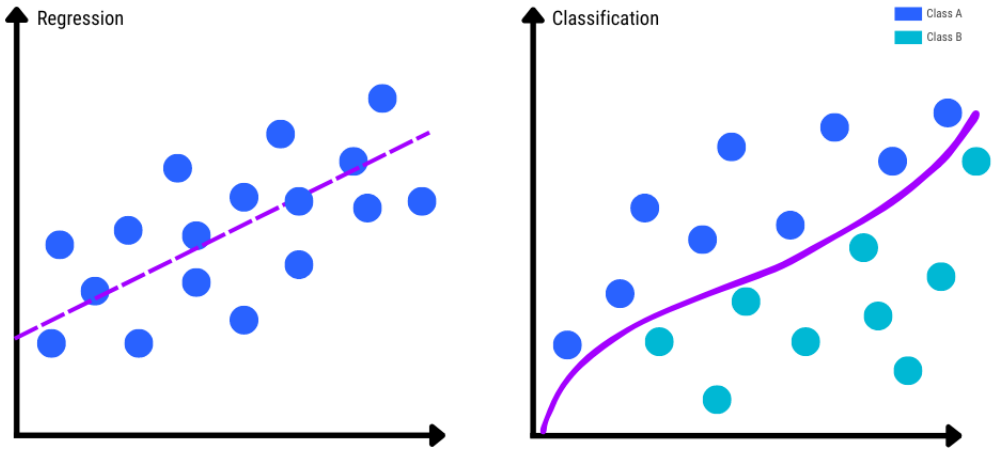
2.5.1 Regression

Regression, as shown on the left of Figure 4, is used when the goal is to predict a continuous outcome or numerical value based on input data. It models the relationship between a dependent variable (the outcome you want to predict) and one or more independent variables (the input features). The output of a regression model is typically a real number, such as predicting the price of a house, the temperature, or the amount of rainfall [9, 63].

2.5.2 Classification

Classification, as shown on the right of Figure 4, is used when the goal is to predict a categorical outcome, meaning the output falls into one of several predefined classes or categories. The model learns from the input data to categorize new data points into one of these classes. The output of a classification model is typically a label, such as "spam" or "not spam," "disease" or "no disease," and so on [9, 63].

Figure 4: Comparison of Regression and Classification [65]



2.5.2.1 Binary Classification Binary classification [35] is a type of classification task where the objective is to classify elements of a set into one of two distinct groups (or classes). These classes are often labeled as 0 and 1, but they can represent any two categories. The goal is to develop a model $f(\mathbf{x})$ that maps an input vector $\mathbf{x}$ to one of these two classes.

The model's prediction $\hat{y}$ can be described as follows:

$$\hat{y} = \begin{cases} 1 & \text{if } f(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where $f(\mathbf{x})$ is the output of the model, which could be a score, probability, or decision function that indicates the likelihood of the input belonging to class 1. The decision boundary, which separates the two classes, is defined by the condition $f(\mathbf{x}) = 0$.

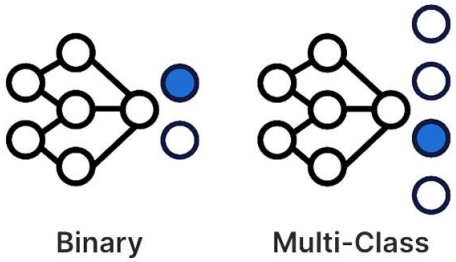
2.5.2.2 Multi-class Classification Multi-class classification [35] extends the idea of binary classification to situations where there are more than two classes. In this case, the objective is to classify an input vector $\mathbf{x}$ into one of K possible classes, where $K > 2$. The goal is to construct a model $f(\mathbf{x})$ that outputs the predicted class label $y \in \{1, 2, \ldots, K\}$.

The model computes a set of class probabilities or scores, one for each of the K possible classes. The predicted class $\hat{y}$ is the class with the highest probability or score:

$$\hat{y} = \arg\max_k f_k(\mathbf{x})$$

Where $f_k(\mathbf{x})$ is the score or probability associated with class k. The model chooses the class that has the highest value among all possible classes.

Figure 5: Comparison of Binary and Multi-class Classification [43]



2.6 Deep Learning Architectures

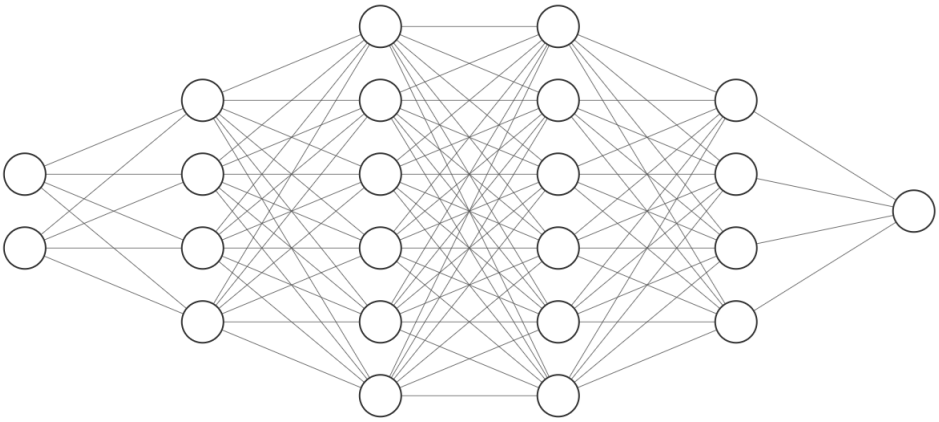
2.6.1 Feedforward Neural Networks (FNNs)

Feedforward Neural Networks (FNNs), also known as Multi-layer Perceptrons (MLPs), are one of the simplest and most widely used types of neural networks. They are called 'feedforward' because the information flows in one direction, from the input layer, through one or more hidden layers, and finally to the output layer, without any cycles or loops in the network, as illustrated in Figure 6 [35, 40, 67, 9, 54].
FNNs consist of the following components:

- **Input Layer:** This is where the network receives the input data, which can represent anything depending on the task, such as images, numerical data, or text features.
- **Hidden Layers:** These layers contain neurons that process the inputs by applying weights, biases, and an activation function. The activation function introduces non-linearity, allowing the network to capture complex patterns in the data.
- **Output Layer:** The final layer produces the output of the network, such as a predicted class in classification tasks or a numeric value in regression tasks.

FNNs learn by adjusting the weights and biases during training, which is done through a process called backpropagation. In backpropagation, the network's errors are calculated and fed back through the layers to optimize the weights, improving the model's accuracy.

Figure 6: Architecture of a FNN



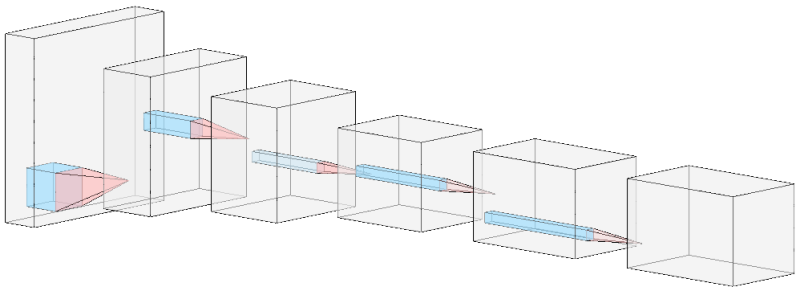
2.6.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized type of neural network primarily designed for processing structured grid data, such as images and, more recently, time series data. CNNs excel in identifying patterns and features within the data by applying convolutional operations, making them highly effective for various tasks, as illustrated in Figure 7 [55, 50, 35, 83].
CNNs consist of the following components:

- **Input Layer:** The input layer receives time series data, which can be represented as a sequence of values over time. This data may come from sensors, stock prices, or any other continuous measurements.
- **Convolutional Layers:** These layers apply convolutional filters to the input data, scanning through the time dimension to extract local features and patterns. Each filter learns to identify specific characteristics, such as trends or spikes in the data.
- **Pooling Layers:** Pooling layers downsample the feature maps generated by convolutional layers, summarizing local regions to reduce dimensionality and maintain important information while making the model invariant to small translations in the input data.
- **Stride:** The stride determines the step size with which the filter moves across the time series data. A larger stride results in a smaller output size and can reduce computational complexity, while a smaller stride provides more detailed feature extraction.

CNNs learn by adjusting the filters and weights during training. This is typically achieved through backpropagation, where the network learns from its errors to improve accuracy over time.

Figure 7: Architecture of a CNN



2.6.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs), as shown on the left in Figure 8, are specialized neural networks designed to process sequential data by leveraging memory through recurrent connections. Unlike Feedforward Neural Networks (FFNs), where inputs are treated as independent entities, RNNs connect inputs across time steps, enabling them to capture temporal dependencies and maintain context throughout a sequence [45, 28, 44, 35, 17].
An RNN unit consists of the following components:

- **Input Sequence:** RNNs receive a sequence of inputs over time, allowing them to handle various types of sequential data, such as time series, text, and audio signals.
- **Hidden State:** At each time step, the RNN maintains a hidden state that incorporates both the current input and the previous hidden state. This structure allows the network to remember information from earlier time steps, enabling it to capture patterns and dependencies over time.
- **Memory and Context:** RNNs effectively remember and leverage context from prior inputs, making them suitable for tasks that require an understanding of sequence relationships. This capability is essential for applications such as natural language processing, where the meaning of a word can depend on the words that came before it.

RNNs learn from data through a process called backpropagation through time (BPTT), allowing them to adjust weights and improve their predictions based on the temporal relationships in the training data.

2.6.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM), as shown in the middle of Figure 8, is a specialized type of Recurrent Neural Network (RNN) designed to effectively capture long-term dependencies in sequential data. They address the vanishing gradient

problem commonly encountered in traditional RNNs by incorporating gating mechanisms that regulate the flow of information [44, 36, 35, 19].
An LSTM unit consists of the following components:

- **Memory Cells:** LSTM utilizes memory cells that maintain information over long periods, allowing the network to carry important context through many time steps. This capability is crucial for tasks that require understanding temporal relationships, such as language processing or time series forecasting.
- **Gates:** The flow of information within an LSTM is controlled by three types of gates:
  - **Forget Gate:** Decides what information from the previous cell state should be discarded, enabling the network to forget irrelevant data.
  - **Input Gate:** Determines which new information should be added to the cell state, allowing the network to update its memory with relevant data.
  - **Output Gate:** Controls what information from the cell state is passed on to the hidden state, impacting the output of the network.

LSTMs process input sequences step-by-step, updating their memory and hidden states at each time step. This mechanism enables them to learn from previous inputs, making them particularly effective for sequential tasks where context is essential.

2.6.5 Gated Recurrent Unit (GRU)

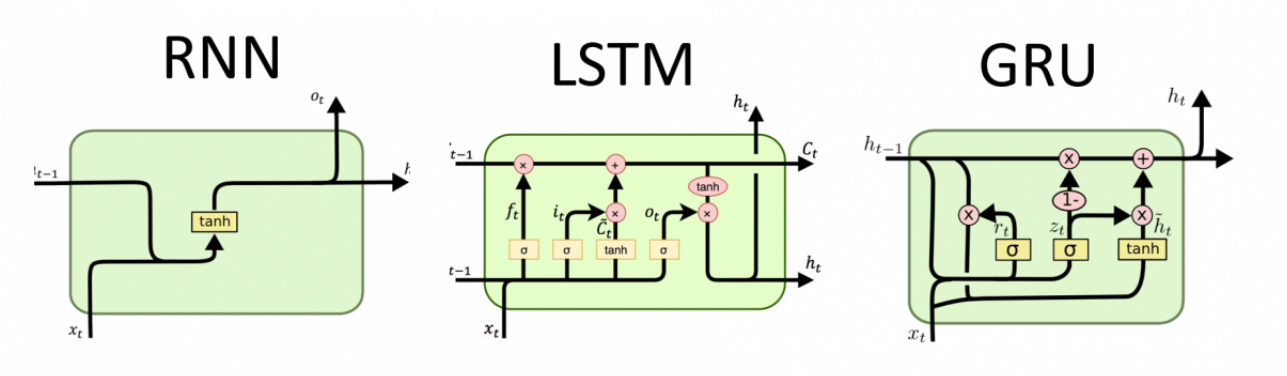
Gated Recurrent Unit (GRU) networks, as shown on the right in Figure 8, are a type of Recurrent Neural Network (RNN) designed to efficiently capture dependencies in sequential data. GRUs simplify the LSTM architecture by combining the forget and input gates into a single update gate, making them less complex and faster to compute [17, 19, 82, 35].
A GRU unit consists of the following components:

- **Hidden State:** At each time step, the GRU receives an input x_t and produces a hidden state h_t. The hidden state acts as a summary of the past inputs and is updated at each step based on the current input and the previous hidden state.
- **Gates:** GRUs utilize two main gates to control the flow of information:
  - **Update Gate:** This gate determines how much of the previous hidden state should be retained and how much of the new candidate hidden state should be added. It effectively allows the GRU to decide when to forget previous information.

– **Reset Gate:** The reset gate controls how much of the previous hidden state to forget. This is particularly useful for capturing new information when the sequence context changes significantly.

GRUs are designed to process input sequences step-by-step, updating their hidden state at each time step. This allows them to capture dependencies in the data efficiently while being computationally less demanding than LSTMs.

Figure 8: Comparison of RNN, LSTM, and GRU Architectures [25]



2.7 Imbalance Mitigation Techniques

2.7.1 Class Weight

The class weight approach relies on assigning different weights to classes in a dataset based on the number of samples in each class. This affects the loss function during the training phase by giving higher importance to minority classes. This approach is simple and effective because it is easy to implement and helpful when data imbalance is not extreme. However, it may not work well with nonlinear algorithms such as tree models. Moreover, the results can be inconsistent, as the effectiveness of this method varies depending on the specific dataset and model used [42, 15, 79, 35].

2.7.2 Random Under Sampling

The random under-sampling technique relies on randomly removing samples from the majority class in the dataset until there is an equal number of samples in each class. This technique can speed up the training phase due to the smaller size of the dataset. However, it can also lead to the loss of valuable information from the dataset [51, 15, 41, 12].

2.7.3 Synthetic Minority Over-sampling Technique (SMOTE)

Synthetic Minority Over-sampling is a technique that generates synthetic samples of the minority class to achieve balance in the dataset. It creates new samples by

interpolating between existing samples from the minority class, which leads to an increase in the minority class. Moreover, it can be used with ML models that do not assume linearity in the data, and it does not result in the loss of information. However, this may lead to an increase in training time [15, 41].

2.8 Explainable Artificial Intelligence (XAI)

Explainable Artificial Intelligence (XAI) focuses on creating ML systems that are not only accurate but also interpretable, enabling users to understand the reasoning behind AI-driven decisions. This transparency is essential for building trust and ensuring accountability in AI systems, as it allows users to manage and evaluate AI actions effectively. XAI addresses the inherent trade-offs between performance and interpretability, emphasizing the importance of designing models that are both understandable and reliable. By enhancing explainability, XAI aims to reduce biases, improve fairness, and support ethical decision-making, fostering greater confidence in AI systems across various domains [14, 56, 4].

2.8.1 Taxonomy of XAI Tecchniques

There different approaches for interpreting ML or DL models:

1. **White Box Versus Black Box Model Techniques**

   (a) **White Box Models:** These are transparent and interpretable AI models with straightforward mechanisms, making their decision-making processes easy to understand. A common example is linear models, where predictions are calculated as a weighted sum of input features. While their simplicity aids in clarity and explainability, it also limits their ability to manage complex datasets effectively. For such cases, advanced models are often necessary to achieve greater accuracy [57].

   (b) **Black Box Models:** These are advanced AI models, such as neural networks and ensemble methods like gradient boosting, known for their high performance on complex tasks but low interpretability. Their opacity stems from the difficulty in understanding how individual features contribute to predictions. For example, in fully connected (FC) neural networks, tracing specific inputs to their impact on the output is challenging, making these models powerful but less transparent [57].

2. **Model-Specific Techniques Versus Model-Agnostic Techniques**

   (a) **Model-specific:** these are techniques designed to analyze and interpret models within a specific algorithm class, leveraging features unique to

that class. Their functionality is typically limited to the specific characteristics of the target algorithm[23].

(b) **Model-agnostic:** these techniques can be applied to any ML model, regardless of the algorithm used. For instance, the Local Interpretable Model-agnostic Explanations (LIME) technique can interpret and analyze predictions and features across diverse models [23].

3. **Global Interpretation Versus Local Interpretation**

(a) **Global Interpretation:** these approaches provide a comprehensive analysis of a model's overall behavior. They involve defining variables, understanding their dependencies and interactions, and assigning importance to each component to gain insights into how the model functions as a whole [69].

(b) **Local Interpretation:** these approaches focus on analyzing individual predictions to understand why a specific decision was made. It examines the data point's context to determine the factors influencing the output, providing insights into the reasoning behind a particular recommendation [69].

2.8.2 XAI Techniques

2.8.2.1 Integrated Gradients (IGs)

Integrated Gradients (IGs) is an interpretability method designed to attribute deep neural network predictions to individual input features. It is commonly used in XAI for providing human-understandable explanations of complex model decisions. Unlike simpler gradient-based methods, IGs offers a more reliable, axiomatic approach to attribution, grounded in theory [80, 60, 61].

1. **Key Concepts in IGs**

- **Baseline**: A reference input, typically a zero vector or neutral input, representing minimal influence on the prediction. The baseline's choice affects the attribution outcome.
- **Path Integral**: IGs computes the integral of gradients along a straight path between the baseline and the actual input. This smooths gradient information and reduces noise, enhancing attribution reliability.
- **Attributions**: These quantify each feature's contribution to the model's output by evaluating how predictions change as input features vary, relative to the baseline.

2. **Mathematical Formulation** The attribution of each feature is calculated by integrating gradients along the path from the baseline x' to the input x:

$$IG_j(x) = (x_j - x'_j) \times \int_{\alpha=0}^{1} \frac{\partial f(x' + \alpha \cdot (x - x'))}{\partial x_j} \, d\alpha$$

Where:

- $f(x)$ is the model's prediction for input x,
- x' is the baseline input,
- x_j and x'_j are the j-th features of x and x',
- α interpolates between x' and x,
- $\frac{\partial f}{\partial x_j}$ is the gradient with respect to the j-th feature.

The final attribution vector is:

$$IG(x) = (x - x') \odot \int_{\alpha=0}^{1} \frac{\partial f(x' + \alpha \cdot (x - x'))}{\partial x} \, d\alpha$$

Where $\odot$ denotes the element-wise product.

3. **Algorithm Outline** The IG algorithm proceeds as follows:

   (a) **Input:**

   - Black-box model f,
   - Input x,
   - Baseline x'.

   (b) **Procedure:**

   - Initialize $IG = 0$.
   - For each step $\alpha \in [0, 1]$, compute the interpolated input $x^\alpha = x' + \alpha \cdot (x - x')$ and the gradient $\frac{\partial f(x^\alpha)}{\partial x}$.
   - Accumulate the gradients scaled by $(x - x')$ over m steps:

$$IG = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial f(x^{\alpha_i})}{\partial x} \cdot (x - x')$$

   - The final attributions are:

$$IG(x) = (x - x') \odot IG$$

   (c) **Output:**

   - A vector representing the feature contributions to the prediction.

4. **Advantages**

   - **Theoretical Foundation**: IGs is grounded in strong theoretical principles, particularly the concept of attributing the prediction of a model to its input features. It is based on the idea of accumulating gradients over a straight path from a baseline to the input, ensuring desirable properties such as sensitivity and completeness in the attribution process [80].
   - **Model-Agnostic**: IGs is a model-agnostic technique, meaning it can be applied to a wide range of ML models, including neural networks and tree-based models, making it highly versatile [80, 58].
   - **Reliability**: By averaging gradients over multiple steps, IGs reduces noise and provides more stable and consistent attributions compared to traditional methods like LIME or SHAP [80].

5. **Limitations**

   - **Baseline Sensitivity**: The choice of baseline is critical for IGs, as a poor baseline selection can lead to misleading attributions. Sensitivity to baseline selection is a known challenge in the method [80, 16].
   - **Computational Cost**: IGs requires multiple model evaluations, which can be computationally expensive, especially for large models or datasets. Various optimizations, such as parallelization or approximations, have been suggested to mitigate this issue [80, 5].
   - **Linearity Assumption**: IGs assumes a linear interpolation from the baseline to the input. This assumption may not hold in highly nonlinear models, potentially limiting the accuracy of the attributions [80].
   - **Saturation Effects**: In regions where activation functions are saturated (such as sigmoid or tanh), gradients can become very small, leading to less reliable attributions for those features [80, 16].
   - **Non-uniqueness**: IGs attributions can vary depending on the choice of baseline, which can complicate the interpretation of model predictions and make consistent explanations more challenging [80].

2.8.2.2 Deep Learning Important FeaTures (DeepLIFT)

Deep Learning Important FeaTures (DeepLIFT) is an attribution method designed to explain deep neural network decisions. It improves upon gradient-based methods like gradients and IGs by offering a more interpretable approach. DeepLIFT compares a model's activations at a given input with those at a reference input, providing insight into the decision-making process [74].

1. **Key Concepts in DeepLIFT**

- **Reference Input**: A baseline input (e.g., zero or average) representing minimal model output. This input is compared with the actual input to assess feature influence.
- **Lifting Transformation**: The difference in activations between the actual input and the reference is distributed across the input features, showing their contribution to the output.
- **Attributions**: These represent each feature's contribution to the model's prediction by comparing output changes as input features vary from the reference to the actual input.

2. **Mathematical Formulation** Attributions for each feature x_j are computed by comparing activations at the reference and actual input. For layer l, the difference in activation $\Delta A_{l,j}$ is given by:

$$\Delta A_{l,j} = A_{l,j}(x) - A_{l,j}(x')$$

Where:

- $A_{l,j}(x)$ and $A_{l,j}(x')$ are the activations of the j-th unit in layer l for the input x and reference x', respectively.
- $\Delta A_{l,j}$ is the activation difference.

Attributions are then computed as:

$$\text{Attribution}_j(x) = \sum_l (\text{Relevance Score}_{l,j} \times \Delta A_{l,j})$$

Where Relevance $\text{Score}_{l,j}$ reflects the layer's contribution to the final output, depending on network architecture and activation relationships.

3. **Algorithm Outline** To compute DeepLIFT attributions:

   (a) **Input:**
   - DL model f,
   - Input x,
   - Reference x' (neutral baseline).

   (b) **Procedure:**
   - Forward propagate x and x' through the model to obtain activations at each layer.
   - Compute the activation differences $\Delta A_{l,j}$ for each layer.
   - Calculate relevance scores $R_{l,j}$ based on network architecture.

- Sum the contributions from each layer to compute the final attribution for each feature:

$$\text{Attribution}_j(x) = \sum_l R_{l,j} \cdot \Delta A_{l,j}$$

(c) **Output:**

- A vector of attributions indicating the contribution of each input feature.

4. **Advantages**
   - **Improvement Over Gradients**: DeepLIFT avoids the instability of gradient-based methods by using activation differences, making it more stable and less sensitive to saturation, which is a common issue with methods like IGs [74].
   - **Model-Agnostic**: DeepLIFT is model-agnostic, meaning it can be applied to a variety of model types, including FFNs, CNNs, and RNNs [74].
   - **Accurate Attributions**: By directly comparing activations, DeepLIFT provides more accurate attributions, especially when gradients are unreliable due to small or saturated gradients [74].
5. **Limitations**
   - **Reference Sensitivity**: The choice of reference input plays a crucial role in attribution accuracy for DeepLIFT. A poor reference can lead to misleading results, as the method relies on comparing activations to a reference input [74].
   - **Computational Overhead**: While DeepLIFT is generally more efficient than methods like IGs, it still requires multiple forward passes through the model, which can become slow and computationally expensive, particularly for large models or datasets [74].
   - **Neuron Behavior Assumptions**: DeepLIFT makes specific assumptions about the behavior of neuron activations, particularly how activation differences contribute to attribution. These assumptions may not always hold for complex or specialized neuron types, limiting the method's applicability [74].

2.8.2.3 Gradient SHAP

Gradient SHAP builds upon Shapley values to provide feature attributions for a model's prediction. It improves upon traditional SHAP methods by using gradients, making it more efficient for DL models, where exact SHAP computation can be computationally expensive [58].

1. **Key Concepts in Gradient SHAP**

   - **Baseline**: Similar to IGs, Gradient SHAP requires a baseline (e.g., zero vector or average input), representing the reference state for comparison.
   - **Shapley Values**: From cooperative game theory, Shapley values distribute the total "payout" (model prediction) across features. Gradient SHAP uses them to compute feature contributions.
   - **Gradient-based Estimation**: Gradient SHAP approximates Shapley values using gradients, making it more computationally efficient for large models like deep neural networks.

2. **Mathematical Formulation of Gradient SHAP** Gradient SHAP approximates feature attributions by combining Shapley values and gradient information. The general formulation is:

$$\hat{v}_j = \mathbb{E}_{S \sim \pi} \left[(f(x_S \cup j) - f(x_S)) \nabla_{x_j} f(x_S \cup j) \right]$$

   Where:

   - $f(x)$ is the model's prediction for input x,
   - x_S is the input with a subset of features excluding the j-th feature,
   - $x_S \cup j$ includes the j-th feature added to x_S,
   - $\nabla_{x_j} f(x_S \cup j)$ is the gradient of the model's prediction with respect to the j-th feature.

   The approximation is computed as:

$$\hat{v}_j = \sum_{S \subseteq F \setminus j} \frac{|S|!(|F| - |S| - 1)!}{|F|!} (f(x_S \cup j) - f(x_S)) \nabla_{x_j} f(x_S \cup j)$$

   Where F is the full feature set and S is a subset excluding j.

3. **Algorithm Outline** To compute Gradient SHAP attributions:

   (a) **Input:**

   - Black-box model f,
   - Input instance x,
   - Baseline x' (or set of reference points).

   (b) **Procedure:**

   - For each feature j, compute the gradient $\nabla_{x_j} f(x)$,

- Compute the feature contribution based on Shapley values and gradient information:

$$v_j = (f(x_S \cup j) - f(x_S)) \nabla_{x_j} f(x_S \cup j)$$

where S is the subset excluding j,

- Approximate the Shapley values by averaging over multiple subsets S:

$$\hat{v}_j = \frac{1}{m} \sum_S (f(x_S \cup j) - f(x_S)) \nabla_{x_j} f(x_S \cup j)$$

(c) **Output:**

- A vector representing the contribution of each feature to the model's prediction.

4. **Advantages of Gradient SHAP**

- **Theoretical Foundation**: Gradient SHAP is grounded in Shapley values, ensuring fair and principled feature attributions, while using gradients for efficiency. This combination allows it to maintain the fairness of Shapley values while overcoming the computational challenges of traditional methods [58].
- **Efficient Computation**: By leveraging gradients, Gradient SHAP approximates Shapley values efficiently, significantly improving scalability for large models, such as deep neural networks, where exact Shapley computation can be prohibitively expensive [58].
- **Model-Agnostic**: Gradient SHAP is model-agnostic and can be applied to any model capable of computing gradients, such as neural networks and DTs, making it versatile across different ML frameworks [58].

5. **Limitations of Gradient SHAP**

- **Gradient Dependence**: Gradient SHAP may not work well for models where gradients are undefined or poorly defined, such as some discrete models. In these cases, relying on gradients for feature attribution may yield inaccurate results [58].
- **Computational Cost**: Despite being more efficient than exact Shapley value computation, Gradient SHAP still requires multiple gradient evaluations and model predictions, which can be computationally expensive for large models, especially DL models [58].
- **Non-linearity Challenges**: Gradient SHAP assumes that gradients provide meaningful information about feature importance, which may not

always hold for highly non-linear models, where the relationship between input features and output may not be adequately captured by gradients [58].

- **Saturation Effects**: Like other gradient-based methods, saturation in activation functions (e.g., sigmoid or tanh) may cause gradients to vanish, which can affect attribution reliability in saturated regions of the model [80, 58].

2.8.2.4 DeepLIFT SHAP

DeepLIFT SHAP is a model interpretability method that combines the advantages of both Shapley values and the DeepLIFT method. Specifically designed for DL models, DeepLIFT SHAP provides feature attributions efficiently by using DeepLIFT for feature contribution estimation and gradient-based methods for Shapley value approximation [74, 58]

1. **Key Concepts in DeepLIFT SHAP**

   - **Baseline**: Like other attribution methods, DeepLIFT SHAP uses a baseline to compare the input and attribute contributions of features. This baseline often represents the absence of information (e.g., a zero vector or average input).
   - **DeepLIFT**: DeepLIFT computes differences in activations of neurons compared to a reference (baseline). It assigns feature contributions based on how much the activations deviate from the baseline in each layer.
   - **Shapley Values**: DeepLIFT SHAP incorporates Shapley values to fairly distribute the model's output among features based on their contributions.
   - **Combination of DeepLIFT and Shapley**: DeepLIFT SHAP uses DeepLIFT to quickly estimate feature contributions and combines them with Shapley value theory for efficient and reliable attributions.

2. **Mathematical Formulation of DeepLIFT SHAP** The key idea behind DeepLIFT SHAP is to combine DeepLIFT's layer-wise contributions with Shapley value computations. The formulation is as follows:

$$\hat{v}_j = \mathbb{E}_{S \sim \pi} \left[(f(x_S \cup j) - f(x_S)) \cdot \Delta_{x_j} f(x_S \cup j) \right]$$

   Where:

   - $f(x)$ is the model's prediction for input x,
   - x_S is a subset of input features excluding feature j,

- $x_S \cup j$ is the input with feature j added to the subset x_S,
- $\Delta_{x_j} f(x_S \cup j)$ is the DeepLIFT contribution for feature j, which represents the difference in activations for input $x_S \cup j$ relative to the baseline.

The attribution for feature j is computed as:

$$\hat{v}_j = \sum_{S \subseteq F \setminus j} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \left(f(x_S \cup j) - f(x_S)\right) \cdot \Delta_{x_j} f(x_S \cup j)$$

Where F is the full set of features and S is a subset excluding j.

3. **Algorithm Outline** The procedure to compute DeepLIFT SHAP attributions is as follows:

   (a) **Input:**

   - Black-box model f,
   - Input instance x,
   - Baseline instance x' (or a set of reference points for approximation).

   (b) **Procedure:**

   - For each feature j, compute the DeepLIFT contribution by evaluating the difference in activations at input x and the baseline x':

     $$\Delta_{x_j} f(x) = f(x) - f(x')$$

     where $f(x)$ is the model's output for input x, and $f(x')$ is the output at the baseline.
   - Calculate the Shapley value for each feature j by considering all subsets S of features, and computing the contribution of j based on DeepLIFT contributions:

     $$v_j = \left(f(x_S \cup j) - f(x_S)\right) \cdot \Delta_{x_j} f(x_S \cup j)$$

     where x_S is the input with all features except j.
   - Approximate the Shapley values by averaging the contributions over multiple subsets S:

     $$\hat{v}_j = \frac{1}{m} \sum_{S} \left(f(x_S \cup j) - f(x_S)\right) \cdot \Delta_{x_j} f(x_S \cup j)$$

     where m is the number of subsets used in the approximation process.

   (c) **Output:**

- A vector representing the contribution of each feature to the model's prediction based on DeepLIFT and Shapley values.

4. **Advantages of DeepLIFT SHAP**

   - **Efficient Computation**: DeepLIFT SHAP accelerates computation by combining DeepLIFT's fast activation difference method with Shapley value approximations, making it suitable for DL models. This approach leverages the strengths of both methods to provide scalable feature attributions [74, 58].
   - **Theoretical Soundness**: DeepLIFT SHAP inherits the fairness and consistency of Shapley values, while DeepLIFT enhances computational efficiency by addressing the challenges of backpropagating feature attributions in DL models [74, 58].
   - **Model-Agnostic**: DeepLIFT SHAP can be applied to any model with accessible gradient information and activations, including neural networks and other ML models, making it versatile and widely applicable [74, 58].

5. **Limitations of DeepLIFT SHAP**

   - **Gradient Dependence**: Similar to Gradient SHAP, DeepLIFT SHAP depends on gradients, which may not be well-defined for certain models. This reliance on gradients can be problematic in cases where gradients are undefined or poorly defined [58, 74].
   - **Approximation Error**: While more efficient than exact Shapley value computation, DeepLIFT SHAP may still introduce approximation errors, especially for high-dimensional input spaces. The use of gradient-based methods for Shapley approximation can lead to errors when estimating the contributions of features in complex models [58, 74].
   - **Computational Complexity**: DeepLIFT SHAP still requires multiple model evaluations for different subsets of features, which can make it computationally expensive for large models and datasets. Although it is more efficient than traditional Shapley computations, the requirement for numerous forward passes can still be costly [58, 74].
   - **Saturation Effects**: Like other gradient-based methods, DeepLIFT SHAP may be affected by saturating activation functions, which can cause gradients to vanish, affecting the reliability of feature attributions in saturated regions of the model [80, 58].

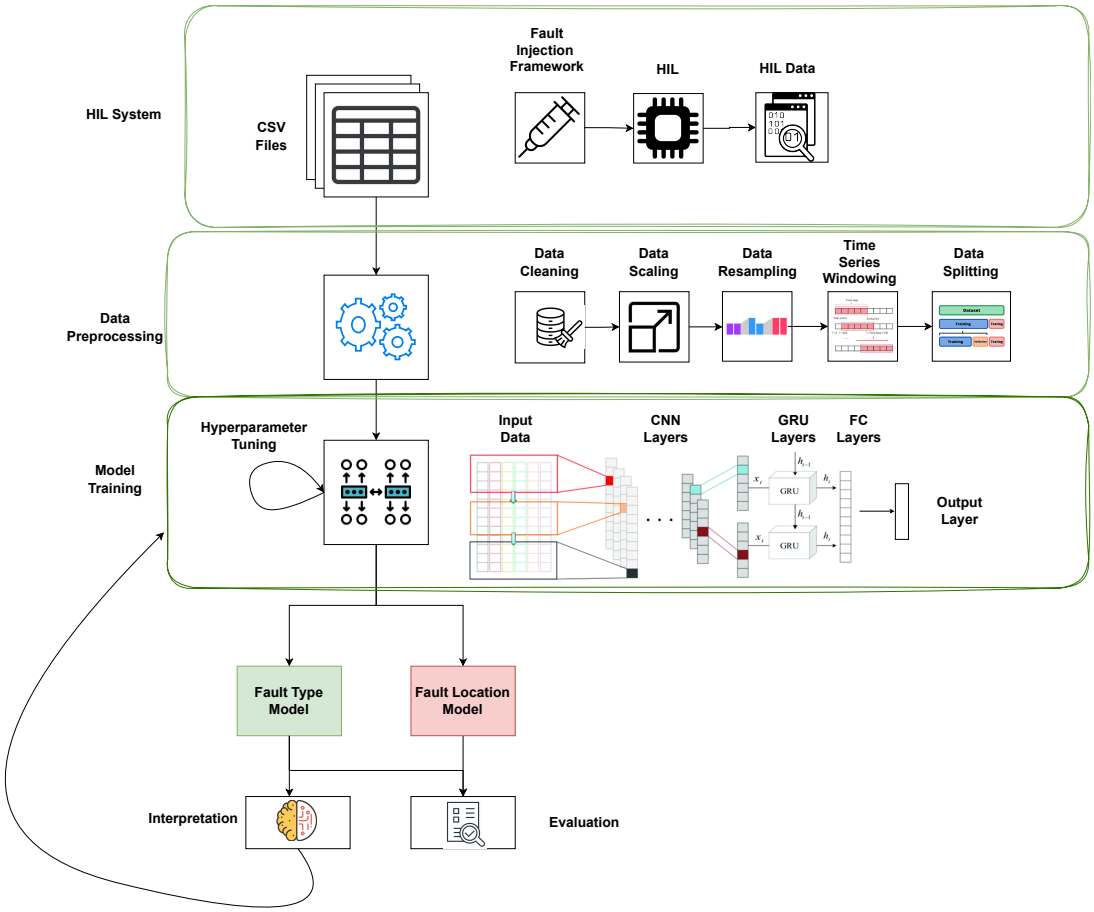
2.9 Summary

This section provides an overview of MBD and related methodologies, including testing approaches like MIL, SIL, PIL, HIL, and VIL. It explores fault injection techniques, focusing on fault types (Noise, Gain, Offset/Bias) and their locations in the system (e.g., accelerator, brake, engine). There are various methods for developing fault detection and diagnoses models, which can be broadly categorized into signal-based, model-based, and data-driven approaches, as briefly outlined. .It also covers ML paradigms like supervised and unsupervised learning, along with regression and classification techniques for prediction and categorization. Advanced DL models, including FNNs, CNNs, RNNs, LSTMs, and GRUs, are discussed for handling complex data. The section also addresses methods for managing data imbalance, such as class weighting, random under sampling, and SMOTE. Finally, it introduces XAI techniques, such as IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP, to enhance the interpretability of AI models.

3 Methodology

This chapter presents a detailed overview of the flowchart for the proposed DL model, highlighting the key phases—from data collection and preprocessing to model training, evaluation, and interpretation using XAI techniques. Each phase is depicted in Figure 9 and elaborated upon in this chapter.

Figure 9: Overview of the Proposed DL Model Methodology



3.1 Data Collection

Data collection in systems involves capturing both healthy and faulty data. Healthy data is gathered during normal system operation, while faulty data is obtained by intentionally injecting faults to simulate failure conditions. Once generated, the data is recorded in various file formats and manually converted into CSV format for further processing. Key components of data collection in HIL testing include:

Data collection in systems involves capturing both healthy and faulty data. Healthy data is collected during normal system operations, whereas faulty data is obtained by intentionally injecting faults to simulate failure scenarios. After generation, the data is stored in multiple file formats and subsequently converted into CSV format for

further processing. The primary elements of data collection in HIL testing include:

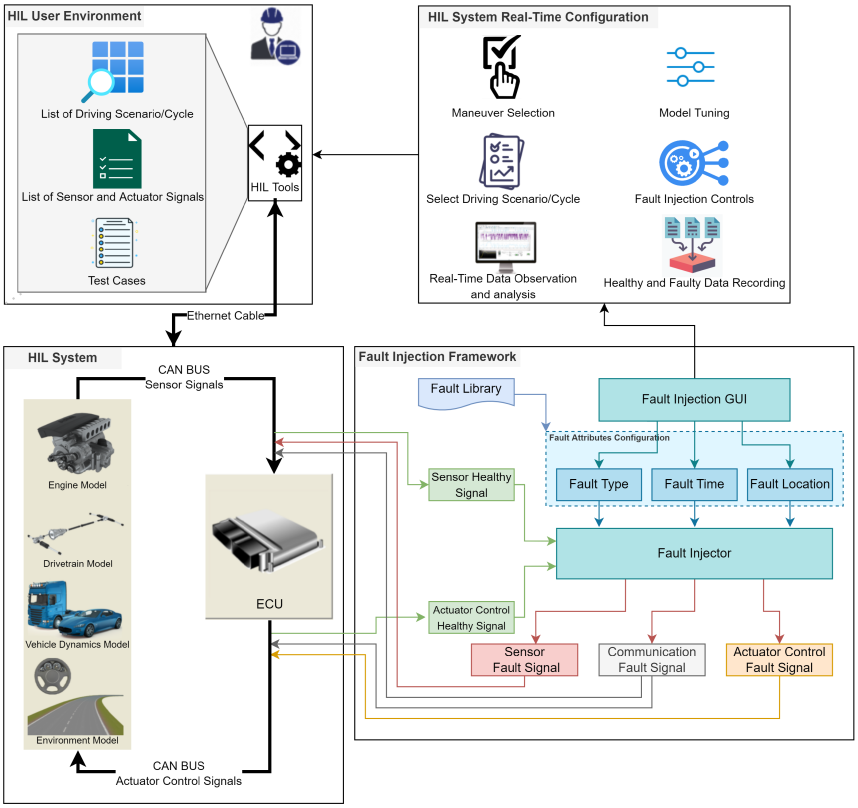
3.1.1 HIL System Confirguration

The initial step in implementing a fault injection strategy for data collection involves comprehending the HIL system setup. This setup combines physical hardware components—such as real-time simulators, real ECUs, actuators, and sensors—with software elements, including simulation models and the HIL software framework. Collectively, these components replicate the behavior of the real system, establishing a controlled environment for fault injection and data acquisition.

3.1.2 Fault Injection Framework

The integration of GUI of fault injection with HIL tools simplifies the setup and management of fault injection scenarios by allowing users to configure parameters such as fault type, fault location, timing, magnitude, duration, and propagation patterns. Its intuitive design enables quick adjustments and efficient simulations, supporting the evaluation of system fault tolerance and response characteristics [1]. Figure 10 illustrates the fault injection framework [1], which integrates the HIL system with the user environment.

Figure 10: Fault Injection Framework with HIL System [1]



3.1.3 Fault Injection Parameter

The fault injection GUI allows users to configure and simulate various fault types, including sensor, communication, actuator, and system faults. Users can specify key parameters such as fault size, duration, and initiation time, either individually or concurrently. The GUI also supports time scheduling for injecting single or simultaneous faults at custom start times, enabling flexible testing of complex fault scenarios.

3.1.4 Real-Time Data Collection

Real-time data collection methods record system behavior during fault injection experiments by tracking critical parameters and actuator responses. Best practices include optimizing sampling rates, employing robust data collection methods, and ensuring synchronization of data streams. These strategies provide an accurate depiction of the system's behavior, facilitating comprehensive performance evaluation and precise fault detection and diagnosis.

3.1.5 Data Logging and Storage

Data from fault injection trials is stored in databases or structured file formats, enabling efficient management of large datasets. Preprocessing and filtering techniques enhance data quality by minimizing noise and removing artifacts, ensuring the data is clean, well-structured, and ready for comprehensive analysis.

3.2 Data Preprocesing

The data preprocessing stage focuses on cleaning and organizing raw data to ensure it is suitable for training AI models.

3.2.1 Data Cleaning

After data collection, the generated CSV files go through a cleaning process to prepare them for model training. This process includes removing irrelevant metadata from the simulation software and correcting column names to ensure they are both descriptive and consistent. This cleaning step is crucial for organizing the data accurately, making it suitable for model development [37].

3.2.2 Data Scaling

Data scaling is a preprocessing step used to normalize or standardize numerical features, ensuring all features contribute equally to the developed model. This step is essential for ML and DL algorithms, as differing feature scales can lead to biased performance or convergence issues. Scaling methods like normalization or

standardization enhance the model's learning ability, preventing any single feature from disproportionately influencing the outcome [2].

3.2.3 Data Resampling

In the dataset, the number of healthy samples significantly exceeds the number of faulty samples, resulting in class imbalance. To address this issue, three effective approaches can be employed:

- **Random Under-sampling**: involves randomly reducing the number of samples from the majority class (healthy samples) to create a more balanced dataset. Applied as a preprocessing step, it helps improve model performance by preventing bias toward the majority class [13].
- **Class Weights**: adjusts the influence of each class during the training process without modifying the dataset. By assigning higher weights to the minority class (faulty samples), the model is encouraged to focus more on these samples, mitigating the imbalance without discarding valuable data from the majority class [85].
- **SMOTE**: increases the number of samples in the minority class by generating synthetic examples through interpolation, improving the class representation and enabling the model to learn from a more diverse set of examples [30, 13].

3.2.4 Time Series Windowing

Time series windowing is a step used to transform a continuous time series dataset into a structured format suitable for model development. This method involves segmenting the time series data into fixed-size sequences, or windows, allowing models to learn temporal patterns and dependencies within the data [49].

3.2.5 Data Splitting

Data splitting is a step in the data preparation process for developing ML or DL models. It involves dividing the dataset into distinct subsets to evaluate model performance and reduce the risk of overfitting. The primary objective is to ensure that the model is trained on one portion of the data while being validated or tested on another, unseen portion [8, 52, 39]. Typically, the dataset is divided into three main subsets:

- **Training Set**: is used to train the model, enabling it to learn patterns and relationships within the data.
- **Validation Set**: is used to tune hyperparameters and model architecture, offering an intermediary evaluation of the model's performance during training.

- **Testing Set**: is reserved for the final evaluation of the model's performance, serving as an unbiased assessment of how well the model generalizes to unseen data.

3.3 Model Development

3.3.1 Baseline Model Development

Establishing a baseline model is a fundamental first step in model development. The baseline model serves as a simple reference point against which more complex models can be compared. By providing an initial benchmark for model performance, the baseline helps understand the minimal level of performance metrics achievable using a straightforward approach [18].
For multi-class time series classification, baseline models typically consist of relatively simple architectures, such as RNNs, LSTM networks, or GRUs. These models are well-suited for processing sequential data and provide foundational tools to assess baseline performance for a given problem.

- **RNNs**: are often the first choice as a baseline model due to their inherent design for processing sequential data. These networks excel at capturing short-term dependencies by passing information through a series of temporal states. However, RNNs are prone to the vanishing gradient problem, which hampers their ability to learn long-term dependencies, making them less effective for more complex tasks that require capturing longer-term patterns. Despite this limitation, RNNs remain a valuable baseline due to their simplicity and ability to handle basic sequence tasks [48].

- **LSTMs**: represent a significant advancement over traditional RNNs. LSTMs address the vanishing gradient problem by utilizing gating mechanisms, allowing the model to retain and update information over longer time horizons. This enhancement enables LSTMs to capture long-term dependencies more effectively than RNNs. As a result, LSTMs are often employed when the baseline RNN fails to perform adequately, making them a natural next step in model development. LSTMs are widely recognized for their improved accuracy and efficiency in handling sequences with complex temporal dependencies [48].

- **GRUs**: are a simpler alternative to LSTMs. While they share many of the advantages of LSTMs in handling long-term dependencies, GRUs use fewer parameters, making them computationally more efficient. This reduced complexity comes at a slight cost in performance, but GRUs often provide comparable results to LSTMs, especially when computational resources are limited or when a more streamlined model is required. In practice, GRUs serve as

a useful model for comparison against more complex architectures, offering a balance between performance and computational efficiency [19, 17].

3.3.2 Proposed CNN-GRU Model

The proposed model integrates CNNs and GRUs in a sequential architecture. The design leverages the complementary strengths of CNNs for feature extraction and GRUs for capturing temporal dependencies. FC layers are added at the end, mapping the learned features to the desired number of output classes for multi-class time series classification.

1. **CNN Layers**: is the first component of the model, which consists of CNN layers, which are essential for extracting spatial features from raw time series data. These layers apply convolutional filters to the data, allowing the model to detect local patterns such as trends, anomalies, and periodic fluctuations. This process is crucial for uncovering the underlying structure of the data, which may not be immediately obvious in its raw form. By learning these local patterns, the CNN layers create a more abstract and informative representation of the data [84]. The convolution operation is mathematically expressed as:

$$\mathbf{X}_{\text{out}} = \sigma\left(\mathbf{W}_c * \mathbf{X}_{\text{in}} + \mathbf{b}_c\right)$$

   where:

   - $\mathbf{X}_{\text{in}}$: Input time-series data.
   - $\mathbf{W}_c$: Convolutional filter weights, which are learned during training.
   - $*$: Convolution operation, applied to extract localized patterns from the input.
   - $\mathbf{b}_c$: Bias term added to the result of the convolution operation.
   - σ: Non-linear activation function (e.g., ReLU), applied element-wise to introduce non-linearity.
   - $\mathbf{X}_{\text{out}}$: Output of the CNN layer, representing extracted features that capture essential patterns in the input.

2. **GRU Layers**: is the second component of the model after the CNN layers, the model incorporates GRU layers to capture the temporal dependencies in the time series data. GRUs utilize gating mechanisms that regulate the flow of information over time, addressing challenges such as the vanishing gradient problem that often arises in deep recurrent networks. These layers process the spatial features extracted by the CNN layers in a sequential manner, learning

the temporal dynamics and long-term dependencies in the data. By doing so, the GRU layers help the model track the evolution of time-dependent patterns, enabling predictions based on both short-term and long-term context [19, 17]. The GRU unit consists of an update gate (z_t) and a reset gate (r_t). The equations for these gates are as follows:

$$z_t = \sigma\left(\mathbf{W}_z \mathbf{h}_{t-1} + \mathbf{U}_z \mathbf{X}_t + \mathbf{b}_z\right)$$

$$r_t = \sigma\left(\mathbf{W}_r \mathbf{h}_{t-1} + \mathbf{U}_r \mathbf{X}_t + \mathbf{b}_r\right)$$

The candidate activation $\tilde{\mathbf{h}}_t$ is computed as:

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_h \left(r_t \cdot \mathbf{h}_{t-1}\right) + \mathbf{U}_h \mathbf{X}_t + \mathbf{b}_h\right)$$

Finally, the GRU output at time step t, denoted as $\mathbf{h}_t$, is given by:

$$\mathbf{h}_t = (1 - z_t) \cdot \mathbf{h}_{t-1} + z_t \cdot \tilde{\mathbf{h}}_t$$

where:

- $\mathbf{X}_t$: Input at time step t, typically derived from preceding layers (e.g., CNN layers in the architecture).
- $\mathbf{h}_{t-1}$: Hidden state from the previous time step, encapsulating past information.
- $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h$: Weight matrices corresponding to the update gate, reset gate, and candidate hidden state, respectively.
- $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}_h$: Weight matrices associated with the input for the update gate, reset gate, and candidate hidden state, respectively.
- $\mathbf{b}_z, \mathbf{b}_r, \mathbf{b}_h$: Bias terms for the update gate, reset gate, and candidate hidden state, respectively.
- σ: Sigmoid activation function, used for computing gating mechanisms.
- tanh: Hyperbolic tangent activation function, applied to compute the candidate hidden state.
- $\mathbf{h}_t$: Output of the GRU unit at time step t, serving as the updated hidden state for the next time step.

3. **FC Layers**: is final component of the model, which integrate the features learned from the preceding CNN and GRU layers. After the CNN layers have extracted spatial features and the GRU layers have captured temporal dependencies, the FC layers combine these learned representations into a format

suitable for classification. By fully connecting each neuron in one layer to every neuron in the next, the FC layers ensure a comprehensive, unified view of the extracted features, facilitating accurate decision-making and prediction [35]. The operation of a FC layer is expressed as:

$$\mathbf{y} = \sigma\left(\mathbf{W}_f \mathbf{h}_{\text{GRU}} + \mathbf{b}_f\right)$$

where:

- $\mathbf{h}_{\text{GRU}}$: Output from the final GRU layer, encapsulating temporal information.
- $\mathbf{W}_f$: Weight matrix of the FC layer, mapping GRU outputs to the target dimension.
- $\mathbf{b}_f$: Bias term added in the FC layer.
- σ: Activation function (e.g., softmax for multi-class classification or sigmoid for binary classification).
- $\mathbf{y}$: Model output, representing the predicted probabilities for each class.

For multi-class classification, the softmax activation function is applied to the output of the FC layer:

$$\mathbf{y}_i = \frac{e^{\mathbf{z}_i}}{\sum_j e^{\mathbf{z}_j}}$$

where:

- $\mathbf{z}_i$: Logit (raw score) corresponding to class i.
- $\mathbf{y}_i$: Predicted probability for class i, normalized across all classes.

In summary, the CNN layers extract features, the GRU layers capture temporal dependencies, and the FC layers map the learned representations to the output classes.

3.3.3 Model Evaluation

Model evaluation for multi-class classification relies on metrics such as accuracy, precision, recall, and F1-score. The confusion matrix highlights correct and incorrect predictions for each class, while the ROC curve assesses the trade-off between true positive and false positive rates. The precision-recall curve, in contrast, emphasizes the balance between precision and recall, particularly in imbalanced datasets [78].

3.3.4 Hyperparameters Tuning

Hyperparameter tuning focuses on selecting the optimal settings, such as learning rate and batch size, to maximize a model's performance. These parameters govern the training process and significantly affect the model's capacity to generalize to unseen data. Common approaches, including grid search, random search, and Bayesian optimization, systematically explore the hyperparameter space. By evaluating various configurations on a validation set, the most effective combination of hyperparameters is identified, boosting model accuracy while optimizing computational resources [18].

3.4 XAI Integration

Integrating XAI enables a deeper understanding of model behavior through GFI, PCFI, and FIs, enhancing transparency and interpretability

3.4.1 Global Feature Importance (GFI)

Global Feature Importance (GFI) is pivotal for understanding and improving model performance. By identifying the features with the greatest influence on predictions, GFI enables targeted testing efforts, ensuring the model is both precise and dependable. Moreover, GFI validates feature relevance—unexpected dominance by a feature may indicate a model issue, necessitating further testing or adjustments. In fault analysis and debugging, GFI helps identify specific features contributing to errors, uncovering issues such as data leakage or spurious correlations. It also optimizes test data collection by focusing resources on the most significant features, enhancing coverage and robustness while reducing complexity. GFI further guides feature engineering by identifying redundant or irrelevant features that can be eliminated to streamline the model, improving performance and interpretability. Finally, GFI insights aid in refining the model architecture or parameters, ensuring critical features are appropriately weighted, thus boosting efficiency and reliability. Attribution methods used to calculate GFI include:

- **IGs**: attributions are calculated by accumulating gradients along a path from a baseline (such as a zero vector or mean value) to the input sample. This method evaluates how the output changes as each feature is incrementally altered from the baseline to its observed value.
- **DeepLIFT**: attributions are computed by comparing the model's output relative to a reference (or baseline) using layer-wise relevance propagation. This method is efficient in capturing non-linear relationships between features, particularly in deep neural networks.

- **Gradient SHAP**: is a combination of Shapley values and gradient-based approaches is used to compute attributions. Shapley values, originating from cooperative game theory, are used to fairly distribute the contribution of each feature by considering all possible combinations of features.
- **DeepLIFT SHAP**: combines DeepLIFT with Shapley values, merging the strengths of both approaches. It provides efficient computations and ensures fairness in the attribution process.

3.4.2 Per-Class Feature Importance (PCFI)

Per-Class Feature Importance (PCFI) offers valuable insights into how specific features influence fault category predictions in ASSs. The most relevant features for each fault type or location are identified, enabling focused testing efforts on the most significant attributes, ensuring more efficient and effective test coverage. This targeted approach enhances the accuracy of real-time driving or HIL tests by prioritizing the most likely fault scenarios. Furthermore, class-wise feature importance aids in early error detection and proactive system calibration, thereby reducing the risk of failure. From a model perspective, interpretability is improved by clarifying the reasons behind predictions, building trust in the system and assisting in troubleshooting. Techniques such as IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP are used to analyze feature contributions at the sample level, thereby supporting better decision-making, reducing overfitting, and optimizing model performance. Overall, PCFI enables more accurate fault detection, improved diagnostic efficiency, and enhanced model transparency and optimization.

3.4.3 Feature Interactions (FIs)

Feature interaction (FIs), facilitated by XAI techniques like IGs, DeepLIFT, Gradient SHAP and DeepLIFT SHAP, enhances the interpretability of DL models for fault detection and classification in ASSs. Unlike traditional correlation methods, which capture only linear relationships between feature pairs, FIs uncovers non-linear dependencies and joint feature effects that are crucial for fault detection. Many faults in ASSs arise only when certain features interact in specific ways, which correlation methods often fail to capture. To analyze these interactions, individual feature attributions are first calculated to quantify each feature's contribution independently. Next, pairs of features are simultaneously perturbed, and the combined attribution scores are calculated. If the combined attribution score significantly exceeds the sum of individual attributions, it signals a strong interaction between features. This helps uncover hidden dependencies that influence the model's predictions, offering a more nuanced understanding of feature relationships. By detecting these complex interactions, FIs improves fault detection, enhances model interpretability, and

provides actionable insights for improving ASSs.

3.5 Summary

This section outlines the methodology for a DL model aimed at fault detection and diagnosis using XAI techniques. It covers the stages of data collection, where both healthy and faulty data are gathered through HIL testing and fault injections. The data is then preprocessed, including cleaning, scaling, and balancing, to prepare it for training. The model development begins with baseline models, progressing to more complex CNN-GRU architectures. Model evaluation uses metrics such as accuracy, precision, recall, and F1-score, while XAI methods are applied to interpret feature importance and interactions, enhancing the model's transparency and performance.

4 Implementation

This section describes the implementation process, starting with a gasoline engine case study that employs detailed simulation models. Key steps include data processing (collection, cleaning, labeling, scaling, and splitting), model development (baseline and proposed CNN-GRU), and integration of XAI for feature analysis and interpretability.

4.1 Case Study: Gasoline Engine

Gasoline engines power most modern vehicles by converting the chemical energy in gasoline into mechanical energy. This case study examines the functions of a gasoline engine using dSPACE's Automotive Simulation Models (ASM), a toolkit for simulating and analyzing automotive systems. ASM allows engineers to enhance performance, efficiency, and control systems through detailed simulations.

4.1.1 Automotive Simulation Models (ASM)

ASM offers a detailed simulation of a turbocharged 6-cylinder engine, supporting both direct and manifold injection systems, as well as manual and automatic transmissions. It precisely models engine components such as the fuel system, air path, cooling system, and exhaust system, allowing engineers to analyze and optimize engine designs [26].

4.1.1.1 Fuel System

The fuel system simulation models a common-rail system that ensures fuel is delivered at the correct pressure and flow rates. Key components include a high-pressure pump, injectors, and a pressure control valve. These features help optimize fuel delivery and sustain engine performance..

4.1.1.2 Air Path Components

The air path simulation includes components such as turbochargers, intercoolers, and Exhaust Gas Recirculation (EGR) systems. These components enhance efficiency and reduce emissions by regulating air pressure and flow through the engine.

4.1.1.3 Coolant System

The coolant system simulation ensures the engine operates within safe temperature limits. It includes models for coolant flow, temperature regulation, and thermostats, which safeguard engine components from overheating.

4.1.1.4 Exhaust System

The exhaust system simulation demonstrates how exhaust gases interact with engine components and influence performance and emissions. It models gas flow and treatment to meet environmental standards while optimizing efficiency.

4.1.1.5 Drivetrain and Vehicle Dynamics

The drivetrain model transmits engine torque to the wheels, while the vehicle dynamics model simulates the effects of factors such as road friction, slopes, and aerodynamics. Together, these models facilitate the analysis of vehicle performance and handling under varying conditions.

4.1.2 Electronic Control Unit (ECU) Model

The ECU simulation replicates the behavior of real-world engine controllers. It manages critical operations such as fuel injection, pressure control, and air-fuel mixture adjustments. The dSPACE MicroAutoBox II and SCALEXIO hardware communicate via the CAN bus to ensure precise simulation.

4.1.3 Controller Area Network (CAN) bus

The CAN bus simulation models the communication network between sensors, actuators, and the ECU, ensuring reliable data exchange in the simulated environment that reflects real-world operations.

4.1.4 Sensor and Actuator Models

ASM includes models for sensors such as engine speed, coolant temperature, and fuel pressure, as well as actuators such as injectors and throttle valves. These models simulate realistic scenarios, enabling engineers to test and refine system performance.

4.1.5 Fault Injection Framework

The fault injection framework enables engineers to simulate system faults, such as sensor errors or actuator malfunctions. This is accomplished by manipulating signals within MATLAB Simulink, with faults such as bias, offset, drift, or noise introduced into sensor readings. The framework assists in evaluating how the system responds to real-world failures and ensures that fault detection models are effectively tested.

4.1.6 Configuration and Control Desk

4.1.6.1 Configuration Desk

The Configuration Desk facilitates communication between different simulation components by generating a variable description file that enables seamless data exchange between the models.

4.1.6.2 Control Desk

The Control Desk offers an interface for real-time interaction with the simulation. Engineers can manipulate sensor signals, initiate fault injections, and monitor the system's response, making it an essential tool for testing and validation.

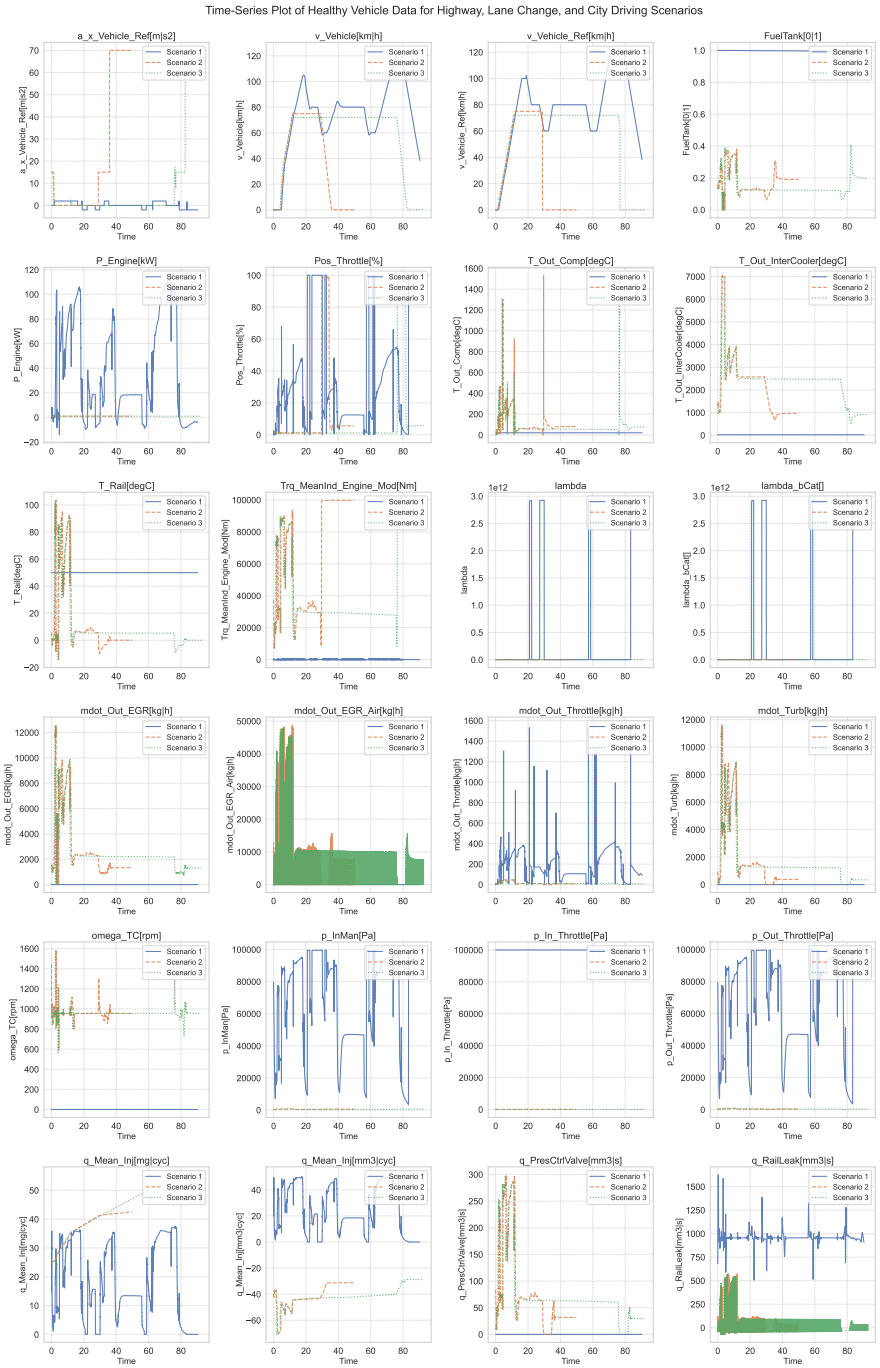
4.2 Data Collection

The main objective of HIL data collection is to create a dataset that reflects the behavior of the simulated vehicle system in real-world scenarios. This process generates two separate datasets:

- **Fault Types Dataset**: captures various types of faults, including noise, gain, and offsets.
- **Fault Locations Dataset**: focuses on identifying fault locations within the system. Faults are injected at positions such as the accelerator pedal, steering wheel angle, and engine speed.

To ensure the dataset accurately reflects real-world driving conditions, data is collected from three distinct scenarios: highway driving, lane changes, and city driving. Figure 11 presents the time-series data from these scenarios, highlighting variations in system behavior under each condition.

Figure 11: Healthy Time-Series Data Across Highway, Lane Change, and City Driving Scenarios



The data collection process begins with the selection and recording of specific features from the HIL system for training models. Table 2 provides an overview of the key features collected during this process.

Table 2: Overview of the Collected Dataset: Feature Names, and Descriptions

Feature Name	Description
a_x_Vehicle_Ref[m/s²]	Target vehicle acceleration.
v_Vehicle[km/h]	Current vehicle speed.
v_Vehicle_Ref[km/h]	Predefined speed for comparison.
FuelTank[0,1]	Fuel remaining (0: empty, 1: full).
P_Engine[kW]	Output power of the engine.
Pos_Throttle[%]	Position of the throttle valve.
T_Out_Comp[°C]	Compressor outlet temperature.
T_Out_InterCooler[°C]	Intercooler outlet temperature.
T_Rail[°C]	Temperature in the fuel rail.
Trq_MeanInd_Engine_Mod[Nm]	Average indicated torque.
lambda	Air-fuel ratio.
lambda_bCat[]	Lambda at the catalytic converter.
mdot_Out_EGR[kg/h]	Mass flow rate of exhaust gas.
mdot_Out_EGR_Air[kg/h]	Air mass flow from EGR.
mdot_Out_Throttle[kg/h]	Air mass flow at throttle valve.
mdot_Turb[kg/h]	Mass flow rate exiting turbine.
omega_TC[rpm]	Turbocharger speed (rpm).
p_InMan[Pa]	Pressure in the intake manifold.
p_In_Throttle[Pa]	Pressure at throttle inlet.
p_Out_Throttle[Pa]	Pressure at throttle outlet.
q_Mean_Inj[mg/cycle]	Average fuel injection quantity.
q_Mean_Inj_Alt[mm³/cycle]	Alternate fuel injection quantity.
q_PresCtrlValve[mm³/s]	Flow through pressure control valve.
q_RailLeak[mm³/s]	Leakage flow in the fuel rail.

4.3 Data Cleaning

The CSV file exported from the HIL system contains unnecessary metadata that must be removed. Additionally, it is essential to ensure that column names are correctly aligned with their corresponding data points. Any irrelevant columns generated by the system should also be deleted. Table 3 shows the first five samples of the cleaned dataset.

Table 3: First Five Samples of the Cleaned Dataset

Feature Name	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
a_x_Vehicle_Ref [m/s²]	1.50×10^{1}	1.50×10^{1}	1.50×10^{1}	1.50×10^{1}	1.50×10^{1}
v_Vehicle [km/h]	2.14×10^{-2}	2.05×10^{-2}	1.96×10^{-2}	1.88×10^{-2}	1.79×10^{-2}
v_Vehicle_Ref [km/h]	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}
FuelTank [0, 1]	7.76×10^{-1}	7.76×10^{-1}	7.76×10^{-1}	7.76×10^{-1}	7.76×10^{-1}
P_Engine [kW]	1.00×10^{0}	1.00×10^{0}	1.00×10^{0}	1.00×10^{0}	1.00×10^{0}
Pos_Throttle [%]	1.22	1.22	1.22	1.21	1.21
T_Out_Comp [°C]	7.58×10^{-1}	7.59×10^{-1}	7.60×10^{-1}	7.61×10^{-1}	7.62×10^{-1}
T_Out_InterCooler [°C]	4.90×10^{2}	4.90×10^{2}	4.90×10^{2}	4.89×10^{2}	4.89×10^{2}
T_Rail [°C]	-1.65×10^{-1}	-1.76×10^{-1}	-1.88×10^{-1}	-1.99×10^{-1}	9.01×10^{-1}
Trq_MeanInd_Engine_Mod [Nm]	7.43×10^{4}	7.42×10^{4}	7.41×10^{4}	7.40×10^{4}	7.39×10^{4}
lambda	1.25×10^{2}	1.26×10^{2}	1.26×10^{2}	1.27×10^{2}	1.27×10^{2}
lambda_bCat []	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}
mdot_Out_EGR [kg/h]	2.74×10^{3}	2.74×10^{3}	2.74×10^{3}	2.74×10^{3}	2.74×10^{3}
mdot_Out_EGR_Air [kg/h]	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}	0.00×10^{0}
mdot_Out_Throttle [kg/h]	3.01×10^{1}	3.02×10^{1}	3.02×10^{1}	3.03×10^{1}	3.59×10^{1}
mdot_Turb [kg/h]	7.37×10^{2}	7.39×10^{2}	7.40×10^{2}	7.41×10^{2}	8.78×10^{2}
omega_TC [rpm]	1.03×10^{3}	1.03×10^{3}	1.04×10^{3}	1.05×10^{3}	1.05×10^{3}
p_InMan [Pa]	5.53×10^{2}	5.53×10^{2}	5.53×10^{2}	5.53×10^{2}	5.53×10^{2}
p_In_Throttle [Pa]	1.70×10^{1}	1.70×10^{1}	1.70×10^{1}	1.70×10^{1}	1.70×10^{1}
p_Out_Throttle [Pa]	3.47×10^{2}	3.47×10^{2}	3.47×10^{2}	3.47×10^{2}	3.47×10^{2}
q_Mean_Inj [mg/cycle]	2.50×10^{1}	2.50×10^{1}	2.50×10^{1}	2.50×10^{1}	2.50×10^{1}
q_Mean_Inj_Alt [mm³/cycle]	-3.66×10^{1}	-3.66×10^{1}	-3.66×10^{1}	-3.66×10^{1}	-3.66×10^{1}
q_PresCtrlValve [mm³/s]	3.33×10^{1}	3.31×10^{1}	3.29×10^{1}	3.27×10^{1}	5.41×10^{1}
q_RailLeak [mm³/s]	9.37×10^{1}	8.88×10^{1}	8.28×10^{1}	7.57×10^{1}	6.76×10^{1}

4.4 Data Labeling

Data labeling is the process of tagging data points. In multi-class classification, each data point is assigned to a specific class, enabling the model to distinguish between different classes. Table 4 shows the labeling for fault locations, while Table 5 presents the labeling for fault types.

Table 4: Fault Locations with Corresponding Textual Labels

Fault Location	Textual Label
Healthy	H
Position Accelerator Pedal	L1
Angle Steering Wheel	L2
Engine Speed	L3
Position Accelerator Pedal & Angle Steering Wheel	L1L2
Position Accelerator Pedal & Engine Speed	L1L3
Angle Steering Wheel & Engine Speed	L2L3

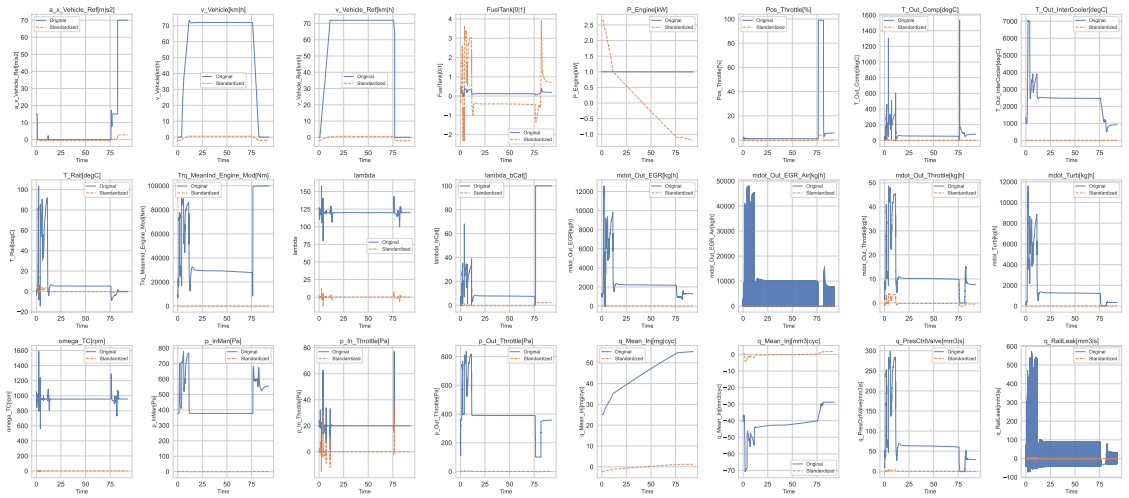
Table 5: Fault Types with Corresponding Textual Labels

Fault Type	Textual Label
`Healthy`	H
`Noise`	F1
`Gain`	F2
`Offset`	F3
`Noise & Gain`	F1F2
`Noise & Offset`	F1F3
`Gain & Offset`	F2F3

4.5 Data Scaling

After cleaning the CSV data, standardization is applied to ensure all features have a mean of zero and a standard deviation of one. This transformation is essential for ensuring consistent scales across features. Without standardization, features measured in different units or ranges could unevenly impact the model's performance. Figure Figure 12 illustrates the difference between the original and standardized data.

Figure 12: Comparison of Original vs. Standardized Data



4.6 Data Resampling

After standardizing the data, resampling techniques, such as random undersampling and SMOTE, are applied to address class imbalance. Random undersampling reduces the majority class size to match the minority class, minimizing training bias but potentially losing some data. SMOTE generates synthetic samples for the minority class, enabling the model to learn from underrepresented cases without duplicating data. These techniques work together to balance the datasets and improve the model's performance. Table 6 shows the counts of the original, random

undersampled, and SMOTE oversampled data samples for each fault location and fault type class.

Table 6: Data Distribution Across Fault Locations and Types: Original, Undersampled, and SMOTE

	Fault Location Data Count				Fault Type Data Count		
Location	Original	Undersampled	SMOTE	Type	Original	Undersampled	SMOTE
H	232,899	232,899	298,397	H	231,698	231,698	256,998
L1	298,397			F1	240,899		
L2	244,901			F2	245,796		
L3	237,001			F3	239,198		
L1L2	254,700			F1F2	243,399		
L1L3	246,298			F1F3	256,998		
L2L3	252,996			F2F3	249,299		

4.7 Time Series Windowing

Time-series windowing divides continuous data into smaller "segments", or "windows," to capture temporal patterns more effectively. The process is as follows:

Algorithm 1 Sliding Window for Time Series Data

Input: Time series data X, number of samples N, window size W
Output: List of windows with corresponding labels

for $i = 0$ **to** $N - W$ **do**
    $window \leftarrow X[i : i + W]$
    $label \leftarrow y[i + W - 1]$
    Store $(window, label)$
end for

Each window, defined by a `window_size`, is created by sliding across the dataset with a fixed step size, allowing for overlapping segments. The label for each window is assigned based on the final time point within that window, preserving the temporal order. This approach effectively captures time-based dependencies and structures the data for supervised learning tasks.

4.8 Data Splitting

The two datasets are split into three subsets: 70% for training, 15% for validation, and 15% for testing. The resampling method—whether random undersampling, SMOTE oversampling, or using the original data—affects the final size of each subset. Additionally, the number of data points in each subset depends on the selected window size and step size, which determine how the data is segmented.

4.9 Model Development

4.9.1 Baseline model Development

Baseline models are implemented using PyTorch [20] to establish a benchmark for evaluating RNN, LSTM, and GRU architectures. The default hyperparameters provided by PyTorch for each model, such as hidden size and the number of layers, were used without considerable tuning. This approach ensured a fair comparison among the three architectures and the proposed models, focusing on performance evaluation without the influence of hyperparameter optimization.

4.9.2 Proposed CNN-GRU Model

The proposed model employs a sequential architecture that integrates CNNs, GRUs, and FC layers for multi-class classification of time series data. Initially, the model uses CNN layers to extract spatial features from the input data. These layers apply convolutional filters to capture local patterns, enhancing the representation of the time series and establishing a strong foundation for subsequent classification tasks. After the CNN layers, GRU layers are added to handle the temporal aspects of the data. The gating mechanisms of GRUs regulate the flow of information, effectively addressing the vanishing gradient problem common in traditional recurrent networks. This sequential arrangement enables the model to learn and maintain long-term dependencies, essential for accurately classifying distinct classes based on historical data. The GRU layers process features extracted by the CNN layers, capturing the temporal dynamics that influence class predictions.
The model concludes with FC layers, which integrate the information learned from the CNN and GRU components. These layers transform high-level features into a format suitable for multi-class classification. By connecting each neuron in the previous layer to every neuron in the next, the FC layers facilitate the comprehensive integration of spatial and temporal information, supporting informed decision-making for class predictions.
This structured design combines the strengths of CNNs and GRUs to improve predictive performance in multi-class classification tasks.

4.9.3 Evaluation Metrics

Evaluation metrics are essential for evaluating the performance of DL models. Below are the key metrics commonly used in classification tasks:

4.9.3.1 Confusion Matrix The confusion matrix summarizes the performance of a classification model by displaying the counts of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN).

Table 7: Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	TP	FN
Actual Negative	FP	TN

4.9.3.2 Metrics The following metrics is used to evaluate the performance of the classification models:

- **Accuracy:** The proportion of correct predictions (both positive and negative) among all instances.

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

- **Precision:** The proportion of true positive predictions among all positive predictions made.

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

- **Recall:** The proportion of true positive predictions among all actual positive instances.

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

- **F1 Score:** The harmonic mean of precision and recall, providing a balance between both metrics.

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

4.9.3.3 ROC Curve The Receiver Operating Characteristic (ROC) curve is used to visualize the trade-off between sensitivity (true positive rate) and specificity (1 - false positive rate) across various thresholds. It provides insight into the model's ability to discriminate between positive and negative classes.

4.9.3.4 Precision-Recall Curve The Precision-Recall curve is particularly useful for imbalanced datasets. It illustrates the trade-off between precision and recall at different thresholds, helping evaluate a model's performance when positive instances are much fewer than negative ones.

4.9.4 Model Training and Hyperparameter Tuning

In DL, effective model training and hyperparameter tuning are crucial for optimizing performance. Table 8 lists the common hyperparameters used across various trials, offering a consistent basis for comparison.

Table 8: Shared Hyperparameters and Their Values Across All Trials

Hyperparameter	Value
Learning Rate	0.001
Batch Size	1024
Number of Epochs	256
Optimizer	Adam
Dropout Rate	0.3
L1	0.0001
L2	0.0001
Patience	8
Scheduler	Cosine Annealing Scheduler
Sampling Technique	None

Table 9 presents the key hyperparameters for various components of the neural network architecture, including CNN, GRU, and FC layers, as well as training parameters. It outlines a range of values for each hyperparameter, such as the number of layers, hidden sizes, resampling techniques, and window/step sizes to be tested in the model. These values are essential for evaluating the model's performance across different configurations and identifying optimal settings.

Table 9: Key Hyperparameters for Neural Network Architecture Components

Component	Hyperparameter	Range
Convolutional Layers	Number of Conv Layers	1 to 7
GRU Layers	Number of GRU Layers	1 to 4
	Hidden Size	32, 64, 256, 512
FC Layers	Number of FC Layers	1 to 7
Training Parameters	Resampling Technique	None, Undersample, SMOTE
	Window Size	10, 50, 100, 500, 1000
	Step Size	10, 50, 100, 500, 1000

4.9.4.1 Window Size and Step Size

Two key hyperparameters are considered: window size and step size. To prevent information loss, the step size is set to be equal to or smaller than the window size. A simple model with fewer layers is used to isolate the effects of these hyperparameters, enabling faster training and clearer results. This approach provides a better understanding of how window size and step size affect performance, without the added complexity of deeper architectures.

For the FLMs (Table 10), the optimal configuration is a window size of 500 and a step size of 10. This setup yields an accuracy of 62.66%, precision of 63.25%, recall of 63.14%, and an F1 score of 63.19%. It requires a substantial training time of 1244.39 seconds and a test time of 5.06 seconds.

Table 10: Performance Metrics of FLMs Based on Window Sizes and Step Sizes

Window Size	Step Size	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
10	10	49.46	51.15	50.72	50.93	500.97	4.71
50	10	59.60	60.23	60.51	60.37	755.81	4.60
	50	50.88	50.47	49.88	50.17	155.44	2.08
100	10	58.57	58.59	58.47	58.53	480.97	4.56
	50	50.69	49.19	48.79	48.99	121.66	2.08
	100	41.71	42.69	42.50	42.59	71.72	1.74
500	**10**	**62.66**	**63.25**	**63.14**	**63.19**	1244.39	5.06
	50	45.16	47.99	46.04	46.99	268.34	2.18
	100	42.03	47.66	45.95	46.79	174.22	1.81
	500	33.83	32.44	32.22	32.33	29.04	1.62
1000	10	49.20	50.61	50.86	50.73	876.49	6.60
	50	45.66	42.45	42.89	42.67	286.73	2.58
	100	45.05	44.82	46.68	45.73	290.35	1.95
	500	34.00	32.62	33.89	33.24	46.86	1.72
	1000	31.00	29.39	29.43	29.41	**25.73**	**1.47**

For the FTMs, as shown in Table 11, the best performance is achieved with a window size of 500 and a step size of 10, yielding an accuracy of 67.24%, precision of 68.39%, recall of 68.12%, and an F1 score of 68.25%. This configuration requires a training time of 316.76 seconds and a test time of 4.41 seconds.

Table 11: Performance Metrics of FTMs Based on Window Sizes and Step Sizes

Window Size	Step Size	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
10	10	64.23	69.83	68.15	68.98	471.93	4.35
50	10	64.71	69.46	68.21	68.83	935.87	5.44
	50	58.25	59.43	59.30	59.36	70.17	1.87
100	10	64.88	65.74	65.86	65.80	352.46	4.61
	50	54.39	59.59	57.48	58.52	72.70	2.13
	100	56.37	55.69	55.93	55.81	35.46	1.78
500	**10**	**67.24**	**68.39**	**68.12**	**68.25**	316.76	4.41
	50	57.38	60.16	58.04	58.47	99.98	2.28
	100	51.82	53.86	53.88	53.87	67.72	1.79
	500	46.22	49.19	48.79	48.99	36.05	1.46
1000	10	66.52	70.82	68.90	69.85	1599.13	6.42
	50	54.71	57.31	57.30	57.30	193.69	2.38
	100	51.36	57.80	55.02	56.38	140.10	2.04
	500	47.59	51.90	49.37	48.33	59.38	**1.51**
	1000	44.67	49.69	47.61	48.63	**18.80**	1.59

4.9.4.2 Number of Convolutional Layers Models with varying numbers of CNN layers are tested while keeping all other hyperparameters constant. The configurations are evaluated using the optimal window size and step size determined in the previous section. The model's performance is assessed by gradually increasing the number of CNN layers.

In the FLMs (as shown in Table 12), the best performance is achieved with four layers, resulting in an accuracy of 93.90%, a precision of 94.10%, a recall of 94.5%, and an F1 score of 94.30%. This configuration demonstrates exceptional predictive performance, with a training time of 3438.89 seconds and a test time of 5.09 seconds.

Table 12: Performance Metrics of FLMs Based on Number of CNN Layers

Number of Layers	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	88.91	89.16	90.32	89.74	4325.20	6.42
2	91.91	92.05	92.94	92.49	4421.86	6.02
3	91.41	91.49	92.60	92.04	4685.09	6.01
4	**93.92**	**94.05**	**94.55**	**94.30**	3438.89	5.09
5	93.70	93.83	94.47	94.15	**2069.09**	**4.92**
6	91.74	92.26	91.89	92.07	2099.00	5.11
7	88.58	88.64	90.10	89.36	2598.76	6.23

In the FTMs (Table 13), optimal performance is achieved with five convolutional layers, resulting in an accuracy of 85.65%, a precision of 86.69%, a recall of 85.99%, and an F1 score of 86.34%. This configuration required a training time of 5726.90 seconds and a test time of 6.80 seconds

Table 13: Performance Metrics of FTMs Based on Number of CNN Layers

Number of Conv Layers	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	69.41	74.77	70.85	72.74	4936.85	7.83
2	65.77	66.90	67.00	66.95	**1976.16**	7.38
3	77.87	78.29	79.25	78.77	8382.03	7.77
4	81.61	82.40	83.25	82.82	6171.83	**6.70**
5	**85.65**	**86.69**	**85.99**	**86.34**	5726.90	6.80
6	81.42	84.50	84.33	84.42	5787.48	8.03
7	83.89	84.26	84.70	84.48	6629.99	10.64

4.9.4.3 Number of GRU Layers and Layer Hidden Size The number of GRU layers and their hidden sizes are systematically tested in various combinations, while keeping all other hyperparameters—such as the optimal window size, step size, and the number of convolutional layers—constant, as determined in the previous sections.

In the FLMs (Table 14), the optimal result is achieved with 3 GRU layers and a hidden size of 512, yielding an accuracy of 94.25%, a precision of 94.43%, a recall

of 94.94%, and an F1 score of 94.68%. The training time for this configuration is 3523.99 seconds, with a test time of 5.20 seconds.
In the FLMs (Table 14), the optimal result is achieved with 3 GRU layers and a hidden size of 512, yielding an accuracy of 94.25%, a precision of 94.43%, a recall of 94.94%, and an F1 score of 94.68%. The training time for this configuration is 3523.99 seconds, with a test time of 5.20 seconds.

Table 14: Performance Metrics of FLMs Based on Number of GRU Layers and Hidden Size

Num of GRU Layers	Hidden Size	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	64	77.83	77.74	78.38	78.06	**753.43**	4.55
	128	86.53	87.16	88.20	87.68	1522.20	**4.21**
	256	88.01	87.91	89.30	88.60	1648.32	4.65
	512	86.69	86.76	87.54	87.15	1611.33	4.33
2	64	86.53	87.16	88.20	87.68	1522.20	**4.21**
	128	88.21	88.33	88.67	88.50	1602.48	4.47
	256	90.66	90.96	91.66	91.31	3602.52	4.73
	512	92.11	92.37	93.15	92.76	2462.76	5.12
3	64	88.01	87.91	89.30	88.60	1648.32	4.65
	128	90.08	90.32	91.31	90.81	1980.90	4.63
	256	92.65	92.83	93.89	93.36	2663.04	4.89
	512	**94.25**	**94.43**	**94.93**	**94.68**	3523.99	5.20
4	64	86.69	86.76	87.54	87.15	1611.33	4.33
	128	86.31	86.32	87.34	86.83	1351.59	4.77
	256	91.13	91.23	92.28	91.75	2328.81	5.17
	512	92.70	93.15	93.99	93.57	3060.26	5.67

In the FTMs (Table 15), the best performance is achieved with 2 GRU layers and a hidden size of 512, yielding an accuracy of 85.24%, a precision of 86.55%, a recall of 86.16%, and an F1 score of 86.36%. The training time for this configuration is 4785.67 seconds, with a test time of 6.00 seconds.

Table 15: Performance Metrics of FTMs Based on Number of GRU Layers and Hidden Size

Num of GRU Layers	Hidden Size	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	64	79.62	80.78	81.10	80.94	4516.03	8.11
	128	80.62	81.65	82.66	82.15	**3410.30**	5.61
	256	83.11	83.96	84.67	84.31	4637.55	5.90
	512	84.09	84.94	84.52	84.73	5168.72	6.76
2	64	77.69	82.41	81.95	82.18	3840.58	7.84
	128	83.12	84.10	83.93	84.01	4041.52	5.83
	256	84.11	84.82	85.40	85.11	4809.30	5.99
	512	**85.24**	**86.55**	**86.16**	**86.36**	4785.67	6.00
3	64	77.55	82.07	81.87	80.18	5031.38	6.89
	128	80.84	86.44	85.40	83.61	4537.15	**5.80**
	256	81.72	83.65	82.44	82.88	3734.32	6.07
	512	83.26	83.73	84.91	84.32	4835.79	7.81
4	64	80.30	82.32	82.59	82.45	5095.18	5.91
	128	83.80	85.05	86.06	85.55	5606.89	7.14
	256	83.69	84.95	85.31	85.13	4484.93	6.41
	512	80.45	84.93	84.48	84.70	6414.27	8.17

4.9.4.4 Number of FC Layers

Models with varying numbers of FC layers are tested, while all other hyperparameters are kept fixed, including window size, step size, number of convolutional layers, number of GRU layers, and hidden size. The impact of the number of FC layers and the doubling of neurons per layer on model performance is analyzed.

For the Location Models (Table 16), the best performance is achieved with a single FC layer, yielding an accuracy of 94.30%, a precision of 94.20%, a recall of 95.00%, and an F1 score of 94.60%. The training time is 3688.41 seconds, with a test time of 5.80 seconds.

Table 16: Performance Metrics of FLMs Based on the Number of FC Layers

Number of FC Layers	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	**94.30**	**94.20**	**95.00**	**94.60**	3688.41	5.80
2	88.40	88.80	89.90	89.34	**2893.31**	5.27
3	89.60	90.00	91.50	90.75	3719.44	5.44
4	90.50	91.10	92.60	91.35	3714.94	5.23
5	90.30	90.80	92.10	91.44	3715.40	5.29
6	89.30	90.20	91.50	90.85	3737.80	**5.15**
7	84.80	86.50	87.20	86.85	3786.37	5.23

For the Type Models (Table 17), the best performance is achieved with a FC layer, yielding an accuracy of 82.11%, a precision of 83.79%, a recall of 83.04%, and an F1 score of 83.50%. The training time is 1629.88 seconds, with a test time of 5.16 seconds.

Table 17: Performance Metrics of FTMs Based on the Number of FC Layers

Number of FC Layers	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
1	82.11	83.79	83.04	**83.50**	1629.88	5.16
2	77.21	**84.43**	81.97	83.19	1573.53	4.83
3	81.10	81.54	81.70	81.62	1892.03	5.00
4	79.46	80.10	81.21	80.65	1363.30	5.06
5	74.85	77.14	77.48	77.31	**808.85**	5.01
6	80.88	83.25	**83.38**	83.31	1609.49	**4.59**
7	78.55	82.12	83.04	82.58	1781.09	4.74

4.9.4.5 Resampling Technique

In the experiments, optimal hyperparameters from previous runs are selected, and model performance is fine-tuned iteratively. The evaluation is conducted using three sampling techniques: SMOTE, oversampling, and using the original dataset.
For the FLMs (Table 18), the best performance is achieved with the undersampling technique, yielding an accuracy of 97.7%, precision of 97.7%, recall of 97.7%, and

an F1 score of 97.7%. This configuration requires a training time of 3045.39 seconds and a test time of 5.01 seconds.

Table 18: Performance Metrics of FLMs Using Different Resampling Methods

Sampling Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
Undersample	**97.7**	**97.7**	**97.7**	**97.7**	**3045.39**	**5.01**
Original Data	94.5	94.6	95.2	94.8	3269.09	5.74
SMOTE	97.4	97.4	97.4	97.4	5437.57	7.92

In the FTMs (Table 19), the oversampling method achieves the highest performance, with an accuracy of 93.54%, precision of 94.57%, recall of 93.55%, and an F1 score of 93.57%. This configuration requires 7,926.13 seconds for training and 9.96 seconds for testing.

Table 19: Performance Metrics of FTMs Using Different Resampling Methods

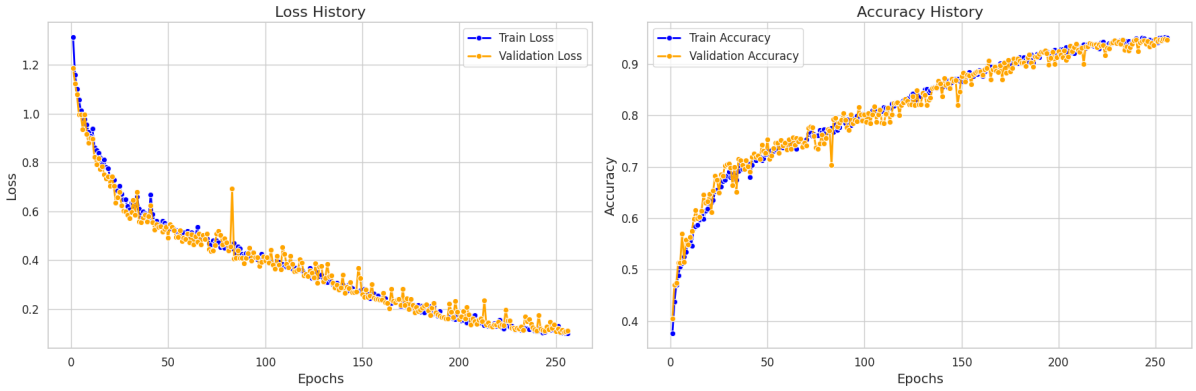
Sampling Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
Undersample	88.74	89.55	88.73	88.68	**4973.72**	7.88
Original Data	84.65	89.44	88.75	87.20	6248.59	**6.59**
SMOTE	**93.54**	**94.57**	**93.55**	**93.57**	7926.13	9.96

4.9.4.6 Final Models

Figure 13 shows the combined loss and accuracy history plots over 250 epochs, illustrating the model's performance during training.

- **Left Plot (Loss History)**: Both the training and validation losses decrease steadily, converging around 0.2 by the final epochs. This indicates that the model is learning effectively and generalizing well to new data, with minimal overfitting.
- **Right Plot (Accuracy History)**: The training and validation accuracies both increase consistently, reaching around 0.9 by the end of training. The close alignment between the two curves indicates stable model performance and continuous improvement throughout the training process.

Figure 13: Training and Validation Loss and Accuracy for the FLM over 250 Epochs



As shown in Table 20, the final FLM starts with four Conv1D layers (32, 64, 128, 256 channels) to capture spatial features, each using a kernel size of 3 with padding. A 3-layer GRU with 512 hidden units follows to capture temporal dependencies. A FC layer with 128 neurons is then applied. Finally, a linear layer reduces the output to 7 classes, providing the final output

Table 20: Layer-wise Summary of the FTM Model Architecture

Layer (Type)	Output Shape	Parameters (#)
Conv1d-1	[-1, 32, 500]	2,336
BatchNorm1d-2	[-1, 32, 500]	64
ReLU-3	[-1, 32, 500]	0
MaxPool1d-4	[-1, 32, 499]	0
Conv1d-5	[-1, 64, 499]	6,208
BatchNorm1d-6	[-1, 64, 499]	128
ReLU-7	[-1, 64, 499]	0
MaxPool1d-8	[-1, 64, 498]	0
Conv1d-9	[-1, 128, 498]	24,704
BatchNorm1d-10	[-1, 128, 498]	256
ReLU-11	[-1, 128, 498]	0
MaxPool1d-12	[-1, 128, 497]	0
Conv1d-13	[-1, 256, 497]	98,560
BatchNorm1d-14	[-1, 256, 497]	512
ReLU-15	[-1, 256, 497]	0
MaxPool1d-16	[-1, 256, 248]	0
GRU-17	[[-1, 248, 512], [-1, 2, 512]]	0
Linear-18	[-1, 128]	65,664
ReLU-19	[-1, 128]	0
Dropout-20	[-1, 128]	0
Linear-21	[-1, 7]	903

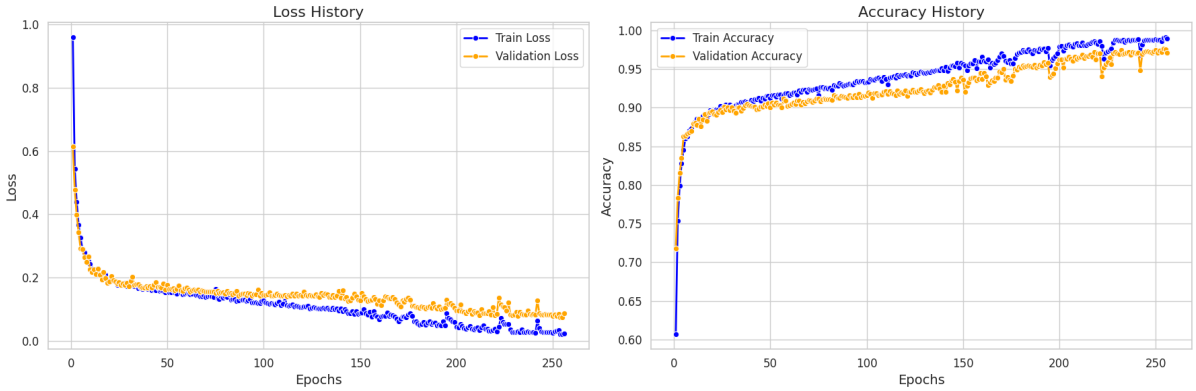
Total Parameters:	199,335
Trainable Parameters:	199,335
Non-Trainable Parameters:	0

Figure 14 shows the model's performance over 250 epochs for both the training and validation datasets.

- **Left Plot (Loss History)**: Both the training and validation losses decrease steadily, converging at approximately 0.2 by the final epoch. This trend suggests effective generalization and minimal overfitting.
- **Right Plot (Accuracy History)**: Both training and validation accuracies increase steadily, approaching 0.9 by the end of training. The close alignment between the two curves reflects stable performance and suggests that the model

generalizes effectively to the validation data.

Figure 14: Training and Validation Loss and Accuracy for the FTM over 250 Epochs



As shown in Table 21, the FTM consists of a series of Conv1D layers, with the number of channels progressively increasing from 24 to 512. Initially, a stride of 1 is used, transitioning to a stride of 2 in the deeper layers. This configuration effectively captures spatial features at multiple levels. A 2-layer GRU with 512 hidden units follows, capturing temporal patterns. The final linear layers reduce the feature dimensions first to 128, then to 7, producing the final output.

Table 21: Layer-wise Summary of the FLM Model Architecture

Layer (Type)	Output Shape	Parameters (#)
Conv1d-1	[-1, 32, 500]	2,336
BatchNorm1d-2	[-1, 32, 500]	64
ReLU-3	[-1, 32, 500]	0
MaxPool1d-4	[-1, 32, 499]	0
Conv1d-5	[-1, 64, 499]	6,208
BatchNorm1d-6	[-1, 64, 499]	128
ReLU-7	[-1, 64, 499]	0
MaxPool1d-8	[-1, 64, 498]	0
Conv1d-9	[-1, 128, 498]	24,704
BatchNorm1d-10	[-1, 128, 498]	256
ReLU-11	[-1, 128, 498]	0
MaxPool1d-12	[-1, 128, 497]	0
Conv1d-13	[-1, 256, 497]	98,560
BatchNorm1d-14	[-1, 256, 497]	512
ReLU-15	[-1, 256, 497]	0
MaxPool1d-16	[-1, 256, 496]	0
Conv1d-17	[-1, 512, 496]	393,728
BatchNorm1d-18	[-1, 512, 496]	1,024
ReLU-19	[-1, 512, 496]	0
MaxPool1d-20	[-1, 512, 495]	0
GRU-21	[[-1, 495, 512], [-1, 2, 512]]	0
Linear-22	[-1, 128]	65,664
ReLU-23	[-1, 128]	0
Dropout-24	[-1, 128]	0
Linear-25	[-1, 7]	903

Total Parameters: 594,087
Trainable Parameters: 594,087
Non-Trainable Parameters: 0

4.10 XAI Integration

The implementation of the feature importance computation methods, as discussed in the methodology section, is described. Several attribution techniques—including

IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP—are used to compute GFI, PCFI, and FIs.

4.10.1 GFI Computation

GFI provides an overall measure of how individual features influence the model's predictions across the entire dataset. The process involves calculating feature attributions for each sample, which are subsequently aggregated to compute the global importance of each feature.

Algorithm 2 GFI Computation

Input: Trained model M, dataset D, baseline type B (zero, mean, median, or random)
Output: GFI scores

procedure COMPUTE GLOBAL IMPORTANCE
    1. Prepare the model and dataset:
        Ensure that model M supports gradient-based attribution methods.

    2. Select baseline:
        Choose baseline B from the following options: Zero, Mean, Median, or Random.

    3. Compute local feature attributions:
    for each sample $x \in D$ **do**
            Compute feature attributions for sample x using one of the following methods:
                IGs, *DeepLIFT*, *Gradient SHAP*, or *DeepLIFT SHAP*.
    end for

    4. Aggregate local attributions:
        Compute the average of the attributions for all samples in D to obtain GFI scores.

    5. Validate results:
        Compare the global importance scores across different attribution methods and baselines to ensure consistency and reliability.
end procedure

4.10.2 PCFI Computation

PCFI extends GFI by focusing on the contribution of features specific to each class in a classification model. This approach helps in understanding how the model's decision-making process varies across classes.

Algorithm 3 PCFI Computation

Input: Trained model M, labeled dataset D, baseline type B (e.g., zero, mean, or domain-specific)
Output: PCFI scores

procedure COMPUTE CLASSWISE IMPORTANCE
    1. Prepare the model and dataset:
        Ensure that model M supports class-specific gradient-based attribution methods.

    2. Select baseline:
        Choose baseline B, such as zero vector, mean values, or domain-specific baselines.

    3. Compute local feature attributions:
    for each sample $x \in D$ **do**
            Compute feature attributions for sample x using one of the following methods:
                IGs, *DeepLIFT*, *Gradient SHAP*, or *DeepLIFT SHAP*.
    end for

    4. Aggregate attributions by class:
    for each class $c \in C$ **do**
            Group samples $x \in D$ by their true class label.
            Compute the average attribution scores for each feature within class c.
    end for

    5. Analyze class overlaps:
        Compare Per-Class attribution scores to identify shared and unique features across different classes.
end procedure

4.10.3 FIs Computation

FIs aims to uncover the combined effect of two or more features on the model's prediction, which may exceed the sum of their individual contributions. Understanding FIs provides deeper insights into how features interact to influence the model's output.

Algorithm 4 FIs Computation

Input: Model M, dataset D
Output: FIs scores

procedure ComputeFeatureInteractions
    1. Prepare the model: Ensure M supports gradient-based attribution methods.

    2. Compute individual feature attributions:
    for each feature $f_i \in D$ **do**
        Compute attribution for f_i using methods such as *IGs*, *DeepLIFT*, *Gradient SHAP*, or *DeepLIFT SHAP*.
    end for

    3. Analyze FIs:
    for each feature pair $f_i, f_j \subset D$ **do**
        Perturb both f_i and f_j, and compute the combined attribution score.
        If the combined score is significantly higher than the sum of individual scores, mark f_i and f_j as interacting features.
    end for

    4. Visualize FIs:
      Use heatmaps to display the interactions.
end procedure

4.11 Summary

This section outlines the implementation process, starting with a case study on a gasoline engine and simulation models. It covers the identification of optimal hyperparameters, including the configurations of CNN and GRU layers, the setup of FC layers, and time-series settings. XAI techniques were integrated to enhance model interpretability and feature analysis.

5 Results and Discussion

5.1 Models Evaluations

5.1.1 FLM

The evaluation plots (Figure 15) summarize the model's classification performance across seven classes, (H, L1, L2, L3, L1L2, L1L3, L2L3), with high accuracy and effective discrimination. Key observations:

- **Normalized Confusion Matrix:** The diagonal values indicate strong performance, with the L1L3 class achieving perfect accuracy (1.00). The L2 class shows the lowest accuracy at 0.88, with misclassifications primarily occurring with H (8%) and L1L2 (3%), suggesting some difficulty in distinguishing L2.
- **Probability Histogram:** The predicted probabilities are concentrated near 0 and 1, reflecting high confidence in predictions with minimal ambiguity.
- **Precision-Recall Curve:** High average precision (AP) scores are observed for each class, with L1, L1L3, and L1L2 achieving a perfect AP of 1.00, while L2 has an AP of 0.98. This indicates effective management of false positives and negatives.
- **ROC Curve:** The ROC Curve displays an AUC of 1.00 across all classes, demonstrating excellent sensitivity and specificity in distinguishing true from false positives.

Figure 15: Evaluation Plots of the FLM's Classification Performance

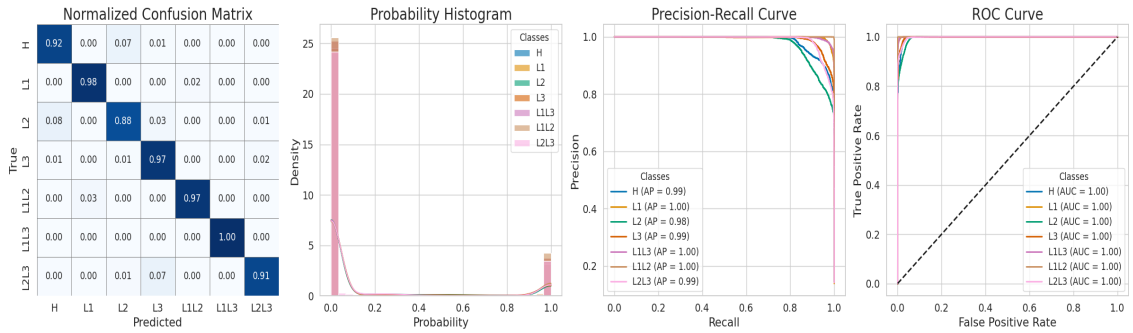


Table 22 presents the performance metrics for four DL models: RNN, LSTM, GRU, and FLM.

- **RNN**: This model had the lowest performance, with an accuracy of 43.07%, precision of 43.10%, recall of 43.06%, and an F1 score of 40.94%. It required 373.92 seconds for training and 7.36 seconds for testing.

- **LSTM**: The LSTM model showed significant improvement, achieving an accuracy of 57.73%, precision of 61.30%, recall of 57.70%, and an F1 score of 57.87%. Training took 699.93 seconds.
- **GRU**: The GRU model further enhanced performance, with an accuracy of 74.45%, precision of 74.84%, recall of 74.37%, and an F1 score of 74.45%. Its training time was 1356.55 seconds.
- **FLM**: The FLM model delivered exceptional results, achieving 97.40% across all metrics, though it required significantly more training time (22,998.75 seconds).

Overall, the table highlights a clear trend: as model complexity increases, performance improves, but there is a trade-off between accuracy and training time efficiency.

Table 22: Performance Metrics for the Baseline Models (RNN, LSTM, GRU) and FLM

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
RNN	43.07	43.10	43.06	40.94	**373.92**	7.36
LSTM	57.73	61.30	57.70	57.87	699.93	7.77
GRU	74.45	74.84	74.37	74.45	1356.55	7.92
FLM	**97.40**	**97.40**	**97.40**	**97.40**	22998.75	**5.01**

5.1.2 FTM

The evaluation plots (Figure 16) indicate excellent classification performance across the seven classes (H, F1, F2, F3, F1F2, F1F3, F2F3), with high accuracy and effective discrimination. Key observations:

- **Normalized Confusion Matrix**: The model classifies most classes accurately, with diagonal values close to 1.00. Minor misclassifications, particularly in F3, suggest some overlap with other categories.
- **Probability Histogram**: Predictions are made with high confidence, with probabilities clustering near 0 or 1. Any intermediate distributions indicate lower certainty in specific cases.
- **Precision-Recall Curve**: Near-perfect AP scores reflect strong precision and recall, showing minimal false positives and high sensitivity.

- **ROC Curve**: Scores are nearly perfect for all classes, highlighting the model's consistent ability to differentiate between true and false positives effectively.

Figure 16: Evaluation Plots of the FTM's Classification Performance

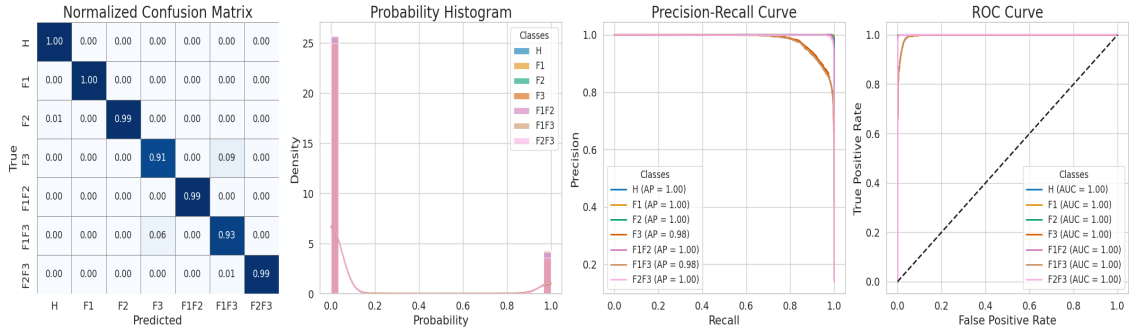


Table 23 presents the performance metrics for four DL models: RNN, LSTM, GRU, and FTM.

- **RNN**: This model had the lowest performance, with an accuracy of 69.25%, precision of 67.96%, recall of 69.10%, and an F1 score of 66.38%. It required 3885.03 seconds for training and 7.08 seconds for testing.
- **LSTM**: The LSTM model showed a slight drop in accuracy to 67.99%, but improved on other metrics: precision of 68.64%, recall of 67.77%, and an F1 score of 66.17%. Training took 2493.00 seconds.
- **GRU**: The GRU model achieved significant gains, with an accuracy of 85.66%, precision of 86.98%, recall of 85.57%, and an F1 score of 85.76%. Its training time was 4702.98 seconds.
- **FTM**: The FTM model demonstrated the best performance, achieving 97.19% accuracy, 97.22% precision, 97.21% recall, and an F1 score of 97.21%. However, it had the longest training time at 7896.56 seconds.

Overall, the results highlight a clear trend: more complex models like GRU and FTM achieve better performance but require longer training times.

Table 23: Performance Metrics for Baseline Models (RNN, LSTM, GRU) and FTM

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
RNN	69.25	67.96	69.10	66.38	3885.03	7.08
LSTM	67.99	68.64	67.77	66.17	**2493.00**	6.65
GRU	85.66	86.98	85.57	85.76	4702.98	**5.84**
FTM	**97.19**	**97.22**	**97.21**	**97.21**	7896.56	6.10

5.2 XAI of FLM

5.2.1 GFI of FLM

Figure 17 presents the GFI results of FLM, comparing attribution methods (IGs, DeepLIFT, Gradient SHAP, DeepLIFT SHAP) and baseline methods (zero, mean, median, random).

5.2.1.1 Attribution Method Comparison

- **IGs**: displayed stable and moderate importance scores, highlighting a gradual accumulation of contributions along the input path. This method tended to smooth sharp transitions, concentrating on steady feature influence.
- **DeepLIFT**: assigned sharper and higher-magnitude scores to certain features. This behavior indicated a heightened sensitivity to baseline changes, making it particularly responsive to features with strong contrasts.
- **Gradient SHAP**: produced a wider distribution of scores, reflecting its sensitivity to subtle variations in both the data and the model's gradients. This method highlighted dynamic feature relationships, making it prone to increased variability.
- **DeepLIFT SHAP**: captured feature importance in a manner similar to Gradient SHAP but with less variability. This approach provided more balanced and consistent importance scores, making it suitable for stable attributions.

5.2.1.2 Baseline Comparison

- **Zero Baseline**: represented a scenario in which all features were set to zero. This approach often emphasized features with strong linear relationships to the target variable. Notable examples included `P_Engine[kW]` and `T_Out_Comp[degC]`, which received higher importance scores under this baseline.

- **Mean Baseline**: used the mean values of features as a reference point. This baseline reduced sensitivity to extreme values, leading to smoother distributions of importance scores. Features generally displayed less pronounced contrasts under this approach.

- **Median Baseline**: applied the median as the reference point, providing robustness against outliers. Moderate variations in feature importance were observed compared to the zero baseline, which makes this strategy effective for datasets with skewed distributions.

- **Random Baseline**: introduced variability by randomly sampling baseline values. This stochastic approach led to broader spreads in importance scores, particularly noticeable for features like `a_x_Vehicle_Ref[m|s2]` and `lambda_bCat[]`.

5.2.1.3 Feature Level Analysis

- **Consistently Important Features**: `P_Engine[kW]` and `q_PresCtrlValve[mm3|s]` consistently ranked high in importance, especially under the zero baseline and using DeepLIFT. These features are key contributors to the model's predictions.

- **Highly Variable Features**: `a_x_Vehicle_Ref[m|s2]` and `v_Vehicle[km|h]` showed significant variability across baselines, particularly under the random baseline. This suggests dynamic relationships with the target variable, influenced by baseline values.

- **Baseline Sensitive Features**: `lambda` and `lambda_bCat[]` showed substantial importance under the mean and median baselines, but their significance dropped under the random baseline, indicating sensitivity to baseline selection.

- **Intermediate Features**: `T_Out_Comp[degC]` and `q_Mean_Inj[mg|cyc]` displayed intermediate importance, with moderate scores from IGs, reflecting their gradual influence on predictions.

- **Low-Impact Features**: `q_RailLeak[mm3|s]` and `mdot_Out_EGR_Air[kg|h]` had low and stable importance across all methods and baselines, suggesting they have minimal impact on the model's predictions.

Figure 17: GFI of FLM Interpretability Methods and Across Baselines

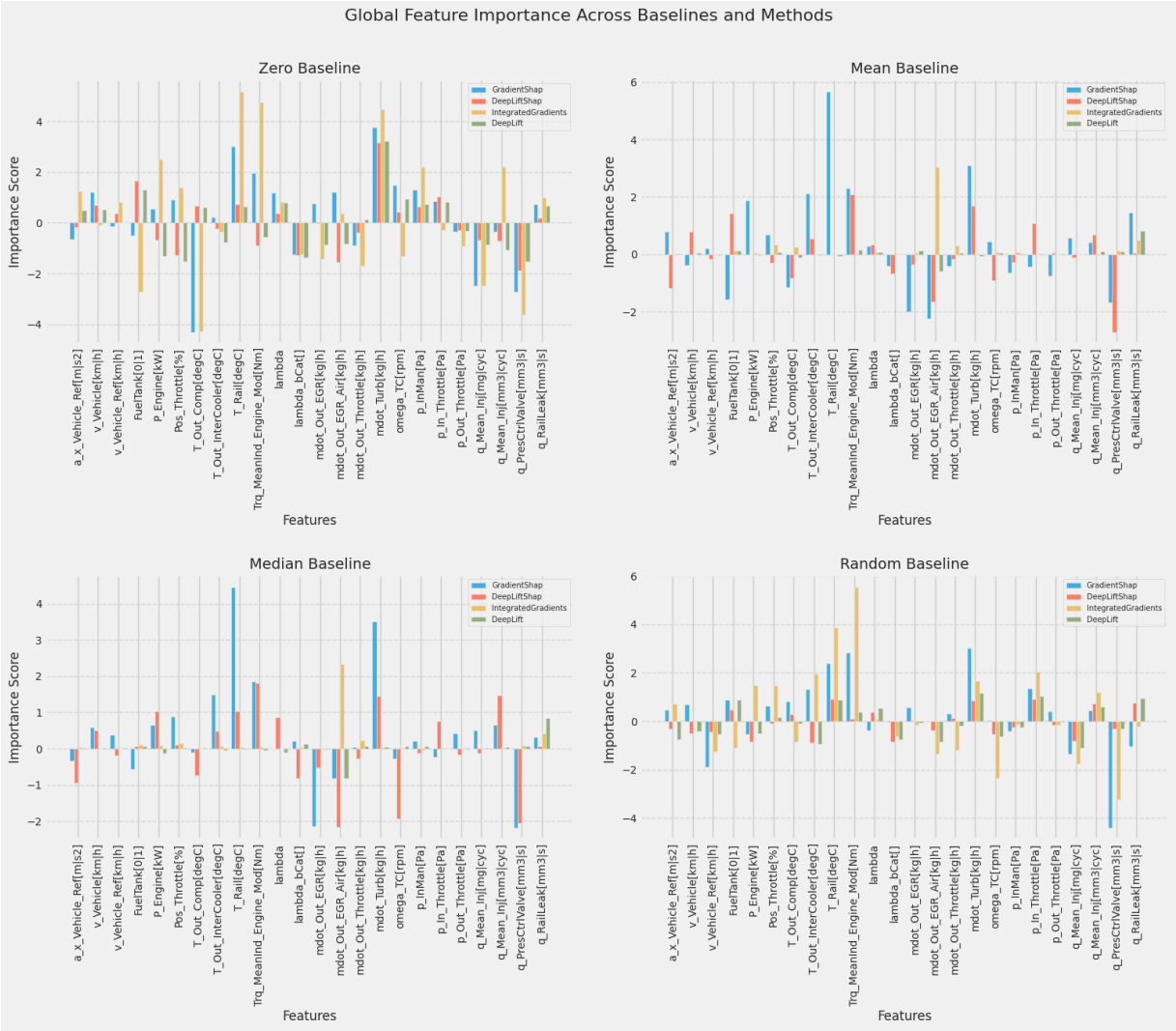


Table 24 presents a ranking of important features influencing the model's predictions based on GFI metrics calculated using IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP.

Table 24: Ranked Features for the FLM via IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP

Rank	Feature
1	P_Engine[kW]
2	q_PresCtrlValve[mm3|s]
3	lambda
4	T_Out_Comp[degC]
5	q_Mean_Inj[mg|cyc]
6	v_Vehicle[km|h]
7	a_x_Vehicle_Ref[m|s2]
8	T_Out_InterCooler[degC]
9	Pos_Throttle[%]
10	mdot_Out_Throttle[kg|h]

The FLM is retrained and evaluated using the features listed in Table 24. Table 25 shows that the retrained FLM has a slight reduction in accuracy, precision, recall, and F1 score (around 2%) compared to the original model. However, it achieves a substantial decrease in training time (from 22,999s to 5,413s) and a slight reduction in test time, enhancing computational efficiency.

Table 25: Performance Comparison of Original and Retrained FLM

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
Original FLM	**97.40**	**97.40**	**97.40**	**97.40**	22998.75	5.01
Retrained FLM	95.62	95.60	95.60	95.60	**5412.72**	**4.54**

5.2.2 PCFI of FLM

Figure 18 shows the PCFI results, comparing different attribution methods (IGs, DeepLIFT, Gradient SHAP, DeepLIFT SHAP) and various baselines (zero, mean, median, random), providing insights into the impact of method behavior and baseline choice.

5.2.2.1 Attribution Method Comparison

- **IGs**: exhibits the least variability, assigning importance values close to zero for most features. It suppresses noise effectively and is particularly well-suited for cases where a conservative approach to feature importance is required. However, this stability comes at the cost of potentially underemphasizing subtle but significant contributions, especially when compared to DeepLIFT SHAP.

- **DeepLIFT**: is relatively more stable compared to Gradient SHAP but demonstrates higher variability than IGs. It consistently highlights features like `a_x_Vehicle_Ref[m/s²]`, `Pos_Throttle[%]`, and `p_Out_Throttle[Pa]`. DeepLIFT's attributions are influenced by the baseline choice, and the zero baseline often exaggerates feature importance, while the median baseline provides more balanced outputs.

- **Gradient SHAP**: shows high variability and sensitivity, often emphasizing extreme feature contributions. Features like `Pos_Throttle[%]`, `p_InMan[Pa]`, and `mdot_Out_Throttle[kg/h]` stand out with large variations across baselines. Gradient SHAP's sensitivity makes it suitable for detecting sharp contrasts but can lead to overestimation of less significant features.

- **DeepLIFT SHAP**: refines the original DeepLIFT algorithm, adding a probabilistic dimension to feature attribution. It demonstrates smoother and more interpretable trends compared to both Gradient SHAP and DeepLIFT. For example, features such as `mdot_Turb[kg/h]` and `p_InMan[Pa]` consistently rank as important across classes with lower variability. DeepLIFT SHAP provides stable and balanced insights across baselines, making it a highly reliable attribution method.

5.2.2.2 Baseline Comparison

- **Zero Baseline**: often inflates the importance of features with values far from zero. Gradient SHAP and DeepLIFT are particularly sensitive under this baseline, with exaggerated peaks for features like `Pos_Throttle[%]` and `p_InMan[Pa]`. DeepLIFT SHAP and IGs exhibit more controlled outputs, though features such as `a_x_Vehicle_Ref[m/s²]` still show sharp importance peaks.

- **Mean Baseline**: smooths feature importance distributions across all methods. DeepLIFT SHAP and DeepLIFT, under this baseline, emphasize features like `mdot_Out_Throttle[kg/h]` and `mdot_Turb[kg/h]` consistently across most classes.

- **Median Baseline**: provides the most balanced and interpretable results across methods. DeepLIFT SHAP, in particular, highlights moderate importance for features such as `q_Mean_Inj[mm³/cyc]` and `lambda_bCat[]` without introducing excessive noise. DeepLIFT also benefits from this baseline, showing stable trends without overemphasizing outliers.

- **Random Baseline**: displays the most variability (from -1 to 4), with notable fluctuations in features like `P_Engine[kW]` and `Pos_Throttle[%]`. Gradient

SHAP highlights these fluctuations more than other methods, while DeepLIFT remains more stable.

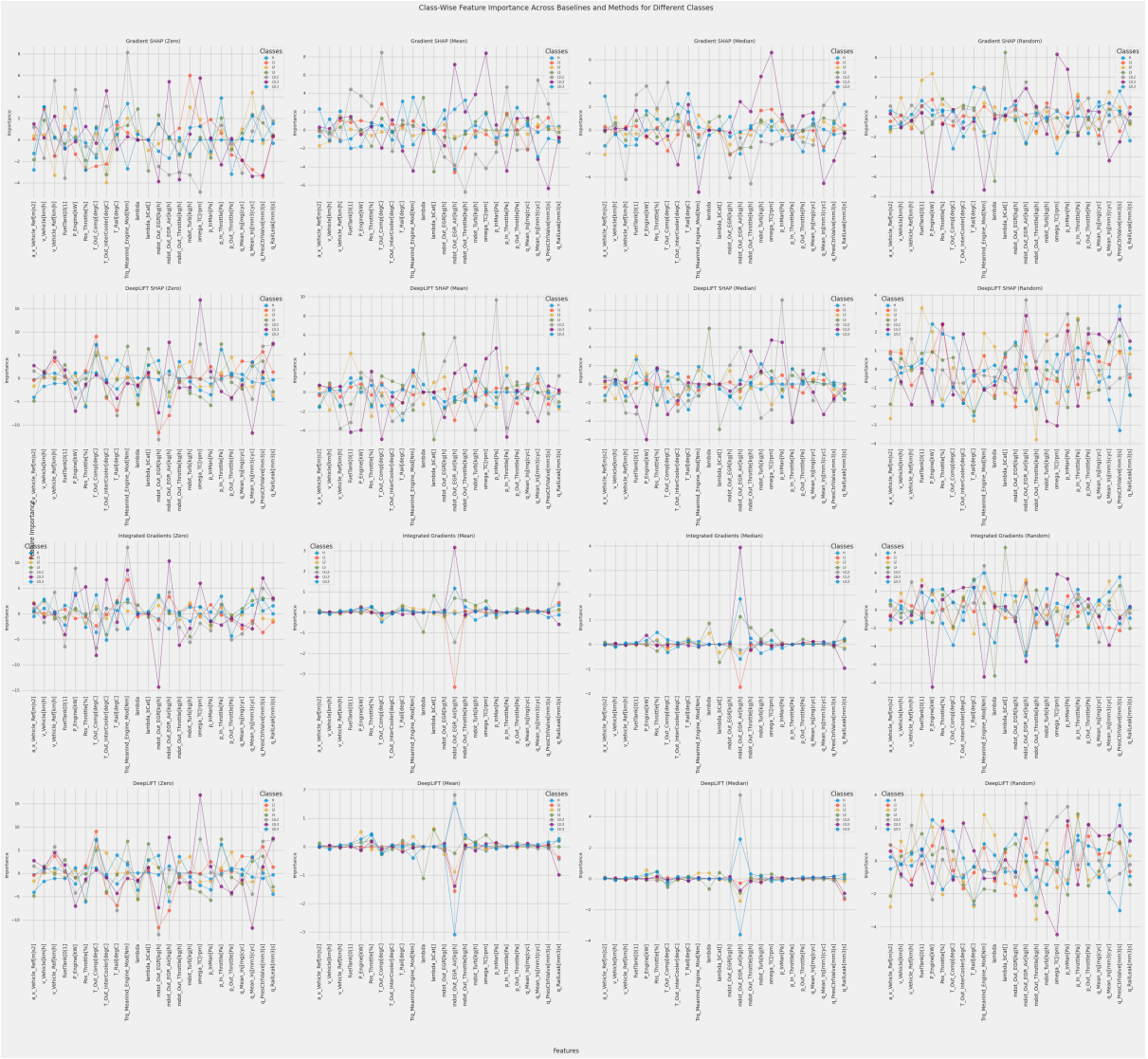
5.2.2.3 Per-Class Comparison

1. **Class H**: shows significant variability in feature importance, especially in Gradient SHAP and DeepLIFT. DeepLIFT SHAP identifies `Pos_Throttle[%]` and `p_InMan[Pa]` as critical features, particularly under the median and mean baselines. IGs, by contrast, assigns lower but consistent importance values to these features.

2. **Classes L1, L2, L3**: exhibit smoother trends in DeepLIFT SHAP compared to Gradient SHAP. Features like `mdot_Turb[kg/h]` and `a_x_Vehicle_Ref[m/s²]` consistently rank as moderately important across baselines. DeepLIFT also highlights these features but introduces greater variability, particularly in the zero baseline.

3. **Classes L1L2, L1L3, L2L3**: display uniform patterns of feature importance in DeepLIFT SHAP and IGs, indicating shared underlying feature contributions. Features such as `q_RailLeak[mm³/s]` and `lambda_bCat[]` emerge as moderately important for these classes.

Figure 18: PCFI of FLM Interpretability Methods and Baselines



5.3 XAI of FTM

5.3.1 GFI OF FTM

Figure 19 presents the GFI results of FTM, comparing attribution methods (IGs, DeepLIFT, Gradient SHAP, DeepLIFT SHAP) and baseline methods (zero, mean, median, random).

5.3.1.1 Attribution Method Comparison

- **IGs**: delivers a smoother and more evenly distributed importance score, highlighting moderate contributions from features across baselines. In the random baseline, it shows comparatively lower importance magnitudes, which may indicate a smoothing effect or a property specific to the gradient method.
- **DeepLIFT**: shows moderate feature importance, often aligning with DeepLIFT SHAP, but with differences in magnitude. It particularly focuses on features

like P_Engine[kW] and Pos_Throttle[%].

- **Gradient SHAP**: consistently produces higher magnitude importance scores than the other methods across all baselines, especially for high-sensitivity features. This is likely due to its use of gradients to approximate the impact of features. In the zero and median baselines, the method highlights significant contributions from features such as a_x_Vehicle_Ref[m|s2], v_Vehicle[km|h], and lambda, among others.

- **DeepLIFT SHAP**: strikes a balance between subtle and extreme importance scores, typically showing moderate values for most features. It often aligns with Gradient SHAP in identifying the same impactful features. However, it tends to highlight a wider range of features, assigning relatively non-zero scores to secondary ones, such as T_Out_InterCooler[degC].

5.3.1.2 Baselines Comparison

- **Zero Baseline**: where all features are compared against a baseline of zero values, some features such as a_x_Vehicle_Ref[m|s2], v_Vehicle[km|h], and lambda are shown to dominate across methods. The Gradient SHAP and IGs methods emphasize the importance of lambda and other features related to vehicle dynamics.

- **Mean Baseline**: when the mean of all feature distributions is used as the baseline, the importance scores become more pronounced for highly nonlinear features. This leads to a significant spike in importance for features like a_x_Vehicle_Ref[m|s2] and q_RailLeak[mm3|s]. Gradient SHAP dominates in magnitude, while IGs distributes importance across a broader range of features.

- **Median Baseline**: offers a robust central tendency for comparison, closely aligning with the patterns observed in the mean baseline but with slightly reduced variation. For instance, features like P_Engine[kW] and Pos_Throttle[%] gain prominence in importance across methods.

- **Random Baseline**: introduces variability, leading to more dispersed importance values. Features like a_x_Vehicle_Ref[m|s2] and lambda_bCat[] remain influential across methods, though their magnitudes are less extreme compared to structured baselines like zero or median.

5.3.1.3 Feature Level Analysis

- **Consistently Important Features**: a_x_Vehicle_Ref[m|s2] and v_Vehicle[km|h] are the most important features, highlighting their key role

in the model's predictions. a_x_Vehicle_Ref[m|s2] is especially crucial in the zero and mean baselines, while v_Vehicle[km|h] is significant in Gradient SHAP and DeepLIFT SHAP. lambda, related to engine efficiency, is also consistently important, along with q_RailLeak[mm3|s], which emphasizes system efficiency.

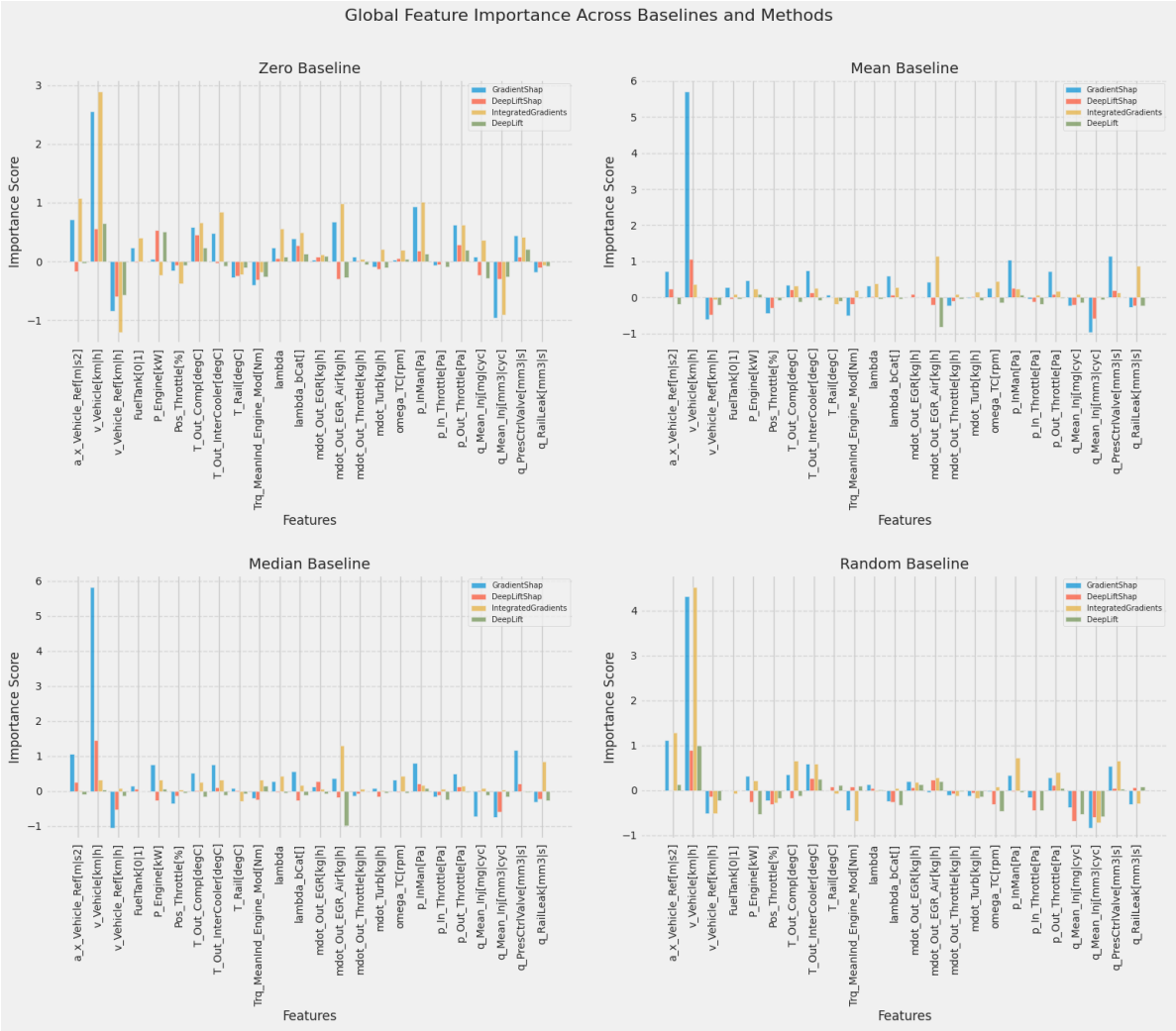
- **Highly Variable Features**: P_Engine[kW] shows significant variation, being prominent in the median baseline but less so in the random baseline, suggesting sensitivity to reference conditions. Pos_Throttle[%] is important in some baselines and methods like DeepLIFT but negligible in others, indicating conditional relevance. q_PresCtrlValve[mm3|s] also varies across baselines, highlighting its secondary yet context-dependent role in the model's predictions.

- **Baseline Sensitive Features**: q_RailLeak[mm3|s] shows high importance in the mean baseline, highlighting its relevance in scenarios with averaged feature conditions. lambda_bCat[] is more significant in the random baseline, capturing variability in less structured data. mdot_Out_EGR[kg|h] gains importance in the median maseline, reflecting its relevance when data is centered around typical values.

- **Intermediate Features**: T_Out_Comp[degC], mdot_Turb[kg|h], and p_InMan[Pa] all show moderate importance across methods and baselines. T_Out_Comp[degC] contributes to temperature-related predictions, mdot_Turb[kg|h] offers steady support, and p_InMan[Pa] is relevant in specific baselines, indicating its contextual utility.

- **Low-Impact Features**: q_Mean_Inj[mm3|cyc], q_Mean_Inj[mg|cyc], T_Rail[degC], p_Out_Throttle[Pa], and T_Out_InterCooler[degC] all show minimal importance, indicating they have little impact on the model's predictions. These features likely contribute little or are redundant within the model.

Figure 19: GFI of FTM Across Interpretability Methods and Baselines



The table 26 presents a ranking of important features that influence the model's predictions based on the GFI metrics calculated through IGs, DeepLIFT, Gradient SHAP and DeepLIFT SHAP.

Table 26: Ranked Features for the FTM via IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP

Rank	Feature
1	a_x_Vehicle_Ref[m|s2]
2	v_Vehicle[km|h]
3	lambda
4	q_RailLeak[mm3|s]
5	P_Engine[kW]
6	Pos_Throttle[%]
7	mdot_Out_EGR[kg|h]
8	q_PresCtrlValve[mm3|s]
9	T_Out_Comp[degC]
10	lambda_bCat[]

The FTM is retrained and evaluated using the features listed in Table 26.Table 27 shows that the retrained FLM has a slight reduction in accuracy, precision, recall, and F1 score (around 3%) compared to the original model. However, it achieves a substantial decrease in training time (from 22,999s to 5,413s) and a slight reduction in test time, enhancing computational efficiency.

Table 27: Performance Comparison of Original and Retrained FTM

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Train Time (s)	Test Time (s)
Original FTM	**97.19**	**97.22**	**97.21**	**97.21**	7896.56	6.10
Retrained FTM	94.45	94.47	94.48	94.46	**5869.24**	**4.77**

5.3.2 PCFI of FTM

Figure 20 shows the PCFI results, comparing different attribution methods (IGs, DeepLIFT, Gradient SHAP, DeepLIFT SHAP) and various baselines (zero, mean, median, random), providing insights into the impact of method behavior and baseline choice.

5.3.2.1 Attribution Method Comparison

- **IGs**: shows low variability, often assigning near-zero importance to many features, reflecting a conservative approach that downplays subtle contributions. However, features like a_x_Vehicle_Ref[m|s²] and Pos_Throttle[%] consistently emerge as important. The zero baseline produces minimal peaks

while maintaining steady importance for key features. The mean and median baselines provide smooth distributions, emphasizing features such as q_Mean_Inj[mm³|cyc]. While the random baseline introduces some variability, it remains more controlled than Gradient SHAP.

- **DeepLIFT**: balances the variability seen in IGs and Gradient SHAP, effectively highlighting key features like Pos_Throttle[%] and p_InMan[Pa] with moderate peaks. The zero baseline exhibits higher variability and exaggerated peaks, resembling Gradient SHAP. The mean and median baselines maintain balanced trends, emphasizing features such as mdot_Out_Throttle[kg|h] and lambda_bCat[]. The random baseline introduces moderate variability, with features like P_Engine[kW] and q_RailLeak[mm³/s] showing notable fluctuations.

- **Gradient SHAP**: reveals high variability in feature importance, with sharp peaks and dips, especially under the zero and random baselines. Features like Pos_Throttle[%], p_InMan[Pa], and mdot_Out_Throttle[kg|h] show consistently exaggerated importance. The zero baseline emphasizes features with large deviations from zero, such as Pos_Throttle[%] and p_InMan[Pa]. In comparison, the mean and median baselines exhibit smoother trends with less variability. The random baseline, however, shows the greatest fluctuations, highlighting its instability and dynamic nature.

- **DeepLIFT SHAP**: provides smoother and more balanced importance trends across baselines compared to Gradient SHAP. Features like mdot_Turb[kg|h] and p_InMan[Pa] consistently show high importance with minimal variability. The zero baseline reveals controlled peaks, though features like a_x_Vehicle_Ref[m|s²] occasionally experience sharp increases. The mean and median baselines exhibit the greatest stability, highlighting features such as mdot_Out_Throttle[kg|h] and q_Mean_Inj[mm³|cyc]. The random baseline shows moderate variability but remains more stable than Gradient SHAP.

5.3.2.2 Baseline Comparison

- **Zero Baseline**: consistently inflates importance for features with values significantly deviating from zero across methods. Gradient SHAP and DeepLIFT are particularly sensitive to this effect, producing more pronounced peaks, while IGs and DeepLIFT SHAP provide more controlled and balanced outputs. Features frequently exaggerated under this baseline include Pos_Throttle[%] and p_InMan[Pa], which consistently rank as highly important.

- **Mean Baseline**: smooths feature importance distributions, ensuring more stable trends. Features such as mdot_Out_Throttle[kg|h] and

q_Mean_Inj[mm³/cyc] consistently emerge as important across methods. While Gradient SHAP demonstrates reduced variability compared to the zero baseline, IGs and DeepLIFT SHAP excel in delivering the most balanced and refined outputs.

- **Median Baseline**: delivers the most balanced and interpretable results among the baselines. DeepLIFT SHAP consistently highlights features like q_RailLeak[mm³/s] and lambda_bCat[] with moderate importance, ensuring clarity and stability. DeepLIFT, in particular, benefits significantly from this baseline, exhibiting stable trends without the excessive peaks seen in other methods.

- **Random Baseline**: exhibits the highest variability across all methods, with Gradient SHAP being particularly sensitive, often emphasizing extreme feature importance values. Features such as P_Engine[kW] and Pos_Throttle[%] display notable fluctuations, highlighting the baseline's inherent instability and dynamic behavior.

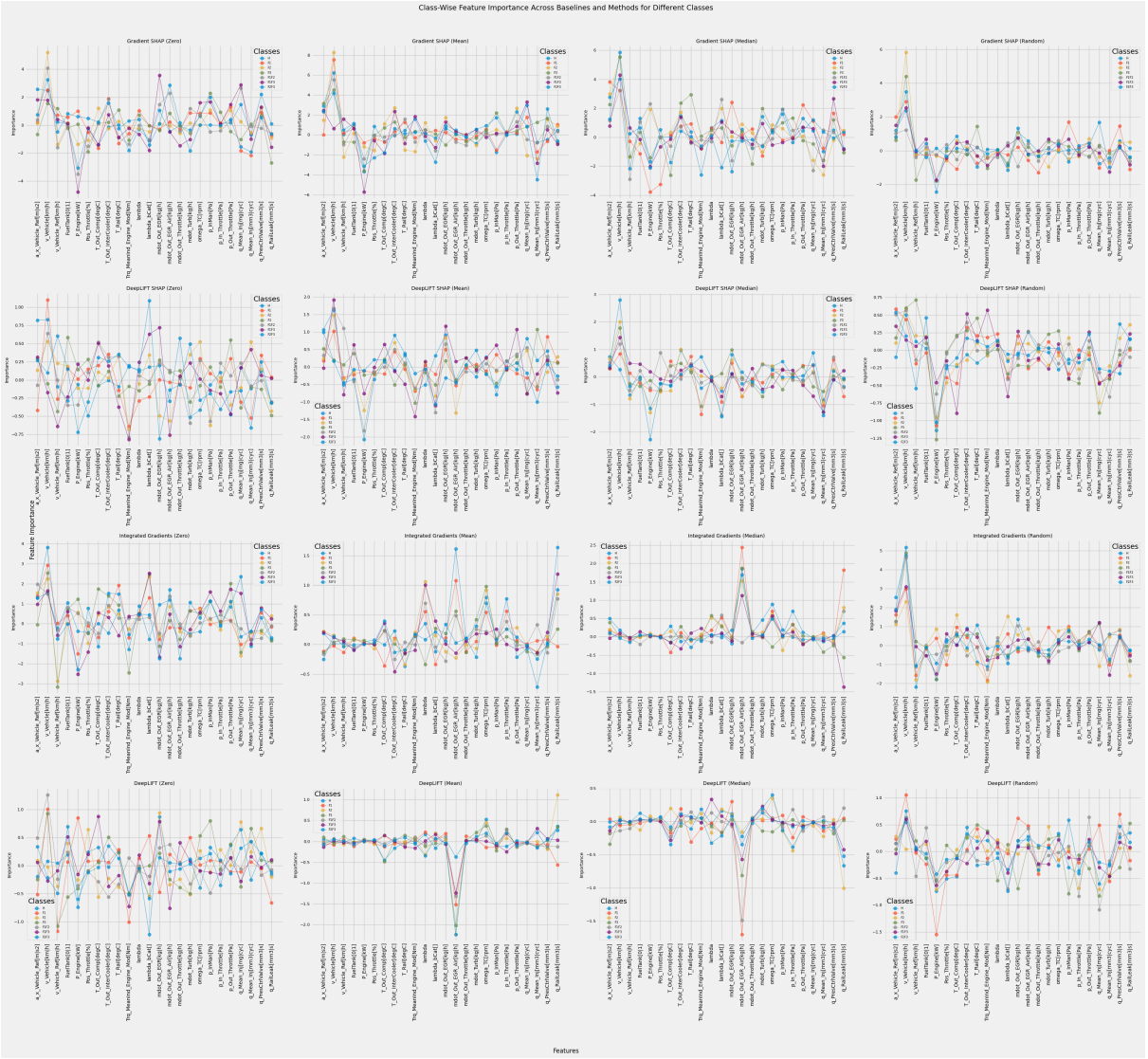
5.3.2.3 Per-Class Comparison

1. **Class H**: Gradient SHAP and DeepLIFT exhibit high variability, reflecting their sensitivity to feature importance fluctuations. In contrast, DeepLIFT SHAP effectively identifies critical features, such as Pos_Throttle[%] and p_InMan[Pa], particularly under the median and mean baselines, where it provides balanced and interpretable outputs. IGs consistently take a conservative approach, assigning low importance values across features.

2. **Classes L1, L2, L3**: DeepLIFT SHAP demonstrates smoother trends compared to the more variable Gradient SHAP. Features such as mdo_Turb[kg|h] and a_x_Vehicle_Ref[m|s²] consistently rank as moderately important across various baselines. However, DeepLIFT introduces greater variability, particularly under the zero baseline, where feature importance peaks are more pronounced.

3. **Classes L1L2, L1L3, L2L3**: uniform patterns in DeepLIFT SHAP and IGs highlight shared feature contributions across these groupings. Features such as q_RailLeak[mm³/s] and lambda_bCat[] consistently emerge as moderately important across different methods and baselines, reflecting their balanced and stable significance.

Figure 20: PCFI of FTM Across Interpretability Methods and Baselines



5.4 FIs

Figure 21 shows the FIs matrix using the Gradient SHAP attribution method and mean baseline, which highlights the interactions between different features in terms of their combined influence on the model's predictions.

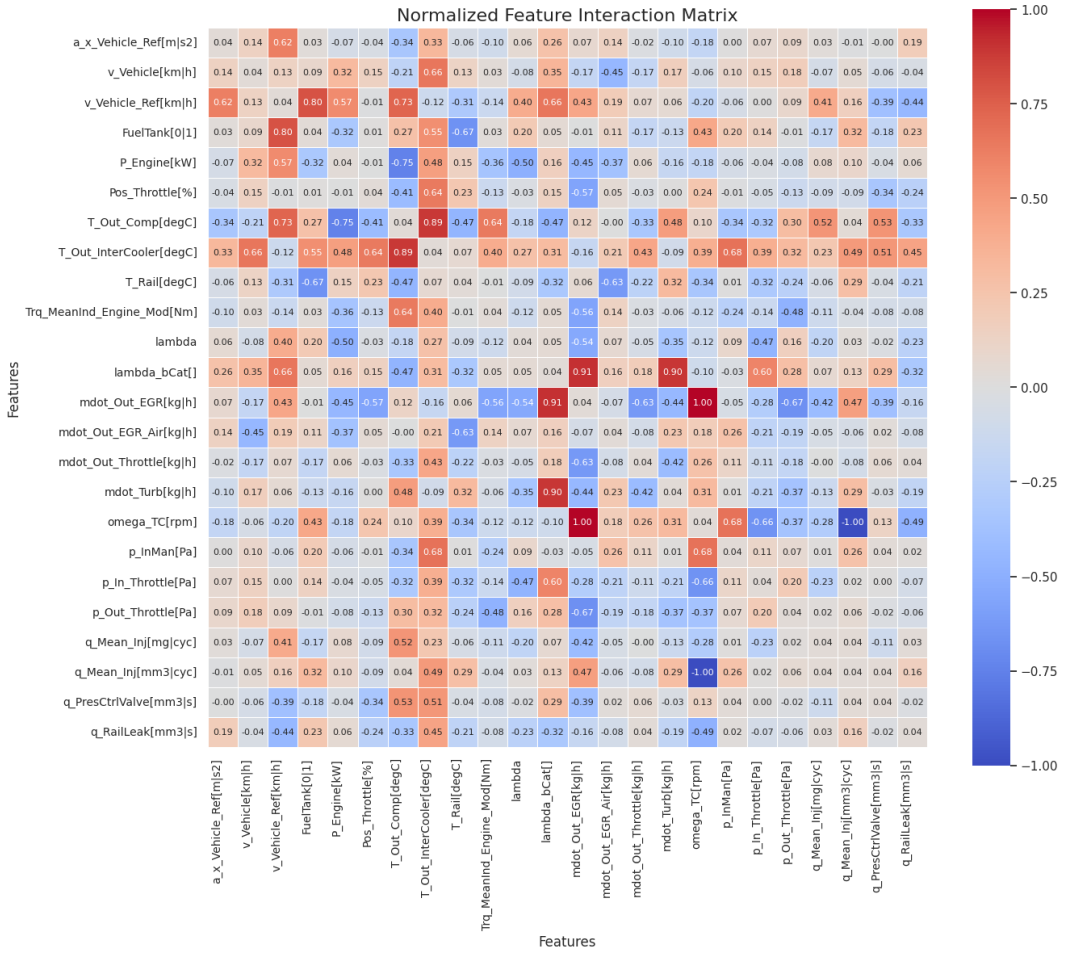
- **Strong Positive Interactions**: `lambda_bCat[]` and `mdot_Out_EGR[kg|h]` have a strong correlation (0.91), influencing emissions. `omega_TC[rpm]` (turbo speed) shows a perfect self-correlation (1.0) and a strong link with `mdot_Turb[kg|h]` (0.90). It also has a moderate correlation with `p_Out_Throttle[Pa]` (0.44). Finally, `FuelTank[0|1]` and `Pos_Throttle[%]` show a moderate interaction (0.52), reflecting fuel management during acceleration.

- **Strong Negative Interactions**: `lambda` and `mdot_Out_EGR_Air[kg|h]` have a strong negative interaction of -0.54, indicating an inverse relationship in

air-fuel dynamics. `q_RailLeak[mm3|s]` shows moderate to strong negative interactions, including -0.42 with `mdot_Out_Throttle[kg|h]`, suggesting a negative link with throttle airflow, and -0.49 with `omega_TC[rpm]`, indicating an inverse relationship with turbocharger speed.

- **Neutral Interactions**: `q_Mean_Inj[mm3|cyc]` and `q_Mean_Inj[mg|cyc]` show neutral interactions with most features, with values close to 0. Similarly, `a_x_Vehicle_Ref[m|s2]` (vehicle acceleration) exhibits weak interactions (around 0), indicating minimal influence on or from other factors.

Figure 21: Normalized FIs Heatmap



5.5 Interpretability vs. Complexity Trade-off

The interpretability-versus-complexity trade-off refers to the balance between the simplicity and speed of models that are easy to understand and the high performance achieved by more complex models, which may be harder to interpret.

5.5.0.1 Attribution Method Execution Times for Fault Location Models

Table 28 compares the time (in seconds) taken by four attribution methods across five models. DeepLIFT is the fastest method in most cases, with the RNN taking

just 0.92 seconds. IGs and Gradient SHAP are slower, especially for the RNN (10.42 seconds) and GRU (8.24 seconds). DeepLIFT SHAP is the slowest across all models, particularly for the Original FLM and Retrained FLM models, where it takes around 32 seconds.

Table 28: Comparison of Attribution Methods Across Fault Location Models (Times in seconds)

Model	IGs	DeepLIFT	Gradient SHAP	DeepLIFT SHAP
RNN	10.42	**0.92**	**1.75**	12.92
LSTM	8.71	3.28	3.76	11.27
GRU	**8.24**	3.53	3.77	**10.23**
Original FLM	19.56	3.65	3.53	32.83
Retrained FLM	19.29	3.55	3.65	32.19

5.5.0.2 Attribution Method Execution Times for Fault Type Models

Table 29 compares the time (in seconds) taken by four attribution methods across five models. DeepLIFT is the fastest method in most cases, with the RNN taking just 0.40 seconds. IGs and Gradient SHAP are slower, especially for the RNN (4.95 seconds) and GRU (4.01 seconds). DeepLIFT SHAP is the slowest across all models, particularly for the Original FTM and Retrained FTM models, where it takes around 13 seconds.

Table 29: Comparison of Attribution Methods Across Fault Type Models (Times in seconds)

Model	IGs	DeepLIFT	Gradient SHAP	DeepLIFT SHAP
RNN	4.95	**0.40**	**0.73**	6.04
LSTM	4.33	1.83	1.91	5.50
GRU	**4.01**	1.83	1.97	**4.86**
Original FTM	6.52	1.12	1.11	13.29
Retrained FTM	6.86	1.09	1.08	13.22

6 Conclusion and Future Work

This thesis presents two hybrid DL models for fault detection and diagnosis of ASSs during the real-time validation process. The models combine CNNs for feature extraction and GRUs to capture the temporal dependencies in sequential data. The FLM and FTM achieved high performance, with accuracies of 97.40% and 97.19%, respectively, demonstrating their effectiveness in classifying fault types and locations.

Four XAI techniques—IGs, DeepLIFT, Gradient SHAP, and DeepLIFT SHAP—to enhance interpretability. IGs balanced sensitivity and stability, DeepLIFT provided consistent attributions, and Gradient SHAP effectively identified key features but sometimes overemphasized them. DeepLIFT SHAP emerged as the most reliable method, offering interpretable results at a higher computational costs. The analysis used four baselines: zero, mean, median, and random. The zero baseline showed stable feature importance, while the mean and median raselines exhibited greater variability. The random raseline introduced the most variability, especially with Gradient SHAP. GFI reduced the feature set from 24 to 10 with minimal accuracy losses—2.74% for FLMs and 1.78% for FTMs. PCFI pinpointed critical features for fault types and locations, enabling targeted diagnostics, while FIs highlighted the role of synergistic feature relationships.

DeepLIFT was the fastest XAI technique, while DeepLIFT SHAP was the slowest, taking 32 seconds for FLMs and 13 seconds for FTMs. Method and baseline selection depend on analysis goals, balancing accuracy, interpretability, and efficiency.

Future work could focus on expanding the range of fault types, particularly concurrent faults, and exploring advanced interpretability techniques, such as Sensitivity Analysis, Uncertainty Analysis, and Counterfactual Explanations. These approaches could further enhance model robustness and clarity. Additionally, investigating XAI at the architectural level could provide deeper insights into the model's decision-making process, ultimately improving transparency and trust.

Bibliography

[1] Mohammad Abboush, Daniel Bamal, Christoph Knieke, and Andreas Rausch. Hardware-in-the-loop-based real-time fault injection framework for dynamic behavior analysis of automotive software systems. *Sensors*, 22(4), 2022.

[2] Ethem Alpaydin. *Introduction to Machine Learning.* MIT Press, Cambridge, MA, 4th edition, 2020.

[3] Aitor Arrieta, Irune Agirre, and Ane Alberdi. Testing architecture with variability management in embedded distributed systems. In *Proceedings of the International Workshop on Embedded Systems*, September 2013.

[4] Alejandro Barredo Arrieta, Natalia Díaz Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, A. Barbado, Salvador García, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Inf. Fusion*, 58:82–115, 2019.

[5] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):e0130140, 2015.

[6] Ali Behravan, Mohammad Abboush, and Roman Obermaisser. Deep learning application in mechatronics systems' fault diagnosis, a case study of the demand-controlled ventilation and heating system. In *Proceedings of the International Conference on Advanced Systems and Electric Technologies (ICASET)*, March 2019.

[7] Ali Behravan, Roman Obermaisser, and Mohammad Abboush. Fault injection framework for demand-controlled ventilation and heating systems based on wireless sensor and actuator networks. In *Proceedings of the IEEE International Conference on Information and Communication Technology Convergence (ICTC)*, November 2018.

[8] Yoshua Bengio and Yann LeCun. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2007.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[10] Lucas Costa Brito, Gian Antonio Susto, Jorge Nei Brito, and Marcus Antonio Viana Duarte. An explainable artificial intelligence approach for unsupervised fault detection and diagnosis in rotating machinery. *ArXiv*, abs/2102.11848, 2021.

[11] Eugenio Brusa, Luca Cibrario, Cristiana Delprete, and Luigi Gianpio Di Maggio. Explainable ai for machine fault diagnosis: Understanding features' contribution in machine learning models for industrial condition monitoring. *Applied Sciences*, 13(4), 2023.

[12] Marius Buda, Armin Maki, and Hans-Georg Müller. A systematic review of the performance of deep learning models for imbalanced data. *Knowledge-Based Systems*, 166:133–147, 2018.

[13] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106:249–259, October 2018.

[14] Diogo Vieira Carvalho, Eduardo Marques Pereira, and Jaime S. Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 2019.

[15] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[16] J. Chen, J. Song, X. Liao, C. Lin, C. Yang, and J. Xu. Learning to interpret neural networks with integrated gradients. *Proceedings of the 35th International Conference on Machine Learning*, pages 2538–2547, 2018.

[17] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.

[18] François Chollet. *Deep Learning with Python*. Manning Publications, New York, NY, 2018.

[19] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS*, pages 1–9, 2014.

[20] PyTorch Contributors. Pytorch, 2024. Accessed: 2024-12-10.

[21] Ana Maria Cornelia, Cristinel Ioan Murzea, Bogdan Alexandrescu, and Angela Repanovici. Expert systems with applications in the legal domain. *Procedia Technology*, 19:1123–1129, 2015.

[22] Xuewu Dai and Zhiwei Gao. From model, signal to knowledge: A data-driven perspective of fault detection and diagnosis. *IEEE Transactions on Industrial Informatics*, 9(4):2226–2238, 2013.

[23] Tina Danesh, Rachid Ouaret, Pascal Floquet, and Stéphane Négny. Hybridization of model-specific and model-agnostic methods for interpretability of neural network predictions: Application to a power plant. *Computers & Chemical Engineering*, 176:108306, 2023.

[24] Pedro Domingos. Supervised learning algorithms: A review. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1558–1572, 2018.

[25] DProgrammer. Rnn, lstm, and gru, 2023. Accessed: 2025-1-15.

[26] dSPACE GmbH. *Automotive Simulation Models (ASM)*, 2019. Accessed: 2024-11-24.

[27] eInfochips Blog. Why is model-based design important in embedded systems?, n.d. Accessed: 2024-11-16.

[28] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[29] Tagir Fabarisov, Ilshat Mamaev, Andrey Morozov, and Klaus Janschek. Model-based fault injection experiments for the safety analysis of exoskeleton system, 2021.

[30] Alberto Fernández, Salvador Garcia, Francisco Herrera, and Nitesh V Chawla. Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. *Journal of artificial intelligence research*, 61:863–905, 2018.

[31] P.M. Frank. Robust model-based fault detection in dynamic systems. *IFAC Proceedings Volumes*, 25(4):1–13, 1992. IFAC Symposium on On-line Fault Detection and Supervision in the Chemical Process Industries, Newark, Delaware, 22-24 April.

[32] Zhiwei Gao, Carlo Cecati, and Steven X. Ding. A survey of fault diagnosis and fault-tolerant techniques—part ii: Fault diagnosis with knowledge-based and hybrid/active approaches. *IEEE Transactions on Industrial Electronics*, 62(6):3768–3774, 2015.

[33] Si-Jie Gong, Shengwei Meng, Benkuan Wang, and Datong Liu. Hardware-in-the-loop simulation of uav for fault injection. *2019 Prognostics and System Health Management Conference (PHM-Qingdao)*, pages 1–6, 2019.

[34] David Gonzalez-Jimenez, Jon del Olmo, Javier Poza, Fernando Garramiola, and Patxi Madina. Data-driven fault diagnosis for electric drives: A review. *Sensors*, 21(12), 2021.

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2016.

[36] Klaus Greff, Rupesh K. Srivastava, Jakub Koutnik, Jürgen Schmidhuber, and Felix A. Gers. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[37] Garrett Grolemund and Hadley Wickham. *R for Data Science: Data Import, Tidy, Transform, Visualize, and Model*. O'Reilly Media, Sebastopol, CA, 2017.

[38] Md Junayed Hasan, Muhammad Sohaib, and Jong-Myon Kim. An explainable ai-based fault diagnosis model for bearings. *Sensors*, 21(12), 2021.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2nd edition, 2009.

[40] Simon Haykin. *Neural Networks and Learning Machines*. Pearson, Upper Saddle River, NJ, 3rd edition, 2009.

[41] Haibo He and Yang Bai. Theoretical and practical challenges in data mining with imbalanced classes. *ACM Computing Surveys*, 41(4):1–36, 2009.

[42] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[43] Aya Hesham. Multi-class classification vs multi-label, 2023. Accessed: 2025-1-15.

[44] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[45] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[46] Sana Ullah Jan, Young Lee, Jungpil Shin, and Insoo Koo. Sensor fault classification based on support vector machine and statistical time-domain features. *IEEE Access*, PP:1–1, May 2017.

[47] Garazi Juez Uriagereka, Estíbaliz Amparan, Ray Lattarulo, Alejandra Ruiz, Joshué Pérez Rastelli, and Huáscar Espinoza. Early safety assessment of automotive systems using sabotage simulation-based fault injection framework. In *Computer Safety, Reliability, and Security: SAFECOMP 2017*, pages 255–269, August 2017.

[48] Kazuya Kawakami. *Supervised sequence labelling with recurrent neural networks*. PhD thesis, Ph. D. thesis, 2008.

[49] Jongseon Kim, Hyungjoon Kim, HyunGi Kim, Dongjun Lee, and Sungroh Yoon. A comprehensive survey of time series forecasting: Architectural diversity and open challenges, 2024.

[50] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 25, pages 1097–1105, 2012.

[51] M. Kubat and S. Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In *Proceedings of the 14th International Conference on Machine Learning*, pages 179–186, 1997.

[52] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer, New York, NY, 2013.

[53] Thi-Thu-Huong Le, Haeyoung Kim, Hyoeun Kang, and Howon Kim. Classification and explanation for intrusion detection system based on ensemble trees and shap method. *Sensors (Basel, Switzerland)*, 22, 2022.

[54] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[55] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[56] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. Explainable ai: A review of machine learning interpretability methods. *Entropy*, 23(1), 2021.

[57] Octavio Loyola-Gonzalez. Black-box vs. white-box: Understanding their advantages and weaknesses from a practical point of view. *IEEE access*, 7:154096–154113, 2019.

[58] S. M. Lundberg and S. I. Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4765–4774, 2017.

[59] Oliver Mey and Deniz Neufeld. Explainable ai algorithms for vibration data-based fault detection: Use case-adadpted methods and critical evaluation. *Sensors (Basel, Switzerland)*, 22, 2022.

[60] Christoph Molnar. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Leanpub, 2020.

[61] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018.

[62] Tiziano Munaro and Irina Muntean. Early assessment of system-level safety mechanisms through co-simulation-based fault injection. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, page 1703–1708. IEEE Press, 2022.

[63] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, 2012.

[64] Alberto Peña, Iñaki Iglesias, Juan Valera, and A Martin. Development and validation of dynacar rt software, a new integrated solution for design of electric and hybrid vehicles. In *Proceedings of the 3rd International Conference on Electric Vehicles*, May 2012.

[65] PI.EXCHANGE. Understanding classification in machine learning, 2023. Accessed: 2025-1-15.

[66] Suresh R, Shanmughasundaram R, and Mohanrajan S R. Model based fault classification method for electric vehicle pertained lithium-ion batteries using multi layer perceptron. In *Proceedings of the IEEE International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–5, February 2020.

[67] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[68] Saeid Safavi, Mohammad Amin Safavi, Hossein Hamid, and Saber Fallah. Multi-sensor fault detection, identification, isolation and health forecasting for autonomous vehicles. *Sensors*, 21(7), 2021.

[69] Rabia Saleem, Bo Yuan, Fatih Kurugollu, Ashiq Anjum, and Lu Liu. Explaining deep neural networks: A survey on the global interpretation methods. *Neurocomputing*, 513:165–180, 2022.

[70] Mustafa Saraoğlu, Andrey Morozov, Mehmet Turan Söylemez, and Klaus Janschek. Errorsim: A tool for error propagation analysis of simulink models. In

Computer Safety, Reliability, and Security: SAFECOMP 2017, pages 245–254. Springer, September 2017.

[71] Scribbr. Supervised vs. unsupervised learning, 2023. Accessed: 2025-1-15.

[72] Di Shang, Emeka Eyisi, Zhenkai Zhang, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. A case study on the model-based design and integration of automotive cyber-physical systems. In *21st Mediterranean Conference on Control and Automation*, pages 483–492, 2013.

[73] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. *IEEE Computer*, 42:53–59, April 2009.

[74] Avanti Shrikumar, Patrick Greenside, Anish Shankar, Anshul Kundaje, Or Caspi, Pritish Pati, Lucas Mongan, Eleanor Choi, Jacob Steinhardt, Tom Sercu, et al. Learning important features through propagating activation differences. *International Conference on Machine Learning (ICML)*, 2017.

[75] Aparna Sinha, Shaik Fayaz Ahmed, and Debanjan Das. Explainable ai for bearing fault detection systems: Gaining human trust. *2023 IEEE Guwahati Subsection Conference (GCON)*, pages 1–6, 2023.

[76] Aparna Sinha and Debanjan Das. An explainable deep learning approach for detection and isolation of sensor and machine faults in predictive maintenance paradigm. *Measurement Science and Technology*, 35, 2023.

[77] Aparna Sinha and Debanjan Das. Xai-lcs: Explainable ai-based fault diagnosis of low-cost sensors. *IEEE Sensors Letters*, 7:1–4, 2023.

[78] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing Management*, 45(4):427–437, 2009.

[79] Yao Sun, A. Wong, and M. S. Kamel. Cost-sensitive learning and the class imbalance problem: A review. *Neural Networks*, 22(5-6):1067–1078, 2009.

[80] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328. PMLR, 2017.

[81] Subramanian V and Thomas O. A survey on supervised machine learning algorithms. *Journal of Machine Learning Research*, 20(1):45–60, 2019.

[82] Zichao Yang, Wen-tau Yih, Christopher Meek, Ruslan Salakhutdinov, and William W. Cohen. A survey of deep learning for sequence modeling. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2655–2671, 2019.

[83] Z. Zhang and Y. LeCun. Time series classification with convolutional neural networks. In *NeurIPS*, 2016.

[84] Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, and J Leon Zhao. Time series classification using multi-channels deep convolutional neural networks. In *International conference on web-age information management*, pages 298–310. Springer, 2014.

[85] Min Zhu, Jing Xia, Xiaoqing Jin, Molei Yan, Guolong Cai, Jing Yan, and Gangmin Ning. Class weights random forest algorithm for processing class imbalanced medical data. *IEEE Access*, 6:4641–4652, 2018.

[86] Haissam Ziade, Rafic Ayoubi, and R. Velazco. A survey on fault injection techniques. *International Arab Journal of Information Technology (IAJIT)*, 1:171–186, January 2004.