UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

ALERT SIGNAL CLASSIFICATION AND PREDICTION USING NATURAL LANGUAGE PROCESSING FOR THE TILE CALORIMETER OF ATLAS

by

Nicholas Perikli

718730

Dissertation

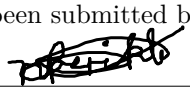
submitted in fulfillment of
the requirements for the degree
Master of Science
by research only
in
Physics
in the Faculty of Science,
University of the Witwatersrand,
Johannesburg, South Africa

Supervisor: **Prof. B. Mellado**

Date: 28 March 2025

Declaration

I, Nicholas Perikli, declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.



Signature of Candidate

28/03/2025

Date of Signage

Johannesburg, South Africa

Location of Signage

Abstract

This study investigated temporal patterns and spatial inhomogeneities in the A Toroidal Large Hadron Collider ApparatuS Tile Calorimeter Detector Control System (ATLAS TileCal DCS) alarm logs to minimize downtime and enhance data collection efficiency for double Higgs boson production during high-luminosity (HL) operations of the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) post the Phase-II upgrade in 2029. By applying Natural Language Processing (NLP) to detector control signals and machine learning (ML) techniques to train Long Short Term Memory (LSTM) models for both classification of future alarm types and forecasting categorical alarm rates in problematic modules, the approach achieved Mean Absolute Percentage Error (MAPE) scores between 12–23% and an overall classification accuracy of 75%. These results demonstrate the feasibility of a ML–driven predictive maintenance strategy, providing a possible early detection mechanism to optimize di-Higgs boson detection. Although further testing is needed before full integration, this LSTM-based framework shows promise for reducing unplanned downtime and accelerating scientific progress in the HL-LHC era.

Contents

Declaration **i**

Abstract **ii**

Contents **iii**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Research Context and Motivation 1

1.2 Research Objectives and Hypotheses 6

2 Theory and Background **8**

2.1 NLP and ML in Predictive Maintenance 8

2.2 NLP of Alarm Data and ML Task Design 18

3 Methodology **21**

3.1 Data Collection and Processing 21

3.2 Data Exploration and Visualization 30

3.3 LSTM Model Development 33

4 Results and Discussion **37**

4.1 Exploratory Data Analysis 37

4.2 LSTM Model Evaluation 42

4.3 Summary of Key Findings 47

4.4 Limitations of Study 48

5 Conclusion **49**

List of Figures

1.1 A sliced 3D view of the ATLAS detector's internal architecture highlighting the main sub-detector components[1]. 2

1.2 A breakdown of the Tile Calorimeter (TileCal) barrel architecture, highlighting the front-end electronics in the drawers of constituent wedge modules[2]. 3

1.3 Hierarchical overview of the TileCal Detector Control System (DCS), including its FE and BE systems, control stations (GCS, SCS, LCS), and integration with system interfaces (DSS, TDAQ, and other supporting infrastructure)[3]. 5

2.1 The internal structure of the central processing unit of an LSTM cell[4]. 12

4.1 The Modular Distribution of NO TOGGLE errors in the EBA partition displayed an uneven spatial distribution of errors with a wide range of counts: maximum of 60 (module 6) and a minimum of 2 (module 60) 38

4.2 Local Heatmaps showing High Modular Activity in LBA (left) and EBC (right) Partitions: Color intensity represents alarm counts (z-axis), module numbers (y-axis), and error categories (x-axis). 39

4.3 Time series comparison of raw (orange) and smoothed (blue) error counts for LBA Partition Module 48. The moving average filter reduces noise while preserving temporal signatures, enabling clearer pattern identification in this high-activity module. 40

4.4 Time series distributions of total error counts per partition (15-minute intervals) before and after smoothing. Colors denote partitions: blue (EBA), green (EBC), red (LBA), cyan (LBC). Chattering alarms in LBA (a) obscure patterns; post-smoothing (b) reveals distinct temporal profiles. 41

4.5 Smoothed time series distributions (15-minute intervals) partitioned by error type. Distinct temporal behaviors across partitions validate feature selection. Partition colors: blue (EBA), green (EBC), red (LBA), cyan (LBC). 41

4.6 Summary of LSTM Forecasting Performance for Highly Active TileCal Modules: Actual (blue) vs. Predicted (orange) trends with error bars. The data is partitioned using a 70:30 train-validation split, indicated by the dotted red vertical line. All models demonstrated strong predictive capabilities. 44

4.7 Graphical comparison of LSTM classifier performance: Top row shows recall-based matrices and bottom row shows precision-based matrices. 46

List of Tables

2.1 Mathematical Formulation of LSTM Cell Dynamics[5,6] 12

2.2 Performance Metrics for Time Series Prediction Tasks[7] 15

2.3 Performance Metrics for Time Series Classification Tasks[8] 16

2.4 Common Hyperparameters and Tuning Ranges in LSTM Models[9]. 17

2.5 Illustrative Extract of ATLAS DCS Alarm Logs for the TileCal Sub-detector, Demonstrating Partition-Based Hardware References and Key Fault Fields . . . 18

3.1 A sample illustration of the alarm log dataset after NLP-based feature extraction and restructuring. The columns shown here were derived from the original fields, capturing only the essential data needed forthe machine learning tasks. 25

3.2 Snapshot of the augmented time-series dataset for predictive modeling. Each sub-table corresponds to a unique combination of Partition (P1,...,P4), Error type (E1,...,E5), and Module (M1,...,M64). Column entries represent event counts (n) and their corresponding time bins (δt). 26

3.3 Detailed stepwise data transformation: merging by timestamp (duplicate entries) and splitting by class (`ALERT`) using 15-minute bins. 27

3.4 A simplified example of dominant error label assignment for two error classes (`EMR` and `ALERT`). 28

3.5 Example of a final classification table after summation and dominant error assignment for two error classes (`EMR` and `ALERT`). 28

3.6 Example of removing chattering alarms: repetitive `E3` alarms are reduced to a single entry with averaged counts per partition, keeping the index value of the first alarm. 29

3.7 Hyperparameter selection for LSTM models (prediction and classification tasks), illustrating the extensive tuning ranges explored during optimization. 35

4.1 The Comparison of TileCal Error Distributions by Partition Revealed Regional Susceptibility Differences for Targeted Maintainance. The table is organized column-wise by partition and row-wise by error type. 37

4.2 Optimal hyperparameter configurations for prediction and classification LSTM models, highlighting variations indicative of the presence of temporal complexities in the respective high activity modules, implying unique leveraging capabilities of the LSTMs. 42

4.3 Comparative Summary of LSTM Model Classifier Performance on their respective test sets. 45

Chapter1 Introduction

This chapter introduces the *Research Context and Motivation*, outlining the background of the Standard Model (SM) of particle physics with a specific focus on the ATLAS experiment at CERN. The section sets the stage by discussing the challenges faced by the Detector Control System (DCS) of the ATLAS detector, particularly from within the Tile Calorimeter (TileCal) subsystem, which is crucial for identifying hardware malfunctions that could disrupt operations. By providing a high-level overview of the dataset and emphasizing the significance of predicting hardware failures, this chapter also introduces the *Research Objectives and Hypotheses* that drive the study. The section concludes by highlighting the importance of predictive maintenance and the potential role of Natural Language Processing (NLP) and machine learning (ML), especially Long Short-Term Memory ($LSTM$[6]) networks, in improving the operational efficiency of high-energy physics (HEP) experiments. This Introduction establishes the problem space and the need for predictive tools, and sets the direction for the subsequent chapters.

1.1 Research Context and Motivation

The Standard Model of particle physics describes the interactions of elementary particles via the electromagnetic, weak, and strong nuclear forces. These interactions are formulated within a Quantum Field Theory (QFT), where particles are excitations of underlying fields governed by gauge symmetries[10].

The SM is structured around three primary QFTs: Quantum Electrodynamics (QED), which describes electromagnetic interactions under the $U(1)$ gauge group, mediated by the photon, γ[10]; Quantum Chromodynamics (QCD), which governs the strong nuclear force under $SU(3)$ symmetry, mediated by gluons, g, that bind quarks into hadrons[11]; and Electroweak Theory, which unifies the weak and electromagnetic forces under $SU(2)_L \times U(1)_Y$ symmetry, mediated by the $\mathcal{W}^{\pm}, \mathcal{Z}$ bosons, and the photon[10].

In the electroweak sector, the initial gauge bosons, $(\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3, \mathcal{B})$, mix after Electroweak Symmetry Breaking (EWSB) to form the observed $\mathcal{W}^{\pm}, \mathcal{Z}$ bosons and the massless photon[10]. EWSB occurs when the Higgs field acquires a non-zero vacuum expectation value (VEV), v, spontaneously breaking $SU(2)_L \times U(1)_Y$ down to $U(1)_{EM}$. This process leaves the photon massless while imparting mass to the weak bosons $\mathcal{W}^{\pm}$ and $\mathcal{Z}$[10]. The Higgs boson, a scalar excitation of the Higgs field, Φ, is central to this mechanism.

The Higgs potential is expressed as:

$$V(\Phi) = -\mu^2(\Phi^\dagger\Phi) + \lambda(\Phi^\dagger\Phi)^2 + \frac{1}{2}m_{12}^2(\Phi^\dagger\Phi) + \frac{\lambda_3}{3!}(\Phi^\dagger\Phi)^3 + \frac{\lambda_4}{4!}(\Phi^\dagger\Phi)^4 + \dots \tag{1.1}$$

[10] ,where μ^2 governs spontaneous symmetry breaking[10], λ controls Higgs self-interactions[12], and higher-order terms such as m_{12}^2, λ_3, and λ_4 introduce modifications beyond the SM[12].

Setting $m_{12}^2 = 0$, $\lambda_3 = 0$, and $\lambda_4 = 0$ reduces this to the SM Higgs potential. Expanding the Higgs field, Φ, around its VEV, given by $\Phi = \frac{1}{\sqrt{2}} \begin{smallmatrix} 0 \\ v+h \end{smallmatrix}$, allows identification of interaction terms, including the trilinear Higgs coupling, given by $\lambda_3 = 3\lambda v$[13]. The trilinear Higgs coupling, λ_3, is critical for Higgs boson pair production at the LHC and provides a direct probe of Higgs self-interactions[14]. Deviations from the SM prediction could signal new physics, such as additional scalar fields or modifications to the Higgs potential[12].

Higher-order terms like λ_4 influence vacuum stability and the electroweak phase transition, offering potential insights into extended Higgs sectors or composite Higgs models[15]. If these terms alter the Higgs potential, they could impact double Higgs production, affecting decay signatures such as $hh \to b\bar{b}\gamma\gamma$[14,12].

CERN hosts the LHC, the world's most powerful particle accelerator, enabling high-energy proton-proton and heavy-ion collisions to probe the fundamental structure of matter[16]. Higgs bosons at the LHC are produced through multiple channels, with a major challenge being the measurement of the trilinear Higgs coupling, primarily accessible via Higgs pair production[17]. Dominated by ggF, this process involves complex loop diagrams and has a low cross-section, making it difficult to observe[14].

Verifying Higgs couplings requires precise experimental techniques, statistical analysis, and optimized detector performance. The trilinear Higgs coupling remains a key focus for understanding the Higgs potential and probing beyond the Standard Model (BSM) physics[15]. Future collider upgrades must enhance luminosity, resolution, and data acquisition reliability to refine Higgs boson studies and explore new physics[18,14].

The LHC has progressively increased its energy and luminosity. Since Run-3 (2022–present), it operates at 6.5 TeV per beam with a peak luminosity of $2{\times}10^{34}$ cm^{-2}s^{-1}[18]. The HL-LHC, expected to be operational after Long Shutdown 3 (2026–2028), aims to boost peak luminosity by nearly a factor of four, reaching $7.5{\times}10^{34}$ cm^{-2}s^{-1}, thus enabling more precise Higgs boson measurements and deeper probes into new physics[16].

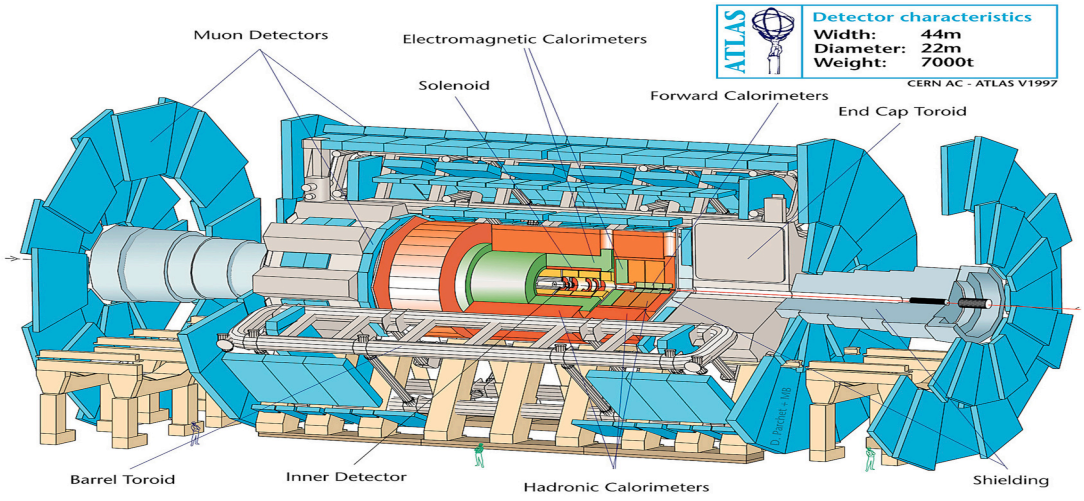


Figure 1.1: A sliced 3D view of the ATLAS detector's internal architecture highlighting the main sub-detector components[1].

ATLAS is a general-purpose detector at CERN[19]. This large-scale cylindrical apparatus, 46 meters in length and 25 meters in diameter, is symmetrically positioned around Interaction Point 1 (IP1). The detector employs multiple sub-detectors for charged particle tracking, energy measurements, and muon detection, enabling the study of particles deflected in various directions relative to the beamline[20]. The design, shown in Figure 1.1, supports high-energy collision analyses.

The ATLAS detector has been instrumental in studying the Higgs boson, contributing to its discovery in 2012 and ongoing analyses of its couplings, including the trilinear Higgs coupling[21,14]. Now, the HL-LHC will allow ATLAS to refine Higgs boson self-coupling measurements, search for rare decays, and investigate EWSB with unprecedented precision[18]. However, higher luminosity introduces challenges such as increased pile-up, greater radiation exposure, and higher data rates.

To address these, ATLAS is undergoing significant upgrades, including improved tracking, enhanced calorimetry, and advanced trigger systems[18,22]. With its increased data collection and improved detector capabilities, ATLAS at the HL-LHC will push the frontiers of particle physics, refining SM tests and expanding the search for new physics, offering deeper insights into the Higgs boson[23].

The Calorimeter System of ATLAS, consisting of the Electromagnetic Calorimeter (ECAL) and Hadronic Calorimeter (HCAL), captures and measures the energy of electrons, photons, and hadrons[22]. TileCal is the hadronic calorimeter of the ATLAS detector. Constructed from alternating layers of steel absorbers and plastic scintillator tiles, TileCal detects energy deposits through light signals collected via wavelength-shifting fibers and transmitted to photomultiplier tubes (PMTs)[24].

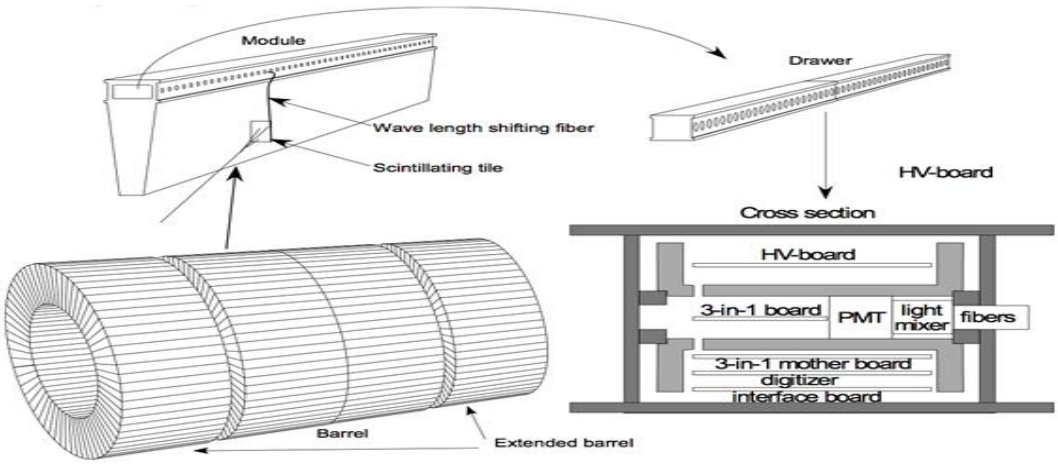


Figure 1.2: A breakdown of the Tile Calorimeter (TileCal) barrel architecture, highlighting the front-end electronics in the drawers of constituent wedge modules[2].

TileCal consists of a central long barrel and two extended barrels surrounding the liquid argon (LAr) calorimeters. These barrel regions are referred to as partitions as follows: Long Barrel A (LBA), Long Barrel C (LBC), Extended Barrel A (EBA), and Extended Barrel C (EBC)[24,25]. The front-end electronics of TileCal are housed in extractable mini-drawers within wedge modules, containing PMTs and high-voltage (HV) distribution cards that regulate HV

from an off-detector back-end supply[26,2]. These electronics amplify, digitize, and store signals before transmitting them to the Read-Out Drivers (RODs), which process and forward data to the ATLAS Data Acquisition (DAQ) system[26,2]. The configuration of TileCal barrels and their front-end electronics is illustrated in Figure 1.2.

The Phase-II upgrade of TileCal introduces a three-stage low-voltage power supply (LVPS) system with an intermediate voltage level, designed to improve efficiency and radiation tolerance[27,22]. The high-voltage power supply (HVPS) upgrade introduces a remote system to reduce radiation exposure, utilizing bulk supplies to distribute power via regulatory and slow control cards[27,22].

System testing and performance validation from 2019 to 2022 demonstrated the feasibility and reliability of these upgrades under HL-LHC conditions, confirming their functionality before large-scale deployment[22,27]. Based on these results, the upgraded LVPS architecture entered pre-production and prototype deployment[27,22]. While the upgrades enhance redundancy, they increase complexity and maintenance demands, necessitating predictive maintenance strategies for system reliability[27,22].

TileCal's electronics are integrated into the DCS of the ATLAS detector to ensure stable operation by continuously monitoring hardware conditions, detecting anomalies, and preventing component failures[28]. The DCS combines both software and hardware components, employing the Supervisory Control and Data Acquisition (SCADA) software WinCC OA, which uses data points (DPs) to monitor and control detector elements via a graphical interface. Real-time data is archived into an Oracle database for diagnostics, and automated responses to anomalies help maintain reliability during physics runs[28,29].

Structured into two layers, the DCS comprises a Front-End (FE) and a Back-End (BE)[28]. The FE hosts monitoring hardware that collects detector data, including LVPSs to power the front-end electronics and an HV distribution system to regulate PMT voltages[28,3].

The BE operates on PCs that process and relay this data through a hierarchy of control stations:Global Control Stations (GCS), overseeing ATLAS-wide operations; Sub-detector Control Stations (SCS), managing specific sub-detectors such as TileCal; and Local Control Stations (LCS), which regulate LV and HV systems for individual TileCal partitions[3,28]. The DCS also organizes TileCal into four main blocks—HV, LV, a central cooling system, and standalone systems—to ensure optimal power distribution and thermal conditions[30,28].

A Finite State Machine (FSM) framework underpins automated diagnostics and hierarchical control by assigning operational states (*OK, WARNING, ERROR, FATAL*) based on severity[28]. Alarm thresholds trigger real-time alerts, displayed in an Alarm Screen for rapid intervention[28]. Integrated sensors and actuators maintain optimal conditions, while safety interlocks can adjust or shut down operations under hazardous circumstances[3,28].

The DCS implements two primary recovery measures: power-cycling of faulty modules (fLVPS trip recovery) and a kill-switch mechanism for severe faults such as overheating or cooling failures. Power is restored only when safe conditions are reestablished, protecting the experiment from potential damage[3,28,29]. These measures also manage HV supplies and cooling systems to maintain detector performance and prevent overheating[29,28].

Additionally, the DCS receives signals from the DAQ system to execute automated recovery processes, log non-critical data in an electronic logbook (e-log), and provide real-time status updates, creating a feedback loop for enhanced stability[3,28,29]. It interfaces with the Trigger and Data Acquisition (TDAQ) system, which processes detector data for physics analyses, ensuring seamless integration between hardware control and data flow[29].

While the DCS governs standard operational control[28], the Detector Safety System (DSS) operates independently, continuously monitoring environmental conditions and system integrity. It autonomously transitions the detector to a safe state when necessary, covering situations beyond the scope of the DCS[29,28].

Figure 1.3 provides a hierarchical overview of the TileCal DCS within ATLAS, illustrating its structure and the interconnection of control domains and hardware/software interfaces that ensure robust operation.

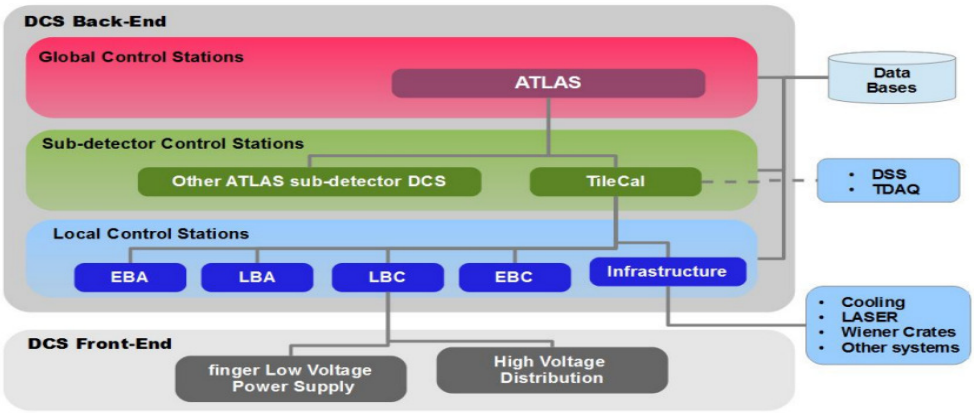


Figure 1.3: Hierarchical overview of the TileCal Detector Control System (DCS), including its FE and BE systems, control stations (GCS, SCS, LCS), and integration with system interfaces (DSS, TDAQ, and other supporting infrastructure)[3].

As the HL-LHC upgrade approaches, ATLAS is set to explore new frontiers in particle physics, particularly through Higgs di-boson detection and the measurement of the trilinear coupling. Achieving optimal experimental conditions is critical for an unambiguous measurement of the trilinear coupling via the main ggF channel, a primary focus of ATLAS HL-LHC operations[28].

The Phase-II upgrade is expected to increase hardware malfunctions due to enhanced operational demands on detector systems. In the ATLAS DCS, hundreds of components and calibration parameters are continuously monitored, generating a large stream of alarm logs. While essential for diagnosing issues, these logs can overwhelm operators if not addressed efficiently[30,31].

To mitigate this challenge, an artificial Intelligence (AI)-driven tool is proposed to predict and classify alarm signals recorded by the ATLAS DCS, with the goal of pre-empting future alarms and minimizing detector downtime. The approach leverages NLP techniques to analyze categorical alarm data, extracting hidden numerical time-series features that feed into ML models.

Two types of forecasting are envisioned: *classification* and *prediction*. The classification models would be trained to predict the *most prevalent alarm type* affecting the entire TileCal system over consecutive time intervals—one using the raw dataset and the other after smoothing the data. The prediction models are partition-specific, designed to forecast the frequency of category-specific alarms for a single type, focusing on one of the *most problematic TileCal modules* in each partition.

This dual-model strategy would provide operators with insights into which alarm types are most likely to occur next and the expected number of malfunctions in specific modules. By integrating this AI-driven early detection mechanism, coupled with the dual-model strategy, into the ATLAS DCS, more robust control systems, structured safety, and real-time preventative measures could be achieved—offering a critical upgrade from the current reactionary approach[32].

1.2 Research Objectives and Hypotheses

This study investigates whether temporal patterns exist within these alarm logs and whether specific detector locations exhibit a higher frequency of hardware malfunctions. If such patterns are present, they may reveal underlying hardware or operational inefficiencies. Given the time-dependent nature of these logs and the potential for spatial correlations, ML techniques—particularly LSTM networks[6]—offer a promising approach for analyzing sequential data. By capturing and leveraging these spatio-temporal patterns, LSTMs can enhance predictive maintenance strategies and reduce unplanned downtime, thereby safeguarding the detector's overall integrity.

This study is built on the following key hypotheses:

- **Temporal Non-Randomness:** Alarm events follow predictable time-series trends that can be effectively modeled using LSTM networks [6].
- **Spatial Inhomogeneity:** Specific problematic locations within TileCal, identified by consistently high error rates, dominate the overall alarm distribution.
- **Increased Hardware Malfunctions:** The Phase-II upgrade will drastically increase the frequency of hardware issues, heightening the need for robust predictive maintenance.
- **Operational Relevance:** An LSTM-driven approach to hardware fault detection can support critical functions, such as Higgs di-boson detection, by minimizing detector downtime[12].

To evaluate the proposed hypotheses, this study utilizes visualization techniques to establish the significance of the research and substantiate the application of LSTM networks. These visualizations are essential in establishing the presence of spatial and temporal patterns within the dataset, ensuring that predictive maintenance strategies are both data-driven and effectively implemented:

- **Frequency Plots:** These visualizations address the *spatial inhomogeneity* hypothesis by demonstrating how certain modules exhibit significantly higher error rates than others and the *increased hardware malfunctions* hypothesis by quantifying alarm distributions across regions and modules.

- **Heatmaps:** Heatmaps complement frequency plots by visually identifying the *most highly active modules* within each TileCal partition. This spatial clustering of errors helps in making informed decisions on where to apply the LSTM models.
- **Time Series Plots:** These plots play a crucial role in addressing the *temporal non-randomness* hypothesis by illustrating distinct time-dependent characteristics of errors at both modular and partition levels.

By identifying these high-error regions, error-specific and partition-specific LSTM predictive maintenance strategies can be formulated. By highlighting areas of concern, frequency plots provide a foundation for selecting critical regions where predictive modelling efforts should be prioritized. Then by pinpointing consistently problematic modules through the use of heatmaps, computational resources can be allocated efficiently, improving the overall effectiveness of predictive maintenance.

Finally, by confirming the existence of meaningful temporal dependencies, these plots reinforce the justification for using LSTMs, which specialize in learning sequential patterns[6].

Beyond justifying the application of LSTMs, these visual analyses contribute to refining the LSTM architecture, training procedures, and hyper-parameter tuning strategies. Factors such as sample size, temporal resolution, and spatial error distributions play a significant role in optimizing model performance. The dataset structure largely determines modifications needed in LSTM configurations, ensuring the final model is robust for predictive maintenance applications.

Thus, these visualization strategies used in this study serve a dual purpose—validating the hypotheses while simultaneously informing key methodological decisions. By analyzing spatial and temporal error distributions, the study establishes a framework for data-driven predictive maintenance using LSTMs, ensuring that the approach is both theoretically justified and practically applicable.

Alongside these hypotheses, the study pursues three primary aims:

- **Proof-of-Principle:** Demonstrate the feasibility of a machine learning–based strategy for predictive maintenance in the TileCal DCS.
- **Model Development:** Create LSTM models capable of accurately predicting future alarm events and classifying error types.
- **Strategic Integration:** Lay the groundwork for integrating ML-driven predictive maintenance into the DCS, ultimately reducing downtime and enhancing detector performance.

By validating these hypotheses through the objectives and achieving the outlined aims, this work aims to establish a solid foundation for ML-integrated predictive maintenance, ensuring that the TileCal system remains robust in the face of the HL-LHC's intensified operational demands.

Chapter2 Theory and Background

The Theory and Background chapter provides the foundational understanding necessary to appreciate the methodology applied later in the thesis. It begins with an exploration of the relevance of *'Natural Language Processing (NLP) and Machine Learning (ML) in Predictive Maintenance'*, especially in complex systems like particle detectors. This section highlights how the use of these technologies in processing raw alarm logs can reveal patterns that are indicative of impending hardware issues. Furthermore, this chapter delves into Long Short Term Memory (LSTM) networks, explaining their architecture, advantages, and suitability for time series prediction and classification tasks by discussing the *'NLP of Alarm Data and ML Task Design.'* The section establishes the LSTM as the ideal model for capturing long-range dependencies in time-series data, which is essential for forecasting malfunctions in ATLAS. This theoretical background ensures a clear understanding of the tools and techniques employed in the research, laying the groundwork for the methodology discussed in the next chapter.

2.1 NLP and ML in Predictive Maintenance

As a branch of AI, the field of ML enables algorithms and models to learn from data and make decisions based on patterns in the data, without explicit programming. In ML, an algorithm is the set of procedures or rules used to learn from data and adjust parameters, while the model is the end product—the learned representation that encapsulates patterns from the data and is used for performing tasks, such as making predictions, on new information[33].

NLP is a discipline that focuses on developing computational methods that enable algorithms and models to interpret, process, and generate human language. It also refers to the application of such methods and the construction of such models and algorithms. NLP models extract and exploit linguistic data for tasks such as text classification, sentiment analysis, machine translation, and speech recognition without explicit task-specific programming[34]. This is not to say that within the fields of ML and NLP, the models learn decision rules without every rule being explicitly programmed—but rather that they largely learn decision rules from data by identifying patterns and adjusting model parameters accordingly[34].

Predictive maintenance is a strategic, data-driven approach to equipment management that anticipates and prevents failures before they disrupt operations. In predictive maintenance, historical sensor data, alarm logs, and operational conditions are analyzed to forecast equipment failures, thus avoiding unexpected breakdowns, optimizing resource use, and preventing costly downtime[35,36].

Modern NLP methods are built on ML models, and the techniques used in both respective disciplines are essential for transforming raw datasets, such as alarm logs, into structured datasets that can be applied to solve real-world problems, such as the predictive maintenance

of critical infrastructure systems[35,34]. In this context, ML techniques, employing statistical algorithms, are applied to build models that learn from historical data, enabling accurate predictions of equipment failures and optimization of maintenance schedules, while NLP techniques are utilized to process and interpret these logs to extract critical time-series features that form the foundation for effective ML model construction[5,34].

Efficient processing of sequential data containing both categorical and numerical information, such as alarm logs, requires several NLP pre-processing steps, which include:

- **Tokenization:**
  - Splitting text into tokens. For example, `"The product is amazing! Love it."` becomes: `["The", "product", "is", "amazing", "!", "Love", "it", "."]`[34].
- **Text Normalization:**
  - **Stopword Removal:** Removes common words (e.g., "the", "is", "and") that do not contribute significantly to meaning or analysis. For instance, the tokenized sentence `["The", "product", "is", "amazing", "!", "Love", "it", "."]` reduces to `["product", "amazing", "love"]`[34].
  - **Stemming:** Reduces words to their root form by removing suffixes, speeding up text processing. For example, `["product", "amazing", "love"]` becomes `["product", "amaz", "love"]`[34].
  - **Lemmatization:** A more refined process than stemming, lemmatization reduces words to their base form while considering context and grammar. For example, `["product", "amazing", "love"]` becomes `["product", "amaze", "love"]`[34].
- **Vectorization:**
  - **TF-IDF:** Term Frequency-Inverse Document Frequency (TF-IDF) transforms tokens into numerical features by considering both their frequency within a document and their rarity across the corpus. Terms that appear frequently in a document but rarely elsewhere are assigned higher importance[37]. For example, consider a corpus with the following documents:

    ```
    Doc-1: "product amazing love"
    Doc-2: "love cats"
    Doc-3: "amazing cat"
    ```

    The TF-IDF vectorizer computes the *Term Frequency* (TF) and *Inverse Document Frequency* (IDF) to assign a TF-IDF value to each term. This is done by multiplying the TF by the IDF for each word, resulting in a sparse vector representation[37].
  - **Word Embeddings:** Rather than representing documents in a high-dimensional, sparse vocabulary space, word embeddings map words to low-dimensional, dense vectors[37]. For instance, words like "joyful" and "happy" will have similar vector representations. Models such as *Word2Vec*[38], *GloVe*[39], or *FastText*[40], are often used. Each word is assigned a fixed-length vector (e.g., 100-300 dimensions) and words with similar meanings have closer vectors. For example:

```
"amazing" -> [0.45, -0.02, 0.78, ...]
  "love" -> [0.39, 0.10, 0.91, ...]
"product" -> [0.12, 0.67, -0.21, ...]
```

- **Encoding:**
  - **One-Hot Encoding:** Converts nominal categorical features into binary vectors. For example, the colors `green', `blue', `red' are converted into [1,0,0], [0,1,0], [0,0,1][34].
  - **Label Encoding:** Maps categories, such as S, M, L, XL to integer values like 0, 1, 2, 3, useful when there is an inherent order in the categories[34].
- **Data Scaling:**
  - **Min-max Normalization:** Scales numerical values to a fixed range, typically between 0 and 1, using the formula: $x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$. It preserves the original relationships but is sensitive to outliers[37].
  - **Standard Scaling:** Transforms data to have a mean of 0 and standard deviation of 1, using the formula: $x_{\text{scaled}} = \frac{x - \mu}{\sigma}$,where μ is the mean and σ is the standard deviation. This method is preferred when data follows a normal distribution or when outliers are present[37].

The ML models that modern NLP methods rely on the use of artificial neural networks (NNs), which are computational architectures composed of layers of interconnected neurons, to mimic human language comprehension through representation learning. The simplest architecture consists of an initial processing layer, one or more dense layers, and an output layer[5].

In a feedforward neural network (FNN), data flows in a single direction—from the initial processing layer, through one or more hidden (dense) layers, to the output layer—with no feedback connections. In each dense layer, every neuron is fully connected to every neuron in the preceding layer. This means that each neuron receives a separate signal from every neuron in the previous layer, and each of these signals is multiplied by a unique weight assigned to that specific connection. These weights indicate the relative importance of each signal[5].

The neuron then sums these weighted signals and adds a bias term—a constant that adjusts the overall sum—to produce a pre-activation value. Next, an activation function transforms this pre-activation value into the neuron's output by scaling, thereby confining the output to a specific range. For example, a sigmoid, σ, function compresses the value to lie between 0 and 1, while the hyperbolic tangent, tanh, function scales it to lie between -1 and 1[5]. Finally, the outputs of all neurons in a layer are gathered into a vector that is provided as input to every neuron in the subsequent layer. Although the same vector is passed to each neuron in the next layer, each neuron processes it differently by applying its own unique set of weights and biases. This process repeats until the final output of the network is produced[5].

Recurrent Neural Networks (RNNs) are designed to handle sequential data by incorporating a feedback loop that allows information from previous time steps to persist. In an RNN, each neuron processes two inputs at every time step: (1) the new input for that time step and (2) the hidden state, which carries information from past computations[5]. In basic RNNs, this hidden state is simply the neuron's previous output, fed back into itself through a recurrent connection.

The neuron processes both the new input and the hidden state by applying its unique set of weights and biases, summing the results, and then passing the pre-activation value through an activation function to generate an output[5].

At the same time, the neuron updates the hidden state, which serves as a memory of past computations. This updated hidden state is not a replacement for new inputs but is instead combined with the next time step's input when the neuron processes future data. Through this continuous feedback mechanism, RNNs integrate past and present information, enabling them to recognize temporal patterns and effectively process sequential data[5].

Despite these improvements, early RNNs often struggled in capturing *long-term dependencies* because of the *vanishing gradient problem*. Vanishing gradients result in negligible updates to early weights, hindering the network's ability to learn long-range dependencies, while exploding gradients can cause unstable, excessively large weight updates causing the model to diverge or converge poorly. Techniques like gradient clipping, adaptive optimizers, and careful initialization are employed to mitigate these issues[5].

Specialized recurrent architectures, such as Gated Recurrent Units (*GRUs*)[41] and *LSTMs*[6], were developed to address the limitations of standard RNNs—namely vanishing gradients and difficulties capturing long-range dependencies. Both GRUs and LSTMs introduce gating mechanisms that regulate information flow through σ and tanh activation functions, enabling more effective learning from sequential data[5].

In GRUs, the reset and update gates manage how past information influences the current state, merging the cell and hidden states into a single representation. By contrast, LSTMs use three gates—input, forget, and output—to control a dedicated cell state, allowing the model to retain relevant information over extended sequences. These gating strategies help both architectures mitigate gradient-related issues and handle long-range dependencies more efficiently than basic RNNs[5].

Unlike standard RNNs, which typically rely on a single activation function (e.g., tanh or ReLU), GRUs and LSTMs incorporate multiple activation functions through their gating structures. The sigmoid function (σ) determines the degree of information flow, while the tanh function scales and refines the stored content. This dynamic interplay ensures that crucial features persist, whereas outdated or irrelevant information is discarded. As a result, LSTMs and GRUs significantly outperform traditional RNNs in tasks requiring long-range dependencies, such as natural language processing, time-series forecasting, and speech recognition[5].

Although both GRUs and LSTMs are specialized RNNS that overcome the shortcomings of vanilla RNNs, the LSTMs offer a more granular control of memory. The inclusion of an additional gate, allowed LSTMs to better isolate and preserve relevant signals over longer sequences, making them particularly effective for complex tasks requiring fine-grained temporal memory[5].

The core structure of an LSTM cell consists of three gates: the forget gate (f_t), which controls what information from past states is retained, the input gate (i_t), which regulates what new information is stored, and the output gate (o_t), which determines the final processed information passed to the next step[5,6]. Figure 2.1 illustrates how these gates interact with the cell state

(c_t) and hidden state (h_t) to regulate information flow at each time step. The mathematical formulation governing the update and transformation of these states at each time step is detailed in Table 2.1, which outlines the functional role of each gate in the computational dynamics of an LSTM cell, followed by the definitions of the mathematical expressions in the equations.

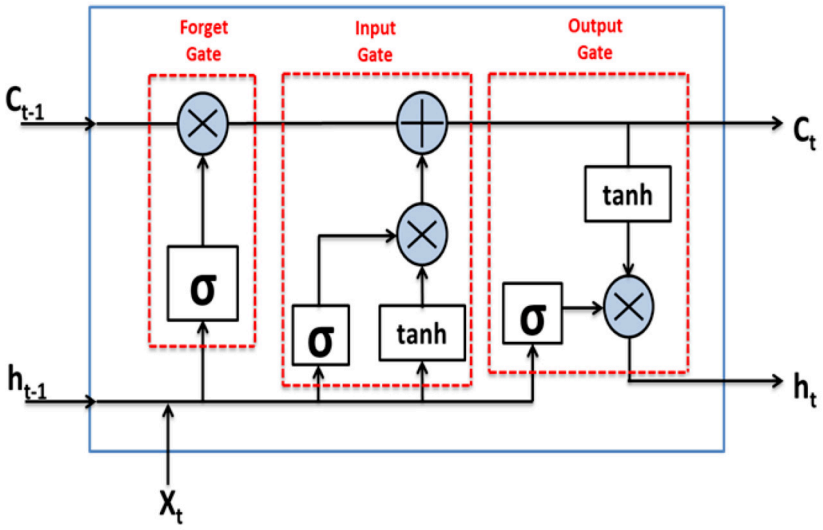


Figure 2.1: The internal structure of the central processing unit of an LSTM cell[4].

Computation	Equation	Function
Forget Gate (f_t)	$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$	Controls memory retention by determining what fraction of the previous cell state is preserved.
Input Gate (i_t)	$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$	Regulates the addition of new information into the cell state.
Candidate Cell State ($\tilde{c}_t$)	$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$	Computes potential updates for the cell state based on the current input and past hidden state.
Updated Cell State (c_t)	$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$	Merges retained information with new memory updates to maintain long-term memory.
Output Gate (o_t)	$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$	Determines how much of the updated cell state is exposed to the hidden state.
Updated Hidden State (h_t)	$h_t = o_t \odot \tanh(c_t)$	Represents the processed output of the LSTM cell for the current time step.

Table 2.1: Mathematical Formulation of LSTM Cell Dynamics[5,6]

- x_t: Input vector at time step t.
- h_t: Hidden state vector at time step t, which serves as the output of the LSTM cell.
- c_t: Cell state vector at time step t, which stores long-term memory.
- f_t: Forget gate activation vector at time step t, controlling how much of c_{t-1} is retained.
- i_t: Input gate activation vector at time step t, determining how much new information is added.

- $\tilde{c}_t$: Candidate cell state, which represents potential new memory updates before being combined with past memory.
- o_t: Output gate activation vector at time step t, controlling how much of the cell state contributes to h_t.
- W_f, W_i, W_c, W_o: Weight matrices applied to the current input x_t for the forget, input, candidate cell state, and output gates, respectively.
- U_f, U_i, U_c, U_o: Weight matrices applied to the previous hidden state h_{t-1} for the forget, input, candidate cell state, and output gates, respectively.
- b_f, b_i, b_c, b_o: Bias terms for the forget, input, candidate cell state, and output gates, respectively.
- $\sigma(\cdot)$: Sigmoid activation function, squashing values between 0 and 1.
- $\tanh(\cdot)$: Hyperbolic tangent activation function, scaling values between -1 and 1.
- $\odot$: Element-wise multiplication (Hadamard product).

These computations allow LSTMs to selectively retain long-term dependencies, making them highly effective for prediction and classification tasks involving sequential data, such as text processing and time-series forecasting[6]

Building an ML model, specifically an LSTM network, involves several key steps: defining the problem, selecting an appropriate architecture, preparing the data, compiling the model, and optimizing it through training and validation[6,5].

The problem is initially defined by examining both the characteristics of the dataset and the nature of the task. In this study, the focus is on two types of time-series problems: *prediction* and *classification*. Time-series prediction involves forecasting future values based on past sequences, while classification assigns categorical labels to given time-series windows—short, consecutive segments of the data that capture local snapshots of temporal patterns over a fixed length. Depending on whether the goal is to *predict continuous values* or *classify windows into categories*, an appropriate LSTM architecture is selected[42].

The most basic LSTM model contains a single LSTM layer as the initial layer that processes temporal input and a dense (fully connected) output layer that produces either a classification or regression output. Data typically enters the network as a 3-D tensor of shape `(batch size, time steps, features)`. Here, batch size represents the number of samples processed simultaneously, time steps denote the sequence length or number of time intervals per sample, and features refer to the number of input variables at each time step[43].

More complex architectures can stack multiple LSTM layers together to capture hierarchical temporal patterns and include additional dense layers with non-linear activations (e.g., ReLU or tanh) to increase representational capacity. Without non-linearity, stacked dense layers effectively collapse into a single linear transformation, which limits model expressiveness[6].

In certain textual tasks, an embedding layer can first convert tokens into dense vector representations before being passed to the LSTM layers. The final layer's activation depends on the task: a softmax function is used for classification to output class probabilities, whereas a linear function is used for regression to predict continuous values[43].

Before training, the data must be prepared. Pre-processing is vital for ensuring that the LSTM model receives meaningful input. This may involve normalizing or scaling features, converting raw alarm logs into sequences for time-series analysis, and handling missing values or noise via smoothing[6]

One crucial aspect of time-series preparation is choosing a *sequence length* (window size) for the input. The sequence length indicates how many past time steps the model includes when generating predictions[42]. Additionally, the*stride* (time step) determines how far the window shifts when forming each subsequent sequence. A common technique, known as the *sliding window approach*, uses overlapping input segments to leverage temporal relationships[6].

In time-series forecasting and classification, data is structured as sequential inputs paired with corresponding target values, enabling models to learn patterns and predict future observations based on past sequences[33]. For example, given a dataset $t_1, t_2, t_3, \ldots, t_n$, consider the input-target pairs formulated using the following window sequencing approach for a time-series prediction task:

$$\left[(t_1, t_2) \rightarrow t_3, (t_2, t_3) \rightarrow t_4, (t_3, t_4) \rightarrow t_5, \ldots\right] \tag{2.1}$$

- (t_i, t_{i+1}) represents the input sequence consisting of two consecutive time steps.
- t_{i+2} is the target value that the model aims to predict based on the input sequence.

This approach is called the sliding window approach, as it systematically partitions the input training samples into overlapping sequences of fixed length. This ensures that data points contribute to multiple training samples, reinforcing temporal patterns across multiple training iterations[33]. The only exception is when non-overlapping sequences are used. For instance, if a dataset has a window size of 5 and a stride of 1, overlapping sequences are created, ensuring that each time step contributes to multiple training examples. However, if the stride is 5, no overlap exists, reducing the number of training samples.

Consider a time-series classification task in *Human Activity Recognition* (HAR). Here, sensor readings are used to predict categorical labels, such as activities involving movement. Data is typically recorded from accelerometers, which measure acceleration along three axes: x (forward-backward), y (side-to-side), and z (up-down) [44,45]. Examples of input-target pairs formulated using the following window sequencing approach for a time-series classification task in HAR include:

$$\left[(\vec{x_1}, \vec{x_2}) \rightarrow \text{“}W\text{”}, (\vec{x_3}, \vec{x_4}) \rightarrow \text{“}R\text{”}, (\vec{x_5}, \vec{x_6}) \rightarrow \text{“}S\text{”}, \ldots\right] \tag{2.2}$$

- $\vec{x_i} = (x_i, y_i, z_i)$ are accelerometer readings in meters per second squared ($\mathrm{m/s^2}$) at time step i.
- $\vec{x_{i+1}} = (x_{i+1}, y_{i+1}, z_{i+1})$ are readings at the next time step.
- The categorical labels “W”, “R”, and “S” denote ‘*Walking*’, ‘*Running*’, and ‘*Sitting*’, respectively.

This discussion does not exclude experiments or models being constructed so that multiple outputs (labels) may be classified or predicted at a single time step. Such experiments and models exist[45,46]; however, these models were not considered in this study[44]. Nevertheless, these studies in HAR motivated the classification task of this study.

In HAR, sensor readings along three spatial axes yield independent time series that, when integrated, determine overall activity[45]. Similarly, the independent time series of total alarm counts from each of the four TileCal partitions serve as the primary features. The dominant alarm label is determined by aggregating these counts and selecting the alarm category with the highest count, thereby reflecting the overall error activity within the entire TileCal sub-detector. In this analogy, the individual partitions correspond to the spatial axes, the total alarm counts are analogous to accelerometer readings, and the alarm label represents the overall activity.

By plotting the time series of total counts per partition separately for each respective alarm category —given that the key hypotheses of this study mentioned in Section 1.2 are proven true— will reveal

correlations among these features and verify that each partition captures unique behaviour that relates to the dominant error label. Moreover, by identifying and removing highly repetitive alarms occurring in short instances of time, either iteratively for each error or collectively by smoothing the total error counts, this approach ensures the features remain robust without the need for additional feature selection or dimensionality reduction[35].

Smoothing methods address noise in time-series data. Moving averages filter short-term fluctuations by taking the mean of adjacent points, whereas exponential smoothing assigns diminishing weights to past data, adapting more readily to changes over time[7]. Iterative smoothing repeats these techniques multiple times until residuals stabilize, but caution is warranted since overly aggressive smoothing can eliminate critical signal information[7]. Once prepared, the data is split into training and validation sets to track model performance and guard against overfitting[7].

After defining and compiling the LSTM model, the training process begins by specifying a loss function, an optimizer, and relevant evaluation metrics. The Adam[47] optimizer is often selected for its adaptive learning capabilities[7].

In time-series forecasting (prediction tasks), different evaluation metrics are required because the goal is to measure numerical accuracy rather than classification performance. Unlike classification tasks, these metrics quantify how close predicted numerical values are to actual values, which is why the loss functions and performance metrics used in time-series forecasting activities are the same[7].

For time-series prediction, the common evaluation metrics are Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE). These can also be used as the loss functions[7]. RMSE and MSE are most commonly used as the loss functions, while binary cross-entropy or categorical cross-entropy are the standard loss functions for binary and multi-class classification tasks, respectively[7].

Depending on the dataset and goals, one performance metric may be preferable over another. For example, in the case of prediction tasks, RMSE penalizes large errors more heavily, and MAE is more robust to outliers. MAPE provides a percentage-based interpretation but can be unstable if actual values are zero. MSE is commonly used when emphasizing larger errors is important, and it also simplifies backpropagation in some frameworks[7].

Table 2.2 summarizes the key performance metrics used in our time series prediction task, detailing the formulas for RMSE, MAE, MAPE, and MSE alongside descriptions of their respective properties and suitability for different error evaluation scenarios.

Metric	**Formula**	**Description**
RMSE	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$	Penalizes large prediction errors more than MAE. Useful when large deviations are especially undesirable.
MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$	Measures the average absolute error magnitude and is more robust to outliers than RMSE.
MAPE	$\frac{1}{n}\sum_{i=1}^{n}\left\lvert\frac{y_i - \hat{y}_i}{y_i}\right\rvert \times 100$	Expresses prediction error as a percentage of actual values, aiding interpretability but potentially problematic if $y_i \approx 0$.
MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Similar to RMSE but without the square root, thus further magnifying large deviations.

Table 2.2: Performance Metrics for Time Series Prediction Tasks[7]

Time-series classification involves evaluating the correspondence between actual and predicted labels, commonly expressed in terms of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN)[8].

Table 2.3 presents a comprehensive overview of widely used classification metrics—including Precision, Recall, and F1-score (with variants such as Macro-F1, Micro-F1, and Weighted-F1 for addressing class imbalance)—detailing their formulas and describing their suitability for different error evaluation scenarios.

Metric	**Formula**	**Description**
Precision	$\frac{TP}{TP+FP}$	Proportion of predicted positives that are correct. Applicable when false positives carry a high cost.
Recall	$\frac{TP}{TP+FN}$	Proportion of actual positives that are correctly identified. Crucial if missing true positives is costly.
F1 Score	$\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$	The Harmonic mean of Precision and Recall, providing a measure that balances both metrics.
Macro-F1	$\frac{1}{n}\sum_{j=1}^{n}\text{F1 Score}_j$	Averages the F1-scores across all the classes, treating each class equally.
Micro-F1	$\frac{\sum TP}{\sum(TP+FP+FN)}$	Weighs classes by their total frequency, to produce a global F1-score.
Weighted-F1	$\frac{\sum(\text{Instances}_j\times\text{F1 Score}_j)}{\sum\text{Total Instances}}$	Adjusts F1-scores by class proportions to account for the presence of class imbalance.

Table 2.3: Performance Metrics for Time Series Classification Tasks[8]

LSTM models contain both learnable parameters (weights and biases) and hyperparameters (externally configured settings) that determine the learning process. Proper selection and tuning of hyperparameters is critical for stability, generalization, and convergence[9].

After data collection and pre-processing, data is typically partitioned into training, validation, and testing sets. Grid search, random search, or Bayesian optimization can systematically explore the hyperparameter space, with packages such as *Optuna*[48], *Hyperopt*[49], and Weights & Biases (*Wandb*[50]) automating this process[9].

Training, validation, testing, and hyperparameter tuning thus form an interconnected cycle of continuous improvement. The training phase iteratively refines model parameters using backpropagation through time (BPTT). Validation is performed on a held-out subset of the training set to detect overfitting, allowing for hyperparameter adjustments or early stopping if performance deteriorates on the validation set[9].

After convergence, testing uses a separate dataset to assess how well the model generalizes, since no further weight updates occur during this phase. After completing the training, validation, testing, and refinement stages, the next phase is model deployment, followed by its application, where the trained model is utilized to perform a specific task[9].

Key hyperparameters in LSTM-based modelling include the number of LSTM units, number of dense units, epochs, batch size, learning rate, dropout, and weight decay, among others. It is important to note that although commonly used values exist for the step size and sequence lengths, the reasonable choice of these values ultimately depends on the size of the dataset used in the study. Table 2.4 highlights some of these common hyperparameters and their definitions.

Hyperparameter	Description
LSTM Units	Defines the number of memory cells in an LSTM layer, affecting the model's ability to capture temporal dependencies.
Dense Units	Specifies the number of neurons in fully connected layers, influencing the model's representational capacity.
Epochs	Determines how many times the entire dataset is passed through the model during training.
Batch Size	Defines the number of training samples processed before parameter updates occur.
Learning Rate	Controls the size of each step in gradient-based optimization.
Weight Decay	Penalizes large weight values to reduce overfitting and improve generalization.
Sequence Length	Specifies how many past time steps each input sequence contains (the window size)
Dropout	Dropout layers use this dropout rate to randomly deactivates neurons during training, mitigating overfitting by preventing co-adaptation of features.
Step Size	Indicates how far the window moves each time step to generate the next input-output pair (the stride).
Optimizers	Adjust model parameters to minimize loss, with common e.g.s including Adam[47], Adamax[47], RMSprop[51] and SGD[52]).

Table 2.4: Common Hyperparameters and Tuning Ranges in LSTM Models[9].

The field of ML is increasingly applied in experimental particle physics for Monte Carlo simulations, event reconstruction, and detector monitoring. Convolutional Neural Networks (CNNs)[53] are used for particle classification, Graph Neural Networks (GNNs)[54] model particle interactions, and Generative Adversarial Networks (GANs)[55] support data augmentation and simulation tasks[56,57,58,59].

LSTMs and related recurrent architectures have been shown to effectively analyze time-dependent data. Hybrid models such as CNN-LSTMs, GNN-LSTMs, and GAN-LSTMs enhance predictive accuracy by integrating spatial and temporal features and can generate synthetic training samples to address data scarcity[56,57,58].

Beyond high-energy physics, LSTMs and hybrid architectures have demonstrated success in diverse applications such as reservoir estimation[60], malware detection[61], and stock prediction[62], further supporting their potential in precision physics experiments[35,56]. Although explicit studies on hybrid architectures such as CNN-LSTMs, GNN-LSTMs, and GAN-LSTMs in high-energy physics remain limited, their conceptual foundation is well-supported by the broader success of deep learning in related domains[56,57,58].

This study aligns itself with CERN initiatives like the Accelerator Fault Tracker (AFT), which emphasize predictive maintenance to reduce detector downtime[63]. Deployment strategies in systems like the ATLAS TileCal include real-time anomaly detection through software integration, edge computing with FPGA/GPU acceleration, cloud-based training via CERN OpenLab, and integration with trigger systems for enhanced event selection[63]. By embedding ML models directly into the DCS, detector efficiency can be improved, minimizing downtime and maximizing data collection for precision physics experiments.

2.2 NLP of Alarm Data and ML Task Design

The ATLAS Digital Data Viewer (DDV) serves as the primary portal for accessing historical alarm logs from the Tile Calorimeter (TileCal) of the ATLAS detector. Due to system constraints, raw alarm logs cannot be directly downloaded; instead, a Python-based extraction tool[64] retrieves the data for specified partitions and time intervals, outputting standardized text files. This standardized raw data forms the basis for two distinct machine learning tasks: time-series prediction and time-series classification.

TileCal is partitioned into four barrel regions (EBA, EBC, LBA, and LBC), each containing 64 modules responsible for particle energy measurements. This structure is reflected in the alarm log entries extracted during RUN-3 operations. Moreover, Each raw alarm log entry contains several key fields that describe the conditions and nature of an alarm event. For example, an entry might read:

`"TIL EBA TRIPPED LVPS_50 HV side"`

This entry indicates that the alarm originated from the *EBA* partition, where there was a power failure involving `LVPS_50`, a specific hardware item within the Low Voltage Power Supply system (in this case, the LVPS itself of module number 50). The phrase `"HV side"` provides additional context on the location of the fault.

Other important fields include:

- **Abbreviation:** Indicates alarm severity, which in increasing order of perceived threat and inverse response time: (W: Warning, E: Error, F: Fatal).
- **Class:** Defines the alarm awareness state i.e. propagated forward (fw), severity level (Error, Warning, Fatal) and Acknowledged (Ack) or Not Acknowledged (Nack) (e.g., `fwWarningAck`, `fwErrorNack`).
- **Alarm Text:** Provides a descriptive text of the fault (e.g., "TRIPPED", "WARNING").
- **Value:** Captures numerical deviations from expected sensor readings.
- **Timestamp:** Records the exact time of alarm registration.
- **Directory:** Specifies the state of the detector for a specific hardware item i.e, whether the alarm was triggered (`came`) or resolved (`went`).

This detailed structure is critical, as it allows subsequent pre-processing steps to be implemented to filter redundant entries and group similar alarms under unified labels, ensuring the data integrity and generalize-ability required for the specific ML tasks.

An example of a TileCal DCS alarm log is provided in Table 2.5, illustrating these important fields and detailing the structure using sample entries:

Item	Abbrev.	Class	Alarm Text	Value	Timestamp	Directory
TIL LBC Drawer 64 PMT 36 Output Voltage status	E	fwErrorAck	TRIPPED	2.00	06-01-2022 18:08:09.745	went
TIL EBA TRIPPED LVPS 42 HV and MB side	E	fwErrorNAck	TRIPPED	0.00	07-01-2022 18:08:20.260	came
TIL LBA Drawer46 Readout Temperature status	E	fwErrorAck	TRIPPED	2.00	07-01-2022 18:33:58.951	went
TIL LBA AuxBoard_15_channel_1 overcurrent	W	fwWarningNAck	WARNING	46.02	07-01-2022 18:58:42.260	came

Table 2.5: Illustrative Extract of ATLAS DCS Alarm Logs for the TileCal Sub-detector, Demonstrating Partition-Based Hardware References and Key Fault Fields

In this study, only alarms at the *Error* severity level or higher were considered. All alarm entries with lower severity levels, such as the last entry in Table 2.5, were discarded to simplify the analysis, as they do not lead to or indicate impending detector downtime and tend to occur in large numbers,

which can obscure the assessment of detector integrity. Excluding these less critical alarms reduces unnecessary complexity and focuses the investigation on issues with a direct impact on detector operations. Consequently, only *Error*-level alarms remained in the dataset.

The overall goal is to build models that (1) predict future alarm counts (a regression task) and (2) classify the most prevalent alarm category affecting TileCal within a given time interval, for subsequent intervals (a classification task).

The dataset for the prediction task contains aggregated sums of alarm counts within uniform time intervals. Each record corresponds to a time bin (e.g., `yyyy-mm-dd HH:MM -- HH:MM`) and contains the total number of alarms recorded during that interval. There are four distinct datasets used for prediction, each corresponding to one of the most highly active modules in each TileCal partition, irrespective of the error category. The raw data components for the prediction task are as follows:

- **Timestamp:** The original time bin for each alarm record.
- **Alarm Count:** The total number of alarms recorded within that time interval.

Prior to modeling, the alarm counts are scaled using Min-Max normalization (for instance, converting a count of 15 into a normalized value of approximately 0.375, given the observed range). The normalized data is then converted into overlapping sequences using a sliding window approach: each input sequence, with a fixed length (e.g., 10 consecutive time bins), is used to predict the alarm count in the immediately following bin.

For the classification task, the dataset is designed for training a model to predict the most prevalent alarm category affecting TileCal for future time intervals based on historical, time-binned, partition-specific error counts The data is multivariate, and the raw components at each time step include:

- **EBA:** Count of total errors recorded in the EBA partition.
- **EBC:** Count of total errors recorded in the EBC partition.
- **LBA:** Count of total errors recorded in the LBA partition.
- **LBC:** Count of total errors recorded in the LBC partition.
- **Label:** The most dominant alarm class (E1–E5) affecting TileCal during that time interval.

Following pre-processing, numerical features are normalized using Standard Scaling (yielding a mean of 0 and standard deviation of 1), and the categorical labels are transformed using One-Hot Encoding (e.g., E1 becomes $[1, 0, 0, 0, 0]$ and E5 becomes $[0, 0, 0, 0, 1]$). The processed data is then organized into sliding window sequences to capture temporal dependencies. For instance, with a fixed sequence length of 3 (three sets of observations) and a step size of 1:

- **Sequence 1: (Input):** Transformed Rows 1–3, **Target:** Row 4's one-hot encoded label.
- **Sequence 2: (Input):** Transformed Rows 2–4, **Target:** Row 5's one-hot encoded label.

The complete sequence dataset is subsequently split into training, validation, and testing sets (e.g., 70% training, 15% validation, and 15% testing). In the current study, the limited data necessitated using the validation set as the testing set; future work will acquire larger datasets to ensure proper evaluation.

Consider a prediction task. An input sequence of normalized alarm counts is given by:

$$[0.2,\ 0.3,\ 0.5,\ 0.7]$$

The LSTM model is trained to forecast the alarm count in the next time bin. For example, the model might predict a value of 0.6, which is then compared to the actual normalized alarm count to compute performance metrics such as MAPE and RMSE.

Consider a classification example. Suppose an input sliding window yields the following standardized feature vectors (for the four partitions):

$$\begin{bmatrix} [0.50,\ 0.00,\ 0.20,\ 0.30] \\ [0.40,\ 0.10,\ 0.70,\ 0.60] \\ [0.90,\ 0.40,\ 0.10,\ 0.60] \end{bmatrix}$$

If the dominant alarm in the subsequent time bin is E1, the target label is represented as:

$$[1, 0, 0, 0, 0]$$

This input-output pair is used to train the LSTM classifier.

Long Short-Term Memory (LSTM) networks were chosen for this study because of their ability to capture long-term temporal dependencies via gated memory mechanisms, which effectively mitigate the vanishing gradient problem common in recurrent neural networks (RNNs). While vanilla RNNs often fail to model long sequences and Gated Recurrent Units (GRUs) might be too simplistic for extended dependencies, LSTMs provide a balanced approach between complexity and performance[5].

Transformers are a relatively new neural network architecture that utilize self-attention mechanisms and feed-forward layers to process input sequences in parallel, rather than sequentially as in RNNs. This design enables them to capture long-range dependencies without relying on recurrence. However, transformers generally require very large datasets—ranging from tens of thousands to millions of training examples—to fully leverage their capacity[5], a condition not met in this study

Similarly, bidirectional LSTMs (Bi-LSTMs) extend standard LSTMs by processing data in both forward and backward directions, thereby capturing contextual information from both past and future. For time-series forecasting tasks where only past information is relevant, the additional computational cost of Bi-LSTMs does not offer a proportional benefit. Hence, standard LSTMs are more appropriate for our purposes[5]

Chapter3 Methodology

In this chapter, the research methodology is detailed, outlining the steps taken to develop and implement the predictive maintenance models. It begins with a description of *Data Collection and Processing*, including the types of alarm logs from the ATLAS TileCal detector and the pre-processing techniques used to prepare the data for analysis. This section discusses how raw alarm logs are transformed into a structured format suitable for machine learning models using NLP techniques. The chapter then presents *Data Exploration and Visualization*, where various strategies—such as frequency plots to reveal modular error distributions, heatmaps to illustrate spatial variations across detector partitions and modules, and time series analyses to capture temporal trends and assess the impact of noise reduction methods—are employed to provide quantitative insights into the study's key hypotheses regarding spatial inhomogeneity and temporal non-randomness. Finally, the methodology covers *LSTM Model Development and Refinement*, detailing the construction of predictive models through techniques such as hyperparameter optimization and the sliding window approach for generating time sequences that effectively handle the time-dependent nature of the data. Together, these sections provide a step-by-step account of the model development and training process, ensuring that readers understand both the technical framework and the rationale behind the chosen methodologies.

3.1 Data Collection and Processing

The dataset contained only unplanned detector faults recorded during the active Physics runs of Run-3 of the LHC. To ensure the data accurately reflected normal operating conditions—unaffected by maintenance, commissioning, or testing activities—entries were restricted to periods when the detector was fully operational.

Data were selected from the following periods:

- **2022:** July, August, September, October, and November.
- **2023:** May, June, July, August, and September.

Periods during which the LHC was non-operational or undergoing commissioning in 2022 and 2023 were not considered. For instance, data from months and days prior to July 5th, when Run-3 officially commenced, were excluded. Furthermore, the months of April, May, and June 2022—dedicated to detector commissioning—were omitted, as were the technical stop from December 2022 through March 2023 and an additional month of commissioning in April 2023.

Testing periods during Run-3 were also removed from the dataset. Since the exact time intervals for testing—identified via the Twiki ATLAS Running Schedule[65]—were only approximate, entire days with testing activities were discarded. Specifically, for 2022, 10 days were removed from September and 1 day from November; for 2023, 5 days in July and 23 days in August were excluded. After these adjustments, the final dataset comprised 18,476 samples collected over a 10-month period.

The Raw alarm logs were obtained from the ATLAS Digital Data Viewer (DDV). Because direct downloading of the alarm data was not possible, a modified Python script originally developed by

Miguel F. Medeiro[64] was used to facilitate the extraction. This script retrieves logs for specified partitions and time intervals, standardizing them into a set of essential columns.

To prevent server crashes—since each output file is limited to fewer than 40,000 entries—the data were sometimes extracted in smaller chunks rather than by month. The extracted logs are returned as four separate text files per partition, with each file clearly labelled according to its respective time frame and partition. This methodical approach ensured that the analysis was based solely on data recorded during genuine operational periods, thereby providing a reliable basis for studying detector faults under normal physics run conditions.

Thereafter all the text files were merged into a single pandas datframe and several NLP preprocessing steps were applied to the `Item` column, including:

- Lowercasing all text (e.g., "AuXboard" → "auxboard").
- Replacing slang terms with standard terminology (e.g., "AUX" → "Auxboard").
- Removing special characters (except `-`, `+`, and `_`).
- Removing redundant descriptors and stopwords.
- Eliminating repeated words within individual entries.

The additional columns: 'Partition', 'Hardware Class' and 'Module Number', were created by extracting relevant features from the existing 'Item' column using regular expressions. The relevant information from the 'Item' column used for the construction of the 'Hardware Class' and 'Module Number' entries were:

- -LVPS: _{1,….,64}
- -Drawer: _{1,…,64}
- -HVPS: _{1,...,4}_channel_{1,..,16}
- -AUX.: _{1,…,16}_channel_{1,...,4}

, where hardware ID = (1,...,4 for HVPS units and 1,...16 for the LVPS units), Module number = (1,...,64) and Channel number = (1,...,4 for the auxboards and 1,...16 for the HVPS units).

In the case of the LVPS and Drawer classes, the module number is explicitly provided as an integer number in the set 1,...,64, for each of the 64 Drawers. In the case of the HVPS and Auxboards, the modular number is implicitly provided and needs to be determined from the ID of the HVPS or Auxboard and its specific channel.

This convention is based on the following pieces of information: There are 4 HVPSss, in which each of these respective 4 HVPSs are connected to a set of 16 modules, except for the last channel, and there are 16 Auxboards, in which each Auxboard is connected to 4 modules.

From the structure of the Item entries, it was possible to extract the exact module that was affected for the affected LVPS and Auxboard using transformation rules[29] The transformation rules are as follows and are accompanied by a few examples:

1. Consider HVPS_N_channel_M, where N and M are integer elements in the respective sets of 1,..,4 and 1,..., 16. The module number is then then M +(N-1)x16.
2. Consider Auxboard_M_channel_N, where M and N are integer elements in the respective sets of 1,..,4 and 1,..., 16. The module number is then N +(M-1)x4.

- HVPS_3_channel_14 will have module number of 46.
- HVPS_4_channel_15 will have module number of 63.
- Auxboard_14_channel_3 will have module number of 46.
- Auxboard_15_channel 4 will have module number of 63.

Understanding the relationship between alarm types and hardware components is essential for standardizing the diverse alarm labels in the dataset. The following methodology was applied to guide the final determination of the standardized alarm labels.

First, the associations between hardware classes and their corresponding alarm texts were determined. Two key columns in the alarm log dataset—`Hardware Class` and `Alarm Text`—were analyzed by grouping the dataset according to hardware classes. This mapping reveals which alarms are triggered for each hardware type.

The results revealed the following associations between hardware classes and alarm texts:

1. **LVPS:** `[TRIPPED, MB SIDE TRIP, ALERT, emergency message received, the ELMB is not in OPERATIONAL state (no toggle != 5)]`
2. **HVPS:** `[ALERT]`
3. **Auxboard:** `[no toggle != 5, emergency message received, ALERT]`
4. **Drawer:** `[NO CONNECTION, ALERT]`

To further validate these associations, the mappings were reversed to examine which hardware components are associated with each specific alarm text. This reverse mapping helps ensure that the conditional grouping of alarms is consistent with the hardware-level behavior.

The reverse mapping results confirmed that certain alarm texts are common across different hardware classes:

1. **emergency message received** : ['LVPS', 'Auxboard']
2. **ALERT**: ['LVPS', 'Auxboard', 'HVPS', Drawer]
3. **(no toggle != 5)**: ['LVPS', 'Auxboard']
4. **NO CONNECTION**: [Drawer]
5. **TRIPPED, MB SIDE TRIP**: [LVPS]

This finding highlights the need for a conditional mapping that consolidates similar alarms into a single standardized category. Based on the above mappings, a conditional mapping was defined to standardize the diverse alarm texts into five generalized categories. This mapping consolidates overlapping alarm labels into a uniform set that reflects the common failure modes observed in the system. The follow mapping was used:

```
alarm_mapping =
    {'MB SIDE TRIP': 'TRIPPED (T)',
    'TRIPPED': 'TRIPPED (T)',
    'the ELMB is not in OPERATIONAL state (no toggle != 5)': 'NO TOGGLE (NT)',
    'no toggle != 5': 'NO TOGGLE (NT)',
    'ALERT': 'ALERT',
    'emergency message received': 'EMERGENCY MESSAGE RECEIVED (EMR)',
    'NO CONNECTION': 'NO CONNECTION (NC)'}
```

Below are the five standardized alarm categories along with a qualitative description for each category based on these findings and an understanding of how these hardware components function:

1. **ALERT (A):**
   This error signifies the detection of a hardware-related anomaly not matching a more specific category. It prompts further investigation into various components or subsystems.
2. **TRIPPED (TRIP):**
   This error occurs when a hardware component operates beyond its permissible limits, triggering an alarm and initiating shutdown of the affected system.
3. **NO CONNECTION (NC):**
   This error occurs when a system or component fails to establish or maintain a required connection, thereby affecting communication or power exchange.
4. **EMERGENCY MESSAGE RECEIVED (EMR):**
   This error signifies the reception of a critical message indicating an emergency situation, requiring immediate intervention.
5. **NO TOGGLE (NT):**
   This error indicates a failure to switch between different operational states or modes, suggesting a malfunction in the control mechanism.

The absence of certain alarm categories in specific hardware items can be attributed to several interrelated factors such as function, safe-guards, design and system architecture.

Firstly, functional specialization plays a key role. Some hardware components like the LVPS units are designed for specialized purposes and are therefore prone only to specific failures. In contrast, HVPS units operate under distinct parameters and may primarily register ALERT errors when anomalies occur.

Secondly, design and communication architecture influence alarm distribution. Hardware positioned remotely, such as Drawer modules, is more susceptible to communication errors than networked components like Auxboards or LVPS units, which are closely integrated with the system's primary circuitry and thus less likely to experience such failures.

Another contributing factor is redundancy and built-in safeguards in certain components, which can prevent specific failure modes from being detected or reported. These protections may lead to the absence of particular alarm categories in hardware with built-in redundancies, even when similar issues arise in less-protected components.

Finally, operational context determines the types of alarms generated. For instance, 'EMERGENCY MESSAGE RECEIVED' (EMR) errors indicate critical conditions requiring immediate intervention. Not all hardware is equally exposed to such severe conditions, and some may report anomalies through alternative error channels.

Together, these factors ensure that standardized alarm categories accurately reflect the operational characteristics and failure modes of each component. The categories (ALERT, TRIPPED, EMR, NO CONNECTION, NO TOGGLE) are rooted in empirical data and the known behaviors of the hardware, providing a structured dataset for further analysis, as shown in Table 3.1.

Partition	Hardware Item	Module Number	Alarm Class	Timestamp
EBA	LVPS	12	TRIPPED	2022-07-18 06:49:08
LBA	Auxboard	15	ALERT	2022-08-07 11:52:33
EBC	Drawer	32	NO CONNECTION	2022-09-06 07:45:56
LBC	Auxboard	8	EMR	2022-10-02 13:20:45
LBA	HVPS	45	ALERT	2022-10-15 18:22:10

Table 3.1: A sample illustration of the alarm log dataset after NLP-based feature extraction and restructuring. The columns shown here were derived from the original fields, capturing only the essential data needed forthe machine learning tasks.

After this NLP-based feature extraction, the data needs to be structured for the two ML tasks: prediction and classification. The objective of the prediction task is to forecast future alarm counts, while the objective of the classification task is to forecast future alarm categories. In order for the datasets to be used for achieving these objectives, a series of steps are required. This dataframe was saved as a dataset in.csv format for later use when extracting features for visualizing the data through frequency plots and heatmaps. The procedure for the construction of the time series prediction datasets is as follows:

The first step is to remove non-essential columns leaving behind the columns:

- `Partition` (`EBA`, `EBC`, `LBA`, `LBC`)
- `Module Number`
- `Timestamp`
- `Alarm Class` (Refined Alarm Text Labels)

These are the essential columns needed for both prediction and classification tasks.

The next step is to assign timestamp bins. The timestamps are floored to fit into predefined intervals of `5 minutes`, `15 minutes`, `1 hour`, `1 day`, `weekly`, or `monthly` bins. A simple example for doing a 5-minute binning is given in Python Implementation:

```
def assign_5min_bin(timestamp):
    return timestamp.floor('5T')
df["Timestamp_5min"] = df["Timestamp"].apply(assign_5min_bin)
```

Continuing with this example, once the data has been binned into time intervals, the next step is to count the occurrences of each error category within each bin. Specifically, each alarm entry is assigned to its corresponding time bin, and the number of alarms is tallied for each category within that bin, ensuring an accurate representation of error frequencies over time.

Whenever the dataset is further subdivided—for example, into smaller datasets focused on a specific partition or error type—these counts are recalculated. This subdivision preserves the overall constraint that the total number of errors across the entire 10-month period remains unchanged. By summing the error counts across all smaller sub-datasets, one can verify that the total still matches the original dataset's overall error count.

Thereafter, the transformed dataframe is then split by error category (`ALERT`, `TRIPPED`, `NO CONNECTION`, `EMR`, `NO TOGGLE`), into five distinct dataframes and each of these are then further split by partition (`EBA`, `EBC`, `LBA`, `LBC`) into four distinct dataframes:

```
# Split dataset by error category
```

```
df_alert     = df[df["Error_Type"] == "ALERT"]
df_tripped   = df[df["Error_Type"] == "TRIPPED"]
df_no_conn   = df[df["Error_Type"] == "NO CONNECTION"]
df_emr       = df[df["Error_Type"] == "EMR"]
df_no_toggle = df[df["Error_Type"] == "NO TOGGLE"]

# Further split by partition (example for ALERT only)
df_alert_eba = df_alert[df_alert["Partition"] == "EBA"]
df_alert_ebc = df_alert[df_alert["Partition"] == "EBC"]
df_alert_lba = df_alert[df_alert["Partition"] == "LBA"]
df_alert_lbc = df_alert[df_alert["Partition"] == "LBC"]
```

Each partition-specific subset is then split according to module number into 64 separate dataframes:

```
for module in range(1, 65):
    df_module = df_alert_eba[df_alert_eba["Module"] == module]
    df_module.to_csv(f"df_eba_module_{module}.csv", index=False)
```

This yields finely-grained sub-dataframes such as `df_eba_module_1`, `df_eba_module_2`, and so forth, each containing a single combination of error class, partition and module number i.e., (`Error Type=ALERT`, `Partition=EBA`, `Module=1`), etc.

The last step is to merge all sub-dataframes together. All sub-dataframes are combined back into a single composite dataframe where each column represents one unique combination of error type, partition, and module number, and where each row provides a time bin and alarm count entry. Table 3.2 illustrates a portion of this final structured dataset. Only the `Count` columns are typically needed for model training, since the data is time-ordered and thus the timestamp column is dropped before further pre-processing and model training.

P1__E1__M1__n	**P1__E1__M1__δt**	**P1__E1__M2__n**	**P1__E1__M2__δt**
21	2022-08-30 07:00-08:00	1	2022-08-30 07:00-08:00
1	2022-08-30 13:00-14:00	1	2022-08-30 13:00-14:00
8	2022-08-30 14:00-15:00	6	2022-08-30 14:00-15:00
6	2022-11-07 09:00-10:00	2	2022-08-30 15:00-16:00

Table 3.2: Snapshot of the augmented time-series dataset for predictive modeling. Each sub-table corresponds to a unique combination of Partition (P1,...,P4), Error type (E1,...,E5), and Module (M1,...,M64). Column entries represent event counts (n) and their corresponding time bins (δt).

After preparing the complete dataset and saving it as a .csv file, the most active modules in each partition can be identified (e.g., via heatmap analysis of cumulative error occurrences). Modules with sufficient and discernible temporal patterns are chosen for the final LSTM predictor training and extracted for use from these extended dataset.

Although the classification dataset construction procedure reuses the same pre-processed data from the prediction dataset construction pipeline, the full series of steps are presented for completeness. The following key steps are identical to those in the prediction dataset construction pipeline:

1. **Column Selection**: Only essential columns are retained. In this instance, however, the `Module Number` column is additionally dropped, as the classification task is done on a partitional and not a modular level.

2. **Timestamp Conversion**: All `Timestamp` entries are converted to `datetime` objects, then truncated or resampled into fixed time bins (e.g., 5 min, 15 min, 1 h, 1 d).
3. **Filtering and Splitting**: After resampling, rows with identical timestamps are merged, and the data is split by error labels.

Transformation Step	Data entries	Table Fields
Before Merging	10:45–11:00, EMR, 0,0,6,0 10:45–11:00, EMR, 0,0,16,0 10:45–11:00, ALERT, 0,0,4,0	Timestamp, Class, EBA, EBC, LBA, LBC
After Merging	10:45–11:00, EMR, 0,0,22,0 10:45–11:00, ALERT, 0,0,4,0	Timestamp, Class, EBA, EBC, LBA, LBC
After Splitting	*excluded* 10:45–11:00, ALERT, 0,0,4,0	Timestamp, ALERT, EBA, EBC, LBA, LBC

Table 3.3: Detailed stepwise data transformation: merging by timestamp (duplicate entries) and splitting by class (`ALERT`) using 15-minute bins.

Table 3.3 illustrates how sample entries are filtered and split in the final step. In this table, rows with identical timestamps are merged by summing numeric columns. Next, only rows with `Class = ALERT` are retained; all others are excluded.

In the *prediction* pipeline, for the remaining (non-excluded) data entries, only one numerical field is retained per split, while the other three in {`EBA`, `EBC`, `LBA`, `LBC`} are dropped. The resulting sub-dataframes are then split by module number. These particular steps are *not* used for the classification dataset construction.

For classification-specific pre-processing, the data is pivoted by partition to form a time series per error category:

```
# Explanation: This snippet transforms the dataset into a pivot table
# keyed by 'Timestamp' and columns determined by 'Partition'.
# It sums the 'Count' values for identical timestamps and partitions.

partition_counts = df.pivot_table(
    index='Timestamp',      # Use 'Timestamp' as the pivot index (row labels).
    columns='Partition',    # Pivot columns by 'Partition' values.
    values='Count',         # Aggregate the 'Count' column.
    aggfunc='sum',          # Summation for identical timestamps/partitions.
    fill_value=0            # Replace missing values with 0.)
```

Multiple error categories can then be merged on the `Timestamp` column to produce a unified table capturing all partition counts per time bin. This is done by applying a series of steps.

Firstly, an error category column is added to each table so that every record is associated with its corresponding error type. The tables are then merged using the common `Timestamp` column. This step ensures that all error records are combined while retaining their category labels. Thereafter, the joined table is then pivoted so that each row represents a time bin and each column shows the count for a specific error category. This unified structure then allows you to determine the dominant error for each time bin.

The next step is to assign class labels based on the dominant error:

1. **Aggregate Partition Counts**: For each error category, sum the partition counts:

$$\text{Total Count}_{\text{error}} = \text{EBA} + \text{EBC} + \text{LBA} + \text{LBC}.$$

2. **Determine the Dominant Error Category**: For each time bin, the error with the highest total count becomes the dominant error.
3. **Append Dominant Error Label**: A new column (e.g., `Dominant_Error`) stores the dominant error category.

Table 3.4 provides a simplified example of a pivot table constructed from separate error-specific dataframes for two error classes, `EMR` and `ALERT`:

Timestamp	ALERT Counts				EMR Counts				ALERT Total	EMR Total	Dominant Error
	EBA	EBC	LBA	LBC	EBA	EBC	LBA	LBC			
10:45–11:00	0	0	4	0	1	0	0	0	4	1	ALERT
11:00–11:15	0	0	2	0	1	2	1	0	2	4	EMR

Table 3.4: A simplified example of dominant error label assignment for two error classes (`EMR` and `ALERT`).

To derive the final classification table, each error category (e.g., `ALERT`, `EMR`) is first pivoted by `Timestamp` and `Partition`. For each time bin, partition-level counts for all errors are merged (by summing duplicates). Then for each row (time bin), now that the dominant error is determined:

1. **Sum Partition Counts**: Aggregate the counts for each partition across both error types. This means summing the ALERT and EMR counts for each partition.
2. **Drop Auxiliary Columns**: After calculating the total counts, drop the original individual ALERT and EMR columns.

Timestamp	Total EBA	Total EBC	Total LBA	Total LBC	Dominant Error
10:45–11:00	1	0	4	0	ALERT
11:00–11:15	1	2	3	0	EMR

Table 3.5: Example of a final classification table after summation and dominant error assignment for two error classes (`EMR` and `ALERT`).

An example of a final classification table is shown in Table 3.5, above. The entries for this table are computed from those of Table 3.4. For example, for the 10:45–11:00 interval, `ALERT` has counts $(0, 0, 4, 0)$; `EMR` has $(1, 0, 0, 0)$. Summation yields `Total EBA` $= 1$, `Total EBC` $= 0$, `Total LBA` $= 4$, `Total LBC` $= 0$. Since $4 > 1$, `ALERT` is dominant.

The second row of Table 3.4 is obtained in the same manner: For the 11:00–11:15 interval, `ALERT` has counts $(0, 0, 2, 0)$; `EMR` has $(1, 2, 1, 0)$. Summation yields `Total EBA` $= 1$, `Total EBC` $= 2$, `Total LBA` $= 3$, `Total LBC` $= 0$. Because $4 > 2$, `EMR` is dominant.

Tables 3.4 and 3.5 are examples where 15-minute time binning is used. This process is repeated for the other selected time bins. The resulting classification dataframes for each bin are then combined into an extended dataframe in the same way as the in the prediction pre-processing pipeline and then saved.

Chattering alarms pose a challenge due to their rapid and frequent occurrences, introducing noise and instability into the data. To mitigate this, we identified any instance of five consecutive `Timestamp` entries of the same type. Each instance was corrected by computing the rounded average of the counts for these five alarms. We then retained only the first `Timestamp` bin and error label entry, replacing each of the partition-specific counts with their rounded average value[35].

Retaining the first timestamp entry for each set of chattering alarms with averaged counts, despite significant noise reduction, can mask legitimate patterns in alarm frequencies that arise under specific detector conditions[35].

Sensitivity analyses in future work could compare multiple chattering-alarm correction methods (e.g., weighted averages or dynamic smoothing windows). Additionally, preserving the original frequency of chattering alarms would facilitate deeper investigation of their potential significance[35].

Furthermore, approaches to handling imbalanced class distributions, such as comparing category distributions before and after data reduction, could help assess representativeness. Introducing both datasets (before and after processing chattering alarms) as validation and/or testing datasets in model development would also clarify the impact of these reductions on real-world applicability[35].

In this study, alarm categories were relabeled for simplicity: `ALERT` → `E1`, `EMR` → `E2`, `NO CONNECTION` → `E3`, `NO TOGGLE` → `E4` and `TRIPPED` → `E5`. Figure 3.6 illustrates an example of removing chattering alarms in a dataframe with multiple consecutive `E3` alarm occurences:

Index	Timestamp	EBA	EBC	LBA	LBC	Class
1000	2023-07-01 10:45–11:00	0	6	0	0	E3
1001	2023-07-01 11:00–11:15	0	6	0		E3
1002	2023-07-01 11:15–11:30	0	4	0	0	E3
1003	2023-07-01 11:45–12:00	0	4	0	0	E3
1004	2023-07-01 12:00–12:15	0	2	0	0	E3
1005	2023-08-02 08:30–08:45	4	0	0	0	E3

⇩

Index	Timestamp	EBA	EBC	LBA	LBC	Class
1000	2023-07-01 10:45–11:00	1	4	0	0	E3

Table 3.6: Example of removing chattering alarms: repetitive `E3` alarms are reduced to a single entry with averaged counts per partition, keeping the index value of the first alarm.

Overall, these pre-processing steps—merging duplicate timestamps, splitting by error category, pivoting, removing chattering alarms, and assigning dominant error labels—produce a clean classification dataset suitable for downstream machine learning tasks. The extended dataset allows for the extraction of time series classification datasets for different time bins, which can then be processed to remove chattering alarms.

This approach enables one to review various time bins to identify the optimal granularity, and allows for ease of dataset comparison by visualizing and analyzing the data, fine-tuning the chattering-alarm removal strategy, and assessing ML model performance on both chattering and non-chattering datasets to evaluate their impact during training and testing.

3.2 Data Exploration and Visualization

This section uses three visualization strategies to provide insights into our study's key hypotheses. Each method is designed to address one of these hypotheses:

- **Time Series Plots:** Directly probe the *temporal non-randomness* hypothesis by examining the evolution of alarm events over time. This visualization helps reveal periodicities and trends that support the predictive potential of LSTM models.
- **Frequency Plots:** Address the *increased hardware malfunctions* hypothesis by quantifying alarm distributions across regions and modules. The frequency plots highlight variations in error occurrences, setting the stage for further spatial analysis.
- **Heatmaps:** Building on the spatial inhomogeneity insights derived from the frequency plots, heatmaps pinpoint specific *spatial inhomogeneity* by mapping error densities across detector modules. This step identifies specific problematic locations within TileCal—namely, the most highly active modules—that dominate the overall alarm distribution.

Frequency plots provide a modular overview of alarm counts across TileCal partitions and individual modules. They illustrate how uniformly errors are distributed within each partition across different modules, thereby supporting the identification of patterns that may indicate hardware issues affecting operational continuity. The procedure for constructing these frequency plots from the nlp-processed dataset illustrated in Table **??** involves a series of steps.

The procedure begins by loading the NLP-processed dataset into a pandas dataframe. The dataset is then segmented into four distinct dataframes corresponding to the physical regions of the TileCal barrel i.e., `df_eba`, `df_ebc`, `df_lba`, `df_lbc`.

Within each partition-specific dataframe, the data is further divided by error class. For instance, in the LBA partition, separate dataframes such as `df_lba_E1`, `df_lba_E2`, ..., `df_lba_E5` correspond to different error categories.

Each error-specific dataframe is then subdivided by module number, leading to naming conventions such as:

$$\texttt{df_lba_M1_E1}, \texttt{df_lba_M2_E1}, \ldots, \texttt{df_lba_M64_E1}.$$

This hierarchical segmentation—from overall partition to error type and finally to individual modules—ensures a granular and modular analysis of error distributions. For each partition and error class, arrays of error counts are generated. For example, for the LBA partition, the alarm counts for a ALERT (E1) errors are aggregated as:

$$\texttt{df_lba_E1_counts} = [\text{len}(\texttt{df_lba_M1_E1}), \ldots, \text{len}(\texttt{df_lba_M64_E1})]$$

Each array is then converted into a dataframe with module numbers serving as the x-values and the corresponding error counts as the y-values, forming bar plots that depict the modular error distribution.

To visualize the overall impact on the TileCal barrel, modular error counts across all four partitions can be summed. For example, the total count for `E3` errors in module number, `j`, of TileCal is computed as:

$$\begin{aligned}\text{df_TIL_Mj_E3} = \texttt{df_eba_Mj_E3} + \texttt{df_ebc_Mj_E3} \\ + \texttt{df_lba_Mj_E3} + \texttt{df_lbc_Mj_E3}.\end{aligned}$$

, and the the array used in producing this modular distribution of errors throughout the whole of TileCal is given by:

$$\texttt{df_TIL_E3_counts} = [\text{len}(\texttt{df_TIL_M1_E3}), \ldots, \text{len}(\texttt{df_TIL_M64_E3})]$$

This aggregated view, however, may not illustrate any clear inhomogeneity of error distributions, which are expected to vary significantly by both partition and error type.

Frequency plots facilitate comparisons in two primary ways:

1. **Across Partitions:** By generating modular frequency plots for a fixed error category across different partitions, it is possible to compare the behavior of the same error type in different physical regions.
2. **Across Error Categories:** For a given partition, generating frequency plots for different error types allows for an assessment on which errors are most prevalent in specific modules.

Heatmaps assist in visualizing the spatial distribution of alarms across the detector modules. By highlighting regions with disproportionately high error counts, they directly address the spatial focus of our analysis i.e., the identification of the most highly active modules of TileCal whose data is to be used for LSTM-based prediction.

The procedure for constructing these heatmaps from the nlp-processed dataset illustrated in Table **??** involves a series of steps. The first few steps follow from the steps used in producing the frequency plots. These steps include loading the NLP-processed dataset is loaded into a pandas dataframe, splitting the data by partition, error type and module number, resulting in dataframes such as `df_eba_M1_E1` .

Once these common steps have been performed, for each partition and error type, error counts are aggregated into a $64 \times 5 \times 5$ matrix. Here, the rows represent the unique identifier given by the combination of partition (EBA,EBC,LBA,LBC) and module number (M1–M64) and the columns represent the different error types (E1–E5), with each cell indicating the number of errors recorded for a given partition, module and error type.

The arrays of error counts are then merged, transposed, and formatted into a structured dataframe. This dataframe is sorted by module number, pivoted along the error type axis, and any empty value entries are replaced with zeros before visualization.

The partition-specific heatmaps, constructed out of data extracted from this extended dataframe,display the total error counts accumulated over the entire dataset (study period), much like the modular frequency plots. To enhance interpretability, the heatmaps are segmented into sub-heatmaps covering four 16-module regions (labeled as Q1, Q2, Q3, Q4) per partition, ensuring a localized analysis of error distributions.

Finally, modules with error counts exceeding a threshold—calculated as the mean error count multiplied by 1.75—are flagged as highly active, so that only potential outliers that may indicate underlying issues or vulnerabilities are targeted. All of these modules are selected for further analysis of their temporal characteristics, after which one module from each partition will be chosen to provide data for training one of the four LSTM prediction models.

Time series plots track alarm events over defined temporal intervals, providing insights into recurring patterns and trends essential for our dynamic modeling approach. With the most active modules identified from prior analyses, the objective now is to verify whether they satisfy critical constraints necessary for robust time series modeling. These constraints include:

- **Non-Random Behavior:** The time series data must exhibit underlying patterns rather than random fluctuations, enabling the model to learn meaningful trends.
- **Negligible Long-Term Trends:** Long-term trends should be minimal or removed effectively to highlight periodic or seasonal patterns clearly.
- **Minimal Noise:** The data should have limited noise, reducible further by smoothing techniques, preserving the integrity of the true signal.
- **Clear Periodicity:** Regular and distinct periodic patterns must remain evident even after preprocessing, ensuring robust forecasts.
- **Sufficient Sample Size:** The dataset must contain an adequate number of data points (at least 20 observations) to reliably capture temporal dynamics.

The analysis procedure begins by determining the optimal granularity or timestep interval, ensuring that it aligns with the constraints listed above. Time series distributions are plotted across various intervals:

$$\delta t = \{5 \text{ min}, 15 \text{ min}, 1 \text{ hr}, 1 \text{ day}, 1 \text{ week}\}$$

In cases where subtle periodic patterns are suspected, finer intervals (seconds or milliseconds) are examined. Since periodicity can differ among modules, an iterative refinement process determines the most suitable timestep interval for each module individually.

For predictive modeling, this assessment is applied to all selected highly active modules. Modules failing to meet the key constraints are filtered out to ensure model reliability.

For the classification task, the raw dataset—including chattering alarms—is initially analyzed to identify an optimal timestep interval that:

- Minimizes the influence of chattering alarms by reducing noise.
- Contains at least 20 data points to guarantee statistical reliability.
- Captures distinct temporal behaviors specific to each partition to justify their inclusion as predictive features.
- Minimizes instances, in which the dominant alarm category within each timestep interval is ambiguous.

Upon establishing optimal timestep intervals, two dataset types are finalized: the raw frequency datasets containing all alarm events for each highly active module (one dataset per partition), and refined classification datasets—one including and one excluding chattering alarms. These processed datasets are subsequently employed in training two distinct LSTM models:

- **LSTM Time Series Classifier:** Trained across all partitions using frequency counts, this model predicts the subsequent dominant categorical error within a sequence.
- **LSTM Time Series Predictor:** Four distinct models, one per partition, utilize differential frequency counts from selected modules to predict future frequency counts.

This systematic methodology ensures that only datasets meeting the essential criteria are employed, thereby optimizing LSTM model performance and maintaining consistency with the detector's intrinsic alarm behavior.

The next section describes the model development process from constructing, training, evaluating and deploying these LSTM-based models for these two distinct tasks, accompanied by a discussion of the chosen architecture and the data pre-processing.

3.3 LSTM Model Development

The model development strategy involves four primary steps: data pre-processing, architecture selection, compilation and training, evaluation and deployment.The model is compiled by specifying a loss function, an optimizer, and evaluation metrics. This compiling process configures the model for training by setting up how errors will be computed and how the weights will be updated, so that when the model is fitted, the training loop uses those configurations in the compiling process[35].

Hyperparameter optimization is integrated within the compilation and training phase by exploring a predefined search space, during model training. First, candidate values for each hyperparameter are specified. Next, a performance metric is chosen as the objective to optimize. Thereafter an optimization algorithm—typically Bayesian optimization—is then used[35].

Bayesian optimization is an efficient method for hyperparameter tuning, integrated within the compilation and training phases. Bayesian optimization is an automated method for efficiently identifying optimal hyperparameters. It operates by systematically constructing a probabilistic representation—known as a surrogate model—that approximates how the model's performance changes across different hyperparameter settings[35].

An acquisition function then guides hyperparameter selection by balancing two objectives: **exploration**, which involves evaluating less-explored regions to reduce uncertainty, and **exploitation**, which prioritizes regions likely to yield better performance based on prior evaluations. Through iterative selection, evaluation, and surrogate model updating, Bayesian optimization efficiently converges to optimal hyperparameter configurations[35].

This cycle—selecting hyperparameters, compiling the model with these settings, training the model, evaluating its performance on a validation set, and updating the surrogate model—is repeated until a pre-determined number of trials is completed or the performance converges. Ultimately, this process identifies the hyperparameter configuration that yields the best performance for the final model[35].

In the first stage of model development, the data is processed further to conform with the model input requirements. This involves pre-processing of time series datasets into compatible model inputs, which is done after the initial pre-processing of raw alarm logs into time series datasets. In both models, a sliding window technique was used to ensure that sequential dependencies were preserved, transforming alarm sequences into structured input-output pairs.

For the prediction model (implemented in PyTorch[66]), the pre-processing begins with normalization. Raw discrete values are scaled using MinMax normalization, which is particularly suitable for regression tasks where maintaining the original data distribution is essential, to map data into the [0, 1] range. This method preserves the relative magnitude relationships crucial for learning sequential dependencies, prevents exploding gradients by constraining the input values, and ensures compatibility with activation functions such as `tanh` and `sigmoid`[35].

A sliding window transformation segments the normalized time series into overlapping sequences, enabling the LSTM model to effectively capture temporal dependencies. Sequence length values typically range between 10–50 while typical values are usually between 1–5 for the stride[35].

Initially, the sequence length is set to 2.5% of the dataset length, or sample size, (before pre-processing and splitting), with a stride of $\frac{1}{10}$th that sequence length, and hyperparameter optimization is conducted under these fixed conditions. After locking in the resulting best hyperparameters, a second hyperparameter search is performed, varying the sequence length between 2.5% and 25% of the dataset

length while maintaining the stride at $\frac{1}{10}$th of the chosen sequence length. This two-phase approach aims to further enhance model performance.

Seperate tuning is used due to the computational cost associated with optimizing these parameters through automated methods in a single step[35], while the choice of ranges was done to ensure a balance between sensitivity to short-term variations and capturing long-term trends, and to minimize redundancy while preserving adequate context for effective learning.

Once the sequences are generated, the dataset is partitioned into training (70%), validation (10%), and testing (20%) sets. The training set is used to update the model parameters, the validation set is employed to guide early stopping and hyperparameter tuning, and the test set provides an unbiased evaluation. Finally, the sequences are converted into PyTorch tensors for efficient computation.

In the classification model (implemented in TensorFlow[67]), the pre-processing differs slightly. Since the data is multivariate in nature, the numerical features are standardized using `Standard Scaler` to extract the relative relationships between features rather than their absolute magnitudes[35].

After standardization, a sliding window transformation is applied using the same initial sequence-length range and stride setting as the prediction model. Finally, categorical target labels are one-hot encoded—for example, converting a label like "E1" into a binary vector ([1, 0, 0, 0, 0])—enabling the use of a softmax activation function in the model's output layer for effective multi-class classification.

The LSTM prediction model forecasts future values in a univariate time series dataset (e.g., alarm counts) through a structured pipeline whose architecture consists of five clearly defined layers:

1. **LSTM Layer:** Processes the input sequences with shape `(batch_size, timesteps, 1)`, capturing temporal dependencies by learning sequential patterns through gated recurrent units.
2. **Dropout Layer:** Randomly deactivates a fraction of the neuronal outputs per training step, preventing overfitting and enhancing generalization.
3. **LSTM Layer:** Extracts long-term dependencies by processing the output from the first LSTM layer, returning only the final hidden state for compact sequential representation.
4. **Dense Layer (ReLU):** A fully connected layer with ReLU activation refines extracted features, introducing non-linearity to improve predictive accuracy.
5. **Output Layer (Linear):** A fully connected layer with a single neuron and linear activation, generating the final numerical prediction for regression.

The classification model follows a different structure due to the multivariate nature of the input. It is structured as follows:

1. **Input Layer:** Defines the input shape `(batch_size, timesteps, features)` without performing computations - due to it's setup, it is required when using Tensorflow to use an Input layer
2. **Dropout Layer:** Immediately follows the input layer, randomly deactivating a fraction of input features to reduce reliance on dominant features and enhance generalization early in training.
3. **Dense Layer:** Applies a fully connected transformation with ReLU activation to refine input features before sequential processing, improving feature representation.
4. **Dropout Layer:** Regularizes the transformed features, preventing overfitting by reducing sensitivity to specific learned patterns.
5. **LSTM Layer:** Processes the refined feature sequence, capturing short- and long-term temporal dependencies while returning the full sequence of hidden states.

6. **Dropout Layer:** Regularizes the sequential output from the first LSTM layer to mitigate overfitting and promote robust learning.
7. **LSTM Layer:** Processes the output from the previous LSTM and returns only the final hidden state, condensing long-term dependencies into a compact representation for classification.
8. **Dropout Layer:** Regularizes the final hidden state to enhance generalization and prevent reliance on specific sequential patterns.
9. **Dense Layer:** Applies a fully connected transformation to refine extracted features, optimizing them for classification.
10. **Dropout Layer:** Adds a final regularization step before the classification output to improve robustness and mitigate overfitting.
11. **Output Layer:** A fully connected layer with a neuron for each class. The softmax activation function converts outputs into probabilities, generating the final output for classification.

Compilation and training for both models follow similar principles. For the prediction model, the loss function is MSE. For the classification model, categorical cross-entropy is used. In both cases, various optimizers are explored during hyperparameter tuning to find the most effective strategy. The hyperparameters in this study are uniformly applied across similar layer types to simplify optimization - for example, the dense units will be the same for all dense layers in the model architecture.

Hyperparameter tuning is performed using Bayesian optimization via Weights & Biases (WandB), which is a popular tool for experiment tracking and hyperparameter optimization that facilitates logging, visualization, and automated searches. It is chosen for its ease of integration with frameworks like PyTorch and TensorFlow, and for improving reproducibility and systematic experimentation[35].

Table 3.7 provides the hyperparameters selected for hyperparameter optimization for the LSTM models for both the classification and prediction, as well as the selected tuning ranges and commonly used values. A wide tuning range was chosen to ensure comprehensive coverage of all possible configurations, balancing optimal performance and computational efficiency.

Hyper-parameter	**Selected Range**	**Typical Values**
Optimizer	{Adam, Adamax, RMSprop, SGD}	Adam
Epochs	100 – 5000	100 – 500
Batch Size	8 – 512	32 – 128
LSTM Units	8 – 512	50 – 200
Dense Units	8 – 512	50 – 200
Learning Rate	1×10^{-7} – 1×10^{-1}	0.001 – 0.01
Weight decay	1×10^{-7} – 1×10^{-1}	0.0001 – 0.001
Dropout	0.01 – 0.75	0.2 – 0.5
Stride	round($\frac{1}{10}$ of sequence length)	1 – 5
Sequence length	round(2.5%–25% of sample size)	10 – 50

Table 3.7: Hyperparameter selection for LSTM models (prediction and classification tasks), illustrating the extensive tuning ranges explored during optimization.

Training employs early stopping. Training halts if the validation loss (or validation accuracy for classification) does not improve within a 10-epoch patience window. This strategy helps prevent

overfitting and reduces unnecessary computation. Evaluation of model performance requires the use of evaluation metrics. Each metric captures a different aspect of a model's abilities and incorrect or hasty conclusions can be made if a one-dimensional view of model performance is employed.

For example, relying solely on MSE might favour a model with low average error that occasionally produces very large mistakes (as shown by a high RMSE). For classification models, accuracy, precision, recall, and various F1-scores are employed. Relying only on accuracy or overall F1-score can be misleading, especially when performance varies across five classes.

Using multiple metrics helps ensure balanced performance and avoids selecting a model that appears superior based on limited indicators. Since a comprehensive set of evaluation metrics offers a multifaceted view of model performance, it was decided that all evaluation metrics would be used in the evaluation phase of model development. For the prediction models, MSE, RMSE, MAE, and MAPE were selected, while for the classification models, accuracy, precision, recall, and the various F1-scores were selected as the evaluation metrics.

Chapter4 Results and Discussion

In this chapter, we present a comprehensive analysis of the experimental outcomes and insights derived from the ATLAS TileCal error logs. The discussion is organized into several key sections. *Exploratory Data Analysis* examines the spatial and temporal distributions of errors across detector partitions and modules, revealing underlying patterns and regional susceptibilities. *LSTM Model Evaluation* follows, detailing the performance of the forecasting and classification models alongside an in-depth discussion of hyperparameter tuning and training strategies. The subsequent *Summary of Key Findings* synthesizes the insights gained from the analysis, emphasizing the implications for predictive maintenance and system reliability. Finally, *Limitations of Study* discusses the challenges encountered during model development and evaluation, including issues related to data quality, sample size, and potential overfitting. Together, these sections provide a clear narrative of the research outcomes and the practical considerations for future work.

4.1 Exploratory Data Analysis

Table 4.1 presents a comparative analysis of error distributions across TileCal barrel partitions, with percentages indicating the proportion of each error type within the respective partitions:

Label/ Partition	TileCal	EBA	EBC	LBA	LBC
EMR	2.92%	7.57%	0%	0.74%	7.63%
ALERT	18.26%	41.01%	50.47%	1.33%	53.69%
TRIPPED	1.16%	0.99%	9.43%	0.34%	2.28%
NO TOGGLE	13.16%	28.94%	26.65%	0.68%	7.92%
NO CONNECTION	64.50%	21.49%	13.44%	96.90%	28.47%
TOTAL	100%	17.48%	4.52%	63.20%	14.95%

Table 4.1: The Comparison of TileCal Error Distributions by Partition Revealed Regional Susceptibility Differences for Targeted Maintainance. The table is organized column-wise by partition and row-wise by error type.

The comparison of TileCal error distributions in Table 4.1 revealed critical disparities in both error type prevalence and partition susceptibility. Three partitions—EBA, EBC, and LBC—were found to be dominated by transient hardware faults, which are typically short-lived and recoverable. For instance:

- EBA and LBC exhibited high ALERT rates (41.01% and 53.69%, respectively), indicative of transient communication or synchronization issues.
- EBC showed negligible EMR (0%) but significant ALERT (50.47%), suggesting transient faults localized to specific modules.

In stark contrast, the LBA partition is overwhelmingly dominated by persistent NO CONNECTION errors (96.90%), pointing to systemic hardware or connectivity failures rather than transient anomalies. This partition alone accounts for 63.20% of total errors, underscoring its outlier status. Additionally, EBC exhibited the lowest overall error rate (4.52%), implying relative stability despite its transient-dominated profile.

These findings confirm that the different errors impact the functioning of TileCal to various different extents and various locations, highlighting the need for error- and partition-targeted maintenance strategies. After plotting the modular distributions of errors for the different partitions across all error categories, it was found that although some similarities exist for these distributions, the distributions remained largely distinct across the different partitions.

These findings confirm that each error distribution is also spatially dependent on module number and not only on the TileCal partition, or barrel region. This modular spatial inhomogeneity of errors is further illustrated in Figure 4.1, below, where NO TOGGLE errors in the EBA partition showed uneven modular clustering.

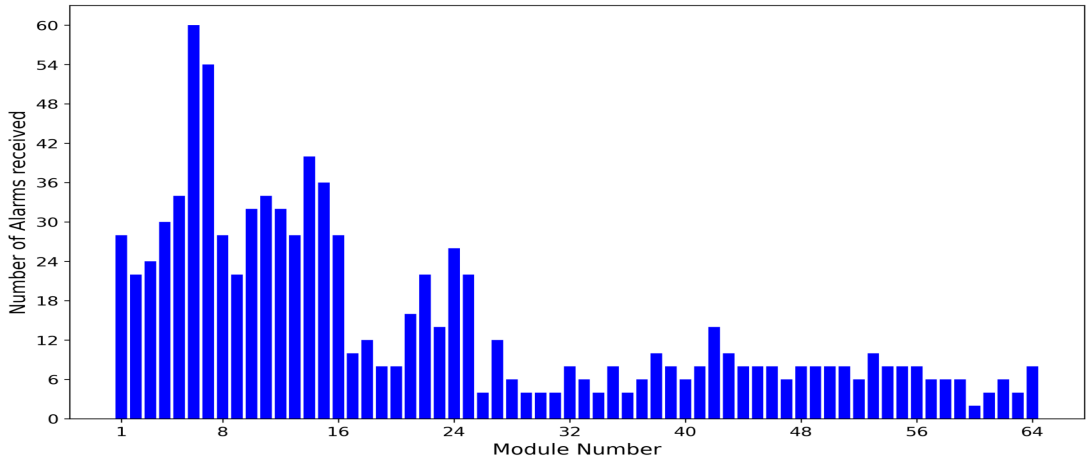


Figure 4.1: The Modular Distribution of NO TOGGLE errors in the EBA partition displayed an uneven spatial distribution of errors with a wide range of counts: maximum of 60 (module 6) and a minimum of 2 (module 60)

The spatial inhomogeneity of TileCal errors, established through partition-level distributions in Table 4.1 was further investigated using heatmaps to identify high-activity regions and modules. A global heatmap (not shown) spanning 10 months of Run-3 data revealed distinct error hotspots across partitions, with specific modules consistently exhibiting anomalously high error counts. These findings again confirm that error occurrences are not random but spatially dependent, correlating with partition and module locations.

Figure 4.2 provides localized heatmap views of two representative partitions (LBA and EBC), providing key observations which include:

- **LBA Partition:** Dominated by persistent `NC` (No Connection) errors, with Module 48 alone accounting for 11,146 counts, indicating severe systemic hardware or connectivity failures.
- **EBC Partition:** Characterized by transient `ALERT` errors, with Module 10 recording 64 counts, suggesting intermittent communication issues.

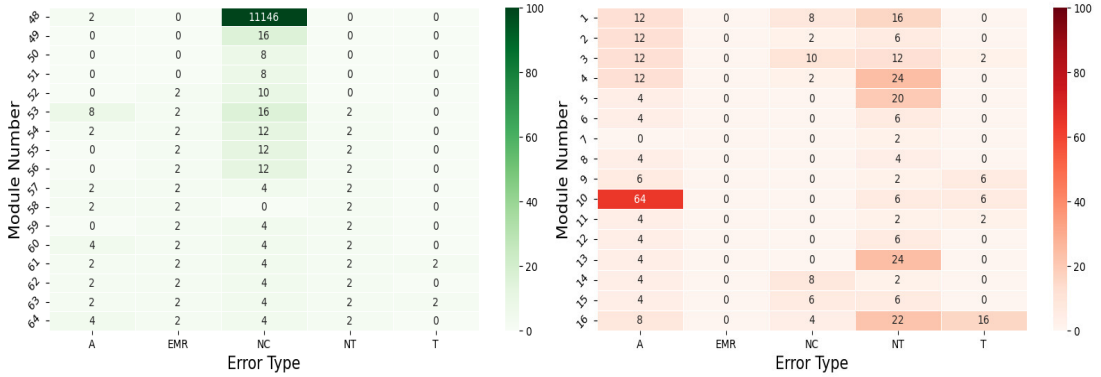


Figure 4.2: Local Heatmaps showing High Modular Activity in LBA (left) and EBC (right) Partitions: Color intensity represents alarm counts (z-axis), module numbers (y-axis), and error categories (x-axis).

Similar analyses for EBA and LBC partitions identified additional hotspots. For the EBA partition, module 6 registered 60 NO TOGGLE errors counts, while for the LBC partition, module 2 registered 180 ALERT error counts.

From the analyses of the local heatmaps, the modules displaying the highest modular activities irrespective of error for each partition with the respective error counts in parentheses are:

1. EBA: Module 6 for NO TOGGLE errors (60)
2. EBC: Module 10 for ALERT errors (64)
3. LBA: Module 48 for NO CONNECTION errors (11146)
4. LBC: Module 2 for ALERT errors (180)

These are the highest activity modules selected for the training and testing of the LSTM prediction models, and are the exact same modules that were identified in the previous section showing strong agreement between the two different methods.

The strong spatial dependencies observed at both partition and module levels underscore the utility of predictive models like LSTMs, which can leverage spatiotemporal error patterns for classification and forecasting. By training on historical error distributions, LSTMs can:

- Anticipate recurrent transient faults (e.g., `ALERT` in EBC).
- Flag persistent failures (e.g., `NC` errors in LBA) for pre-emptive maintenance.
- Prioritize interventions for high-risk modules (e.g., Module 48 in LBA).

Heatmap analysis has pinpointed critical error-prone regions in TileCal, providing actionable insights for targeted diagnostics. However, spatial patterns alone are insufficient for robust prediction; temporal variations in error signals must also be analyzed.

With the most highly active modules identified, the next step was to extract partition-specific temporal signatures from the time series datasets. These signatures must satisfy the key constraints outlined in Section 3.2 of the Methodology, including the temporal non-random hypotheses from Section 1.2 of the Introduction, to ensure the datasets are suitable for LSTM-based classification and prediction tasks. This process involved analyzing the datasets across various time bins and applying smoothing techniques to enhance temporal patterns.

For the prediction task, the optimal time bin size varied by module. Extensive analysis was conducted across the following time bins: 5 minutes, 15 minutes, 1 hour, 1 day, weekly, and monthly. In some cases, smaller bins were necessary to capture the required temporal constraints. When a module's time series failed to meet these constraints, it was replaced with the next most active module in its partition.

The final optimal time bins for the most highly active modules were determined as follows: 15-minute intervals for EBA and LBC modules, 25 ms intervals for the EBC module and Hourly intervals for the LBA module. To further refine the datasets, manual smoothing techniques (e.g., moving averages, exponential smoothing, or a combination) were applied.

The effectiveness of smoothing is illustrated in Figure 4.3, which compares the raw and the smoothed data for LBA Partition Module 48 (NO CONNECTION errors), highlighting the effectiveness of the manual moving average approach. After smoothing, these processed datasets were used to train, evaluate, and refine the respective LSTM predictive models.

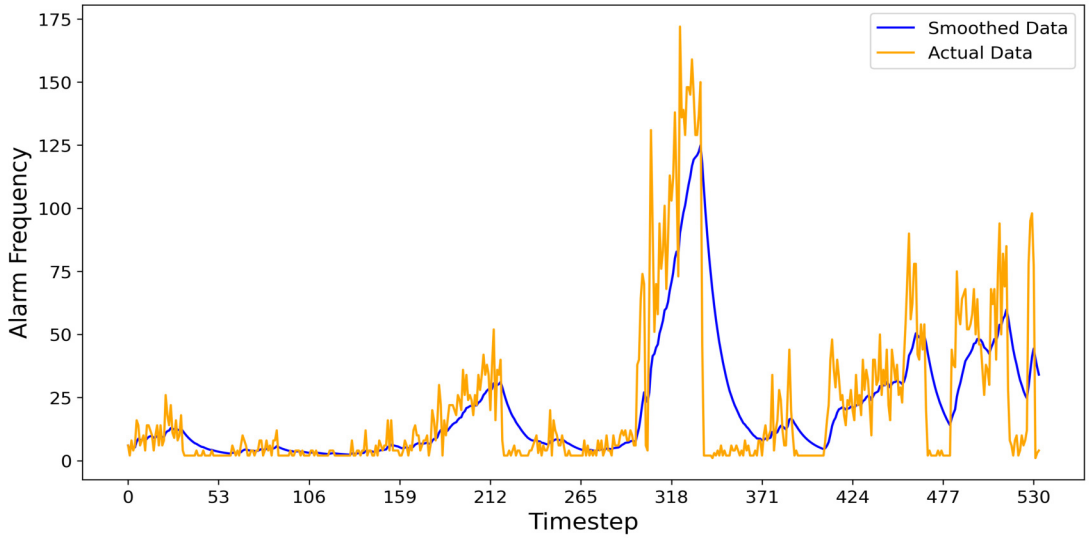


Figure 4.3: Time series comparison of raw (orange) and smoothed (blue) error counts for LBA Partition Module 48. The moving average filter reduces noise while preserving temporal signatures, enabling clearer pattern identification in this high-activity module.

For the time binning and chattering alarm removal needed for the classification task, the same process was followed, but no finer granularity was required. The optimal time bin was determined to be 15 minutes. Section 3.2 of the Methodology also specified specific constraints related to chattering alarms that needed to be considered. The 15-minute binning allowed for clearer temporal patterns to emerge after removing chattering alarms, as shown in Figure 4.4.

To further analyze the datasets, the smoothed time series distributions were decomposed by error type and partition. Figure 4.5 illustrates the spatio-temporal profiles for two of the five error categories:

The key findings based on the temporal distributions in Figure 4.5 were: For ALERT errors, the EBA partition (blue) showed sporadic spikes separated by long zero-error intervals, indicating low spike frequency, but high intensity. In contrast, for the NO CONNECTION errors, the LBA partition (red) maintained a non-zero baseline (≈5–10 errors per interval), suggesting persistent low-level connectivity issues, while the EBA partition (blue) exhibited periodic bursts at ≈100-interval cycles, likely tied to scheduled system checks or hardware duty cycles.

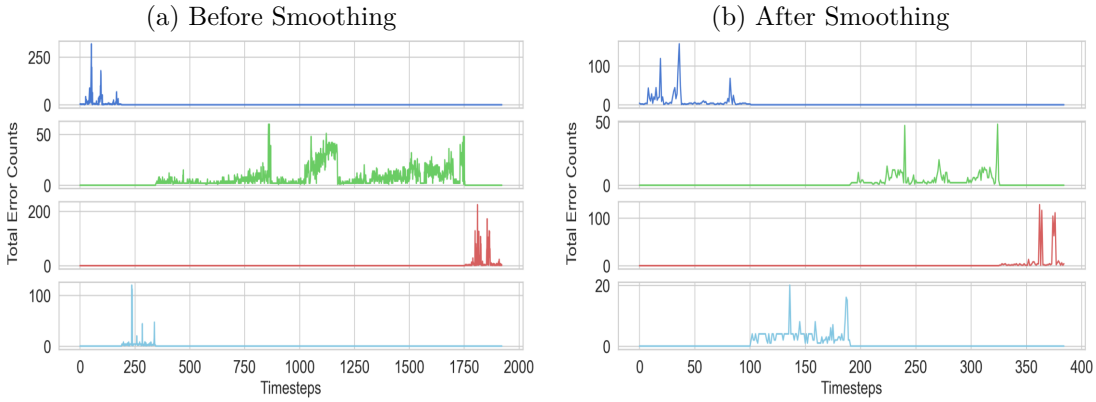


Figure 4.4: Time series distributions of total error counts per partition (15-minute intervals) before and after smoothing. Colors denote partitions: blue (EBA), green (EBC), red (LBA), cyan (LBC). Chattering alarms in LBA (a) obscure patterns; post-smoothing (b) reveals distinct temporal profiles.

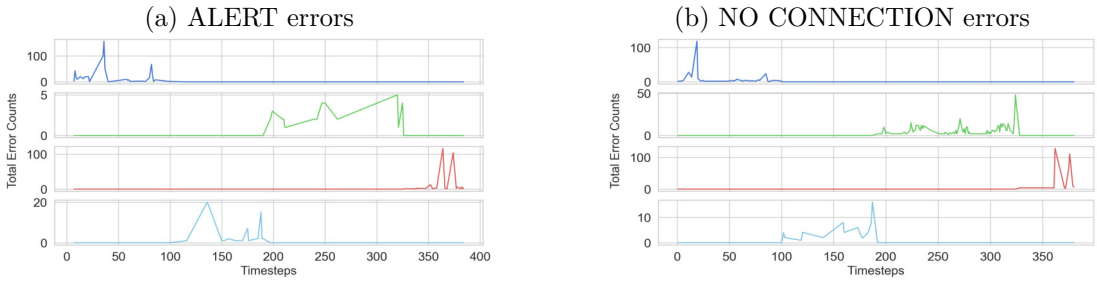


Figure 4.5: Smoothed time series distributions (15-minute intervals) partitioned by error type. Distinct temporal behaviors across partitions validate feature selection. Partition colors: blue (EBA), green (EBC), red (LBA), cyan (LBC).

These distinct profiles justify the use of partition-specific error counts as model features. However, overlapping zero-error regions and non-linear behaviors in some error types (e.g., EMR errors in EBC) complicate feature correlation with the dominant error label. Additionally, the class imbalance with most counts concentrated in the LBC partition for NO CONNECTION errors, further obscures clear feature-label relationships.

Despite these trade-offs, smoothing preserved critical temporal behaviors (e.g., Gaussian-like bursts separated by zero-error intervals) and mitigated class imbalance. However, reducing the dataset from 1,926 to 368 samples introduced compromises: temporal sampling biases favouring dominant features, under-representation of minority classes, incorrect assumption of partition-specific trends, and overfitting risks from modelling smoothed artifacts rather than true system dynamics.

While this approach is justified—as noise reduction was essential to isolate meaningful temporal patterns in the LBA partition, which accounted for an extremely disproportionate 63.2% of total errors and 96.9% of NO CONNECTION errors (see Table 4.1)—it inherently prioritizes LBA's majority-class trends at the risk of further under-representing minority classes such as TRIPPED/EMR errors in EBC/LBC partitions.

Furthermore, additional feature selection or dimensionality reduction could be incorporated as it serves to strengthen feature-label correlations and ensure distinct feature characteristics. For example, the partition-specific error behavior illustrated in Figure 4.5 could justify using and defining distinct

temporal features such as 'spike frequency', 'baseline levels', 'burst periodicity' for each partition in predictive modeling.

In the next section, LSTM-based models are evaluated for both time series forecasting and classification of partition-specific alarm data. Model configuration, hyperparameter tuning, and performance metrics are examined, comparing prediction (MSE, RMSE, MAE, MAPE) and classification (accuracy, precision, recall, F1-score) results. This comparison highlights the models' effectiveness and identifies opportunities for further refinement.

4.2 LSTM Model Evaluation

This section evaluates the performance of LSTM models developed for predicting and classifying hardware malfunction alarms within the ATLAS Tile Calorimeter. The prediction models forecast future alarm counts, while classification models identify specific error types. Both tasks are crucial for maintaining operational reliability and optimizing detector uptime during the HL-LHC data-taking periods, thereby supporting critical physics analyses such as Higgs di-boson detection.

All models underwent a two-stage Bayesian hyperparameter optimization process. Initially, core parameters—including batch size, LSTM units, dense layer units, dropout rate, and learning rate—were optimized. Subsequently, temporal parameters such as sequence length and stride were refined. The initial sequence lengths were set at 2.5% of the dataset size, with strides at one-tenth of these lengths. Both parameters were later optimized across broader ranges.

Each model used a consistent data split of 70% for training and 30% for validation. Due to the limited and inconsistent number of samples, the validation set also served as the test set. Additionally, weight decay, tuned at one-tenth of the learning rate range, was specifically applied to classification tasks to mitigate overfitting, particularly important given the multivariate nature of the datasets and significant class imbalances.

The LSTM classifiers were trained to predict the next dominant alarm type based on four features—total alarm counts per partition in 15-minute intervals—for chattering inclusive and chattering exclusive datasets. This approach allowed assessment of how chattering alarms impacted model performance and evaluated the effectiveness of the smoothing methods applied.

Conversely, the LSTM prediction models were trained to forecast future alarm counts using univariate datasets segmented by module-specific bins per partition. Given the effectiveness of manual smoothing demonstrated in Figure 4.3, the prediction models were trained and tested only on the datasets after manual smoothing was applied.

The optimal hyperparameter configurations across all models are summarized in Table 4.2, highlighting variations indicative of LSTM's leveraging capabilities:

Hyperparameter	**Tuning Range**	**EBA**	**EBC**	**LBA**	**LBC**	**Classification (Processed)**	**Classification (Raw)**
Batch Size	(8, 512)	76	121	97	112	136	256
LSTM Units	(8, 512)	59	35	98	45	32	128
Learning Rate	$(1\times10-71\times10^{-7}, 1\times10-11\times10^{-1})$	0.0082	0.0003	0.006	0.026	0.00024	0.0001
Weight Decay	$(1\times10-81\times10^{-8}, 1\times10-21\times10^{-2})$	-	-	-	-	0.2484	0.01593
Dropout	(0.01, 0.75)	0.046	0.122	0.415	0.403	0.096	0.25
Sequence Length	roundup(2.5%, 25%) data. len.	2	2	100	2	10	25
Stride	roundup(1/10 seq. len.)	1	1	10	1	1	3

Table 4.2: Optimal hyperparameter configurations for prediction and classification LSTM models, highlighting variations indicative of the presence of temporal complexities in the respective high activity modules, implying unique leveraging capabilities of the LSTMs.

Referencing Table 4.2, the classification models reveal that the chattering inclusive dataset required significantly larger batch size, dropout rate, and sequence length (256, 0.25, 25) compared to the chattering exclusive dataset (136, 0.096, 10).

In this case, the chattering inclusive LSTM leverages larger batch sizes and increased dropout rates to effectively mitigate noise from chattering alarms, while the chattering exclusive LSTM employed smaller batch sizes and dropouts, alongside shorter sequence lengths to capitalize on the temporal complexity simplified through smoothing, thereby optimizing predictive performance across distinct alarm behaviors.

These variations in alarm behavior across datasets are not explicitly determinable from Table 4.2 for the prediction models. However, several notable observations illustrate the LSTM's unique ability to leverage different dataset-dependent elements were made.

For example, the EBA model, employs moderate architectural complexity (59 LSTM units, 64 dense units) combined with a moderate learning rate (0.0082), effectively balancing detailed temporal information, model capacity, and training stability.

The LBA model prioritizes high model capacity (98 LSTM units, 128 dense units), aggressive regularization (dropout = 0.415), and a focus on capturing long-term temporal dependencies (sequence length = 100, stride = 10). In contrast, the LBC model adopts moderate architectural complexity (45 LSTM units, 64 dense units), similarly aggressive regularization (dropout = 0.403), and facilitates rapid convergence through a higher learning rate (0.026), like the EBA and EBC models employs a sequence length = 2 and fine-grained temporal resolution (stride = 1) to effectively balance detailed temporal information with computational efficiency.

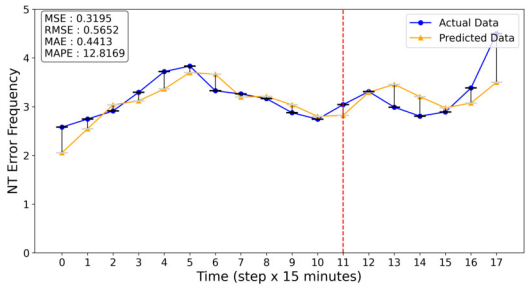
Thus, despite the LSTM architectures being identical for all these models, the EBA and LBA models prioritize predictive power through more complex networks, whereas EBC and LBC favor computational efficiency and stronger regularization, with EBC focusing on minimalism and LBC emphasizing rapid training.

Figure 4.6, above, presents graphical plots for a comparative summary of the LSTM prediction models in terms of performance and reliability.

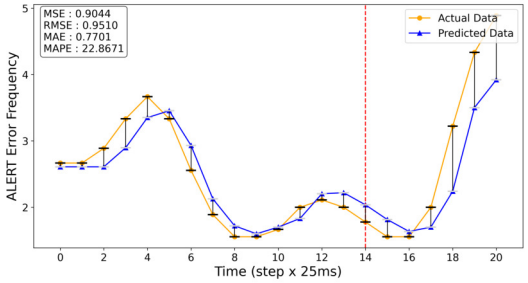
Comparing the models across the different modules, however, is inherently challenging due to the substantial variability in alarm counts and differences in sample sizes - factors that directly influence evaluation metrics and complicate direct comparisons. Moreover, data leakage can distort these metrics by enabling the model to exploit unintended temporal dependencies, thereby artificially improving validation performance.

For instance, a model with low MSE and MAE but high MAPE may perform well in absolute terms, with small errors that appear large as percentages while the actual counts are low - as observed for the EBC and LBC models. Conversely, a model with high MSE and MAE yet a low MAPE might mask significant deviations if its errors are evenly distributed across varying alarm magnitudes - as observed in the LBA model.

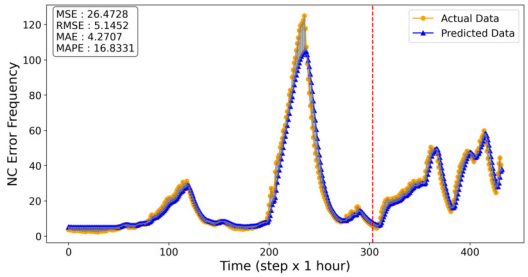
From the content in Figure4.6, it was concluded the EBA model delivered the most accurate and stable predictions: it exhibited low MSE, MAE, and MAPE across the board - its stable, smooth alarm distribution enabled accurate forecasting without large fluctuations. In contrast, the EBC and LBC models struggled with the volatility in the data leading to relatively higher MAPE values and lower MSE and MAE scores, displaying intermediate performance.



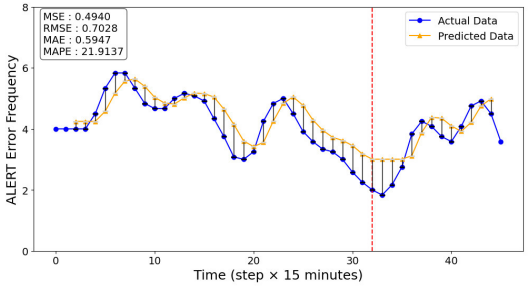
(a) EBA Module 6



(b) EBC Module 10



(c) LBA Module 48



(d) LBC Module 2

Figure 4.6: Summary of LSTM Forecasting Performance for Highly Active TileCal Modules: Actual (blue) vs. Predicted (orange) trends with error bars. The data is partitioned using a 70:30 train-validation split, indicated by the dotted red vertical line. All models demonstrated strong predictive capabilities.

The EBC model registered higher absolute errors (MSE: 0.9044, MAE: 0.7701) due to more irregular peak patterns in the data, leading to less stable predictions, and its high MAPE (22.87%) indicates difficulty in capturing extreme fluctuations. Meanwhile, the LBC model recorded the lowest MSE and MAE among these two (MSE: 0.4940, MAE: 0.5497), yet it still exhibited a high MAPE (21.91%), suggesting that while its absolute error performance was good, challenges remain in low-count regions.

The LBA model, however, despite also displaying high volatility in the data, exhibited a competitive MAPE (16.83% compared to 12.82% for the EBA model) but showed significantly higher values across the other evaluation metrics (MSE: 5.9980, MAE: 4.2707). Further investigation attributed this mismatch to data leakage: a lengthy sequence length (100) was used without a buffer zone, allowing the model excessive overlap between training and validation data.

As a result, the LBA model's apparent stellar performance arose from memorizing historical patterns rather than truly learning them. In contrast, for the other models employing shorter sequence lengths (EBA,EBC, and LBC), any data leakage appears marginal, ensuring more reliable predictions.

To mitigate data leakage and enhance long-term forecasting reliability, future work should consider introducing a buffer zone between the training and validation sets, implementing cross-validation to reduce reliance on a single validation set, and maintaining a separate test set to assess generalization beyond the validation period.

Despite these challenges, the models demonstrate strong predictive potential. Refining hyperparameters—such as reducing sequence lengths or applying additional regularization techniques—could further improve performance while minimizing leakage risks. Overall, the EBA model emerges as the best-performing option, whereas the LBA model's susceptibility to data

leakage undermines its utility, and the EBC and LBC models offer intermediate, yet promising, performance.

Now that the prediction models have been evaluated, we next turn our attention to the LSTM classification models. These models were trained and tested on the raw (chattering-inclusive) and processed (chattering-removed) datasets, respectively, to assess the impact of noise reduction and class imbalance on error classification ability.

Table 4.3 provides a comparative summary of LSTM model performance using key performance metrics—including precision, recall, F1-score, and support—for each error type, along with the macro and weighted averages.

	Raw Dataset Model				**Processed Dataset Model**			
Class	**Precision**	**Recall**	**F1**	**Support**	**Precision**	**Recall**	**F1**	**Support**
E1: ALERT	0.58	0.13	0.22	50	0.64	0.45	0.53	20
E2: TRIPPED	0.00	0.00	0.00	10	0.67	0.29	0.40	7
E3: NO CONNECTION	0.94	0.98	0.96	407	0.80	0.95	0.87	59
E4: EMR	0.00	0.00	0.00	6	1.00	0.29	0.44	7
E5: NO TOGGLE	0.09	0.80	0.15	7	0.43	1.00	0.60	3
Macro Avg	0.32	0.38	0.27	480	0.71	0.59	0.57	96
Weighted Avg	0.87	0.88	0.86	480	0.76	0.75	0.72	96
Accuracy				88%				75%

Table 4.3: Comparative Summary of LSTM Model Classifier Performance on their respective test sets.

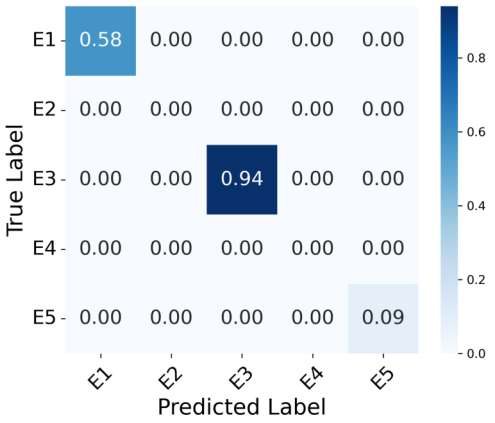
On the raw dataset, the overall accuracy of the raw dataset LSTM model was 88%, primarily driven by the dominant E3 (NO CONNECTION) class, which comprised nearly 85% of the testing samples and achieved high precision and recall (above 90%). However, this came at the expense of the minority classes: both E2 (TRIPPED) and E4 (EMR) registered 0% recall, and although class E5 (NO TOGGLE) reached 80% recall, its precision was only 9%, indicating many misclassifications.

In contrast, on the processed dataset, the processed dataset LSTM model achieved a lower overall accuracy of 75% due to a more balanced class distribution (with E3 constituting 62% of samples). Importantly, the performance on minority classes improved markedly. For example:

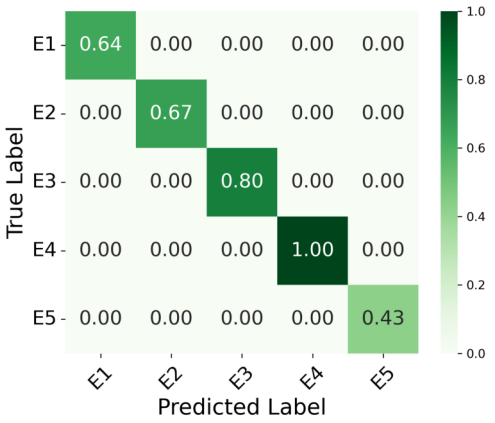
- **E1 (ALERT)**: Improved from 58% to 64% precision and from 13% to 45% recall.
- **E2 (TRIPPED)**: Increased from 0% to 67% precision and from 0% to 29% recall.
- **E4: (EMR)**: Increased from 0% to 100% precision and from 0% to 29% recall.

As a result, the Macro F1-score nearly doubled from 27% to 57%, reflecting a more balanced and reliable performance across all classes.

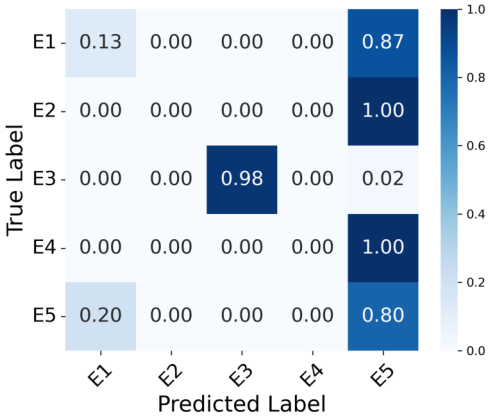
Figure 4.7 compares the performance of the LSTM classifiers graphically using confusion matrices. The top row displays recall-based confusion matrices—each row normalized to 100%—to show the proportion of actual instances correctly classified within each class. The bottom row presents precision-based confusion matrices with column normalization (each column normalized to 100%), which reflects the accuracy of the predictions for each class.



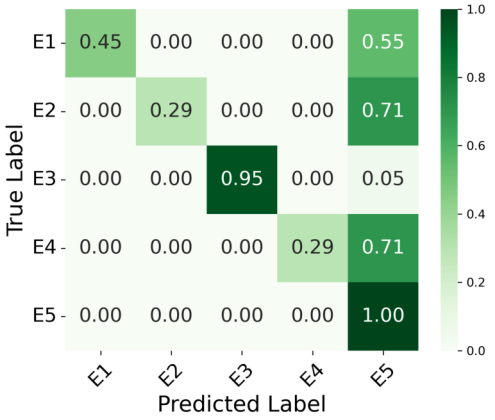
(a) Recall of chattering-inclusive LSTM



(b) Recall of chattering-removed LSTM



(c) Precision of chattering-inclusive LSTM



(d) Precision of chattering-removed LSTM

Figure 4.7: Graphical comparison of LSTM classifier performance: Top row shows recall-based matrices and bottom row shows precision-based matrices.

The confusion matrices further illustrate these findings. For the raw dataset LSTM, the heavy bias toward the E3 class in the test set is evident, with significant misclassifications for the minority classes. The processed dataset LSTM, benefiting from noise reduction, showed a greater diagonal dominance, which signified improved classification confidence and better differentiation between alarm types.

In summary, while the raw dataset model achieved higher overall accuracy, this was largely attributable to the dominance of a single class, resulting in severe imbalance and poor minority-class detection. The processed dataset model, although yielding lower overall accuracy, offers substantially enhanced performance for minority classes and a higher Macro F1-score.

These improvements suggest that noise reduction and class rebalancing not only enhance model reliability for critical applications—such as ensuring the operational efficiency of the ATLAS Tile Calorimeter and supporting key physics analyses—but also pave the way for further model optimization through refined hyperparameters and data quality strategies.

In conclusion, the LSTM models demonstrated significant potential for enhancing the reliability and operational efficiency of the ATLAS Tile Calorimeter by effectively predicting and classifying hardware malfunctions. While challenges such as data leakage and class imbalance were identified, the models'

overall performance underscores their capability to support critical physics analyses, such as Higgs di-boson detection. By refining hyperparameters, addressing data quality issues, and implementing robust validation strategies, these models can be further optimized to ensure accurate and reliable predictions, ultimately contributing to the success of high-energy physics experiments during the HL-LHC era.

4.3 Summary of Key Findings

- **Need for Predictive Maintenance**
  - Partition-specific Faults: Infrequently occurring ALERT and TRIPPED errors in the EBA, EBC, and LBC partitions and frequently occurring NO CONNECTION errors in the LBA partition were identified.
  - High-Risk Hotspots: Modules such as LBA module 48, EBA module 6, and LBC module 2 show high error activity, underscoring the need for targeted maintenance.
- **Utility of NLP and ML**
  - Feature Extraction: NLP-driven data processing enabled structured time-series data to be constructed out of the categorical nature of the alarm logs.
  - Proof of Concept: The notion of an ML-driven maintenance in this context demonstrates potential to reduce downtime and support analyses like Higgs di-boson detection.
- **Efficacy of LSTM Models**
  - Alarm Prediction: Forecasting models of the high activity modules achieved MAPE scores of 12–23%, capturing strong temporal dependencies.
  - Alarm Classification: Models reached 75% overall accuracy with a Macro F1 of 57%, ensuring robust dominant fault classification across the different classes.
  - Feasibility: Results validate the viability of LSTM-based spatio-temporal modeling for fault identification and prevention in high-energy physics.
- **Impact and Reliability**
  - Targeted Maintenance: Identifying high-risk modules enables proactive fault management, and forecasting errors by type provides actionable insights for scheduling interventions. The LSTM framework and proof-of-principle align with ATLAS goals—especially for Higgs di-boson detection—paving the way for improved detector uptime and reliability under high-luminosity conditions.
- **Future Applications**
  - Real-Time Deployment: The approach is promising for seamless integration into the ATLAS DCS, offering real-time predictive maintenance that could significantly enhance operational control and reduce downtime.
  - Extensibility to Other Tasks: The framework is designed to be expandable to additional NLP/ML applications, such as event reconstruction and anomaly detection, providing a versatile foundation for further advancements in high-energy physics.

- **Challenges in Predictive Maintenance**
  - Compatibility challenges: Incompatibility with legacy systems hinder the integration of ML-driven predictive tools into the DCS. This leaves a passive system struggling to capture subtle emerging fault patterns, while dealing with persistent issues.
  - Limited DCS Capabilities: The current passive setup does not extract or leverage information on highly active modules, which is an obstacle to developing an integrated system for targeted, module-specific maintenance interventions.

- **Barriers to Effective NLP and ML Utilization**
  - Limited Data Selection: Relying solely on error-level alarms may omit subtle signals and contextual details, reducing the richness of the dataset.
  - Biased Topic Analysis: The current alarm generalization approach may not capture all informative features, which can undermine the performance of subsequent ML models in fault prediction.

- **Limitations in LSTM Model Efficacy**
  - Data Leakage: Limited separation between training and validation data (with no dedicated test set) increased data leakage risks leading to overestimated model performance.
  - Class Imbalance: Chattering alarms hindered minority class detection; while their removal improved model performance, it significantly reduced dataset size and may have compromised representativeness.
  - Model Reliability Concerns: Although the results serve as a proof of principle, addressing sample size variability, data leakage, and class imbalance is essential before full deployment.

- **Challenges to Impact and Reliability**
  - Evaluation Metric Limitations: Common evaluation metrics (e.g., MAPE, overall accuracy) may not fully capture a model's performance on rare but critical fault events, potentially leading to false alarms and wasted maintenance efforts.
  - Deployment Challenges: Scalability remains a concern, as integrating additional sensor data and deploying the framework within the ATLAS DCS pose significant operational challenges.
  - Experimental Set-up Consistency Issues: Variations in TileCal module composition, performance, run configuration and calibration status hinder consistent fault pattern identification, while statistical fluctuations in experimental conditions obscure the actual characteristic temporal alarm behaviour.

- **Boundaries to Future Applications**
  - Limited Validation and Testing: The models require further testing and refinement and have not yet been validated in a real-time DCS environment, limiting insights into their practical deployment.
  - Scope Limitations: Although the framework is expandable to other NLP/ML tasks, its current implementation is limited to predictive maintenance, and its effectiveness in broader applications remains untested.

Chapter5 Conclusion

This study has successfully demonstrated the feasibility of employing NLP-driven LSTM models for predictive maintenance in the TileCal of ATLAS. The essential spatial and temporal patterns were effectively captured by transforming unstructured alarm logs into structured time series datasets, enabling the development of robust predictive and classification models. The predictive models achieved MAPE scores between 12% and 23%, while the classification model attained an overall accuracy of 75%. These results underscore the potential of ML-driven strategies to enhance operational efficiency and minimize unplanned downtime in high-energy physics (HEP) experiments, especially during high-luminosity operations of the HL-LHC era.

The key contributions of this work include:

- **Proof of Concept for Predictive Maintenance:**
  By leveraging NLP techniques to pre-process alarm logs and LSTM models to capture temporal dependencies, this study provides a proof of concept for predictive maintenance in the ATLAS TileCal. The models' ability to forecast future alarm counts and classify dominant error types demonstrates the viability of integrating machine learning into the DCS for real-time fault detection and pro-active maintenance.
- **Identification of High-Risk Modules:**
  Through exploratory data analysis, specific modules within the TileCal that exhibited disproportionately high error rates were identified. These high-risk hotspots, such as LBA module 48 and EBA module 6, highlight the need for targeted maintenance strategies to mitigate persistent hardware malfunctions.
- **Improved Fault Classification:**
  The LSTM classification model, trained on both raw and processed datasets, showed significant improvements in detecting minority error classes after noise reduction. This balanced performance across error categories ensures that critical faults are not overlooked, even in the presence of class imbalance.

Despite these successes, several limitations and challenges remain. The study faced issues related to data leakage, class imbalance, and the limited size of the dataset, which may have impacted the models' generalizability. Additionally, the current framework has not yet been validated in a real-time DCS environment, and further testing is required to assess its scalability and robustness under operational conditions. Future work should focus on refining data pre-processing techniques, increasing sample sizes, and integrating the models into the DCS for continuous monitoring and real-time deployment.

The implications of this research extend beyond the ATLAS TileCal. The proposed framework is adaptable to other sub-detectors and experimental setups within the ATLAS experiment. By reducing unplanned downtime and optimizing detector performance, this approach has the potential to accelerate scientific progress, particularly in critical analyses such as Higgs di-boson detection.

In conclusion, this study lays the groundwork for a ML–driven predictive maintenance strategy that aligns with the goals of the HL-LHC era. While further refinement and validation are necessary, the results demonstrate the promise of LSTM-based models in enhancing the reliability and efficiency of HEP experiments, ultimately contributing to the success of groundbreaking scientific discoveries.

References

[1] CERN. Photograph: Cern ac. DIGITAL, 2001. Accessed from Bulletin Issue: 46/2001, Mon 12 Nov 2001. © 1998-2024 CERN (License: CC-BY-4.0).

[2] H. Tlou. Integration and commissioning of the ATLAS Tile Demonstrator Module for Run 3. *Journal of Instrumentation*, 17(11):C11010, 2022. doi: 10.1088/1748-0221/17/11/C11010.

[3] F. Martins. The ATLAS tile calorimeter DCS for Run 2. In *2016 IEEE Nuclear Science Symposium, Medical Imaging Conference and Room-Temperature Semiconductor Detector Workshop (NSS/MIC/RTSD)*, 2016. doi: 10.1109/nssmic.2016.8069837.

[4] Adarsha Ruwali, A. J. Sravan Kumar, Kolla Bhanu Prakash, G. Sivavaraprasad, and D. Venkata Ratnam. Implementation of Hybrid Deep Learning Model (LSTM-CNN) for Ionospheric TEC Forecasting Using GPS Data. *IEEE Geoscience and Remote Sensing Letters*, 18(6):1004–1008, 2021. doi: 10.1109/LGRS.2020.2992633. URL https://doi.org/10.1109/LGRS.2020.2992633.

[5] G. Van Houdt, C. Mosquera, and G. Nápoles. A Review on the Long Short-Term Memory Model. *Artificial Intelligence Review*, 53(8), 2020. doi: 10.1007/s10462-020-09838-1. URL https://doi.org/10.1007/s10462-020-09838-1.

[6] Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[7] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018. URL https://otexts.com/fpp3/.

[8] P. Lau and D. Weidong. *Time-Series Classification and Prediction*. Springer, New York, 2009. ISBN 978-1-4419-0514-4.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, chapter 8, pages 307–387. MIT Press, 2016. ISBN 978-0262035613.

[10] V. Parameswaran Nair. *Concepts in Particle Physics: A Concise Introduction to the Standard Model*, chapter 7, pages 181–196. World Scientific Publishing Company, 2018. ISBN 9789813227552.

[11] Gregorio Bernardi and Jacobo Konigsberg. The Higgs Boson Search and Discovery. 2016. doi: 10.1142/9789814733519_0015. URL https://cds.cern.ch/record/2217161.

[12] Steven D. Bass, Albert De Roeck, and Marumi Kado. The Higgs boson implications and prospects for future discoveries. *Nature Reviews Physics*, 3(9):608–624, 2021. ISSN 2522-5820. doi: 10.1038/s42254-021-00341-2. URL http://dx.doi.org/10.1038/s42254-021-00341-2.

[13] Henning Bahl, Johannes Braathen, Martin Gabelmann, and Georg Weiglein. anyH3: precise predictions for the trilinear higgs coupling in the standard model and beyond. *The European Physical Journal C*, 83(12):12173, 2023. doi: 10.1140/epjc/s10052-023-12173-8. URL https://link.springer.com/article/10.1140/epjc/s10052-023-12173-8.

[14] M.L. Mangano, G. Ortona, and M. Selvaggi. Measuring the Higgs self-coupling via Higgs-pair production at a 100 TeV p–p collider. *Eur. Phys. J. C*, 80:1030, 2020. URL https://doi.org/10.1140/epjc/s10052-020-08595-3.

[15] Biagio Di Micco, Maxime Gouzevitch, Javier Mazzitelli, and Caterina Vernieri. Higgs boson potential at colliders: Status and perspectives. *Reviews in Physics*, 5:100045, November 2020. ISSN 2405-4283. doi: 10.1016/j.revip.2020.100045. URL http://dx.doi.org/10.1016/j.revip.2020.100045.

[16] Fabienne Landua. The CERN accelerator complex layout in 2022. Complexe des accélérateurs du CERN en janvier 2022. 2022. URL https://cds.cern.ch/record/2813716. General Photo.

[17] ATLAS Collaboration. Search for Elusive 'di-Higgs Production' Reaches New Milestone. https://atlas.cern/updates/briefing/new-milestone-di-Higgs-search#:~:text=ATLAS%20physicists%20are%20looking%20for [Accessed 12 Feb. 2024], 2021.

[18] J. T. Boyd. LHC Run-2 and Future Prospects. In *Proceedings of the 2019 European School of High-Energy Physics, St Petersburg, Russia, 4–17 September 2019*, volume 5, pages 247–258. CERN Yellow Reports: School Proceedings, 2021. doi: 10.23730/CYRSP-2021-005.247. URL https://arxiv.org/abs/2001.04370.

[19] G. Aad, J. Adelman, and S. et al. Arfaoui. Data quality from the detector control system at the atlas experiment. *Journal of Physics: Conference Series*, 219(1 PART 2):022037, 2010. doi: 10.1088/1742-6596/219/2/022037. URL https://doi.org/10.1088/1742-6596/219/2/022037.

[20] Viatcheslav Filimonov. The Detector Control System of the ATLAS Experiment. Scientific Figure on ResearchGate, 2008. URL https://www.researchgate.net/figure/ATLAS-DCS-architecture_fig3_228653457. Accessed 13 Feb, 2024.

[21] ATLAS Collaboration. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716:1–29, 2012. doi: 10.1016/j.physletb.2012.08.020. URL https://doi.org/10.1016/j.physletb.2012.08.020.

[22] C. Clement and ATLAS Collaboration. Upgrade of the ATLAS Hadronic Tile Calorimeter for the High Luminosity LHC. *Physics of Atomic Nuclei*, 84:368–372, 2021. doi: 10.1134/S1063778821030091.

[23] F.C. Argos, S. Moayedi, O. Solovyanov, G. Usai, and H. Wilkens. ATLAS Tile Calorimeter Phase II Upgrade: Entering Production phase. https://ep-news.web.cern.ch/content/atlas-tile-calorimeter-phase-ii-upgrade-entering-production-phase, 2023. Accessed on 14 Feb. 2024.

[24] Technical Design Report for the Phase-II Upgrade of the ATLAS Tile Calorimeter. Technical report, CERN, Geneva, 2017. URL https://cds.cern.ch/record/2285583.

[25] H. Wilkens. Coloured Run 3 Version of TDR Fig. 5.12 (2008) with E3 to eta=1.6, E4 to Eta 1.72. Available at https://twiki.cern.ch/twiki/pub/AtlasPublic/PublishedTilecalFigures/TileCal_Cells_and_Rows_run3_color.pdf, 2024. Accessed 7 Oct. 2023.

[26] The ATLAS Collaboration. *Technical Design Report for the Phase-II Upgrade of the ATLAS Tile Calorimeter, ATLAS-TDR-028, CERN-LHCC-2017-019*, chapter 4: Technical Summary, pages 58–67. CERN, 2017. [Available online] : https://cds.cern.ch/record/2285583.

[27] The ATLAS Collaboration. *Technical Design Report for the Phase-II Upgrade of the ATLAS Tile Calorimeter, ATLAS-TDR-028, CERN-LHCC-2017-019*, chapter 7:Low and High Voltage Power Supplies, pages 111–139. CERN, 2017. [Available online] : https://cds.cern.ch/record/2285583.

[28] R. Turner. *Detector Control Systems and Searches for Leptonic Decays of the Higgs Boson in the ATLAS Experiment at the LHC*. PhD thesis, University of Birmingham, 2020. URL https://etheses.bham.ac.uk/id/eprint/10891/7/Turner2020PhD.pdf.

[29] The ATLAS Collaboration. *Technical Design Report for the Phase-II Upgrade of the ATLAS Tile Calorimeter, ATLAS-TDR-028, CERN-LHCC-2017-019*, chapter 9: System Integration Aspects, pages 157–165. CERN, 2017. [Available online] : https://cds.cern.ch/record/2285583.

[30] C. Marques, A. Gomes, et al. The TileCal DCS Detector Control System. In *Proceedings of ICAP2006*, Chamonix, France, 2006. TUPPP14.

[31] J. Marques et al. Advancing atlas dcs data analysis with a modern data platform. *Journal of Instrumentation*, 19(3):P03012, 2024. doi: 10.1088/1748-0221/19/3/P03012. URL https://doi.org/10.1088/1748-0221/19/3/P03012.

[32] P. Starovoitov. Upgrade of ATLAS Hadronic Tile Calorimeter for the High-Luminosity LHC. *Instruments*, 6(4):54, 2022. doi: 10.3390/instruments6040054.

[33] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006. ISBN 978-0387310732.

[34] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 3rd edition, 2020. ISBN 978-0131873216.

[35] S. Cai, A. Palazoglu, L. Zhang, and J. Hu. Process alarm prediction using deep learning and word embedding methods. *ISA Transactions*, 85:274–283, 2019. doi: 10.1016/j.isatra.2018.10.032.

[36] Raymond S. Beebe. *Predictive Maintenance of Pumps Using Condition Monitoring*. Elsevier, 2016. ISBN 978-0128038282.

[37] Yoav Goldberg. A Primer on Neural Network Models for Natural Language Processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016. doi: 10.1613/jair.4992.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 26, pages 3111–3119, 2013. URL https://arxiv.org/abs/1310.4546.

[39] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL https://aclanthology.org/D14-1162/.

[40] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 135–144, 2017. URL https://aclanthology.org/E17-1013/.

[41] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, Fethi Bougares, and Holger Schwenk. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014. URL https://aclanthology.org/D14-1179/.

[42] Zachary C. Lipton. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv preprint arXiv:1506.00019*, 2015. URL https://arxiv.org/abs/1506.00019.

[43] Alex Graves. *Speech Recognition with Deep Recurrent Neural Networks*. PhD thesis, University of Toronto, 2013. URL https://www.cs.toronto.edu/~graves/phd.pdf.

[44] Analytics Vidhya. Implementing LSTM for Human Activity Recognition using Smartphone Accelerometer Data, 2021. URL https://www.analyticsvidhya.com/blog/2021/07/implementing-lstm-for-human-activity-recognition-using-smartphone-accelerometer-data/. Accessed: 2022-04-07.

[45] D. Alharbi, A. Zahrani, M. Awais, and S. Abid. Human activity recognition using lstm networks: A review of applications and challenges. *Journal of Ambient Intelligence and Humanized Computing*, 13(8):3381–3399, 2022. doi: 10.1007/s12652-022-03812-5.

[46] K. Zhang, Y. Chen, and Z. Xu. Time-series data preprocessing for machine learning: A survey. *Journal of Machine Learning Research*, 20:1–45, 2019. DOI: 10.5555/12345678.

[47] D.P. Kingma and J.B. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1412.6980.

[48] Optuna. Optuna documentation. https://optuna.readthedocs.io/en/stable/, 2019. Accessed: 2022-01-09.

[49] Hyperopt. Hyperopt documentation. https://hyperopt.github.io/hyperopt/, 2013. Accessed: 2022-01-09.

[50] Weights & Biases Team. Weights & Biases Documentation. https://wandb.ai, 2021. Accessed: 2022-01-09.

[51] T. Tieleman and G. Hinton. Lecture 6.5: Rmsprop: Divide the gradient by a running average of its recent magnitude. In *Neural Networks for Machine Learning*, 2012. Lecture slides available at http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[52] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951. doi: 10.1214/aoms/1177729586.

[53] 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.

[54] Franco Scarselli, Michela Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Giampiero Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[55] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[56] D. Guest, K. Cranmer, and D. Whiteson. Deep learning and its application to lhc physics. *Annual Review of Nuclear and Particle Science*, 68:161–181, 2018.

[57] M. Paganini, L. de Oliveira, and B. Nachman. Accelerating science with generative adversarial networks: An application to 3d particle showers in multilayer calorimeters. *Physical Review Letters*, 120(4):042003, 2019.

[58] A. Butter and et al. Graph neural networks for particle reconstruction in high energy physics. *Journal of Instrumentation*, 15:P08002, 2020.

[59] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308, 2014.

[60] A. Mehrabi, M. Bagheri, Majid Nabi Bidhendi, Ebrahim Biniaz Delijani, and M. Behnoud. Improved Porosity Estimation in Complex Carbonate Reservoirs Using Hybrid CRNN Deep Learning Model. *Research Square*, 2024. doi: 10.21203/rs.3.rs-3923665/v1.

[61] V Amrith et al. An Early Malware Threat Detection Model Using Conditional Tabular Generative Adversarial Network. *IEEE*, July 6 2023. doi: 10.1109/icccnt56998.2023.10307903. URL https://doi.org/10.1109/icccnt56998.2023.10307903. Accessed 27 Jan. 2024.

[62] Mohit Beniwal. *Predicting Indian and Global Stock Indices by Applying Machine Learning Techniques*. PhD thesis, Delhi School of Management, 2018. URL http://hdl.handle.net/10603/544321.

[63] A. Apollonio, TE-MPE, and L. Serio. Data-driven operation and maintenance for particle accelerators. URL https://api.semanticscholar.org/CorpusID:266598806.

[64] ATLAS Tile DCS Development Team. PullAlarms: ATLAS TileCal Alarm Handling Repository, 2025. URL https://gitlab.cern.ch/ATLAS-Tile-DCS-Dev/public/-/tree/master/PullAlarms. [Accessed: 2025-03-09].

[65] ATLAS Collaboration. ATLAS Tile Calorimeter Operations, 2025. URL https://twiki.cern.ch/twiki/bin/viewauth/Atlas/TileOperations. [Accessed: 2025-03-09].

[66] PyTorch. Pytorch Documentation, 2016. URL https://pytorch.org. Accessed: 2022-01-07.

[67] TensorFlow. Tensorflow documentation. https://www.tensorflow.org/, 2015. Accessed: 2022-01-07.