BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

SATELLITE TELEMETRY ANOMALY DETECTION

DETEKCE ANOMÁLIÍ V SATELITNÍ TELEMETRII

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. VOJTĚCH ORAVA**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ KAŠPÁREK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2026

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

166536

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Orava Vojtěch, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Satelite telemetry anomaly detection**
Category:	Artificial Intelligence
Academic year:	2025/26

Assignment:

1. Study and describe the possibilities of using machine learning methods and neural networks for anomaly detection in satellite telemetry with a focus on the possibility of deployment as an on-board solution for satellites (Autoencoders, LSTM, graph NN).
2. Propose an overall approach, sub-methods and neural network architecture to detect anomalies in satellite telemetry. Consider single and multi-channel telemetry information.
3. Find or prepare a suitable dataset and its annotation.
4. Implement the proposed solution using appropriate libraries. Consider use on edge (performance limited) devices.
5. Compare the implemented solution using “ESA Anomaly Detection Benchmark” or “NASA Anomaly Detection Dataset SMAP & MSL”.

Literature:

- Amita Kapoor, Antonio Gulli, Sujit Pal, Francois Chollet, Deep Learning with TensorFlow and Keras, Packt Publishing, 2022
- Kotowski, Krzysztof, et al. "European Space Agency Benchmark for Anomaly Detection in Satellite Telemetry." arXiv preprint arXiv:2406.17826, 2024.
- De Canio, G., et al. ESA Anomaly Dataset. 1.0, European Space Agency, 25 June 2024, doi:10.5281/zenodo.12528696.

Requirements for the semestral defence:
Items 1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Kašpárek Tomáš, Ing., Ph.D.**
Consultant:	Javorka Martin, Bc.
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2025
Submission deadline:	20.5.2026
Approval date:	7.11.2025

Abstract

This thesis addresses anomaly detection in satellite telemetry, where anomalies may signal impending system failures with potentially catastrophic consequences. The work investigates the use of artificial intelligence methods for this task. Multiple detection models were implemented and evaluated, including long short-term memory (LSTM) networks, autoencoders, residual networks (ResNet), and convolutional neural networks, using both forecasting-based and reconstruction-based approaches. Model performance was assessed using several metrics, with PR-AUC (area under the precision–recall curve) as the primary criterion. The experiments were conducted on a synthetic telemetry dataset generated by a custom data generator developed as part of this work. The best results were achieved by an LSTM-based autoencoder model, reaching a PR AUC of 0.946. To evaluate deployment feasibility, inference performance was tested on embedded platforms (AMD ZCU104 and NVIDIA Jetson Orin Nano), simulating onboard satellite computing constraints. In addition, real telemetry data from the ESA anomaly detection benchmark were used to assess performance in practical scenarios. Finally, an Anomaly Exploration and Report System (AERS) was developed to complete the detection pipeline, providing an interface for domain experts to implement custom anomaly explanation modules.

Abstrakt

Tato diplomová práce se zabývá detekcí anomálií v satelitní telemetrii, kde mohou anomálie signalizovat hrozící selhání systému s potenciálně katastrofickými následky. Práce zkoumá využití metod umělé inteligence pro tento úkol. Bylo implementováno a vyhodnoceno několik detekčních modelů, včetně sítí s architekturou LSTM, autoenkodérů, reziduálních sítí (ResNet) a konvolučních neuronových sítí, a to s využitím přístupů založených jak na predikci, tak na rekonstrukci. Výkon modelů byl hodnocen pomocí několika metrik, přičemž primárním kritériem byla hodnota PR-AUC (plocha pod křivkou precision-recall). Experimenty byly provedeny na syntetickém datasetu telemetrie vygenerovaném vlastním generátorem dat vyvinutým v rámci této práce. Nejlepších výsledků bylo dosaženo modelem autoenkodéru založeným na LSTM, který dosáhl hodnoty PR-AUC 0,946. Pro vyhodnocení proveditelnosti nasazení byl výkon inferenčního modelu testován na vestavěných platformách (AMD ZCU104 a NVIDIA Jetson Orin Nano), které simulovaly výpočetní omezení na palubě satelitu. Kromě toho byla k posouzení výkonu v praktických scénářích použita reálná telemetrická data z benchmarku detekce anomálií ESA. Nakonec byl k doplnění celého detekčního procesu vyvinut systém pro analýzu anomálií (AERS), který poskytuje rozhraní k implementaci vlastních modulů pro odborníky z dané oblasti.

Keywords

satellite telemetry, anomaly detection, neural networks, TensorFlow, machine learning, forecasting-based detection, reconstruction-based detection, synthetic data generator, AMD ZCU104, NVIDIA Jetson Orin Nano, anomaly analysis, ESA ADB

Klíčová slova

satelitní telemetrie, detekce anomálií, neuronové sítě, TensorFlow, strojové učení, předpovídací a rekonstrukční detekční metody, generátor syntetických dat, AMD ZCU104, NVIDIA Jetson Orin Nano, analýza anomálií, ESA ADB

Reference

ORAVA, Vojtěch. *Satellite telemetry anomaly detection.* Brno, 2026. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Kašpárek, Ph.D.

Rozšířený abstrakt

Lidstvo vyslalo během posledních sedmdesáti let do vesmíru velké množství družic a satelitů, které plní různé úkoly – od pozorování Země přes průzkum jiných planet a měsíců až po meziplanetární mise. Bez ohledu na typ mise jsou tyto systémy vybaveny rozsáhlou sadou senzorů a řídicích jednotek. Provoz v nehostinném prostředí vesmíru klade vysoké nároky na jejich spolehlivost. Stav satelitu lze monitorovat pomocí telemetrických dat, například napětí, proudů, teplot či zátěže jednotlivých subsystémů. Tato data mohou indikovat vznikající závadu nebo jinou nežádoucí anomálii. Čím dříve je problém detekován, tím je vyšší šance na dostatečnou reakci, která může zabránit katastrofě v podobě ztráty kontroly nad satelitem nebo ztráty kontaktu s ním. Jelikož je vyslání každé družice do vesmíru nákladné, je žádoucí, aby k takovým událostem docházelo co nejméně.

Diplomová práce se zabývá detekcí anomálií v satelitní telemetrii s využití detektorů postavených na neuronových sítích. Existující přístupy k detekci anomálií lze obecně rozdělit na statistické metody, klasické metody strojového učení a metody založené na hlubokém učení. Je také důležité určit typ dat, ve kterých anomálie detekujeme: závislá a nezávislá na čase. Data nezávislá na čase je možno shlukovat do skupin a určovat tak anomálie – body ležící mimo shluky. Časově závislá data, mezi něž patří i satelitní telemetrie, protože se vlastně jedná o hodnoty nějakých signálů v čase, je pak nutno analyzovat jinak. Mezi hlavní přístupy k detekci anomálií v takovýchto datech patří (a v této práci se používají): předpovídací přístup (anglicky *forecasting)* a rekonstrukční přístup (anglicky *reconstruction*). Předpovídací metoda predikuje následujících N kroků signálu, které přímo navazují na vstupní sekvenci a porovnává je se skutečnými naměřenými hodnotami. Míra odchylky mezi predikcí a reálným signálem je poté využita jako anomální skóre chyby – výrazně vyšší chyba predikce indikuje přítomnost anomálie. Rekonstrukční metoda zná vzor bezchybného chování signálu a snaží se jej rekonstruovat. Rekonstruovaný signál pak porovnává s reálným a v případě vysoké odchylky signálů je detekována anomálie, protože anomální vzorky signálu nebyly přítomny v trénovací sadě.

Reálné existující detektory anomálií v satelitní telemetrii používají oba přístupy. Systémy se liší také v tom, kde je telemetrický signál zpracováván: na Zemi nebo přímo na palubě satelitu. To pak určuje jak velký a výkonný model je možno použít. Na výkonných počítačích na Zemi se používají k detekci např. transformery, zatímco palubní detektory jsou často postaveny na sítích LSTM (long-short term memory) nebo konvolučních sítích.

Bylo implementováno a experimentálně ověřeno několik architektur neuronových sítí, konkrétně LSTM (Long Short-Term Memory), autoenkodéry, reziduální sítě (ResNet) a konvoluční neuronové sítě. Výsledky byly porovnány také s referenčním detektorem Telemanom vyvinutým společností NASA. Pro experimenty byly využity otevřené datasety (NASA SMAP & MSL, OPS-SAT, ESA AD), přičemž část testování proběhla na podmnožině datasetu ESA AD v rámci Kaggle výzvy. Jelikož dostupné datasety často obsahují anonymizované či normalizované kanály bez detailní dokumentace, byl v rámci práce navržen a implementován generátor syntetických telemetrických dat.

Dataset vytvořený tímto generátorem byl následně použit k otestování vytvořených modelů. Bylo dosaženo výsledku 0,946 v metrice PR AUC (plocha pod křivkou precision-recall) pro nejlepší model postavený na architektuře LSTM + autoenkodér. Podobného skóre dosáhl i klasický autoenkodér. U obou modelů se jednalo o předpovídací verze. Telemanom dosáhl na tomto datasetu výsledku PR AUC 0,768.

Významným aspektem práce je rovněž analýza možností nasazení detekce přímo na palubě satelitu. Byla provedena měření rychlosti, velikosti modelů a energetické náročnosti na platformách Raspberry Pi 4B, AMD ZCU104 a NVIDIA Jetson Orin Nano, které simulují

omezené výpočetní zdroje palubních systémů. Nejrychlejší modely dokázaly zpracovat 1000 vstupních sekvencí o délce 250 vzorků z 20 telemetrických kanálů v řádu jednotek sekund. Velikost modelů se pohybovala v řádu jednotek až desítek megabajtů. Naměřená spotřeba na Raspberry Pi 4B byla 2 – 4 W, u ostatních platforem nepřesáhla 14 W. Dále byly zkoumány možnosti optimalizace pomocí prořezávání (pruning) a kvantizace modelů.

Na základě experimentů byla navržena metodika konstrukce detektoru anomálií pro konkrétní satelit: každý takový satelit se liší množstvím a skladbou senzorů a je nutné na to brát zřetel, protože nelze vytvořit jeden univerzální detektor anomálií, který by byl použitelný na všech satelitech a družicích.

Závěrečným přínosem práce je návrh a implementace modulárního systému pro analýzu detekovaných anomálií a generování reportů (Anomaly Exploration Report System – AERS). Tento systém poskytuje rozhraní pro tvorbu specializovaných modulů pro analýzu a umožňuje zpracování výstupů různých detektorů do strukturovaných zpráv (reportů) určených pro expertní analýzu.

Satellite telemetry anomaly detection

Declaration

I hereby declare that this Diploma's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Kašpárek, Ph.D. Artificial intelligence (AI) and large language model (LLM) tools were utilized for text editing and coding support. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Orava
May 15, 2026

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Kašpárek Ph.D. for his support, feedback and provided consultations. I also thank Zaitra s.r.o. (Bc. Martin Javorka and Ing. Daniel Kyselica) for consultations and for the opportunity to access specific space-ready hardware.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Contents

Chapter 1

Introduction

Humankind has sent many satellites into space in the last 70 years. These satellites serve different purposes, but each of them faces the rough conditions of the space. That implies a problem of anomalies that are occurring and preventing satellites from their work. This thesis focuses on anomaly detection in spacecraft telemetry using deep learning methods.

In the context of a satellite, telemetry refers to the continuous stream of signals and time-series data gathered from its onboard sensors. These data enable engineers to remotely monitor the satellite's health and operational status. Much like a car that transmits telemetry such as throttle position, speed, or engine temperature, a satellite provides telemetry data including battery voltage, attitude (orientation), and communication signal strength.

The theoretical background of anomalies, along with recent approaches to anomaly detection (covering both time-series-based and nontemporal prediction methods) is presented in Chapter 2. Anomaly detection techniques can be categorized into three main groups: statistical methods, traditional machine learning approaches, and deep learning (DL) models. Within the DL domain, architectures such as LSTM networks and autoencoders are among the most commonly applied methods for anomaly detection. Proper evaluation of these models requires suitable metrics, particularly precision, recall, and the area under the precision–recall curve (PR AUC).

Chapter 3 focuses specifically on anomaly detection in satellite telemetry. It reviews state-of-the-art solutions and summarizes the datasets commonly used in this domain. Although four satellite telemetry datasets are widely adopted in research, they have several significant limitations. To overcome these problems, a synthetic dataset generator comprising 20 different channels (using unique 13 sensors) was developed. The proposed generator is capable of producing both sensor-based anomalies and environment-based anomalies.

The core of this work lies in Chapter 4, where the design and implementation of anomaly detectors are detailed. Besides that, optimization techniques (quantization, neural network prunning) used for neural networks are explained here.

The detectors were evaluated on commercially available hardware as well as on an edge platforms represented by a Raspberry Pi 4B, AMD ZCU104 and NVIDIA Jetson Orin Nano. Detailed information on the experiments is provided in Chapter 5.

Although the most widely used satellite telemetry datasets have notable limitations, as discussed earlier, one of them – the ESA anomaly dataset – was selected for real mission–oriented evaluation. Chapter 6 describes the data preprocessing steps applied to this dataset, the modifications required to adapt the models accordingly, and a comparison of the achieved results with those reported in the associated Kaggle competition.

An anomaly exploration and report system (AERS) is introduced in Chapter 7 to complete the telemetry processing pipeline. This pipeline begins with telemetry generation, continues through anomaly detection, and concludes with a modular framework for anomaly exploration and report generation. This chapter presents the final stage of the overall workflow, extending the pipeline with an exploration layer rather than integrating all components into a single unified system.

Chapter 2

Anomaly detection

An anomaly is a deviation from the normal pattern or general distribution of the data [40]. Such deviations are called outliers for some types of data. This section provides an overview of what anomalies look like, the main types of anomalies, and the approaches used to detect them. An illustrative example of an outlier in two-dimensional data is shown in Figure 2.1.

Section 2.1 introduces the formal definition of an anomaly, outlines common anomaly types, and summarizes evaluation metrics used in the anomaly detection domain. Section 2.2 then describes the main methodological approaches to anomaly detection, including statistical techniques, classical machine learning methods, and deep learning–based approaches.

Finally, Sections 2.3 and 2.4 distinguish between anomaly detection in non-temporal data and in time-series data, highlighting the specific characteristics and challenges associated with each category.

2.1 Anomalies

As stated above, an anomaly occurs when the data deviates from the general (usual) distribution of a data. The anomaly could be represented as a single observation or as multiple observations. Anomalies usually take only a small portion of the data in a dataset. There are several types of anomalies that we distinguish [40].

2.1.1 Categorization of data

Data in which we want to detect outliers or anomalies could have different types. That includes video, audio, text, images, and so on. It's also important to keep in mind that different data have different temporal characteristics. Time-based data will play a key role in this document because they follow the narrative of space telemetry. More details are in Section 2.4. There are non time-based data as well. Approaches that are used for those data are described in Section 2.3.

2.1.2 Anomaly types

According to papers [4, 7], anomalies can be classified into 3 main categories:

- **Point anomalies** – individual data point does not follow expected pattern or distribution. This point significantly differs from the majority of observations. Temporal information for one data point is not important. This type of anomaly is considered as the simplest one.

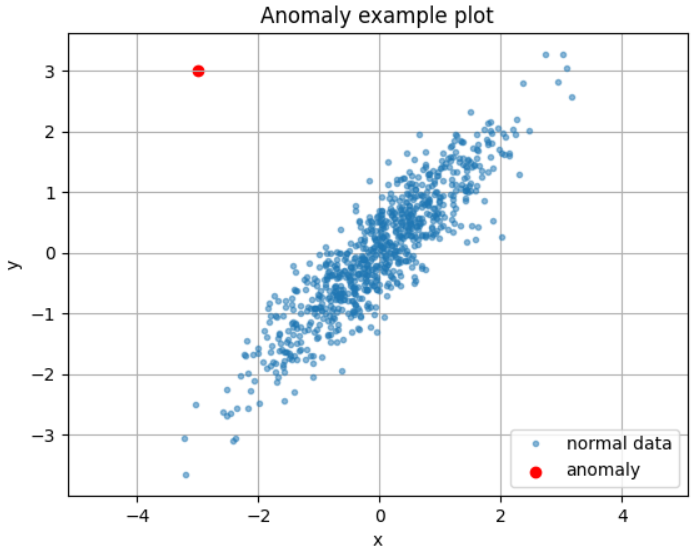


Figure 2.1: Example plot of anomaly or outlier (red dot) in normal data (blue dots) for 2D data.

- **Contextual anomalies** – one data point or sequence of data points is observed as an anomaly only in some specific context. Out of this context, point or points are taken as normal data.
- **Collective (group) anomalies** – occur when related data instances are anomalous in the relationship to the rest of the data. Individually, they can look normal.

This distribution to categories applies mainly for the time-based data. Its visual representation can be seen in Figure 2.2. Data that can be clustered, could have anomalies (deviating points from the distribution) of types called **Cluster-based** and **Correlation Anomalies** [12].

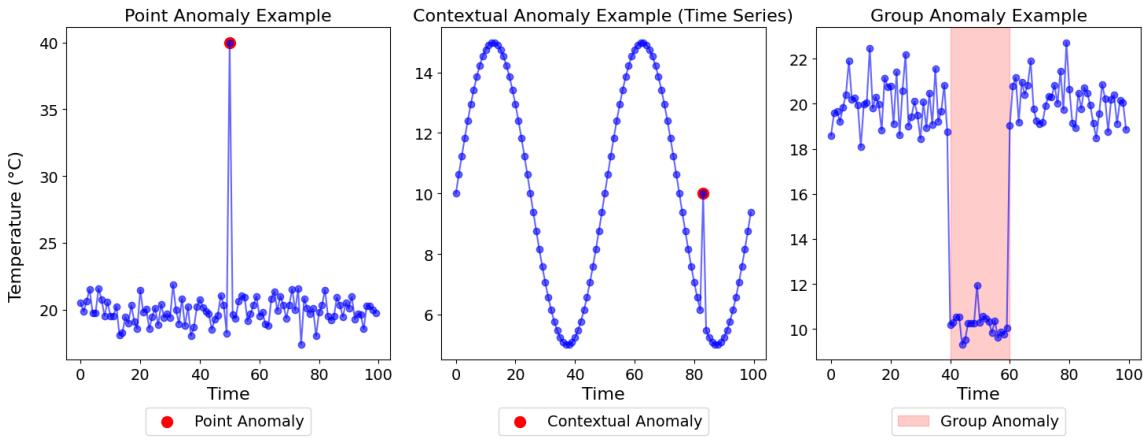


Figure 2.2: Examples of anomaly types (point, contextual and group). Y-axis displays temperature values, X-axis represents time.

2.1.3 Anomaly detection challenges

There are several challenges that must be taken into account when working with anomaly detection (AD) systems [12]. The nature of anomalies, their rarity, and the lack of clearly labeled data often make the development and evaluation of such systems significantly more complex than traditional classification tasks. Furthermore, anomalies may evolve over time, requiring detection methods that are adaptable and robust. The most important challenges include:

- **Dimensionality** – datasets for anomaly detection are often highly dimensional, making it difficult for AD methods to maintain precision. One of the most common approaches to deal with this issue is dimensionality reduction (for example using Principal Component Analysis).
- **Data privacy** – there are fields of human life or industry where privacy is very important (banks, healthcare, cybersecurity, strategical systems). Working with data from those fields presents new challenges (data anonymization).
- **Noise in data** – real-world data frequently contains measurement errors, inconsistencies, or random fluctuations. Noise can obscure the boundary between normal and abnormal behavior, reducing detection accuracy. AD system must be robust to overcome those problems.
- **Class imbalancy** – in most anomaly detection problems, normal samples outnumber anomalies, which can lead to biased models that favor the majority class.
- **Sparsity** – datasets sometime have missing values or they are incomplete. Common solutions of those problems include data imputation, matrix factorization, or algorithms that can natively handle missing data.

2.1.4 Evaluation metrics

To correctly determine the success rate of the anomaly detector (or in general any detector), we need the correct evaluation metrics. These metrics can be categorized into 3 groups: performance metrics, environmental metrics, and hardware complexity metrics.

Performance metrics describe detector accuracy and robustness. It is common to use terms True Positive (TP) for correctly detected anomaly, False Positive (FP) for incorrectly detected anomaly, True Negative (TN) for non-anomaly datapoint detected as non-anomaly, and False Negative (FN) for anomaly datapoint detected as non-anomaly (see Figure 2.3 for illustration). Some of the most used performance metrics in the field of anomaly detection are [6, 40]:

- **Precision** – measures the fraction of true anomalies among all detected anomalies:

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

  Low precision indicates many false alarms.

- **Recall** – measures the fraction of correctly detected anomalies among all anomalies:

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

  Low recall means high miss of true anomalies.

- **F1-Score** – balance both metrics above, called harmonic mean:

$$\text{F1-Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

  Low F1-Score denotes poor balance between precision and recall.

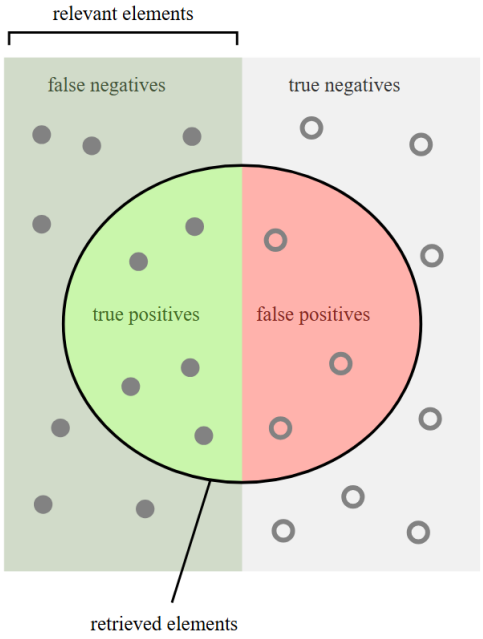


Figure 2.3: Example showing the distribution of TP, FP, FN and FP data points. Source: [38].

- **Average Precision** – evaluates precision across all possible thresholds using precision P_n and recall R_n at threshold n:

$$AP = \sum_n (R_n - R_{n-1})\, P_n$$

- **AU-ROC (Area under the Receiver Operating Characteristic curve)** – represents the model's ability to distinguish between classes across different decision thresholds t. A higher AU-ROC value indicates better overall discrimination performance.
- **PR AUC (Area under the Precision–Recall curve)** – captures the trade-off between precision and recall across different thresholds. This metric is particularly informative for imbalanced datasets, such as anomaly detection tasks, where anomalies are rare. A higher PR AUC value indicates that the model maintains high precision while achieving high recall. An example of this curve for an anomaly detection model is shown in Figure 2.4.

Environmental metrics take into account environmental factors related to detection models, such as temperature, radiation and the vacuum environment. **Hardware complexity metrics** evaluate metrics that are related to inference time or memory usage. Performance and hardware complexity metrics are the most important for on-board deployment.

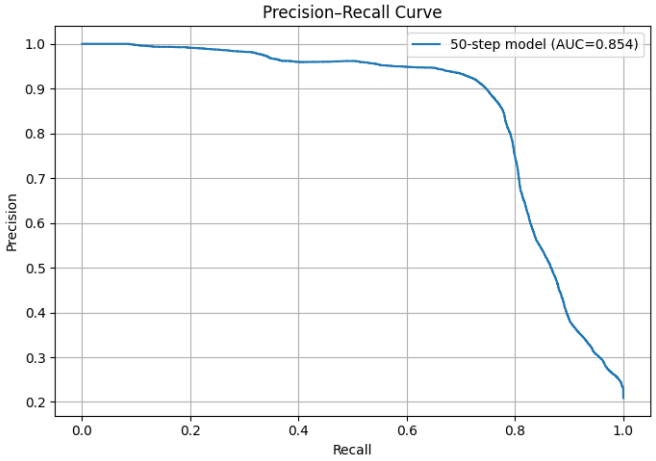


Figure 2.4: Example of Precision – Recall curve for different values of threshold t. Area under this curve is called PR AUC. It is the most used performance metric in this thesis for models comparison.

2.2 Anomaly detection approaches

"Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior. These non-conforming patterns are often referred to as anomalies, outliers, discordant observations, exceptions, aberrations, surprises, peculiarities or contaminants in different application domains" [4].

Today, anomaly detection is used in a wide variety of fields, including the Internet of Things (IoT), medical applications, transportation, communication, security, or industrial automation. Spacecraft, which produce and process a lot of data, also need to have some kind of anomaly detector.

Anomalies can be detected using statistical methods, which don't use machine learning at all or traditional approaches to machine learning like clustering methods. With the development of deep neural networks, they have begun to be used to detect anomalies [7, 12]. The following subsections describe all three approaches, including their pros and cons.

2.2.1 Statistical approaches

The simplest way to distinguish an outlier is to set a **threshold**. All values above/below the specified threshold are considered anomalies. Such methods are often referred to as *statistical*, or *threshold-based* approaches. They rely on fundamental statistical measures such as the mean, standard deviation, minimum, and maximum to describe the expected range of normal behavior.

These approaches are conceptually simple, computationally efficient and easy to interpret, making them suitable for **univariate data** and real-time monitoring scenarios such as sensor readings, server metrics, or financial transactions. However, their performance degrades when dealing with **multivariate**, **non-linear**, or **non-stationary** data, where relationships between variables are complex and cannot be captured by simple statistical boundaries [7]. Table 2.1 summarizes the advantages and disadvantages of these approaches.

Advantages	• No training needed • Simplest approach
Disadvantages	• Struggle with high-dimensional or unstructured data • Inappropriate for non-linear and non-stationary data

Table 2.1: Advantages and disadvantages of statistical approaches to anomaly detection [4, 7].

2.2.2 Traditional machine learning approaches

More advanced approaches use machine learning for detection. The word *traditional* here means that these approaches do not use deep neural networks. The survey [4] divides machine learning anomaly detection methods into 6 groups (classification based, nearest neighbor based, clustering based, statistical, information theoretic and spectral).

Classification based methods are working with the assumption that "a classifier that can distinguish between normal and anomalous classes can be learnt in the given feature space". These methods include Bayesian Networks or a detection system based on Support Vector Machines. **Nearest neighbor based methods** assume that outliers or anomalies occur out of the dense neighborhoods. Examples of these techniques could be k-nearest neighbor distance. **Clustering systems** sort data to clusters and assume that data points which don't belong to any cluster are anomalies. Fejjari et al. describe the same or similar methods for anomaly detection [7]. The advantages and disadvantages of traditional machine learning approaches are summarized in Table 2.2.

Advantages	• Fast training and inference • Lower computational and memory requirements • Easier to interpret and debug • Perform well on small or structured datasets
Disadvantages	• Struggle with high-dimensional or unstructured data • Limited capacity to capture complex nonlinear relationships • Require manual feature engineering

Table 2.2: Advantages and disadvantages of traditional machine learning approaches to anomaly detection [4, 7, 12].

2.2.3 Deep neural network approaches

Newer techniques utilize the power and capabilities of deep neural networks. Huang et al. divide deep learning methods for anomaly detection into 3 categories [12]: **based on**

reconstruction, **based on prediction** and **hybrid approach**. The advantages and disadvantages of the deep learning approaches are summarized in Table 2.3.

Prediction-based (or sometimes refered as forecasting-based) approaches use a model to predict a point or points given the point or a window of recent values. Whether the predicted points are anomalous or not, is determined by comparison with actual values. Most of these methods use a sliding window to predict one point in the future. Prediction-based approaches are a good choice in use cases where we need to capture patterns and trends in data [12, 40].

Models that are marked as **reconstruction-based** are designed to predict future values as well. In this case, anomalies are detected using prediction/reconstruction errors (high error means anomaly). They learn the distribution of normal data and, once trained, attempt to reconstruct the incoming data [12, 40].

Hybrid models include strengths from different approaches to enhance the accuracy of anomaly detection. Specific examples of neural networks belonging to all categories are described in Section 2.3 for non-temporal data and in Section 2.4 for time series.

The survey [40] presents one more category of anomaly detection: **Representation-based models**. These models learn representations in the latent space. That is very useful when we have large data with high complexity (noise, seasonality, non-stationarity).

Advantages	• Ability to learn complex and nonlinear patterns • Scalable to large and high-dimensional datasets • Automatically extracts relevant features from raw data • Can incorporate temporal, spatial, or contextual information • Improves performance with more data (transfer learning, fine-tuning)
Disadvantages	• High computational and memory requirements • Requires large labeled datasets for effective training • Risk of overfitting, especially on small or noisy data • Limited interpretability and explainability • Sensitive to hyperparameter settings and initialization

Table 2.3: Advantages and disadvantages of deep neural network approaches to anomaly detection [7, 12].

2.3 Non-temporal based anomaly detection

This section discusses anomaly detection techniques applied in domains where data samples do not possess an inherent temporal sequence or dependency. In these scenarios, each data instance is treated independently, without consideration of chronological order or

progression. Non-temporal methods are commonly used in various applications. These applications include medical field, (cyber)security matters, or frauds in the finance sector.

In medical applications, anomaly detectors can support early diagnosis by identifying deviations in imaging data. Anomaly detection is also a fundamental part of cybersecurity. It is used for the detection of unauthorized access, identification of malicious activities, or uncovering suspicious network traffic. Models based on autoencoders or Recurrent Neural Networks are used in this field [12].

A standalone sector where anomaly detection finds its place is finance. Common tasks for non-temporal detectors are fraud transactions or unusual trading activities, because these often involve independent events. Anomaly detectors for the financial sector are built on techniques such as SVM or Random Forest [12]. In general, non-temporal based anomaly detectors often rely on statistics, clustering or reconstruction-based approaches.

2.4 Time-series based anomaly detection

A time series is a sequence of data points arranged in chronological order. In most cases, time series have the shape of observations recorded over some time. We can divide time series into *univariate* and *multivariate*. Univariate time series (UTS) are based on single variable, which is changing over time. Example could be measuring temperature value in some location over time. Equation 2.1 represents an univariate time series X with values $x_1, x_2, \ldots, x_t$ at timestamps $T = \{1, 2, \ldots, t\}$ [40].

$$X = (x_1, x_2, \ldots, x_t) \tag{2.1}$$

Multivariate time series (MTS) are made up of multiple variables dependent on time. Each variable is influenced by its past values and other variables (also called dimensions). This influence is dictated by the correlation between the variables. An example of a multivariate time series is measuring humidity and/or other physical quantities together with temperature over time. Equation 2.2 reffers to an multivariate time series of dimension $D = \{1, 2, \ldots, d\}$ at timestamps $T = \{1, 2, \ldots, t\}$ [40].

$$X = (X_1, X_2, \ldots, X_t) = ((x_1^1, x_1^2, \ldots, x_1^d), (x_2^1, x_2^2, \ldots, x_2^d), \ldots, (x_t^1, x_t^2, \ldots, x_t^d)) \tag{2.2}$$

Both traditional and deep learning methods could be used for anomaly detection in time series. In more complex data like satellite telemetry, it's convinient to use deep learning approach. As mentioned in Subsection 2.2.3, main approaches are **forecasting-based**, **reconstruction-based**, **representation-based** and **hybrid models**.

2.4.1 Forecasting-based models

According to studies from authors [3, 35, 40], there are multiple model architectures that can be used for time series forecasting, some of them are: Reccurent Neural Networks (RNN), Convolutional Neural Networks (CNN), Graph Neural Networks (GNN) and Transformers. Forecasting-based models are good choice, when high inference speed is needed, but they struggle with rapidly changing timeseries data (which may result in lower precision compared to reconstruction-based models).

Reccurent Neural Networks

Reccurent Neural Networks process input sequences and have a Reccurent Unit for capturing dependencies in data across time. That is achieved using shared weight matrices over sequences. The Reccurent Unit consists of `input`, `output`, and `hidden state` with feedback connection. The current hidden state depends on the currect input x_t (t denotes timestep) as well as the previous hidden state h_{t-1} (see Equation 2.3). The hidden state can be considered as a summary of past inputs and states. It can be computed by Equation 2.4 where A is weight matrix between states, B is weight matrix between inputs and states, and b_i is bias for state i. The actual output y_t is then described by Equation 2.5, having C as a weight matrix between states and the outputs and b_y representing the bias for the output y [11, 9].

$$h_t = f(h_{t-1}, x_t) \tag{2.3}$$

$$h_t = \tanh(A * h_{t-1} + B * x_t + b_i) \tag{2.4}$$

$$y_t = C * h_t + b_y \tag{2.5}$$

RNNs parameters are updated via Backpropagation Through Time. There are multiple types of RNNs (visualization in Figure 2.5), each used for different task:

- One-to-One RNN – single input, single output. Used for classification tasks where no sequences are involved.
- One-to-Many RNN – single input, multiple outputs. Usable in cases where single input generated sequence of multiple outputs.
- Many-to-One RNN – multiple inputs, single output. Applications focused on sentiment analysis use this approach to determine if sequence of input words is either positive/negative/neutral.
- Many-to-May RNN – multiple inputs, multiples outputs. Utilized in machine language translation.

The RNN architecture described above is called Vanilla RNN. There are more modern and better variation of RNN – Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU) and Bidirectional RNN. **LSTMs** are formed by input gate, forget gate, output gate and memory cell. Each part learns different aspects of the input sequence. LSTMs were introduced to overcome problem of *vanishing gradient* – in long sequences during backpropagation, gradient can become so small that its magnitude is almost zero - it vanishes. More modern version of LSTM is **GRU**. This architecture simplifies LSTM by joining input and forget gate to update gate. **Bidirectional RNNs** just process input in both directions: forward (left to right) and backward (right to left).

Anomaly detection based on forecasting using RNNs was and is being used for both UTS and MTS. Zamanzadeh et al. describe some recent anomaly detectors based on LSTM [40]. There are detectors that use stacks of LSTM units. More modern approaches use LSTM or GRU with some other methods like Gaussian Mixture Models or Support Vector Machines. Another survey by Wang et al. [35] portraits summary of RNN based methods which use dynamic thresholding or new metric – Local Trend Inconsistency.

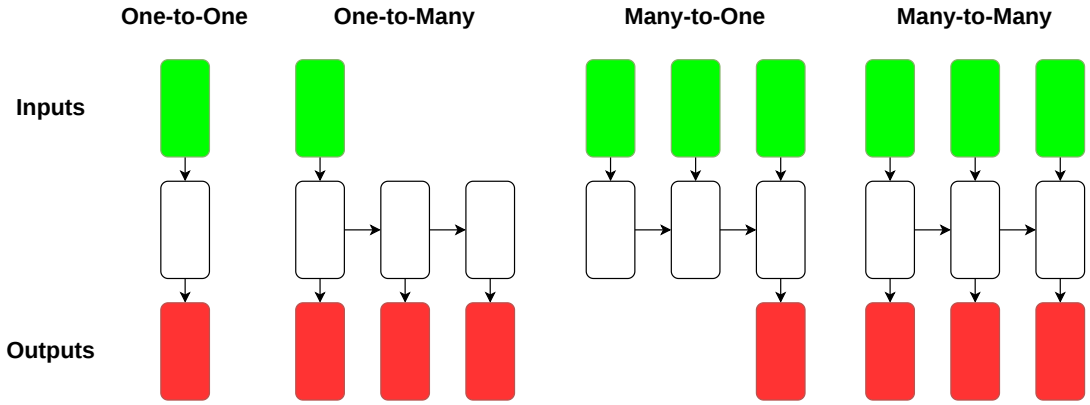


Figure 2.5: Types of RNNs. Red are inputs of a network, green are outputs. White boxes represents cell or hidden states. Image was created based on the [9].

Convolutional Neural Networks

The next forecasting-based approach to anomaly detection employs **Convolutional Neural Networks** (CNNs). These architectures are composed of activation functions and a sequence of convolutional, pooling, and fully connected layers. While CNNs are predominantly associated with computer vision tasks such as image classification or object detection, their core operation (the *convolution*) also makes them suitable for sequential data. An example of the convolution operation is illustrated in Figure 2.6.

Pooling layers act as dimensionality-reduction components by aggregating values from neighboring regions, typically using either maximum or average functions [27]. For time-series applications, the most relevant variant is the **1D convolution**. By stacking multiple 1D convolutional layers, a CNN can effectively capture both short-term and long-term temporal dependencies, forming a hierarchical representation of the signal. This makes CNN-based forecasting models computationally efficient and well-suited for processing long sequences, which is particularly advantageous in anomaly detection for telemetry data.

Anomaly detection surveys [35, 40] describe the DeepAnt model as a representative of the CNN-based forecasting model. This unsupervised anomaly detection model can detect small deviations in time series patterns. It can be used for both UTS and MTS. Another model worth mentioning is Temporal Convolutional Network (TCN), which can better handle timeseries data. Both mentioned models only consider one-dimensional time series. Unlike the aforementioned models, TimesNet maps one-dimensional time series into a two-dimensional structure, enabling more effective modeling of multi-periodic temporal dependencies.

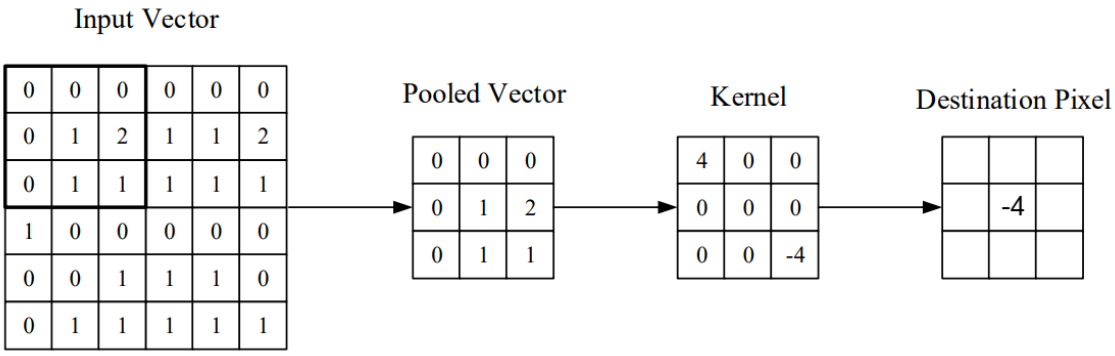


Figure 2.6: Convolution operation demonstration. Input matrix is convolved with kernel, resulting in the new value of one pixel. This demonstration is mostly used on image data. Taken and edited from [27].

Graph Neural Networks

Data in which we want to detect anomalies can be seen as graphs in some cases. For this kind of data arrangement, **Graph Neural Networks** (GNN) can be used. The graph G is formally defined as $G = (V, E)$, where V are the vertices and E are the edges connecting these vertices. Graphs can have different types and scales [41]:

- **Directed/Undirected Graphs** – edges between vertices could be directed or undirected (the same edge for both directions).
- **Homogenous/Heterogenous Graphs** – nodes and edges have the same types in homogenous graphs, different in heterogenous.
- **Static/Dynamic Graphs** – graphs in which topology or input features vary over time are called dynamic graphs.

For GNNs, it is important to clearly define the type of task being performed. Tasks focused on individual nodes—such as classification, clustering, or categorization—are referred to as *node-level tasks*. When the goal is to make predictions about edges, for example determining edge types or inferring whether a connection between two nodes should exist, these are known as *edge-level tasks*. Finally, tasks that consider the entire graph structure and aim to learn a representation of the whole graph are categorized as *graph-level tasks*.

GNN model is learning using pairwise message passing between nodes of the graph. Nodes iteratively update their inner representation (embedding) by the information they receive from neighbors. Equation 2.6 shows the update formula for the embedding h_u of node u at time or step t. As mentioned earlier, update is done with aggregated information from neighbors $N(u)$ of node u as shown in Equation 2.7.

$$h_u^{t+1} = \text{Update}(h_u^t, m_{N(u)}^t) \tag{2.6}$$

$$m_{N(u)}^t = \text{Aggregate}\left(h_i^t, \forall i \in N(u)\right) \tag{2.7}$$

For MTS vertices of graph often represent dimensions of input data and edges display correlations learned from data. There are multiple GNN architectures, some of them are: **Graph Convolutional Networks (GCN)** where a convolutional operator is used for the aggregation of neighbors or **Graph Attention Networks (GAT)** [40].

Transformers

The final category of forecasting-based models considered in this thesis are **Transformers** [34, 40]. Transformers are deep learning architectures built around the *self-attention* mechanism, complemented by positional encodings that supply information about the order of the input sequence. Their standard architecture consists of two major components: an encoder and a decoder. Both parts rely on a multi-head self-attention mechanism, defined by

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.8}$$

where Q is the matrix of queries, K the matrix of keys, V the matrix of values, and d_k the dimensionality of the key vectors used for scaling. The term *multi-head* refers to applying this attention operation multiple times in parallel, allowing the model to learn different types of relationships within the sequence.

Transformers are particularly effective at capturing long-range dependencies, a capability that traditional recurrent architectures often struggle with. This makes them well suited not only for natural language processing tasks but also for anomaly detection in time-series data, where understanding long-term patterns is essential. The core structure of the Transformer model is illustrated in Figure 2.7.

Transformers typically require large training datasets and substantial computational resources to perform well. In many satellite telemetry applications, the available data is relatively limited, and onboard hardware often has strict memory and processing constraints. For these reasons, while theoretically powerful, Transformers may not always offer a practical advantage over smaller and more efficient architectures such as LSTMs or CNN-based models described above.

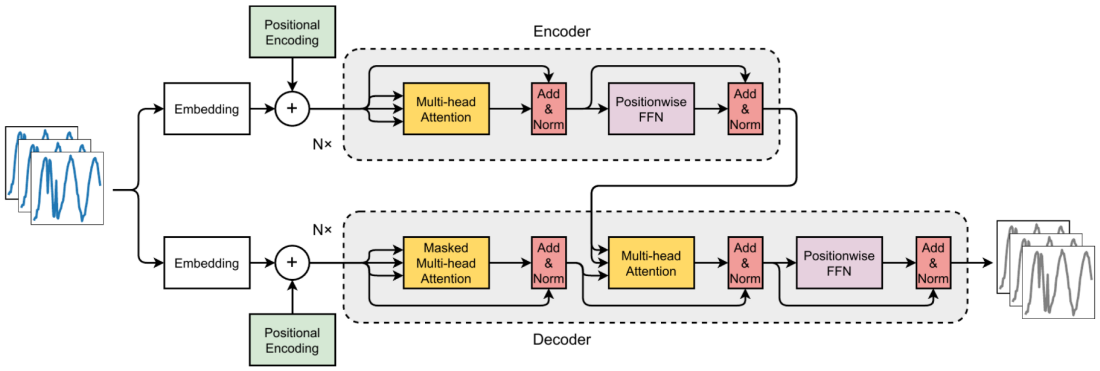


Figure 2.7: Transformer architecture used for anomaly detection with encoder and decoder parts and multi-head attention modules. Source: [40].

2.4.2 Reconstruction-based models

Another important class of models for anomaly detection are reconstruction-based models. Unlike forecasting-based models, which predict future values using only past observations and therefore do not have access to the current data point, reconstruction-based models use the complete current input to reconstruct the expected signal. Anomalies are then identified by comparing the reconstructed output to the actual observed data: a high reconstruction error indicates a deviation from the learned normal patterns. This approach allows reconstruction-based models to achieve higher detection accuracy, although it can introduce a small delay since the full current data must be available before detection. Consequently, these models are particularly suitable for scenarios where precision is critical and minor delays can be tolerated [3, 35, 40]. Common architectures used for reconstruction-based anomaly detection include Autoencoders (AE), Variational Autoencoders (VAE), Generative Adversarial Networks (GAN), and, in some cases, Transformers adapted for reconstruction tasks.

Autoencoders

Autoencoders are neural networks that can reduce the dimensionality of the input by compression into lower representation and then reconstruct the input back to original dimensionality. Autoencoders constist of two parts: *encoder* and *decoder*. The encoder learns important features of input data and reduces the dimensionality. Decoder then reconstructs original data from compressed form. Latent space, the smallest layer of the network, where compressed representation is stored, is called bottleneck. The typical architecture of the autoencoder can be seen in Figure 2.8 in part a). The goal of the autoencoder is to achieve accurate reconstruction and to minimise reconstruction loss [8].

The ability of the autoencoder to reduce dimensionality is used in the domain of anomaly detection as a preprocessing step. Wang et al. describe specific applications of autoencoders with Gaussian Mixture Models (DAGMM) or autoencoders with convolutional reccurent networks (MSCRED) [35].

Variational Autoencoders

Another reconstruction-based architecture relies on **Variational Autoencoders (VAE)**. VAE works similarly to autoencoders, but inputs are not encoded as single points but as a distribution. This distribution is learned by a neural network $q_\phi(Z|X)$ where ϕ are networks parameters. The network transforms d dimensional input X to a k dimensional latent representation Z, where $k < d$. Loss function of VAE is based on Kullback-Leibler divergence which measures similarity of two distributions.

Specific example of VAE usage is LSTM-VAE, which is a variation of VAE with LSTM as feed-forward network. Another methods are CVAE (Conditional VAE), STORNs or HVAE (Hierarchical VAE) [40]. The VAE architecture is shown in Figure 2.8 in part b).

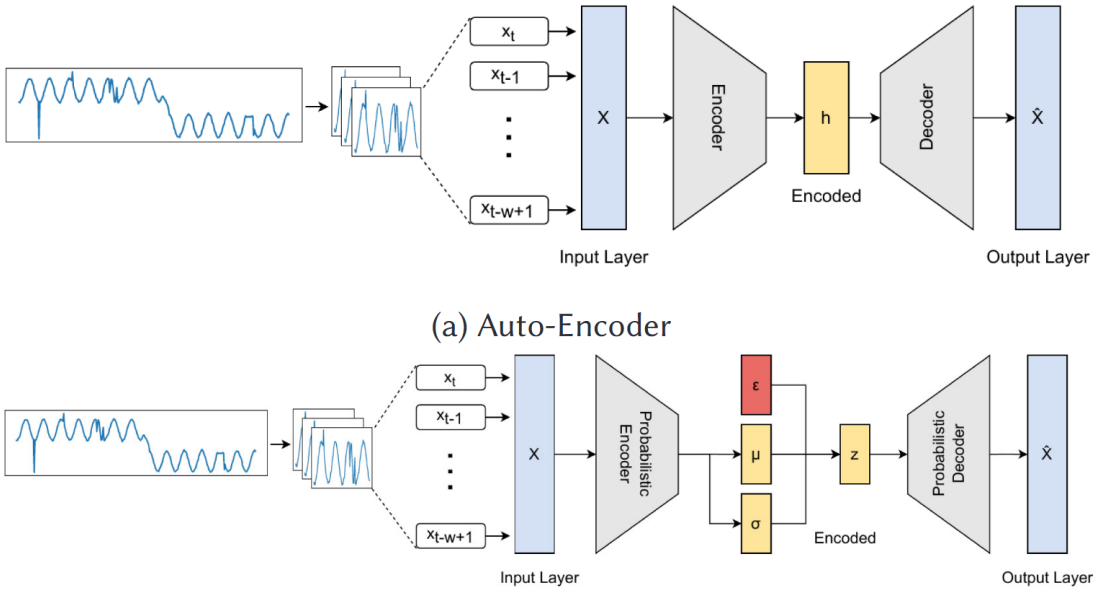


Figure 2.8: Details of the autoencoder (a) and variational autoencoder (b) architectures. Both encoder and decoder parts can be seen in the image, connected with the narrow part called bottleneck. Variational autoncoder have latent state z and latent distribution represented by variables μ (mean), σ (standard deviation) and ϵ (random noise sampled from a standard normal distribution) in the bottleneck. Source: [40].

Generative Adversarial Network

The next architecture which can be utilized for anomaly detection is **Generative Adversarial Network** (GAN). GAN includes 2 main parts: *generator* and *discriminator*. The principle of learning is based on the minimax model, where we want GAN to generate the sample from distribution $p(x)$ with minimal error and at the same time we want GAN to maximize the discrimination between correct and false samples from distribution $p(x)$. Training GAN can be difficult and requires a trade-off between two mentioned optimizations. For anomaly detection, GAN can be combined with autoencoders (system DAEMON) or with LSTM (system MAD-GAN) [3, 35].

Transformers

Earlier mentioned **Transformers** are so versatile, that they can be combined with other approaches to form reconstruction-based anomaly detection model. Examples of these are TranAD, TransAnomaly or MEMTO [35].

2.4.3 Representation-based models

As mentioned in Subsection 2.2.3, representation-based models learn compact feature representations in the latent space, which are then used for anomaly detection. This

approach allows models to effectively process high-dimensional and complex data by capturing meaningful patterns and structures. In most cases, representation learning is applied in unsupervised or self-supervised settings, where labeled anomalous samples are scarce or unavailable [40].

Two of the most common neural architectures for representation learning are Convolutional Neural Networks and Transformers. CNNs are particularly effective for spatially structured data, such as images or multivariate time series with local correlations. Transformers, on the other hand, have shown strong performance on sequential and high-dimensional data because of their ability to model long-range dependencies through self-attention mechanisms.

2.4.4 Hybrid models

As the name suggests, hybrid models are a combination of the above concepts. Different techniques are combined by using the joint objective function. Hybrid models combine the strengths of multiple approaches. In time series anomaly detection, this often involves integrating deep representation learning with classical anomaly detection methods or combining different approaches of neural architectures mentioned above.

The survey [40] provides a guide or manual for choosing the correct model or architecture for the intended purpose. Insights into this guide are presented in Table 2.4.

Use case description	**Suitable architectures**
Multidimensional data + complex dependencies	GNN (GCN, GAT)
Sequential data with long-term temporal dependencies	LSTM, GRU
Large datasets requiring scalability and efficiency	Transformers
Noise in anomaly detection	AE, VAE
High-Frequency data + detailed temporal patterns	CNN, Temporal CNN
Data with evolving patterns and multimodal distributions	combinations of GNN, VAE, LSTM
Generalisation across diverse datasets	Representation based methods

Table 2.4: Architecture selection guideline for anomaly detection tasks based on the information from survey [40].

Chapter 3

Satellites

This thesis aims to provide an anomaly detection system for satellites in space. The use case like this is different from regular anomaly detection in industry or other sectors. This chapter sums up important information about satellites and their computational and memory resources and about their telemetry in Section 3.1. This work is not the first to approach the detection of satelite telemetry anomalies, so existing state-of-the-art systems are described in Section 3.2.

Neural network–based models, including anomaly detectors, require data for training. A variety of existing datasets with different characteristics are available; these are discussed in Section 3.3. During the work on this thesis, it became apparent that a synthetic dataset is needed because of disadvantages of publicly available datasets. Consequently, a satellite telemetry data generator was developed. Further details about this synthetic dataset and its generation process are provided in Section 3.4.

3.1 Anomaly detection on satellites

In this thesis, the term satellite refers to a human-made object (spacecraft) sent to space. Each satellite has different application, most common types are: communication satellites, navigation satellites, astronomical satellites and satellites for Earth observation. Another classification of satellites can be done based on theirs position/orbits: low Earth orbit, medium Earth orbit, geostationary orbit, geostationary transfer orbit. A special type of low Earth orbit is the Sun-synchronous orbit (SSO), which is commonly used for Earth observation missions because it allows the satellite to pass over the same location at approximately the same local solar time. Each orbit has a range of altitudes above Earth's surface [32].

Satellites are equipped with sensors, cameras, and other technological instruments for data collection, part of which consists of telemetry. Satellite telemetry is further explored in Subsection 3.1.1. Anomalies in the telemetry could lead to a fault behavior of a satellite or even to satellite destruction. Typical anomalies are examined in Subsection 3.1.2.

3.1.1 Satellite telemetry

The satellite telemetry is multi-variate timeseries (each time serie is called channel). Compared with other telemetries in which we can perform anomaly detection, satellite telemetry has specific features and complexities. Mainly, available data has relatively few anomalies (compared to the whole dataset size) in most cases. Other challenges are [15]:

- high dimensionality,
- high volume,
- complex dependencies between channels,
- variety of types of channels,
- complex characteristics,
- errors in data due to space environment.

To provide a better understanding of how telemetry data look like, we can look at the OPS-SAT anomaly dataset [29] (more details about this and other space-related anomaly datasets are provided in Section 3.3). The OPS-SAT dataset contains 9 source telemetry channels – 3 magnetometer (device for measuring magnetic field) channels and 6 photo diode channels. Figure 3.1 visualize records of telemetry in 2 channels.

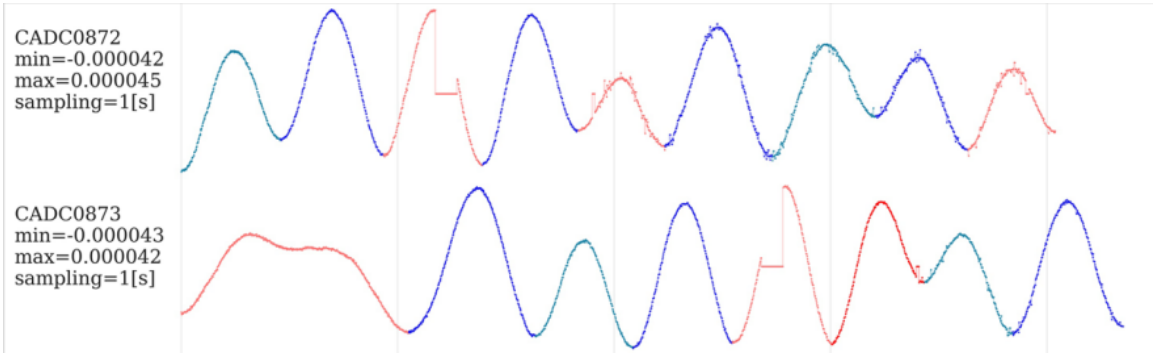


Figure 3.1: Example data from OPS-SAT dataset for 2 channels over a time. Both CADC0872 and CADC0873 are magnetometers telemetry channels. Anomalous segments are highlighted red. Source: [29].

We can see that signal flow in non-anomalous sector follows a periodical pattern. We can detect anomaly by using a model which predicts next "standard" value and comparing it with real telemetry signal. Difference in those values might signalize an anomaly.

3.1.2 Typical anomalies in telemetry

In general, anomalies in satellite telemetry can be categorized into two main types. The first category includes **sensor-related anomalies**, such as hardware malfunctions, measurement drift, calibration errors, or permanent damage to onboard instruments. The second category consists of **environmental anomalies**, which arise from external conditions affecting the satellite, including radiation spikes, temperature extremes, space weather events, or unexpected interactions with the surrounding environment [2, 15, 29]. This was kept in mind when creating the sythetic dataset and its generator, further described in Section 3.4.

3.2 State-of-the-art satellite telemetry anomaly detectors

This part contains a description of state-of-the-art anomaly detection systems and models in domain of satellite telemetry. Most of the papers on this topic use the NASA SMAP /

MSL dataset [2] as a comparative dataset. A more detailed view of this particular dataset is provided in Section 3.3. The survey [7] yields a comparison of anomaly detection models in terms of performance metrics. The NASA datasets mentioned above have been used for measurements. Table 3.1 shows the results of this study. The authors state the precision and recall of different approaches to anomaly detection. Most of these architectures have been described in Section 2.4.

We can see that **GCN** has good results in both precision and recall together with Transformer-based anomaly detection (denoted as TranAD). Similarly, **attention based temporal CNN** and **LSTM + GAN** performed well. It's interesting that simple model like LSTM was able to achieve relatively high precision (87.5 %). The same survey also provides information about computational times for some architectures. The fastest training time (71.45 s) was measured for LSTM architecture with transfer learning.

Approach	**Precision (%)**	**Recall (%)**	**F1-score**
LSTM	87.5	80.0	0.836
LSTM + transfer learning	79.4	83.5	0.814
Attention TCN	94.74	94.17	0.945
ResNet	80.76	58.87	0.681
FCN	69.31	58.09	0.632
ELMs	76.4	77.9	0.771
SVM	84.1	85.7	0.849
Bi-Transformer	85.40	72.4	0.784
TranAD	85.40	99.99	0.921
GCN	94.86	99.59	0.972
GNU	89.2	89.6	0.894
GRU + VAE	81.41	94.46	0.875
LSTM + GAN	90.28	92.45	0.914
LSTM + multi-scale strategy	87.37	86.54	0.870

Table 3.1: Comparison of different approaches performance metrics. Precision and recall values are from [7], F1-score was calculated using standard formula. NASA SMAP & MSL have been used as dataset.

The authors of survey [19] used NASA SMAP and MSL datasets as well. They selected 5 channels for experiments (A-2, F-5, G-7, P-1 and P-3). They tested multiple anomaly detection approaches: machine learning, deep learning, thresholding, ARIMA – Auto-Regresive Integrated Moving Average and hybrid methods. AUC-ROC was selected as a performance metric. Table 3.2 displays the results from this survey. Each model is ranked for each channel and the best results are marked in bold. The last column shows the average rank for the models. According to these data, the best model is **LSTM AE** with average rank 2.2, followed by the hybrid model (consisting of LSTM, AE and CNN) with average rank 2.6. The simple LSTM model did not perform well and was ranked last in this survey.

Models	A-2		F-5		G-7		P-1		P-3		Average Rank
	AUC ROC	Rank	AUC ROC	Rank	AUC ROC	Rank	AUC ROC	Rank	AUC ROC	Rank	
ARIMA	0.492	10	0.690	10	0.677	9	0.513	3	0.723	10	8.4
KNN	0.808	4	0.860	5	0.846	3	0.510	4	0.987	4	4.0
Isolation Forest	0.717	7	0.776	8	0.808	8	0.499	6	0.940	8	7.4
Dense AE	0.761	6	0.857	6	0.840	4	0.654	2	0.981	5	4.6
Convolutional AE	0.917	2	0.863	3	0.813	6	0.318	10	0.990	3	4.8
Variational AE	0.794	5	0.770	9	0.830	5	0.432	8	0.973	6	6.6
LSTM AE	**0.934**	1	0.903	2	0.924	2	0.503	5	**0.996**	1	**2.2**
LSTM Predictor	0.372	11	0.682	11	0.337	10	0.421	9	0.599	11	10.4
Transformer	0.810	3	0.863	3	0.813	6	0.318	10	0.917	9	6.2
GAN	0.660	9	0.836	7	0.300	11	0.454	7	0.964	7	8.2
Hybrid	0.661	8	**0.907**	1	**0.925**	1	**0.665**	1	0.995	2	2.6

Table 3.2: Performance of models on different channels from NASA SMAP & MSL datasets. Metric is AUC-ROC and each channels has models ranked by this metric [19].

Study by Murphy et al. [18] presents various anomaly detection models on edge boards. This time, the authors selected channel F-5 from NASA SMAP dataset for experiments. The models included in the experiments were:

- Autoencoder (AE),
- Convolutional Autoencoder (CNNAE),
- Variational Autoencoder (VAE),
- Long-Short Term Memory Autoencoder (LSTMAE),
- Hybrid Autoencoder (encoder part as CNN, decoder part as LSTM),
- Transformer,
- Isolation Forest.

The authors have chosen a set of commercially available egde boards: Raspberry Pi 4, Google Coral, NVIDIA Jetson Nano, and two Myriad-based systems (Ubotica CogniSAT-XE1 and Ubotica CogniSAT-XE2). CPU Intel i9 of 14th generation was used to obtain the baseline performance. The very useful information from this study is, that the mentioned edge devices had a small power consumption. Table 3.3 shows the measured relative power consumptions (total power consumption minus idle power consumption) as well as ROC area scores and F1-scores.

Model Architecture	Device	ROC Area	F1 Score	RPC (W)
Basic AE	CPU	0.857	0.730	–
	XE1	0.857	0.730	0.92
	XE2	0.784	0.669	0.96
	Jetson Nano	0.857	0.848	1.33
	Raspberry Pi	0.754	0.730	1.02
	Google Coral	0.857	0.730	1.12
CNN AE	CPU	0.860	0.545	–
	XE1	0.861	0.545	1.04
	XE2	0.842	0.545	1.04
	Jetson Nano	0.860	0.545	2.77
	Raspberry Pi	0.860	0.545	1.56
	Google Coral	0.860	0.545	0.87
LSTM AE	CPU	0.903	0.892	–
	XE1	–	–	–
	XE2	0.679	0.646	1.46
	Jetson Nano	0.885	0.848	2.32
	Raspberry Pi	0.885	0.848	2.15
	Google Coral	–	–	–
Hybrid AE	CPU	0.907	0.810	–
	XE1	–	–	–
	XE2	0.729	0.465	1.13
	Jetson Nano	0.585	0.413	1.97
	Raspberry Pi	0.585	0.413	1.47
	Google Coral	–	–	–
Transformer	CPU	0.863	0.764	–
	XE1	–	–	–
	XE2	0.754	0.703	1.03
	Jetson Nano	0.754	0.703	2.15
	Raspberry Pi	0.754	0.703	1.85
	Google Coral	–	–	–
VAE	CPU	0.770	0.779	–
	XE1	–	–	–
	XE2	–	–	–
	Jetson Nano	0.729	0.733	1.44
	Raspberry Pi	0.729	0.733	1.02
	Google Coral	–	–	–
Isolation Forest	CPU	0.776	0.514	–
	XE1	–	–	–
	XE2	–	–	–
	Jetson Nano	0.776	0.514	3.74
	Raspberry Pi	0.776	0.514	3.92
	Google Coral	–	–	–

Table 3.3: ROC Area, F1-score and relative power consumption (RPC) for different (edge) devices and various anomaly detection architectures. Maximum power draw was set to 5W. CPU in this table is Intel i9 14th generation. Missing values (marked as "-") means that given model architecture couldn't be compiled and run on given device. This was caused by using unsupported layers in most cases. Source: [18].

NASA researchers demonstrated the effectiveness of LSTM networks for anomaly detection in their work [13]. Their approach, referred to as **Telemanom**, leverages sequence-to-sequence LSTM models to forecast future telemetry values and identify anomalies through deviations between predicted and actual measurements. Using this method, the authors reported a prediction error of 5.9 % on the combined SMAP and MSL datasets – two widely used benchmarks in satellite telemetry anomaly detection (see Section 3.3 for further information about these datasets).

A significant advantage of the Telemanom system [13] is the availability of its complete source code, released publicly on GitHub. For the purposes of this thesis, the Telemanom implementation was adopted as one of the reference models. Only minor adjustments were made to adapt the codebase to the specific experimental setup and synthetic dataset needs. The adapted model served as a baseline for comparison in Section 4.3, providing a useful reference point for evaluating the performance of newly designed architectures.

3.3 Anomaly datasets

Datasets play a very important role in the domain of neural networks. Each task has its appropriate datasets. There are four main and most used datasets for satellite telemetry described in Subsection 3.3.1. Because these datasets offer only small or no documentation about individual channels (values are either normalized and/or channel names are anonymized), it was decided to create a synthetic dataset (see 3.4). Values of telemetry sensors in this dataset can be generated with full control over the generating process, which allows to model real-life problems and anomalies. Thanks to this, it's possible to analyze the data and explain what probably happed and why anomaly occured. That is something we can't always tell for the publicly available datasets.

3.3.1 Existing datasets

The most popular and most used datasets with expert-labeled anomalies of satelite telemetry are the following:

- NASA Soil Moisture Active Passive (SMAP) dataset [2],
- NASA Mars Science Laboratory (MSL) dataset [2],
- ESA Anomalies Dataset (ESA AD) [15],
- OPS-SAT dataset [29].

NASA SMAP dataset

NASA SMAP dataset has 55 data channels, each channel is associated with 1 out of 12 modules. Modules or sensors include power system voltages and currents, reaction wheel speeds, a gyroscope, or thermal sensors. Raw data have a form of multivariate time-series. The anomalies in the data set were annotated and labeled by space engineers.

NASA MSL dataset

The NASA MSL dataset includes 27 channels and 27 modules. Modules are devices from rover engineering, because data in this dataset come from Curiosity rover Mars mission from

2012. Telemetry represents thermal subsystems, power bus readings, or sensor voltages and currents. The form of data is multivariate time-series.

ESA AD

ESA AD channels were anonymized before publication, so we cannot tell which sensors or modules they represent. Data come from three ESA missions and represents several years of telemetry recordings. Dataset is provided with ESA Anomaly detection benchmark including various anomaly detectors. Although this dataset is not ideal, to show the performance of detection models not only on synthetic data but also on real data, ESA AD was utilized. A brief overview of anomaly detection using the data from this dataset is provided in Chapter 6.

OPS-SAT dataset

The OPS-SAT dataset contains 9 named channels and data in this dataset are in the form of univariate timeseries. As mentioned in Section 3.1, there are 3 magnetometer channels and 6 photo diodes channels. Raw data varies in length or sampling frequency. The anomalies have been manually labeled. The authors provide this dataset with extracted features like mean, standard deviation, and variance.

3.4 Synthetic anomaly dataset

The need for a synthetic dataset was raised from unavailability or incompleteness of channels documentation in publicly available and well known datasets mentioned above. A program was created for generating data from virtual satellite with multiple sensors. The signals it generates have been designed based on the information found on the Internet and inspired by signal flows in above mentioned datasets. This allows to generate data for any timespan with option to plan anomalies or create them randomly. It's possible to create real-time scenarios which may occur on actual satellite.

The mentioned artificial satellite and its sensors are further described in Figure 3.2. Sensors behavior was designed with awareness of real conditions like Sun light and dark periods and satellite movement around Earth. The program can generate 2 main classes of anomalies: **sensors failures** and **environment anomalies** (solar storm, Earth eclipse, reaction wheel imbalance, software fault/memory leak, telescope overload).

The dataset generation program can generate datasets based on multiple input options. The dataset can be obtaned for a given number of **days**, with variational **sampling frequency**. **Number of cycles** around Earth in one day can be specified as well as **starting seed** for random generators, ensuring replication of results. Anomalies can be generated (**the number of them can be specified**) or created based on **the anomaly plan file** – the user can specify which, when and how long anomalies he wants to include; code 3.4 shows an example of an anomaly plan file (CSV format). The user can also set **how much environment-based anomalies** should be generated.
The generator can inject 6 variations of anomaly signal change:

- **signal spike** – injects a spike into the signal,
- **stuck of signal at value X** – temporarily sets the sensor signal values to a given value at a given index for a given duration,

- **signal drift** – temporarily injects a drift into the sensor signal,
- **noise increase** – injects increased noise into the sensor signal,
- **signal froze** – temporarily freezes the signal (at current value),
- **rapid decrease of signal value** – temporarily decrease the signal values by a multiplicative factor.

```
sensor,type,start_idx,duration,kwargs
gyro_x_deg_s,increased_noise,8754,2021,{'extra_std': 0.1}
gyro_x_deg_s,spike,60992,4923,{'magnitude': 1.0121141512376388}
comms_snr_dB,decreased,40511,3990,{'factor': 0.5}
cpu_temp_C,frozen,12343,1564,{}
cpu_temp_C,drift,47489,621,{'drift_per_step': 0.01}
telescope_sensor_V,stuck_at,7489,4621,{'value': 1.3}
```

Table 3.4: Example structure of CSV file for the anomaly configuration plan, defining the specific parameters for injecting faults into generated telemetry signals. The plan incorporates all previously mentioned signal distortion types. While these particular anomalies are sensor-specific, environmental-based anomalies (which impact multiple sensors simultaneously) would share same starting indexes.

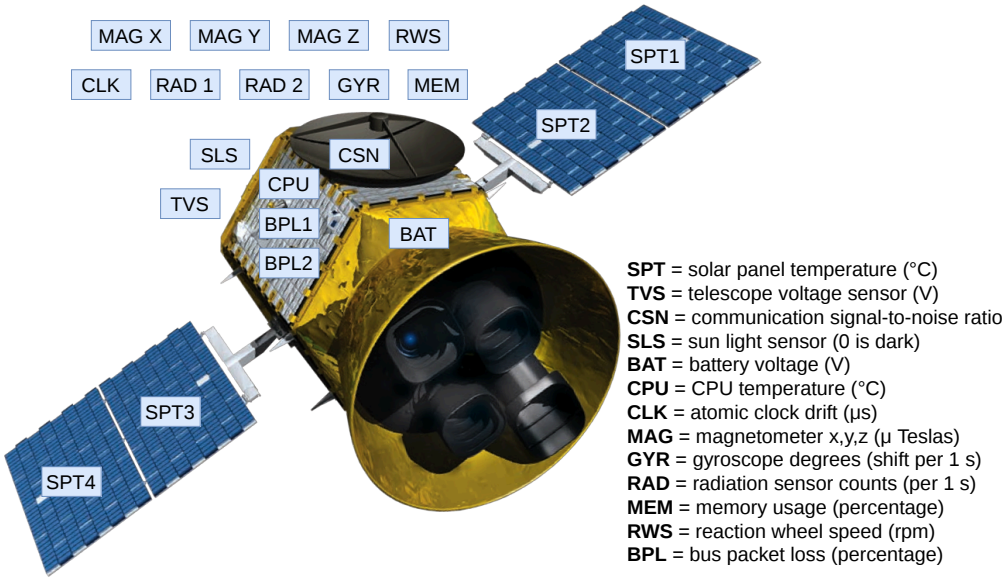


Figure 3.2: Visualization of a imaginary satellite with various sensors. Satellite serves as a model for telemetry generation. All used sensors (from which telemetry is generated) are displayed here with additional description.

Figures 3.3 and 3.4 illustrate how normal data and anomalies look in the generated dataset. The first image shows both sensor-specific and environment-based anomalies across telemetry from 4 solar panel tempretature sensors. All above mentioned types of injected faults/anomalies are visualized in the second image with labels. The dataset generator was made public and is available under the MIT license on Github [26]. The telemetry generator has a form of console application written in Python language. All information describing how to work with the dataset generator is listed in the attached `README.md` file in the Github repository.

Schefels et al. took a similar approach to telemetry generation in their work Synthetic satellite telemetry for machine learning [31]. They created a **closed-source** satellite telemetry generator with anomalies based on Python for the German Aerospace Center (DLR). Their program can generate various signals with different shapes, such as sinusoidal, square, flat and cosine. They also employed similar techniques for anomaly injection, such as adding noise, creating outliers or gaps, and shifting signals.

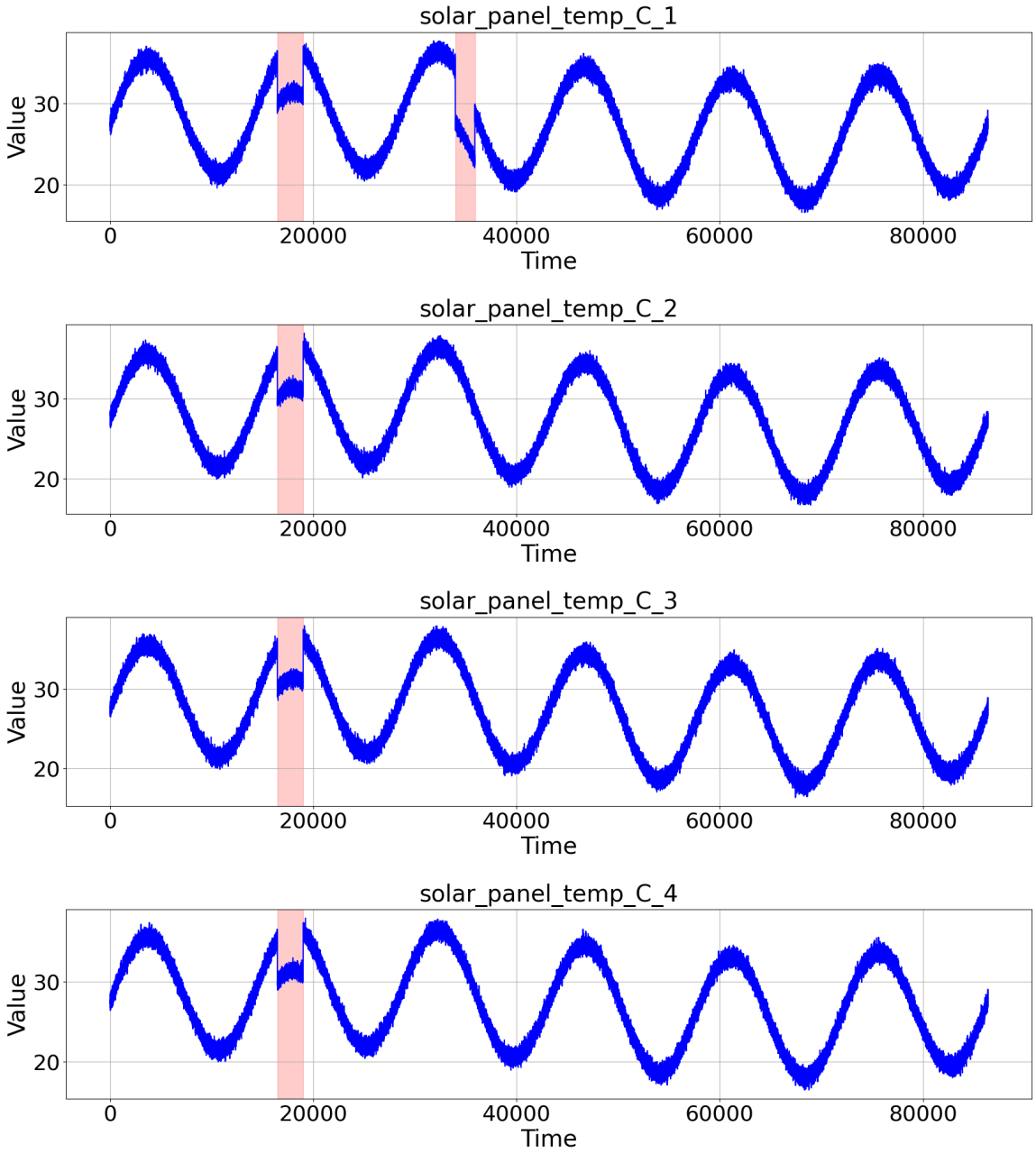


Figure 3.3: A set of data from test dataset used in Chapter 4 generated via dataset generator program. Red segments highlight anomalies. Each graph has a title describing which sensor telemetry it belongs to. All data have been sampled every 1 s for a day (86400 timestamps). Progress of four temperature sensors of solar panel are displayed here, with 1 environmental anomaly occuring (across all sensors) and 1 sensor anomaly at solar panel number 1.

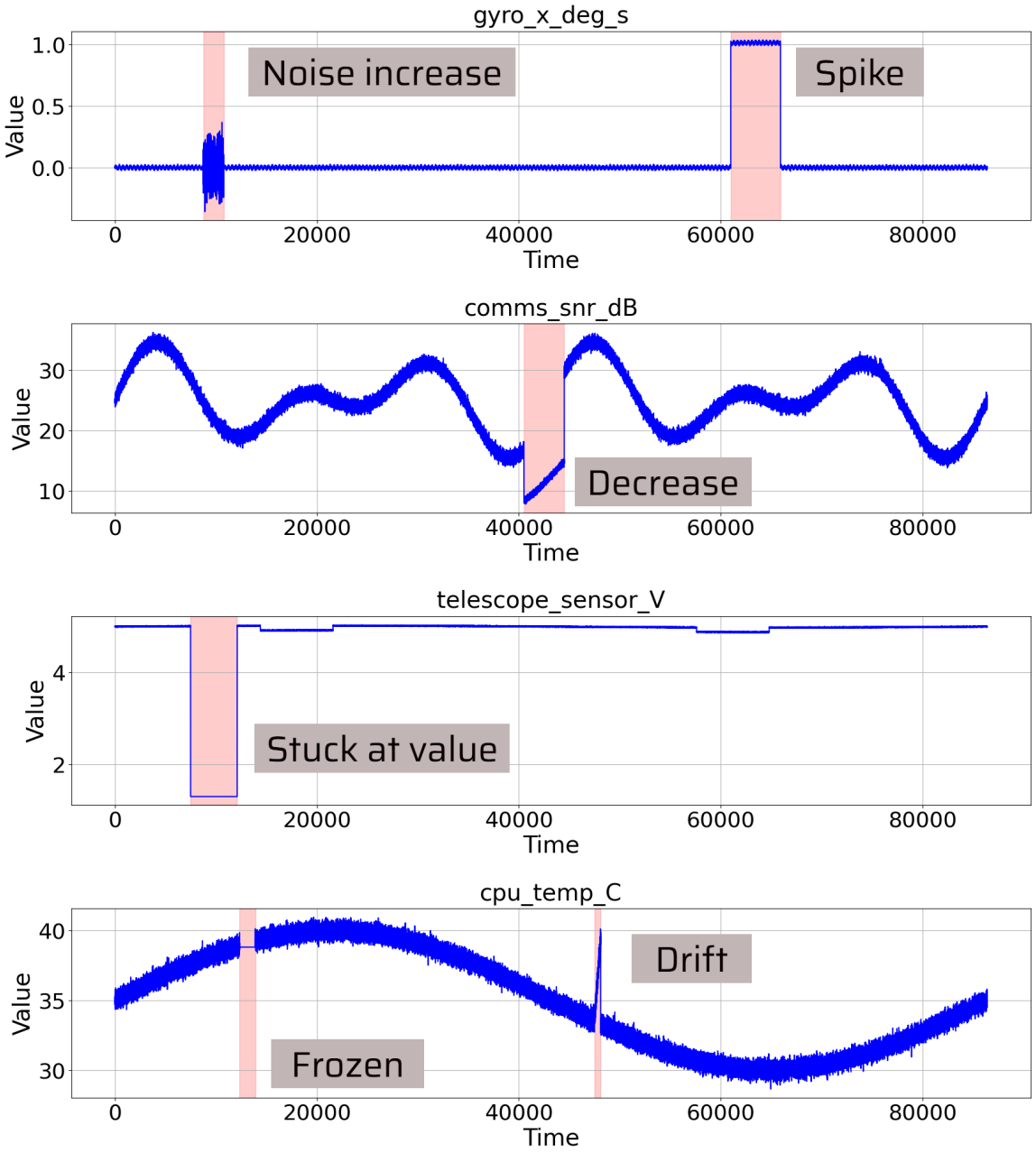


Figure 3.4: Telemetry signal of four different sensors. First one shows gyroscope x-axis data with two anomalies (*noise* and *spike*). The second signal represents signal-to-noise ratio of radio communication. Major signal *decrease* anomaly was generated here. Third graph displays telescope sensor voltage where anomaly *stuck at* 1.3 V happened. Last plot shows 2 types of anomalies (*freeze* of sensor and signal *drift*). These anomalies have been generated using anomaly plan presented in code in Table 3.4.

Chapter 4

Detector design and implementation

The design of a system plays a fundamental role in software development, as the implementation is built directly upon it. This chapter plays a core role in this thesis as it outlines all components and steps required to develop an effective anomaly detection system for satellite telemetry.

Section 4.1 introduces the software and hardware tools used throughout this thesis. It provides an overview of the key libraries, frameworks, and computational resources required for training, validating, and testing large neural networks.

Section 4.2 focuses on optimization techniques such as pruning and quantization, as well as the use of Pareto frontiers for multiobjective comparison. Section 4.3 then presents the search for an optimal anomaly detector architecture, describing the initial network prototypes and the advanced experiments performed on more complex models.

To evaluate real-world deployability, Section 4.4 reports tests conducted on a Raspberry Pi 4B, serving as a stand-in for an on-board space computing platform with limited resources. These sections include multiple visualizations and graphs to illustrate design decisions, constraints, and observations.

Finally, Section 4.5 provides a summary of the findings and offers recommendations for designing and deploying anomaly detection models in satellite telemetry applications.

4.1 Software and hardware tools

Both software and hardware tools and resources are described in the following section. Software tools include a brief survey of the programming libraries used, mostly specialized in machine learning and neural networks. There is also a short paragraph about training servers. The hardware section provides detailed information about the devices used for experimentation, including specifications, performance capabilities, and the rationale behind their selection. This includes microcontrollers, single-board computers, and edge devices suitable for running lightweight neural network models.

4.1.1 Software tools

Python was used as the primary programming language for the practical component of this thesis, alongside several machine learning libraries. Some experiments and demonstrations

scripts were developed and executed in the Jupyter[1] environment, which provides an interactive interface for incremental code execution. All neural network architectures were implemented using the TensorFlow[2] and Keras[3] frameworks. TensorFlow lightweight version called TensorFlow Lite was utilized for models deployment on edge computer board. Additional Python libraries used throughout the work include `numpy` for numerical computation and `pandas` for data manipulation. Visualization of results was performed using the `matplotlib` and `seaborn` plotting libraries.

The servers of the **Metacentrum**[4] service have been used for computationally intensive tasks such as neural networks training. The Metacentrum is the service of official National Grid Infrastructure (NGI). It is an activity of the CESNET association that operates and manages a distributed computing and data storage infrastructure. It provides researchers and students with access to high-performance computing (HPC), cloud computing, and large-scale data processing capabilities needed to solve complex, demanding tasks that would exceed the capacity of a single institution or common user PCs.

4.1.2 Hardware tools

The Metacentrum grid mentioned above provides a variety of GPUs for neural network training. Some of the most common are NVIDIA A40 or NVIDIA L40S. Among those, first demo versions of neural networks have been trained using NVIDIA MX940 and NVIDIA GTX1070 Ti in author PCs.

Satellites do not have such powerful hardware, so it was necessary to test models on real space boards or at least at hardware with limited computational power. For the experimental evaluation, some of the following hardware boards were used (selected devices are bold and are described in further detail together with tests in Chapter 5):

- Enclustra ZX2 + Mars ST3 (based on Zynq 7000 platform)
- Enclustra XU5 + Mercury+ ST1 (based on Zynq UltraScale+)
- **AMD ZCU104 (based on Zynq UltraScale+)**
- **NVIDIA Jetson Orin Nano**
- Trenz TE0950 (AMD Versal)
- **Raspberry Pi 4B 4GB RAM**
- SKAIDOCK + Xiphos Q8 (Zynq UltraScale+) – Engineering Model

4.2 Optimization techniques

Because edge devices need optimalized networks for efficient performance, this part is dedicated to optimization techniques. Specifically, it is about quantization and pruning. Both of these processes can be used independently or together to improve optimization in the sense of speeding up inference or model compression. Neural networks can also be accelerated

[1] https://jupyter.org
[2] https://www.tensorflow.org
[3] https://keras.io
[4] https://metavo.metacentrum.cz/cs/about/index.html

using proper hardware, such as NPU (neural processing unit) or other accelerators. In real-world applications, devices often perform these optimizations automatically through vendor-provided tools or runtime frameworks tailored to their specific hardware architecture.

Subsection 4.2.1 describes the quantization. It is a process of converting floating numbers in the neural network to integers for faster inference. The second optimization method, pruning, is explained in Subsection 4.2.2. Because optimization decision is often based on single objective, multi-objective technique called Pareto frontier is described in Subsection 4.2.3.

4.2.1 Neural network quantization

The most represented operation in neural networks is matrix-vector multiplication:

$$y = Wx + b$$

where W is the matrix of weights, x is the input value and b is bias. We want these operations to be computed as parallel as possible for better performance. Taking an array of input values and computing a dot product in parallel with weights using the formula 4.1 constructs an operation called MAC (multiple and accumulate). A_n is an accumulator for row n, $C_{n,m}$ is the result of the multiplication of input at position m with weight at position n. The illustration is provided in Figure 4.1 [20].

$$A_n = b_n + \sum_m C_{n,m} \tag{4.1}$$

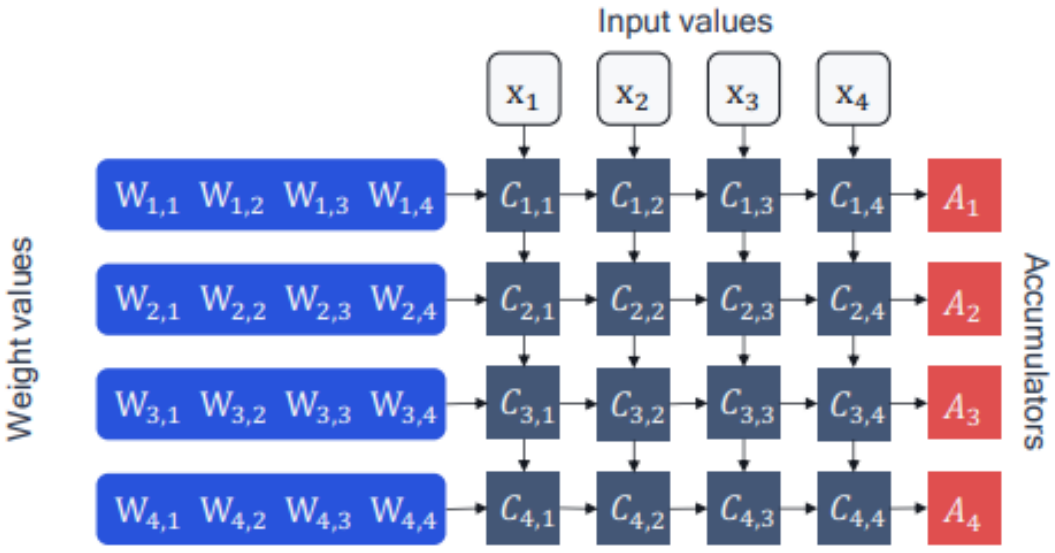


Figure 4.1: Example of parallel MAC operations in neural network [20].

To make these computations more efficient and faster, we can use integers instead of floating point numbers. This denotes the move from floating-point operations to fixed-point operations. This conversion is called **quantization**. The quantized integer value is an

approximation of the original floating-point value, which implies a loss of precision, but saves computational resources.

The most commonly used quantization method is called *uniform affine quantization*. It's defined by 3 parameters:

1. scale factor s – size (floating-point number) of the step of the quantizer,
2. zero-point z – represents real zero,
3. bit-width b – size of the output integer.

The values s and z represent parameters for mapping a floating point number to an integer. The size of the integer is defined by the parameter b. Quantization to the integer can be calculated using equation 4.2, where *round* is a rounding function to the closest integer, *clip* is clipping function. This function (equation 4.3) takes arguments a and b, which are the lower and upper bounds.

$$x_{int} = \text{clip}\Big(\text{round}\Big(\frac{x}{s}\Big) + z\,; a, b\Big) \tag{4.2}$$

$$\text{clip}(x; a, b) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq b, \\ b, & x > b. \end{cases} \tag{4.3}$$

The real value of the quantized input can be obtained via equation 4.4. The described computational process is the general asymmetric case of quantization. The symmetric version sets the zero point z to 0. For use of symmetric quantization, one has to choose between unsigned and signed variations. They differ in their cliping bound intervals: $(0; 2^b - 1)$ for unsigned integers; $(-2^{b-1}; 2^{b-1} - 1)$ for signed integers. Unsigned type is good fit for ReLU activations, signed form for distributions centered around zero [20].

$$x \approx \hat{x} = s\,(x_{int} - z) \tag{4.4}$$

Using the previous steps, we can get quantization function q (equation 4.5). The limits of the quantization grid are defined by values (q_{min}, q_{max}).

$$\hat{x} = q(x; s, z, b) = s\left[\text{clip}\left(\text{round}\left(\frac{x}{s}\right) + z; 0, 2^b - 1\right) - z\right] \tag{4.5}$$

Quantization has two main approaches: **Post-training quantization** and **Quantization-aware training**. Post-training quantization is applied to the model after it has been trained. The goal is to reduce the size of the model and accelerate the inference speed without the need for additional training. If the weight distribution is not gaussian-like, it might be difficult to get reasonable accuracy of the model after quantization. To overcome this issue, post-training quantization has to calculate appropriate quantization parameters (correct quantization range for each quantizer – q_{min} for minimal value and q_{max} for maximal one, see equation 4.5). This calculation involves methods to determine the trade-off between clipping and rounding error. Such methods are listed below [20, 36]:

- Min-max – q_{min} is set to minimum value of quantized tensor V, q_{max} has maximum value from tensor V,
- Mean Squared Error (MSE) – this method minimizes MSE between original and quantized tensor and tries to find corresponding q_{min} and q_{max},

- Cross entropy,
- Kullback-Leibler Divergence.

For summary, we can describe post-training quantization in these steps [36]:

1. Get fully trained neural network model.
2. Sometimes models have to be calibrated before quantization using a small calibration dataset.
3. Calculate the quantization parameters of the weights and activations.
4. Quantize model weights.
5. Quantize model activation values during inference and get model output using inverse quantization.

The other technique, called Quantization-aware training, integrates quantization process directly into training of neural network model. This approach solves the problem of quantization to low bits (such as 4-bit or less), which causes a large quantization error in post-training quantization. Higher accuracy is redeemed by the need for neural network training.

The main idea of quantization-aware training is that during the forward pass of computation graph of the quantized network, the weights and activations are quantized to lower precision (integers). Thanks to this, network's predictions can reflect what will happen at inference time. In the backward pass, gradients are computed as usual, ignoring quantization steps. This allows the model to update the weights with full precision. The whole process is shown in Figure 4.2.

The rounding operator in Equation 4.2, which derivation is either undefined (for numbers at boundary of two integers – like 1.5) or 0 (everywhere else), is problematic for gradient calculation. To avoid this problem, *straight-through estimator* (STE) is used. STE approximates the gradient of the rounding operator as shown in Equation 4.6.

$$\frac{\partial \,\mathrm{round}\,(y)}{\partial y} = 1 \tag{4.6}$$

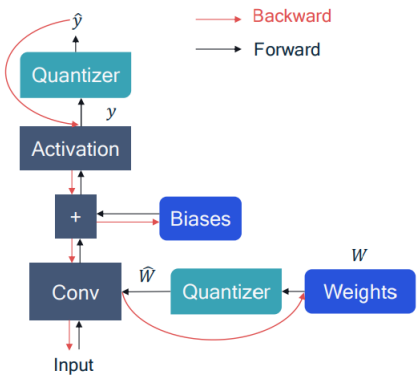


Figure 4.2: Computation of forward and backward pass in quantization-aware training using STE. Source: [20].

The approach for the neural network quantization has to be chosen based on the problem to be solved and based on the trade-off between simplicity and accuracy requirements.

4.2.2 Neural network pruning

Neural network model pruning is a technique used to reduce the size of a model (model compression) and/or improve its efficiency. However, removing parameters can lead to a decrease in model performance. Prunning is achieved by removing minor parameters (weights) from a model. In most scenarios, only weights of parameters are removed, retaining the biases unchanged.

An important consideration is how much of the model should be pruned. It depends on whether we need to get a model with some specific size or not. The paper [21] considers pruning 30 % of all parameters as a starting point for pruning, with a pruning safe zone between 30–50 %. Pruning a model is useful in scenarios where computational resources are limited or need to be conserved. The list below marks several cases where the use of prunning should be considered [21]:

- **Real-time applications** – real-time models (e.g. autonomous driving in vehicles) require low latency. Pruning achieves this by creating a model with fewer parameters performing fewer computations, which means faster inference.
- **Edge devices** – these devices have limited computational resources, so they can meaningfully use only lightweight models. Thanks to pruning, we can shrink the model size to save memory and comply with given restricted resources.
- **Bandwidth constraints** – pruned models can save resources needed for data transmission on edge devices or remote locations.

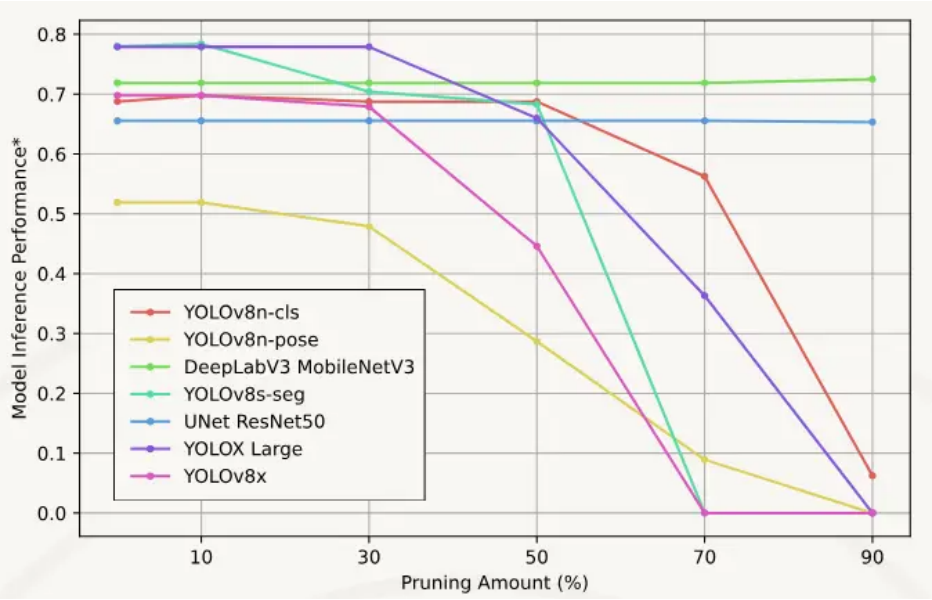
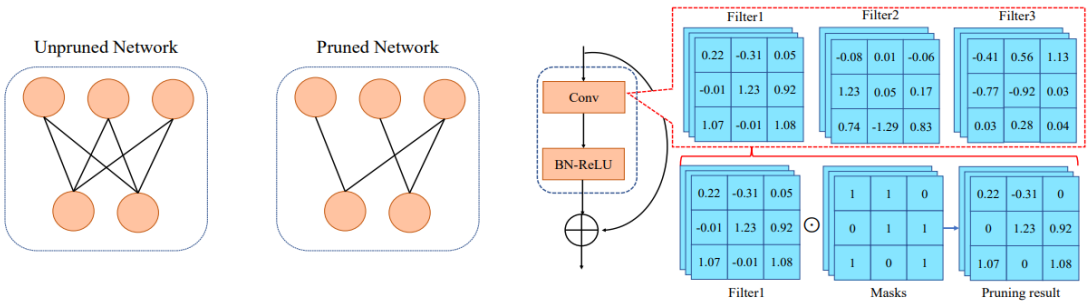


Figure 4.3: Inference performance changes of different neural network models given the different pruning percentage. Source: [21].

Two main approaches to pruning are **train-time pruning** and **post-training pruning**. As the name suggests, train-time pruning is integrated directly into the model training process. Pruning decisions are made together with weight updates during model training. This makes the training process more complex; however, pruning is more efficient.

Post-training pruning is applied after the model has been trained. This pruning removes unnecessary connections, neurons, or larger structures from a fully trained model. The pros of this approach are simpler implementation and pruning parameters are easily adjustable. On the other hand, pruning decisions are made by the user, which means that they may not be optimal [21].

Pruning can be categorized into **unstructured**, **structured** and **semi-structured**. The unstructured approach uses minimum weight thresholds to determine if the parameters should be pruned. It's common practice to set pruned weights to zero. This can be done directly or using masks as shown in Figure 4.4. The second category is structured pruning. It's merit is to remove entire filters, channels, layers, or attention heads (see Figure 4.5). It aims to find a pruned neural network with maximum speed inference and minimum performance decrease, given a prune percentage. Semi-structured pruning is based on patterns. As an example we can choose different parts of the weight structures to be pruned in different layers/filters [5, 21].



(a) From the view of neurons and connections

(b) From the view of weights and masks

Figure 4.4: Illustration of different ways of performing unstructured pruning. Source: [5].

When pruning, one has to analyze if the weight should be pruned or not. This can be done using some criteria or conditions. This list presents some criteria that are being used in real applications [5]:

- **Magnitude-based pruning** – all weights and/or activations that have an absolute value below a defined threshold are pruned. The real-world example of this pruning is provided by the TensorFlow framework [33].
  The function `tfmot.sparsity.keras.prune_low_magnitude` could be used to prune a given percentage of weight with low values (close to zero). It can be applied to a trained model by retraining it with a pruning wrapper.
- **Loss change** – this method evaluates loss of the neural network with the specific weight (default scenario) and without the weight (pruned).

- **Sensitivity/saliency** – computes a sensitivity and/or saliency of the weight to determine how much the weight is important in the network.

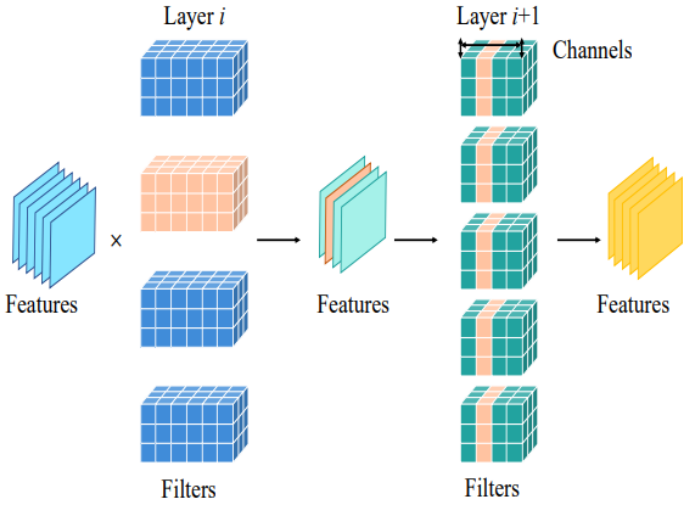


Figure 4.5: Visualization of structured pruning. Orange blocks are pruned. Source: [5].

The pruned neural network can be further improved by combining it with quantization or neural architecture search (NAS).

4.2.3 Pareto frontier

Pareto frontier (also called Pareto front or Pareto curve) is a tool used in multi-objective optimization. Multi-objective means that there are more than one objective that needs to be optimized simultaneously. Pareto frontier represents set of solutions for which no objective can be improved without making at least one other objective worse. The front of solutions helps to make tradeoff choices, rather than considering full range of every parameter [37].

Solution X that lies on the Pareto front is *dominating* solution Y if:

- X is **at least equally good** in all objectives than Y and
- X is **better** at least in one objective than Y.

For minimization of a problem, we can mathematically write Pareto domination as follow ($f_i(A)$ is a function of objective i for solution A):

$$\forall i : f_i(X) \leq f_i(Y), \exists j : f_j(X) < f_j(Y) \tag{4.7}$$

The concept of Pareto frontier and Pareto dominance is used practically in Sections 4.3 and 4.4 to compare trained models in more metrics at the same time. Example visualization of Pareto frontier is provided in Figure 4.6.

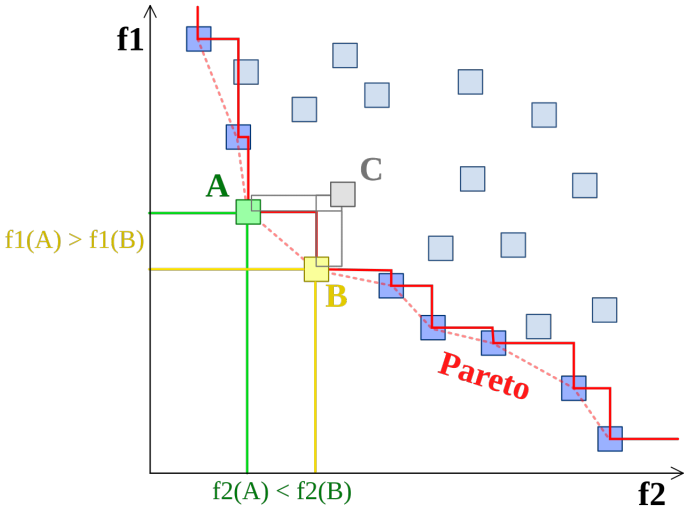


Figure 4.6: Pareto frontier example. There are two objective functions $f1$ and $f2$, which both we want to minimise in this case. Blue squares creates Pareto frontier. Point C is not included in Pareto frontier because its dominated by both point A and point B. These two point are not dominated by any other, so they are on the frontier. Source: [37].

4.3 Proposed anomaly detector

This section conducts a search for the optimal architecture for an anomaly detector without direct testing on edge devices, which is analysed in the next section. It starts with searching for an optimal neural network for detection in Subsection 4.3.1 by trying and evaluating multiple architectures. This is followed by Subsection 4.3.2 where advanced measurements are calculated on various model variations from the first part. Although this approach yields satisfactory results, other approaches have been attempted and are described in Subsection 4.3.3 to achieve a wider coverage of the issue.

4.3.1 Detection network search

Several approaches were explored to identify a suitable neural network architecture for anomaly detection. The following section provides a detailed description of these methods. All detection models were trained and evaluated using synthetic datasets generated by the program described in Section 3.4. Unsupervised learning is more suitable for the task of detecting satellite telemetry anomalies, because the detector should identify anomalies based on recent normal behavior rather than being trained in a supervised way to recognize predefined anomaly patterns. A supervised detector would likely perform poorly on new, unseen data, since anomalies in telemetry are rare and often differ from those seen during training.

From a data perspective, the main difference between the two most common categories of detectors — **forecasting-based** and **reconstruction-based** — lies in what is considered the ground truth output. In forecasting-based methods, the ground truth corresponds to the subsequent sequence of data over a certain length, allowing the detector to compute the error between its forecast and the true sequence. On the other hand, a reconstruction-based

detector learns the distribution of normal time-series signals and identifies anomalies as points with a high reconstruction error.

Various detector architectures have been tested to find the most suitable one. As an appropriate neural network designs, models based on the existing state-of-the-art approaches described in Section 3.2 have been selected. Choosen architectures are: **classical autoencoder**, **LSTM based autoencoder**, **ResNet network** and **Temporal Convolutional Neural Network (TCNN)**.

The information about those models are summarized in Table 4.1. Each architecture has information about layers it consists of, the category of detection and performance data. If not declared else, models have been tested on 1 day long dataset (86400 timesteps), with environmental and sensor-based anomalies (38 in total). Some models have been trained as forecasting models, and some have been trained for signal reconstruction. Reconstruction-based LSTMAE was trained using only half the size of the dataset due to limited memory resources at the time of these experiments. To compare new models with reference solution, the Telemanom model from paper [13] was used. The original Telemanom[5] model predicts values only for one feature at a time, so for the demonstration purposes here, it was modified to predict all features at once.

Model	Type	Layers	Size	Precision	Recall	AU ROC
LSTMAE	forecasting	256->128->256	3.86 MB	0.782	0.782	**0.870**
LSTMAE	reconstruction	128->64->128	900 KB	0.718	0.718	0.753
Autoencoder	forecasting	512,256,128,64	13 MB	0.793	0.793	**0.854**
ResNet like model	forecasting	3 residual blocks	65 MB	0.738	0.738	0.845
TCNN	forecasting	5 conv1D res. blocks	1 MB	0.676	0.676	0.765
Telemanom	forecasting	80,80	330 KB	0.680	0.680	0.768

Table 4.1: Performance data of various detection models. All forecasting models predict next 50 steps based on the current signal window with length of 250 timestamps. Telemanom model was recreated from NASA paper and it predicts only next 10 steps. Precision and recall have been computed for the optimal threshold (it vary for different models). All models have been trained for 15 epochs. Training data had length of 1 day (86400 timestamps), only reconstruction-based LSTMAE was trained on half-day long dataset.

Three out of five models outperformed baseline Telemanom model in PR AUC (area under precision-recall curve – more details are provided in the next Subsection) metric. All of these models architectures are further explored in the following sections.

A note about used synthetic dataset and model training

As was mentioned above, the test dataset for all evaluations provided in this chapter (if not declared else) consists of 86400 datapoints or timesteps. There are 20 features measured in each timestep, which match the number of telemetry sensors from the generated synthetic dataset. The same applies for training and validation datasets. The key distinction lies in the composition of the datasets: both the training and validation datasets contain **no anomalies**, while the testing dataset includes **38 anomaly events**, covering both sensor-related and environmental cases. This ensures that the detector learns what **normal** data look like.

When training, the input batch of data has shape of `(batch size, sequence length, number of features)`, where batch size was set to 32 (for LSTM it was set to 16), se-

[5] https://github.com/khundman/telemanom/tree/master

quence length is 250 in most cases and number of features is 20 as was determined earlier. Sequence length sets size of the past signal window. On the other end of the network, the shape of the output is (again in most cases, if not determined else) as follows: `(number of predicted steps, number of features)` for forecasting-based models or `(sequence length, number of features)` for reconstruction-based models. Usually, models predict 50 future steps of a signal for each telemetry sensor.

4.3.2 Advanced measurements

After first evaluations to find the best detector architecture described above, it was decided to train multiple versions of the models from Table 4.1. Each model has been recreated with different number of layers and number of neurons in these layers. The goal of this approach was to find out if making a larger architecture of neural network anomaly detector has a positive influence on the results (metrics). To measure performance and to compare the models, the following metrics have been recorded:

- **Precision** – described in Section 2.1.4.
- **Recall** – described in Section 2.1.4.
- **Area under Precision-Recall Curve** – also described in Section 2.1.4. Main comparison metric used in this thesis.
- **Optimal threshold value** – more than a metric, this values determine the best anomaly threshold to use to balance precision and recall. Using this variable threshold for each model is the reason why precision and recall both have the same value.
- **Average inference time** – average time in seconds spent inferencing on 1000 input sequences. The shape of the input sequence data is (sequence length, 20), where 20 is the number of features in the synthetic dataset. In other words, this time means how long it takes to predict future values 1000 times. Further details regarding the measurement hardware are provided in the table notes below.
- **Model size** – size of the model (architecture + weights). This value is also a good reference point for predicting how much RAM will the model need when deployed and running inference.
- **FLOP** – floating point operations – the total number of operations like addition or multiplication, which the model computes in one forward pass. Unlike FLOPS (metric to compute GPU performance), FLOP is used to determine computational complexity of a model.
- **Average training time** – average time needed to train a model on 1000 input sequences. Measured in seconds. As for average inference time, further details regarding the measurement hardware are provided in the table notes below.
- **Batch size** – how many datapoints are in one batch when training a model.

All models have been trained, validated and tested with the same three (disjunct) datasets with the length of 1 day (86400 timestamps). All models have been trained in 30 epochs, with batch size 32 or 16 as was described above in previous subsection. The Adam optimizer[6]

[6] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

was used and *mean absolute error* was chosen as a loss function. All forecasting based models used sequence of data for training with length of 250 and forecasted next 50 steps of time-series. Only models called **denseae1** and **denseae1_drop** have been trained with sequences of length = 500. This **denseae** models predicted 100 next steps. Training and evaluation of models has been performed on high-end GPUs (NVIDIA A40 and NVIDIA L40S) on Metacentrum servers.

Tables 4.2 and 4.3 summarize the computed results. The first table shows the properties of the model such as size or simplified architecture. The second table collects all the performance metrics. In total there were 23 models trained; all of them are variations of model architectures from Table 4.1. Table captions describe all columns in more detail. It is important to note that the average training time and the average inference time were measured after models were trained on the same hardware at Metacentrum servers. These values do not represent times achievable at space boards because of large differences in hardware performance. The time values in these two tables serve only as a relative comparison between model architectures. The real simulation time values are presented in Table 4.8, where measurements have been made using a Raspberry Pi 4B, an onboard-like computer.

Taking a closer look at the Table 4.2, we can see that except for architectural details there are two technical measurements: Model size and MFLOP (Mega FLOP). The size of the model varies considerably for different types of architecture. For classical autoencoders (denoted as denseae), the greatest influence on model size (aside from number of neurons in layers) has been the type of method. Models **denseae1**, **denseae2**, and **denseae3** have significantly different sizes for forecasting-based and reconstruction-based variations. The reason for this is that the reconstruction-based form has an output made of number of neurons of the same size as input (sequence length). On the other hand, the forecasting-based version of the autoencoder predicts (size of output) only the number of next step values, which is shorter than the input sequence.

Another information we can get from Table 4.2 is that autoencoders with LSTM layers (marked as LTSMae) are smaller than classical dense autoncoders. The same applies for TCNN (Temporal Convolutional Neural Network). Only ResNet models are significantly larger than all other architectures. The MFLOP values indicate that ResNet models are the most computationally demanding, while LSTM autoencoders appear at the opposite end of the complexity spectrum.

Referring to Table 4.3, we observe the performance metrics for each model. The first two columns act as identifiers corresponding to Table 4.2, while the remaining columns contain the measured metric values. Precision and recall are identical for each model because their values were selected as mutual optima. These metrics were calculated using the optimal threshold listed in the OT column. Adjusting this threshold would increase precision at the cost of recall, or vice versa. The most important indicator is PR AUC. Its value represents the area under the precision-recal curve, which is used as the main comparison metric between models. The highest PR AUC was achieved by model **LSTMae2** (0.946), closely followed by **LSTMae1** (0.944). The second best architecture in this metric is the classic autoencoder (0.870). ResNet outperformed TCNN (maximum PR AUC value 0.841).

As noted earlier, the time-related values in the AIT and ATT columns are included solely for comparison. These results show that dense autoencoders achieved the shortest training time. The value of 0.066 s per 1000 input sequences is even lower than the inference time, which is likely an artifact of rounding. This discrepancy arises because the ATT values were derived from the time required to train a single batch of input, and such short durations

are highly sensitive to measurement precision. As in the case of model size and MFLOP, ResNet model is again the worst one with ATT ranging from 2.054 s to 6.846 s and AIT between 0.603 s and 1.777 s. The best results were recorded by dense autoencoders and TCNN models.

Model name	Method	Architecture	B/D/K	Model size [MB]	MFLOP
denseae1	FC	[512, 256, 128]	64	74.35	12.3787
denseae2	FC	[256, 128, 64]	32	**18.64**	3.0956
denseae3	FC	[256, 128, 64]	64	18.68	3.1034
denseae1	REC	[512, 256, 128]	64	121.31	20.1988
denseae2	REC	[256, 128, 64]	32	30.40	5.0525
denseae3	REC	[256, 128, 64]	64	30.45	5.0604
denseae1_drop	REC	[512, 256, 128]	64	121.32	20.1988
denseae2_drop	REC	[256, 128, 64]	32	30.41	5.0525
denseae3_drop	REC	[256, 128, 64]	64	30.46	5.0604
LSTMae1	FC	[256, 128]	64	12.23	0.4892
LSTMae2	FC	[128, 64]	32	3.21	0.2451
LSTMae3	FC	[256]	128	11.65	0.4892
LSTMae4	FC	[128]	64	3.04	0.2451
LSTMae1	REC	[256, 128]	64	11.84	2.4462
LSTMae2	REC	[128, 64]	32	3.10	1.2255
LSTMae3	REC	[256]	128	10.13	2.4462
LSTMae4	REC	[128]	64	**2.65**	1.2255
resnet1	FC	[256, 256, 256]	256	195.09	222.2912
resnet2	FC	[128, 128, 128]	256	**97.93**	64.5154
resnet3	FC	[256, 256, 256]	512	385.53	254.0298
tcnn1	FC	[64, 64]	[3, 3, 3]	**4.63**	56.921
tcnn2	FC	[64, 64]	[5, 5, 5]	5.51	93.7355
tcnn3	FC	[128, 128]	[3, 3, 3]	8.70	220.1338

Table 4.2: Models configuration overview. In Method collumn, FC stands for forecasting-based, REC for reconstruction-based. Architecture briefly describes size of models layers. B/D/K number represents size of bottleneck for autoencoders, size of last dense layer for Resnets, or kernel sizes for TCNNs. MFLOP means mega FLOP, which was described above. Mark `_drop` in models name means that model has Dropout layers in the architecture.

Model name	Method	Precision	Recall	PR AUC	OT	AIT [s]	ATT [s]
denseae1	FC	0.783	0.783	0.827	0.440	0.215	0.586
denseae2	FC	0.805	0.805	**0.870**	0.416	0.108	0.177
denseae3	FC	0.699	0.699	0.802	0.404	0.100	0.173
denseae1	REC	0.767	0.767	0.715	0.413	0.123	0.080
denseae2	REC	0.792	0.792	0.812	0.399	0.111	0.066
denseae3	REC	0.734	0.734	0.777	0.393	0.091	0.066
denseae1_drop	REC	0.599	0.599	0.614	0.489	0.124	0.089
denseae2_drop	REC	0.690	0.69	0.697	0.505	0.085	0.075
denseae3_drop	REC	0.679	0.679	0.685	0.518	0.09	0.075
LSTMae1	FC	0.899	0.899	0.944	0.380	0.367	1.478
LSTMae2	FC	0.903	0.903	**0.946**	0.367	0.326	1.323
LSTMae3	FC	0.912	0.912	0.939	0.349	0.269	1.060
LSTMae4	FC	0.861	0.861	0.928	0.346	0.246	0.963
LSTMae1	REC	0.637	0.637	0.723	0.132	0.513	2.069
LSTMae2	REC	0.586	0.586	0.629	0.142	0.490	1.882
LSTMae3	REC	0.687	0.687	0.688	0.088	0.571	1.290
LSTMae4	REC	0.843	0.843	0.838	0.076	0.330	1.253
resnet1	FC	0.729	0.729	0.807	0.355	1.655	5.229
resnet2	FC	0.745	0.745	**0.841**	0.473	0.603	2.054
resnet3	FC	0.785	0.785	0.838	0.375	1.777	6.846
tcnn1	FC	0.679	0.679	0.774	0.366	0.131	0.287
tcnn2	FC	0.741	0.741	**0.788**	0.376	0.129	0.290
tcnn3	FC	0.678	0.678	0.760	0.371	0.145	0.301

Table 4.3: Models performance metrics. Abbreviations in Method collumn are the same as in the case of Table 4.2. Precision, recall and PR AUC (Area Under Precision-Recall Curve) have been described above. **OT** stands for optimal threshold, **AIT** is Average Inference Time – average time (in seconds) model needed to inference on 1000 input sequences, measured in 10 runs. **ATT** is short for Average Training Time and its value means how many seconds it takes to train model on 1000 input sequences. This metric was calculated as mean from batch training time and batch size. Best values of PR AUC for each model architecture type are marked bold.

Pareto frontier graphs have been created to better visualize model performances in different objectives. The following images (4.7, 4.8, 4.9) show some of the Pareto frontier graphs for the models from tables 4.2 and 4.3. Models that were too far from the front have been removed, so the graph can be more precise in the dense space where other models are situated. In other words, the graphs have been zoomed in to interesting areas.

The first image (4.7) presents a comparison of the *PR AUC* values against the *average training time*. The results show that the `LSTM-based models` (models 2, 3, and 4), together with the `denseae2` model in both forecasting- and reconstruction-based variants, form the Pareto frontier. In other words, these models outperform all remaining approaches when considering both performance and training efficiency. The final model selection therefore depends on the priorities of the user or application – whether they prefer higher *PR AUC* or shorter training time.

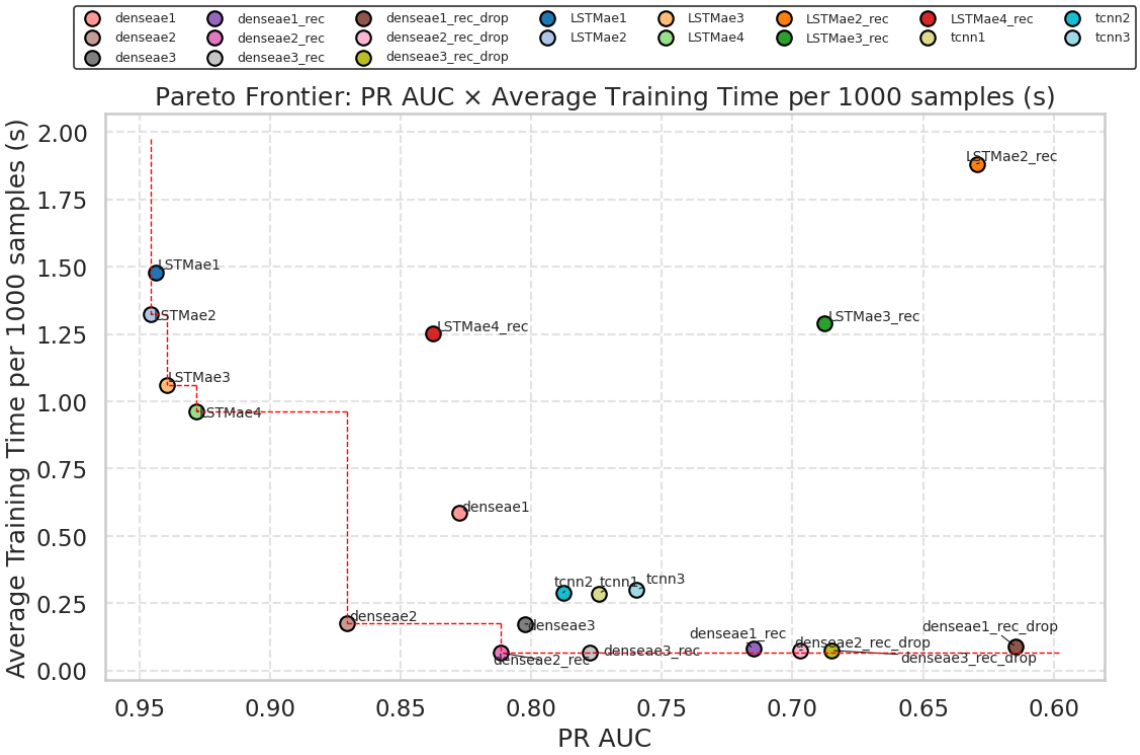


Figure 4.7: Pareto frontier of PR AUC and average training time on 1000 input sequences in seconds. X axis is inverted: better values are on the left. Solutions that make Pareto frontier are located in the left bottom corner (the more is model located to the bottom left corner, the better it is in this two objectives). Solutions laying on dashed red line form Pareto frontier.

The plot in Figure 4.8 shows that the models `LSTMae2` and `LSTMae4` (in both forecasting and reconstruction variants) dominate the remaining solutions when comparing *model size* and *PR AUC*. Due to their relatively large size, the `ResNet`-based models do not appear in this zoomed view of the graph. Since most evaluated models are relatively small, this

Pareto frontier representation is less informative than others; however, it is included here for completeness and illustrative purposes.

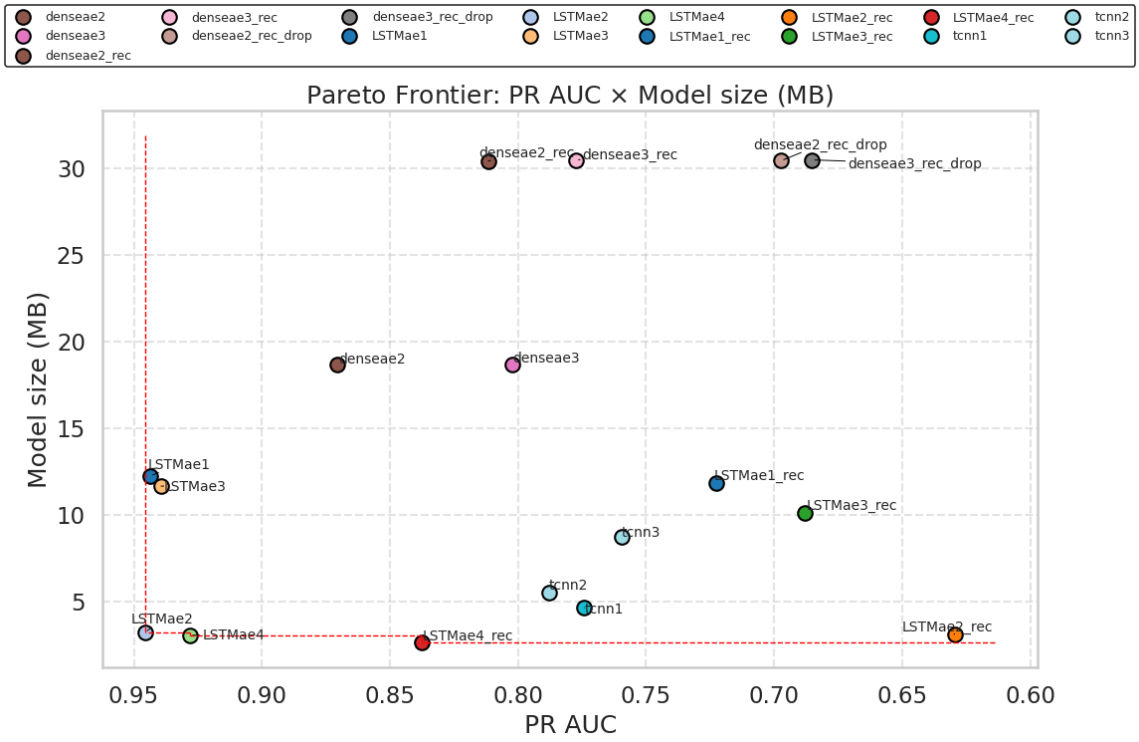


Figure 4.8: Pareto frontier of PR AUC and model size. X axis is inverted: better values are on the left. Solutions that make Pareto frontier are located in the left bottom corner (the more is model located to the bottom left corner, the better it is in this two objectives). Solutions laying on dashed red line form Pareto frontier.

The last graph (see Figure 4.9) presents a comparison of the two objectives: *PR AUC* and *average inference time*. The results are very similar to those observed in the comparison of PR AUC and average training time. Once again, the `LSTM-based models` together with `denseae2` form the Pareto frontier. This outcome further highlights the relationship between inference efficiency and training complexity in anomaly detection models.

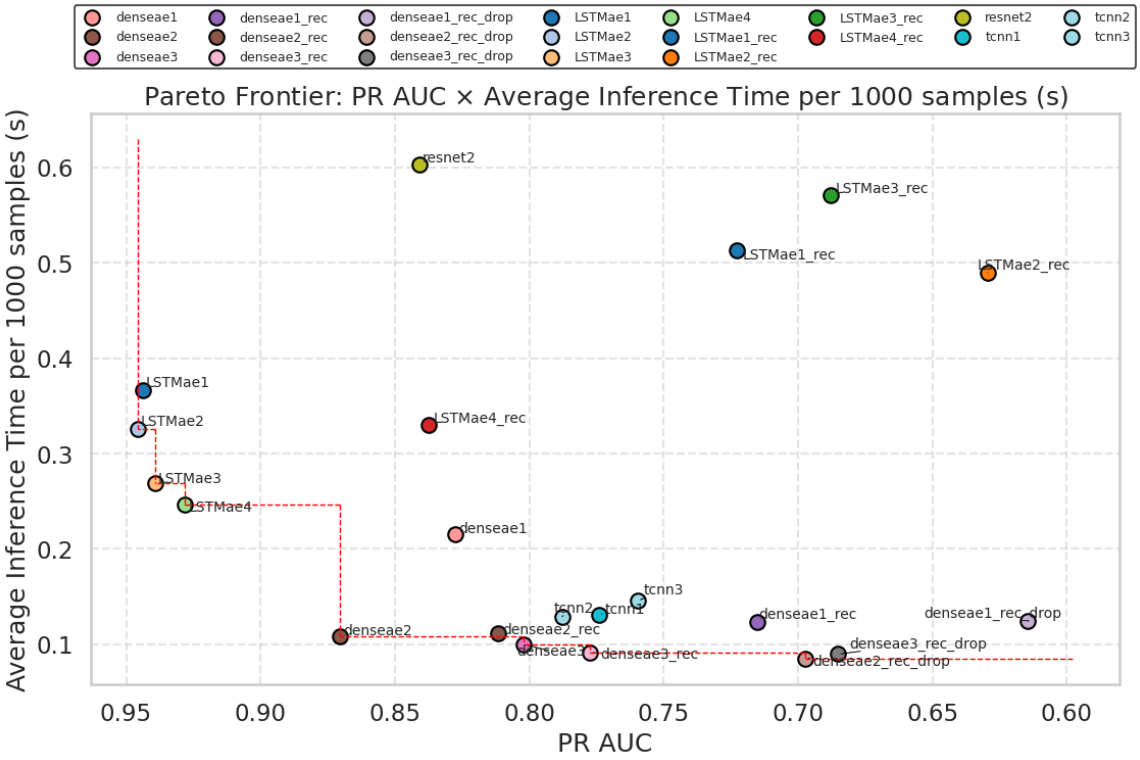


Figure 4.9: Pareto frontier of PR AUC and average inference time on 1000 input sequences in seconds. X axis is inverted: better values are on the left. Solutions that make Pareto frontier are located in the left bottom corner (the more is model located to the bottom left corner, the better it is in this two objectives). Solutions laying on dashed red line form Pareto frontier.

It is common practice to include loss function plots in theses involving neural networks. However, in the context of this work, which focuses on neural network–based anomaly detection, such plots are not particularly informative. Figure 4.10 shows the loss curve of the best-performing model (based on the highest PR AUC score on the test dataset). At first glance, the model appears to be overfitting, as the validation loss does not decrease significantly during training.

This behaviour is not necessarily problematic. In practical applications, an anomaly detector must first learn the structure of the **normal** data. Since this structure may vary from one satellite to another, even a slightly different validation dataset can represent a different pattern of normal behaviour. As a result, the model may effectively be attempting to learn two different normal distributions, which leads to a higher validation loss without negatively affecting anomaly detection performance. For this reason, this loss function graph is the only one included in this thesis.

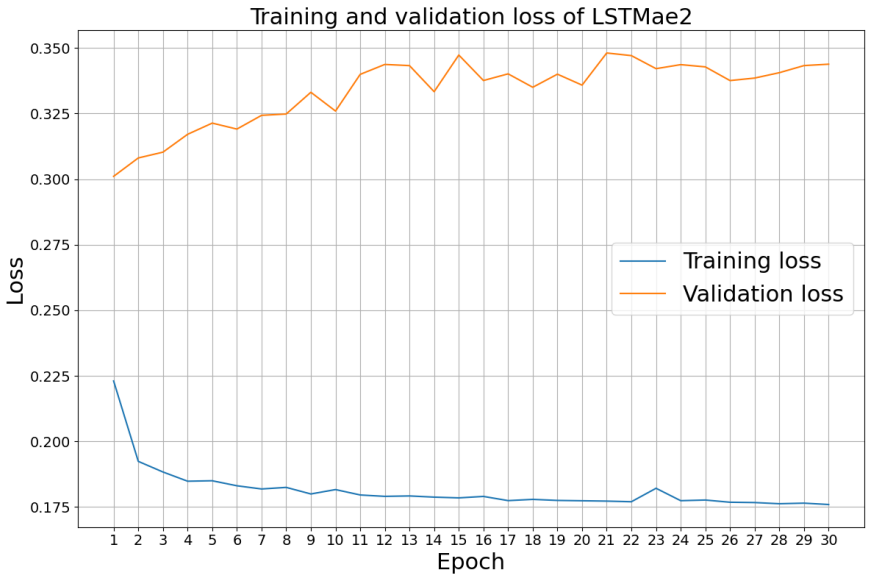


Figure 4.10: Loss value recorded over time for highest PR AUC model **LSTMae2**. Blue line represents training loss, while orange line shows progress of validation loss.

4.3.3 Another approaches in architecture search

With the goal to confirm or deny results presented in the previous part, it was decided to train and test some more specific models. That included two steps or cases:

1. creation of a very large models for each category,
2. use of GRU (Gated Recurrent Units) instead of LSTM for LSTM-based autoencoders – because GRU should be faster than LSTM.

Table 4.4 provides an overview of the models created in step 1. The number of layers and neurons in each model was increased to produce larger architectures, which is reflected in the recorded model sizes. For example, the largest dense autoencoder in Table 4.2 originally had a size of 121.32 MB, whereas large variant now takes 538.78 MB. Similar growth can be observed for the LSTM autoencoder, ResNet, and TCNN models, with the most significant increase recorded for the TCNN model (from 5.51 MB to 215.68 MB). Comparative visualization of model sizes is provided in graph in Figure 4.11.

Model name	Method	Architecture	B/D/K	Model size [MB]	MFLOP
denseae_large	FC	[4096, 2048, 1024, 512, 256, 256]	128	538.78	89.767
LSTMae_large	FC	[1024, 512, 512, 256, 256]	128	253.01	2.687
resnet_large	FC	[512, 512, 512, 512]	512	780.13	1132.448
tcnn_large	FC	[512, 512, 512]	[5, 5, 5]	215.68	8891.428

Table 4.4: Overview of a large versions of models from previous subsection. Abbreviations in the header row are explained in the caption of Table 4.2. The main differences are in the model architectures and model sizes.

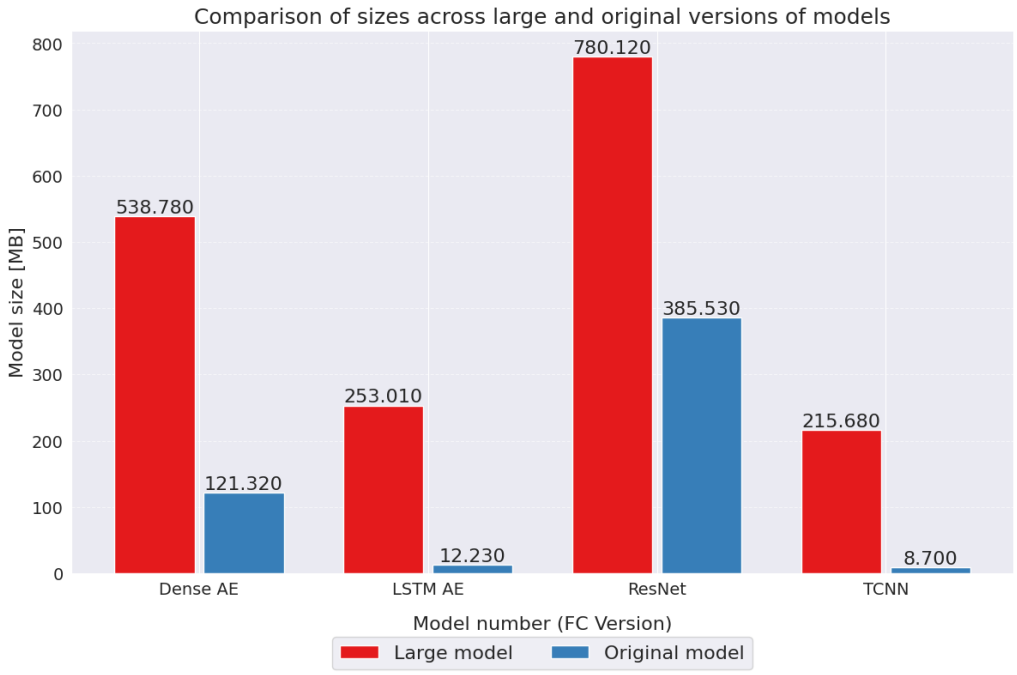


Figure 4.11: The graph shows comparison of sizes of largest original models and newly created ones. Highest growth is recorded for TCNN architecture, smallest for ResNet.

The rationale behind creating larger models was the expectation that they might achieve higher precision, recall, and, in particular, higher *PR AUC* compared to the original models. Performance metrics for these larger models are summarized in Table 4.5. Examining the ATT statistic, we observe that two models show higher ATT values than the slowest original models listed in Table 4.3 (LSTM and TCNN architectures). These differences are due to varying GPU availability during training on the Metacentrum servers, so ATT values should only be used for comparisons within a single table, not across different tables.

Model name	Method	Precision	Recall	PR AUC	OT	AIT [s]	ATT [s]
denseae_large	FC	0.806	0.806	0.814	0.591	**0.223**	**0.305**
LSTMae_large	FC	**0.807**	**0.807**	**0.835**	0.458	1.36	2.861
resnet_large	FC	0.748	0.748	0.816	0.367	0.372	0.770
tcnn_large	FC	0.76	0.76	0.834	0.391	1.073	3.015

Table 4.5: Large models performance metrics. Column in headers have the same meaning as in Table 4.3. Best values are marked bold.

For this work values from column *PR AUC* of Table 4.5 are more important. The highest PR AUC was measured for the LSTMae model (0.835). The highest precision and recal also holds LSTMae autoencoder (0.807). For better visualization of the results for each model type, the PR AUC results of the best original models and large models have been summarized in graph in Figure 4.12. Its clear that hypothesis about larger models having potentionally better results was not confirmed. In 3 out of 4 cases, original models outperformed large versions (despite that results are relatively close to each other). The only case where making a bigger model actually helped, was TCNN architecture. New larger model achieved PR AUC 0.834 which is higher about 0.046 than best original TCNN result.

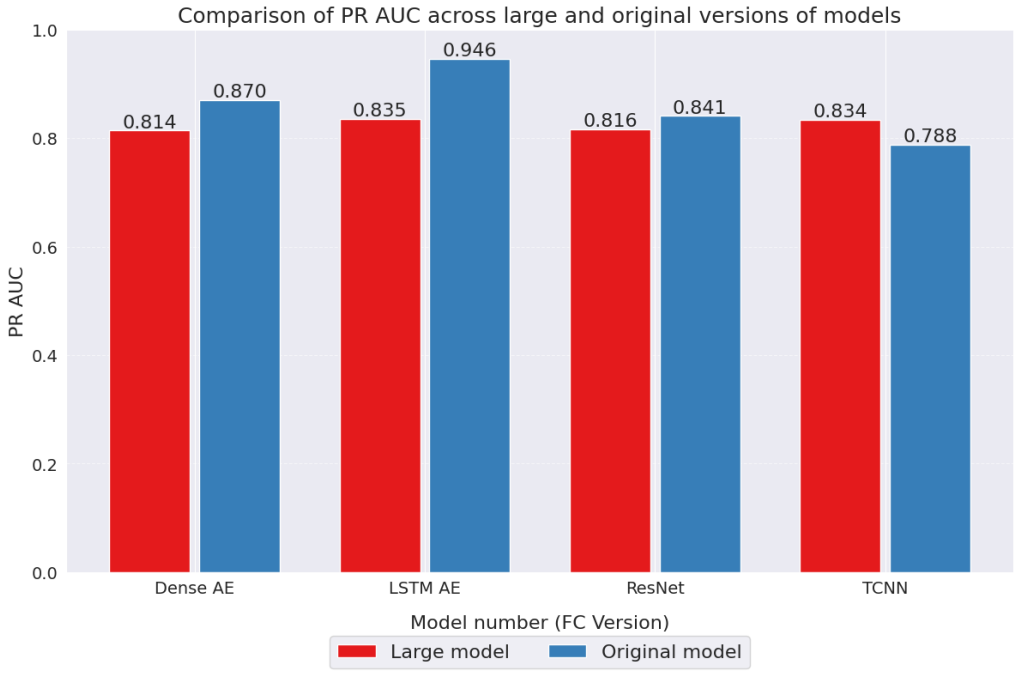


Figure 4.12: Graph compares PR AUC values for each model type. Highest PR AUC stays for LSTM autoencoder. The only case where model scaling improved detection was TCNN architecture. Higher is better.

The presented measurements indicate that creating larger models did not provide any significant benefit in this case. This outcome suggests that simply increasing the size of the models does not guarantee improved performance. It is possible, that using a more complex dataset or adopting a different detection approach could yield different results, where larger

architectures might offer an advantage. However, in the context of the current study and the generated synthetic dataset, it is evident that larger models do not necessarily lead to better results. This finding emphasizes the importance of matching model complexity to the characteristics of the data and the specific requirements of the anomaly detection task, rather than assuming that bigger models are inherently superior. For space-oriented applications, this is encouraging, as smaller models are easier to deploy and require less computational and memory resources on satellite hardware.

Although the larger autoencoder models (whether based on LSTM layers or dense blocks) or ResNet architecture achieved training losses comparable to or even lower than their smaller (original) counterparts, their anomaly detection performance (measured by PR AUC) was worse. This behaviour can be explained by a mismatch between model capacity and dataset size. The larger networks contain far more parameters and thus have enough expressive power to memorize the normal training patterns instead of learning their underlying structure. When trained exclusively on normal data, they tend to reconstruct or forecast normal sequences extremely well, but also generalize this capability to patterns that should be considered abnormal. In forecasting-based anomaly detection, this reduces the separation between the error distributions of normal and anomalous samples, directly lowering PR AUC even though the overall training loss remains low.

This effect is amplified by the nature of the dataset: anomalies represent only a small part of dataset for these big model sizes, making PR AUC highly sensitive to even slight reductions in the relative error difference between normal and anomaly behaviour. Smaller models, due to their limited capacity, generalize more appropriately and maintain a stronger separation between normal and anomalous reconstruction or prediction errors.

The other approach included LSTM units replacement with GRU units. GRU units are a simplified variant of LSTM, which makes them more computationaly efficient [10]. GRU uses only 2 gates instead of 4 in LSTM:

- **Update gate** – states how much information from the past should be caried forward.
- **Reset gate** – determines how much information from the past should be ommited or forgotten.

The decision to try to replace LSTM units with GRU units was motivated by inference efficiency considerations for sequence models. Compared to LSTM, GRU has a simpler internal structure and lower computational complexity, which can lead to faster inference. Based on this assumption, the original LSTM autoencoders were converted to GRU-based architectures and trained on the same dataset as the remaining models. The resulting performance metrics are presented in Tables 4.6 and 4.7.

Model name	Method	Architecture	B/D/K	Model size [MB]	MFLOP
GRUae1	FC	[256, 128]	64	9.24	0.4892
GRUae2	FC	[128, 64]	32	2.45	0.2451
GRUae3	FC	[256]	128	8.79	0.4892
GRUae4	FC	[128]	64	2.32	0.2451

Table 4.6: Overview of a GRU autoencoders models. Abbreviations in the header row are explained in the caption of Table 4.2. Model sizes are very close to the ones measured for LSTM-based detectors.

GRU-based models have been built with the same architecture details as LSTM models which is described in Table 4.6 in columns **Architecture** and **B/D/K**. Again, in the same way as for other models, model sizes and MFLOP values are denoted here as well. The largest GRU model is model type 1, with a size of 9.24 MB. For comparison, the largest LSTM model – also model type 1, with identical layers and neuron counts – has a size of 12.23 MB. This difference is small in absolute size, but significantly different (around 25 % increase) in relative comparison.

Model name	Method	Precision	Recall	PR AUC	OT	AIT [s]	ATT [s]
GRUae1	FC	0.817	0.817	**0.874**	0.307	0.631	2.167
GRUae2	FC	0.811	0.811	0.867	0.301	0.610	2.040
GRUae3	FC	0.792	0.792	0.840	0.297	0.483	1.504
GRUae4	FC	0.778	0.778	0.837	0.283	**0.444**	**1.392**

Table 4.7: GRU models performance metrics. Again, abbreviations in header correspond to the ones in Table 4.3. Best values are marked bold.

More interesting information is gathered in Table 4.7, where performance metrics are stored. Precision and recall (like in previous tables) have the same values because they have been measured at the threshold (optimal threshold) coresponding to the maximum of both values. The highest precision (0.817), recall (0.817), and PR AUC (0.874) have been recorded for the model **GRUae1**.

Measurements of the PR AUC metric are used as the primary basis for comparing the GRU-based models with their LSTM counterparts in order to determine which sequence architecture is better suited for the intended use case. The graph in Figure 4.13 visualizes the extracted PR AUC values from Tables 4.7 and 4.3. It is clear that the LSTM models outperform the GRU models in every instance. These results indicate, that despite the slightly smaller model sizes of GRUs, the performance advantage is consistently better for LSTMs. This suggests that for tasks where predictive quality is critical, LSTM-based architectures remain the more reliable choice.

To conclude this subsection and to provide a more comprehensive comparison, radar charts summarizing multiple objectives for each model type are shown in Figure 4.14. These charts illustrate that, for both precision/recall and PR AUC, the LSTM models consistently achieve better performance. The MFLOP values remain nearly identical across the two architectures. GRU models do offer a slight advantage in model size; however, the difference between 12 MB and 9 MB is minor and does not represent a meaningful practical benefit in the context of satelite telemetry anomaly detection.

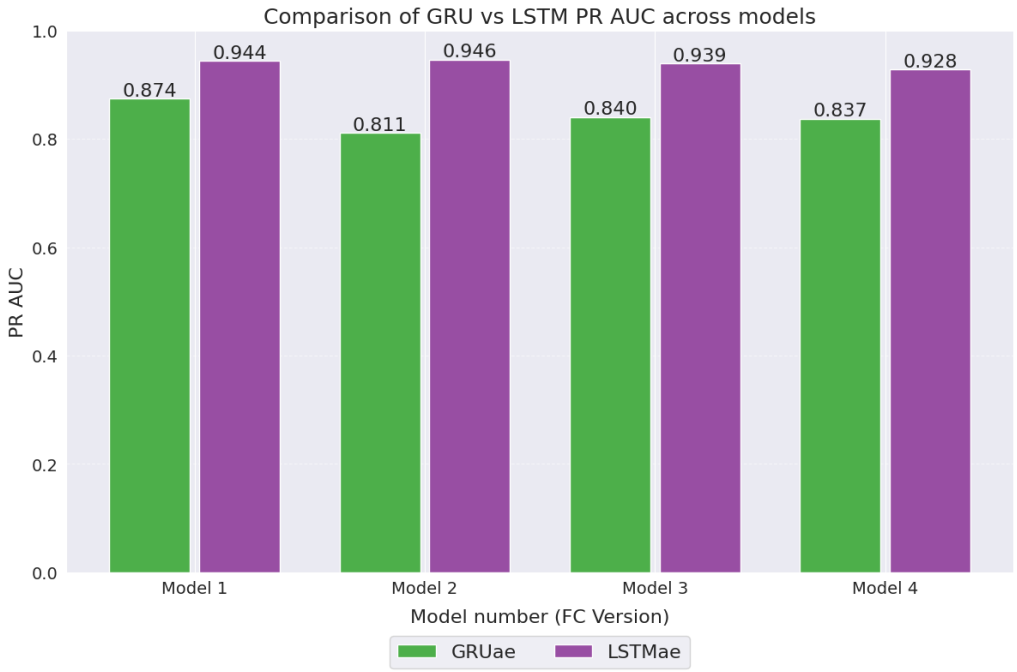


Figure 4.13: Graphical comparison of PR AUC values for 4 model types using LSTM or GRU units (higher values are better). In each case, usage of LSTM performed better than GRU. Best GRU model variation was the first one, best result recorded by LSTM model was in second architecture.

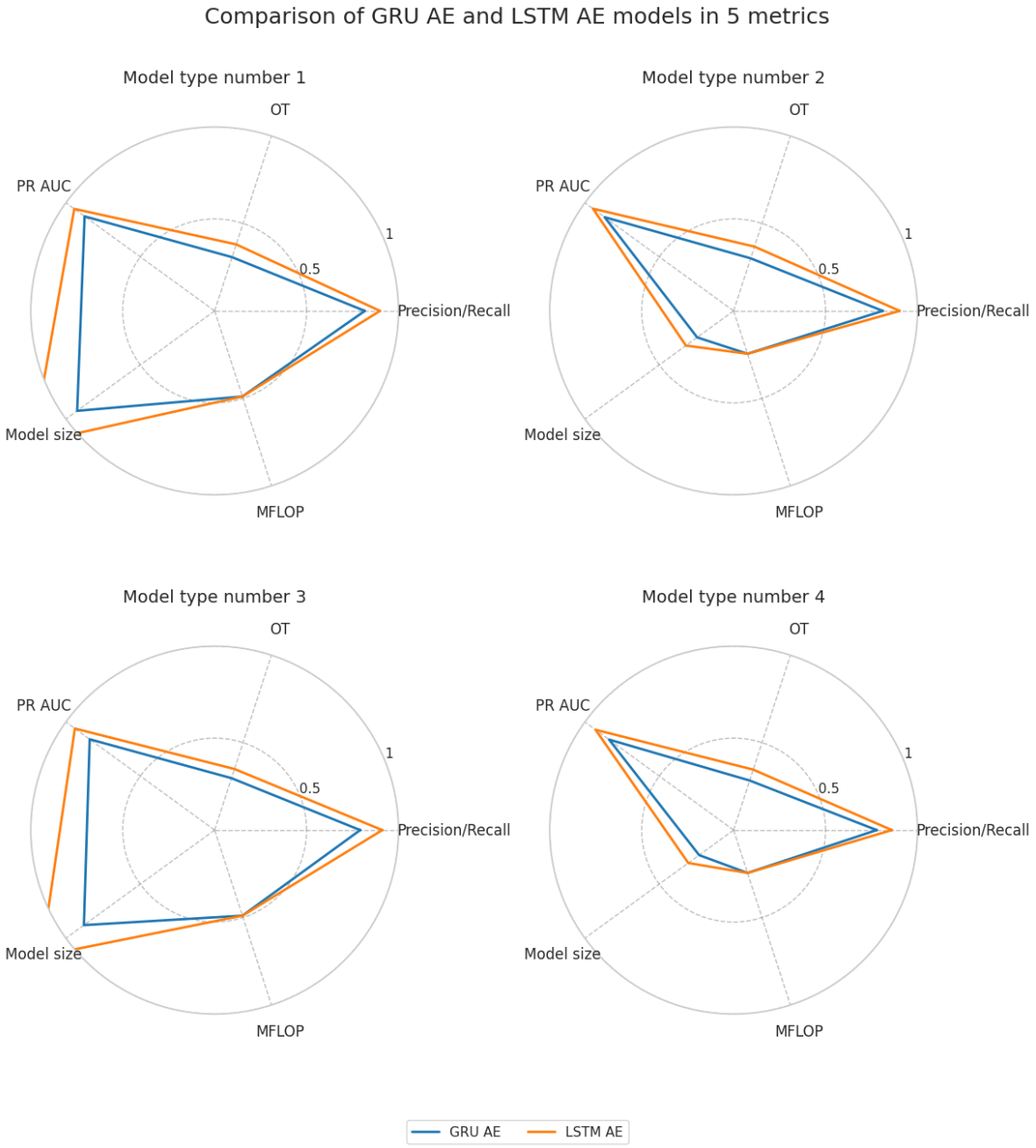


Figure 4.14: Radar charts for all forecasting based models built with LSTM and GRU units. LSTM outperforms GRU at two most important metrics: PR AUC and precision/recall. The key to read the graphs is following: PR AUC and Precision/Recall are better when higher, Model size and MFLOP is better when lower. OT stands for optimal threshold and its value influences precision and recall. For this graph OT is not better or worse if it's low or high, it was included just for illustrative purpose.

4.4 On-board performance

To gain more realistic results in terms of inference times and model sizes, it was decided to measure these on Raspberry Pi 4B with 4 GB of RAM. This device has 4-core ARM Cortex-A72 CPU (1.5 GHz). The results are presented in Table 4.8. Models have been transformed from TensorFlow to **TensorFlow Lite (TF Lite)**[7], because Raspberry Pi 4B does not support full TensorFlow. This caused a decrease in the size of the models but significant increase of inference time for LSTM based autoencoders. TF Lite does not fully support standard LSTM layers and instead requires them to be converted into unrolled versions (parameter `unroll=True`), which means the recurrent operations are expanded into a long sequence of simpler operations. While this workaround enables deployment on edge devices like Raspberry Pi, it significantly increases the number of computation steps needed during inference. As a result, models that rely on LSTMs are dramatically slower in terms of inference speeds compared to native LSTM execution in full TensorFlow. Similar results have been measured for GRU-based neural networks, where inference speed drop was not that notable, but still relatively high compared to other architectures. TF Lite provides support for 1D convolutions, which had positive influence on TCNN based models.

Table 4.8 holds 2 types of PR AUC values. Values denoted as *PR AUC Orig* in the table are not computed directly on Raspberry Pi but instead are copied from default TensorFlow full versions of models. Values in column *PR AUC* are particular values measured on Raspberry Pi, but using a smaller version of test dataset. This decision was made because some models exceeded the available memory during inference on the complete 86,400-timestep test dataset. To run those models on the Raspberry Pi, it was necessary to reduce the dataset size or run inference only on a portion of it, which led to inconsistent PR AUC results across models. To avoid these discrepancies and ensure fair comparison, the original TensorFlow PR AUC values were displayed as well as new measured ones. This issue would not appear in the real-life scenario, since in that case only 200 previous values need to be stored in RAM, not the whole big dataset.

Looking at Table 4.8 we can see that fastest models are classical simple autoencoders. TF Lite constraints didn't have impact on them. The Model size column further shows that TensorFlow Lite achieved strong compression on TCNN-based architectures—two of them becoming the smallest models overall.

Denseae2 seems to be the best model from this table, having high value of original models PR AUC (0.870) while staying small (6.2 MB) with inference time around 3 s per 1000 input sequences. Significant inference time increase (caused by problems mentioned above) was recorded for model **LSTMae2** – model with the highest PR AUC. This might be partially solved by using GRU units instead of LSTM, which is little bit faster in TF Lite implementation, achieving similar PR AUC values. Very interesting solution is now represented by TCNN models, because they are small and fast. Since the new, smaller dataset contained a relatively high number of anomalies (15), representing approximately 20 % of the original dataset, all measured *PR AUC* values are correspondingly high.

The image 4.15 shows the Pareto frontier for the Raspberry Pi results comparing PR AUC original values with inference time on 1000 input sequences. The Pareto front is still mainly made up of LSTM-based models and the **denseae2** autoencoder. We can clearly see that many classical autoencoders lie on or close to the Pareto frontier low boundary of inference time.

[7] https://www.tensorflow.org/api_docs/python/tf/lite

Model name	Method	IT [s]	Model size [MB]	PR AUC	PR AUC Orig
denseae1	FC	7.794	24.770	0.902	0.827
denseae2	FC	**2.933**	6.200	0.969	0.870
denseae3	FC	2.950	6.216	0.921	0.802
denseae1	REC	15.181	40.425	0.960	0.715
denseae2	REC	4.551	10.121	0.974	0.812
denseae3	REC	4.827	10.137	0.969	0.777
GRUae1	FC	115.783	5.552	0.957	0.874
GRUae2	FC	34.902	3.337	0.906	0.811
GRUae3	FC	105.556	4.612	0.924	0.840
GRUae4	FC	**28.743**	2.455	0.932	0.837
LSTMae1	FC	163.737	6.266	0.968	0.944
LSTMae2	FC	42.23	3.302	0.963	0.946
LSTMae3	FC	147.864	5.378	0.974	0.939
LSTMae4	FC	**34.551**	2.511	0.966	0.928
LSTMae1	REC	266.835	7.085	0.982	0.723
LSTMae2	REC	62.364	4.171	0.887	0.629
LSTMae3	REC	217.04	5.256	0.925	0.688
LSTMae4	REC	49.808	2.765	0.934	0.838
resnet1	FC	55.054	65.023	0.943	0.807
resnet2	FC	**19.193**	32.635	0.961	0.841
resnet3	FC	81.615	128.501	0.972	0.838
tcnn1	FC	**5.142**	1.519	0.942	0.774
tcnn2	FC	9.179	1.811	0.929	0.788
tcnn3	FC	18.44	2.875	0.898	0.760

Table 4.8: Inference time, model size and Precision-Recall Area Under Curve (PR AUC) measured with Raspberry Pi 4B. Inference time was measured as time needed to infer on 1000 input sequences, the same way as it was done in Table 4.3. Model size reffers to the size of the TensorFlow Lite version of a model. Values in PR AUC Orig column have been copied from previous measurements. Dense autoencoder models with dropout layers have been ommited because they didn't show any improvement in term of PR AUC from autoencoders without dropout layers.

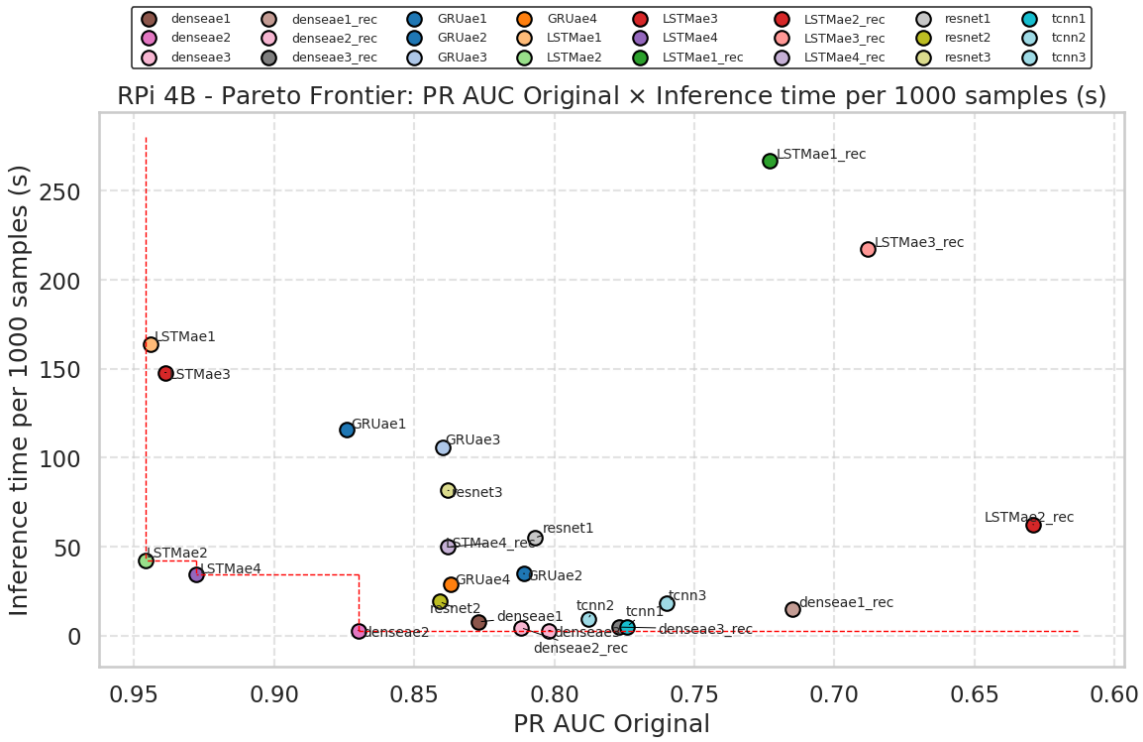


Figure 4.15: Pareto frontier of PR AUC and inference time on 1000 input sequences in seconds measured on Raspberry Pi 4B. X axis is inverted: better values are on the left. Solutions that make Pareto frontier are located in the left bottom corner (the more is model located to the bottom left corner, the better it is in this two objectives). Solutions laying on red line are creating Pareto frontier. LSTM-based solutions and denseae2 model dominates.

To even more demonstrate the influence of TF Lite not natively supporting LSTM and GRU layers or units (these models are executed using fallback kernels that are significantly slower and not optimized for Raspberry Pi hardware), images 4.16 and 4.17 show correlation heatmaps of models evaluation metrics for both cases: with LSTM and GRU based models and without them. In Figure 4.16 we can observe almost zero (-0.029) correlation between **inference time for 1000 input sequences** and **model size**. But after removing LSTM and GRU based anomaly detectors from this heatmap calculation, the correlation changes dramatically. In Figure 4.17, the correlation of the inference time and the size of the model is equal to 0.81, which is a relatively high correlation.

A correlation of 0.81 indicates a strong positive relationship between model size and inference time. In other words, larger models tend to require more time for inference, and this trend is consistent across the evaluated architectures (once LSTM/GRU models are excluded). Such a high value suggests that model size is a reliable predictor of inference time performance for TF Lite–compatible architectures.

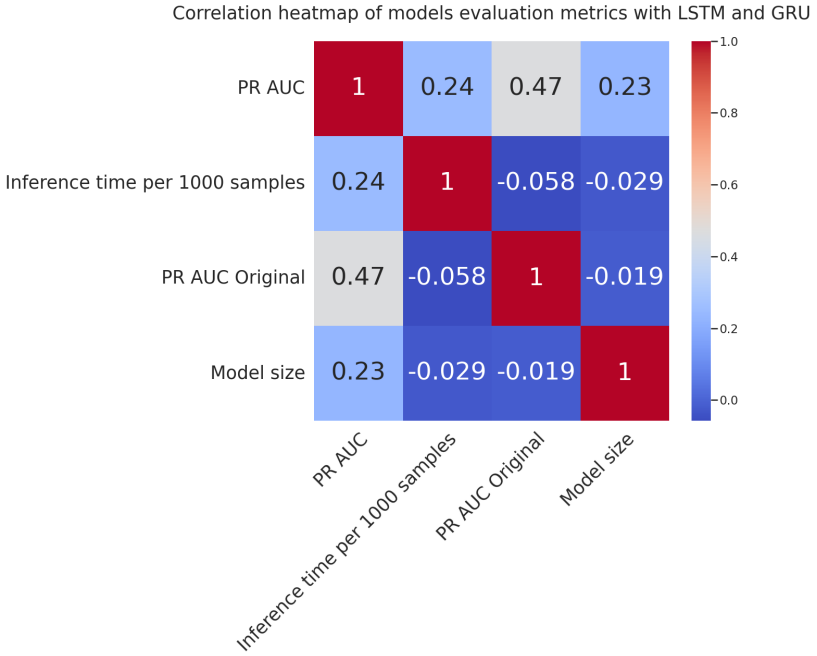


Figure 4.16: Heatmap shows correlation of metrics of models presented in Table 4.8. If slow LSTM and GRU based models included, then correlation between model size and inference time is almost zero (-0.029), resulting in no correlation.

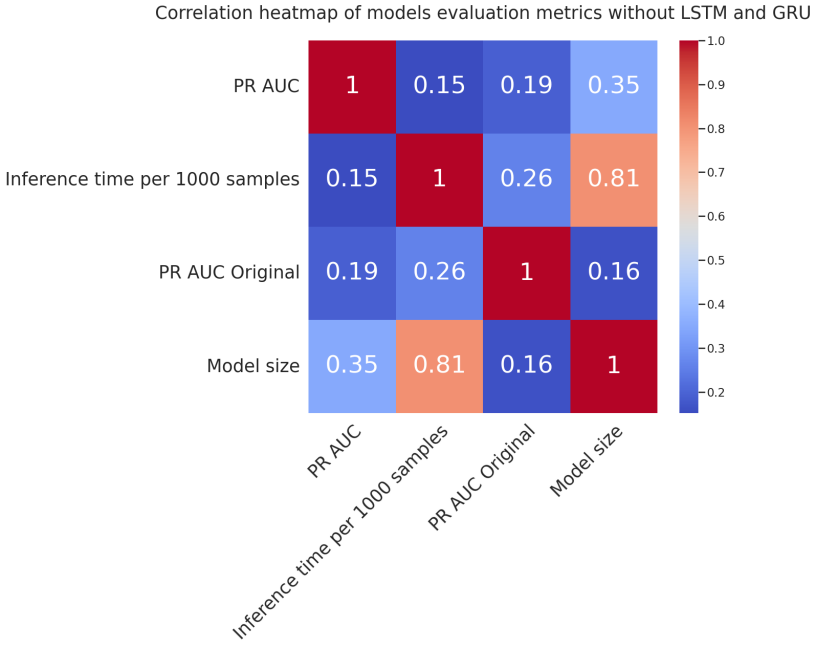


Figure 4.17: Heatmap shows correlation of metrics of models presented in Table 4.8, this time without LSTM and GRU models. Correlation between model size and inference time is notably higher (0.81).

One final comparison that can be observed from results in Table 4.8 is that GRU layers are a little bit faster than LSTMs. The graph in Figure 4.18 presents the times needed for inferences on 1000 input sequences between these two architectures. This highlights an important trade-off that may need to be considered in practical deployments: **speed** $\times$ **accuracy**, because GRU models are faster in each model instance, but LSTM models have a higher PR AUC score as was shown in Subsection 4.3.3. Consequently, the choice between these two similar architectures depends on the specific constraints and priorities of the target application, such as computational limits versus detection performance.

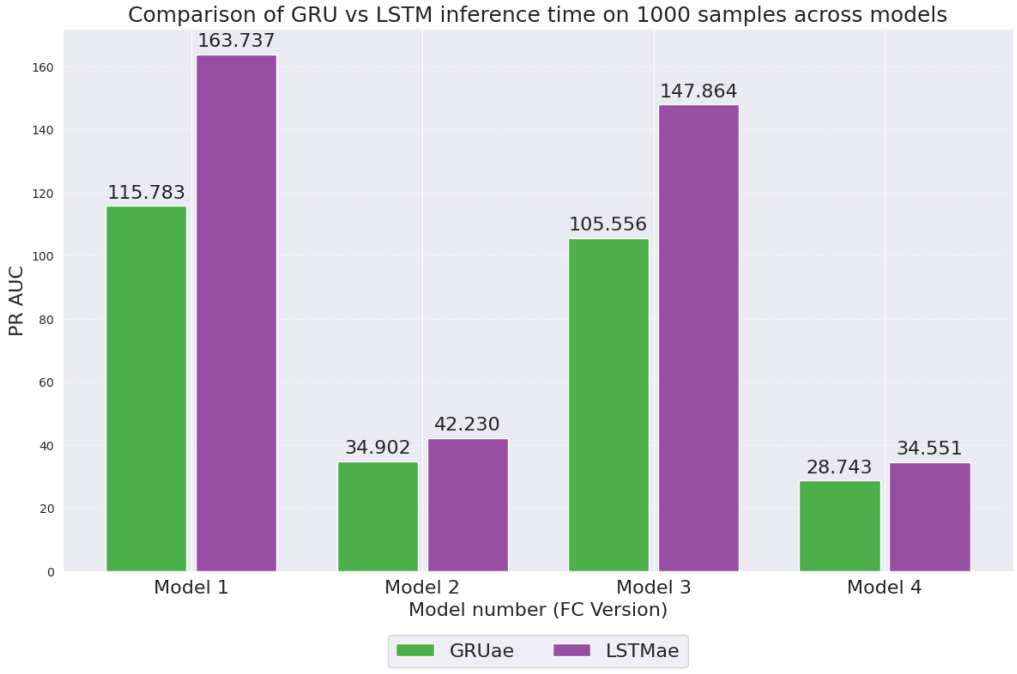


Figure 4.18: Measured times (in seconds) needed for inference on 1000 input sequences (samples) by GRU- and LSTM-based models. Smaller is better.

In practical applications, even a relatively high inference time may not pose a problem. The testing dataset used in this work assumes that every sensor is sampled once per second. In real-world scenarios, however, (a) sampling frequency often varies across sensors, and (b) many sensors do not sample data nearly as frequently. Even for a satellite with the same 20 sensors as in the generated dataset, the onboard computer would not be forecasting the next 50 telemetry values every second, as was done here for evaluation purposes.

Another important point is, that the inference times reported in Table 4.8 represent the time required to process **1000** input sequences. In other words, the listed values correspond to performing 1000 independent predictions of the next 50 steps for each telemetry signal. In real-world conditions, such extensive repeated forecasting is unnecessary, since predicting 50 future values is not required every second.

Even if we assume a satellite sampling frequency of $1\,s$, the actual time required to predict the next 50 values for the best-performing model (**LSTMae2**) will be:

$$\frac{42.23\,s}{1000} = 0.04223\,s = 42.23\,ms$$

which represents the time needed to compute the next 50 steps – still a very fast and practical inference speed for onboard deployment.

A very important factor for space-oriented applications is **power consumption**. This metric is not included in Table 4.8, as accurately measuring the power draw during model inference is non-trivial. Power measurements in Watts were performed using two devices: a **socket power meter** and a **USB energy tester**.

The socket power meter was only capable of measuring power with a precision of $1\,W$. This turned out to be insufficient, since both the standby consumption of the Raspberry Pi 4B and its consumption during inference remained within the range of $2 - 4\,W$, making any difference in power consumption impossible to observe. The evaluation using the USB energy tester was more informative, as this device provides readings with a precision of $0.01\,W$, enabling a more accurate assessment of power usage during model inference.

Because the USB energy tester provides only a small informative display of voltage, current and power, values of power consumption have been read manually. The illustrative Figure 4.19 of Raspberry Pi 4B with heatsinks on CPU and RAM, together with used USB energy tester is provided below. The observation resulted in inference power draw to be around $3.4\,W$ for any model, while stand by Raspberry Pi 4B consumption remained around $2.15\,W$.

Figure 4.19: Rapsberry Pi 4B with 4GB RAM used for measurements in this section. There is also a USB energy tester connected to power supply, which was used to read power drawn.

4.5 Anomaly detector architecture

The final design decisions and proposals for the anomaly detection architecture are described in this section. The subsections below bring together the key results of this chapter, covering both the posibilities of optimization methods and the final architecture of the anomaly detector. This section also outlines several practical considerations that must be addressed when deploying the proposed detectors in real-world scenarios.

Beyond the technical design, it is important to consider factors such as computational overhead, scalability, and robustness to noise, as these can significantly influence the operational performance of the system. Furthermore, integrating the anomaly detector into existing monitoring workflows require adjustments to thresholding strategies, alert frequency, and the handling of uncertain or borderline cases. Each satellite also carries different types of sensors and produces its own specific telemetry data, which creates the need to generate or obtain training data tailored to each individual satellite. The synthetic dataset generator created within this work can be utilized for that.

4.5.1 Optimization technique

As discussed at the end of the previous section, even best performing (highest PR AUC) LSTM and GRU models are relatively fast for their intended purpose. This might raise the question of whether further model optimizations are necessary: if the models are already fast enough, why optimize them at all? There are, however, several reasons to consider optimization. Three important ones are highlighted below:

1. **Unique use cases** – there might be satellites or board computers that can use only a small portion of CPU time for anomaly detection, so the forecasting must be done quickly.

2. **Deployment optimizations** – some spaceboards have techniques mentioned in Section 4.2 directly integrated into programs which handle models deployment. This way quantization or pruninng are done automatically, without external interventions into models code. Similar strategies can also be implemented for TF Lite models on Raspberry Pi devices.

3. **Energy efficiency** – optimized models generally consume less power during inference. For space applications, reducing energy consumption is critical, as satellites operate on limited power budgets and every watt saved can improve system longevity and reliability.

After considering this, builders of anomaly detectors can decide if extra optimization is needed for desired application or the one provided by board deployment software is sufficient enough.

4.5.2 Detection architecture

The experiments and measurements in the previous parts of this chapter provide a complex analysis of finding the optimal anomaly detector architecture. The best model with highest *PR AUC* metric was **LSTMae2** in forecasting mode, with PR AUC 0.946 on test dataset. Classical autoencoders also proved as reliant solutions since they provide good precision/recall with fast inference speeds and low memory requirements.

Multiple input sequence lengths were tested, although most models used 250 previous values to predict the next 50 steps. In practice, predicting more than 50 future steps for each telemetry indicator is generally inefficient, as forecasting 100 or 200 steps ahead would likely be highly uncertain and less reliable. In a particular application, future steps can be weighted as proposed in Equation 4.8, giving more realistic prediction results by assigning higher reliability to values closer in time, while predictions further into the future are considered less certain.

$$\text{predicted\_steps}(x_1, x_2, \ldots, x_N) = (a_1 x_1, a_2 x_2, \ldots, a_N x_N) \tag{4.8}$$

Variables $a_1, \ldots, a_N$ in Equation 4.8 are factor multiplicators, influencing the importance of predicted parts of future steps. The weighting scheme presented in this equation can be adapted depending on the mission requirements. For example, in situations where short-term forecasts are critical for real-time decision making, the first m weighting factors can be emphasized, while next $N - m$ factors can be reduced to reflect decreasing confidence in longer-term predictions. This approach allows tuning the anomaly detection system to prioritize reliability where it matters most.

Another key insight is that in real applications, sampling every $1\,s$ is neither necessary nor always desirable. Many satellite telemetry systems operate with variable sampling rates depending on the sensor type, mission requirements, or available bandwidth. Therefore, anomaly detectors must be flexible to handle irregularly sampled data or to aggregate measurements over longer intervals without losing critical information.

In summary, the analysis presented in this chapter highlights that the optimal anomaly detection system for satellite telemetry must balance accuracy, inference speed, and resource efficiency. The **LSTMae2** model in forecasting mode provides the highest *PR AUC*, while classical autoencoders offer a reliable alternative with lower computational and memory requirements. Practical deployment considerations (including flexible handling of sampling rates, weighting of future predictions, and robustness to sensor variability) are essential to ensure that the system performs reliably in real practice.

Chapter 5

Experiments on space-like edge devices

This chapter describes performance tests on space-like computational boards, in addition to the tests performed on the Raspberry Pi 4B with TensorFlow Lite models in the previous chapter. Experiments were run on multiple hardware boards to demonstrate the model's ability on different architectures. The following boards have been selected for experiments: **NVIDIA Jetson Orin Nano** and **AMD ZCU104**, both for CPU inference. Jetson Orin Nano also has a GPU, but it was not tested.

Section 5.1 describes the preprocessing steps and setups needed for successful deployment on those boards, including the conversion of Keras models into the ONNX format. ONNX models have been tested on Raspberry Pi 4B to obtain performance baseline against above mentioned boards. Subsection 5.2.1 is dedicated to this. Measurements performed on CPU of AMD ZCU104 are detailed in Subsection 5.2.2. NVIDIA Jetson Orin Nano tests are outlined in Subsection 5.2.3. The last Section 5.3 holds information about measured results on all boards with metrics comparisons.

5.1 Experiments setup

Spaceboards do not natively support full TensorFlow/Keras or TensorFlow Lite models, therefore the models must be converted into a compatible format before deployment. One of the most widely used formats for AI deployment on edge devices is ONNX[1]. **ONNX** (Open Neural Network Exchange) is an open standard for representing neural network models and enables interoperability between different deep learning frameworks and hardware platforms. It supports models originating from TensorFlow/Keras and is compatible with many edge devices.

Another commonly used execution backend is **TensorRT**[2] developed by NVIDIA. TensorRT is optimized for high-performance GPU inference and is widely used on NVIDIA-based edge platforms, such as the Jetson series. To preserve the same conditions for all boards, it was decided to run the tests on the CPU of the NVIDIA Jetson Orin Nano. Inference on the GPU would likely be even faster than on the CPU.

The above-mentioned options are the simplest solutions for converting the original model. The advanced approach includes the usage of FPGA (Field Programmable Gate Array)

[1] https://onnx.ai
[2] https://developer.nvidia.com/tensorrt

backend (for example Vitis AI [3] or FINN [4]). Since usage of FPGA requires additional work and skills and it is not that straightforward, it was decided to use only the ONNX format. As was mentioned in Subsection 4.5, the created detectors are fast enough even in the default format, so for the purpose of this thesis it is not necessary to use FPGA as the fastest possible solution.

The best model for each architecture category (dense, LSTM, ResNet and TCNN) was converted using the Python and **tf2onnx library** [24]. Experiments with converted models have been run on both testing boards. The results are presented in the following two sections. All models are forecasting-based. The most interesting metrics of these tests are:

- **inference speed on 1000 input sequences** – to show speedup againts Raspberry Pi 4B CPU inference with TensorFlow Lite versions of models,
- **PR AUC** – to see how inference performance changes when models are optimized for smaller devices,
- **power consumption** – important metric for all edge boards.

5.2 Tests on edge devices

Each edge board is introduced from a hardware perspective, including key technical specifications such as processor architecture or connectivity options. To provide a clearer context and improve readability, images of individual devices are also included.

5.2.1 Baseline: Raspberry Pi 4B ONNX

Raspberry Pi 4B with 4 GB of RAM, as described in Section 4.4, is equipped with a quad-core **ARM Cortex-A72** CPU (1.5 GHz). In the experiments presented in Section 4.4, TensorFlow Lite models were evaluated on this device. However, due to their relatively high computational and memory requirements, the tests had to be executed only on a subset of the available dataset. The full dataset could not be processed because of memory limitations of the platform, which would otherwise lead to excessive resource usage or instability during inference.

For this reason, Raspberry Pi 4B was chosen as a reference platform for establishing a baseline when evaluating the ONNX versions of the same models. By first measuring inference performance on this widely available embedded device, it becomes possible to create a consistent point of comparison for the subsequent experiments.

In the following sections, the hardware platforms used for evaluation are changed to different devices that more closely resemble space-grade or spaceboard-like computing platforms. Importantly, the neural network models themselves remain identical across all experiments. This ensures that any differences in inference time or performance can be attributed primarily to the hardware characteristics rather than variations in the models or datasets. Consequently, the results obtained on the Raspberry Pi can be interpreted as a CPU benchmark that serves as a reference for comparing the computational capabilities of the other evaluated devices.

[3] https://www.amd.com/en/products/software/vitis-ai.html
[4] https://github.com/Xilinx/finn

5.2.2 Spaceboard 1: AMD ZCU104

The **AMD ZCU104** development board (see Figure 5.1) is built around the **AMD Zynq UltraScale+ MPSoC**, which combines a multi-core **ARM Cortex-A53** CPU (1.5 GHz) processing system with programmable logic (FPGA) on a single chip. This heterogeneous architecture enables efficient acceleration of compute-intensive tasks, including machine learning inference, digital signal processing, and high-speed data acquisition. The ZCU104 features DDR4 memory, USB 3.0, Ethernet, and FMC expansion connectors, making it suitable for prototyping advanced embedded and edge applications. Thanks to the integration of programmable logic, developers can implement custom hardware accelerators to achieve low-latency and energy-efficient AI processing directly on the device [1].

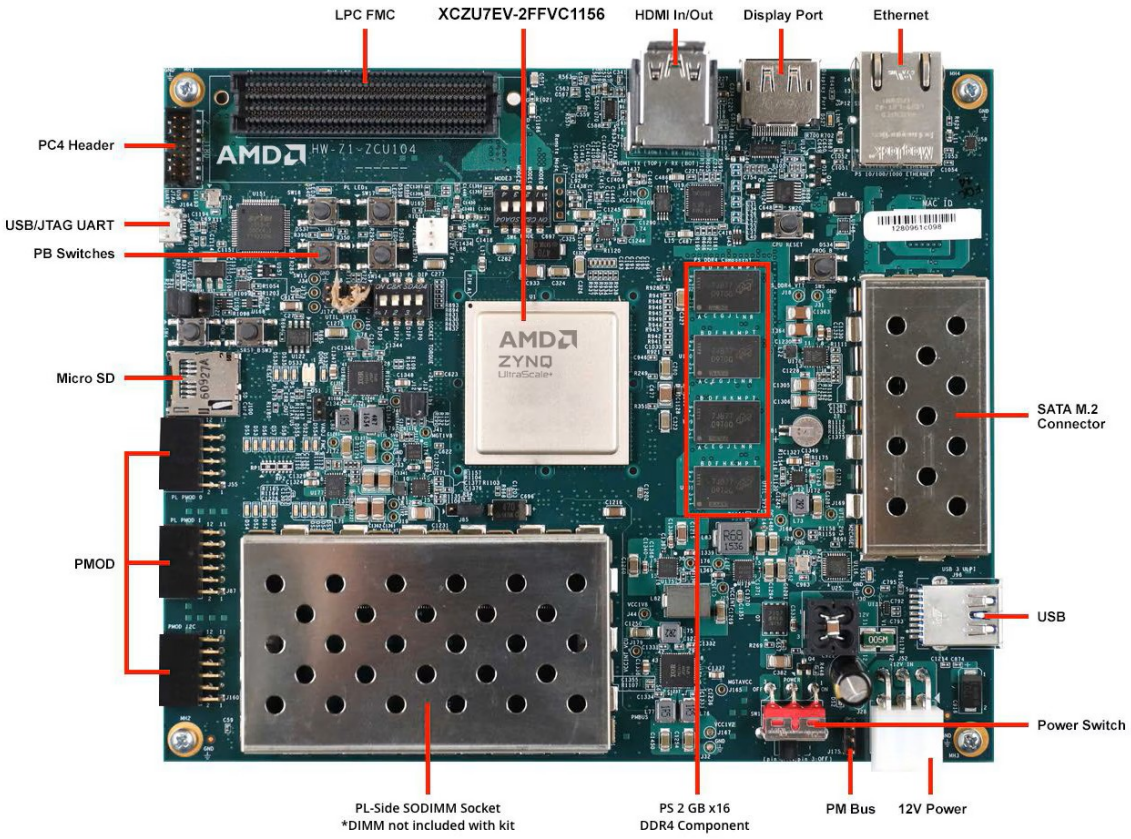


Figure 5.1: AMD ZCU104 evaluation kit. Source: [1].

5.2.3 Spaceboard 2: NVIDIA Jetson Orin Nano

The **NVIDIA Jetson Orin Nano** is a compact, high-performance AI edge computing platform from NVIDIA designed for embedded and robotic applications. It has a **6-core ARM Cortex-A78AE CPU** (1.7 GHz) paired with an NVIDIA Ampere-architecture GPU with Tensor Cores. It delivers up to tens of TOPS (trillions of operations per second) of AI inference performance in a small form factor with low power consumption (configurable from roughly 7 W to 25 W) [23]. Development kit build on NVIDIA Jetson Orin Nano is

presented in Figure 5.2. There are two versions of NVIDIA Jetson Orin Nano that differ in RAM size (4 or 8 GB). The 8 GB RAM device was used in the following measurements.

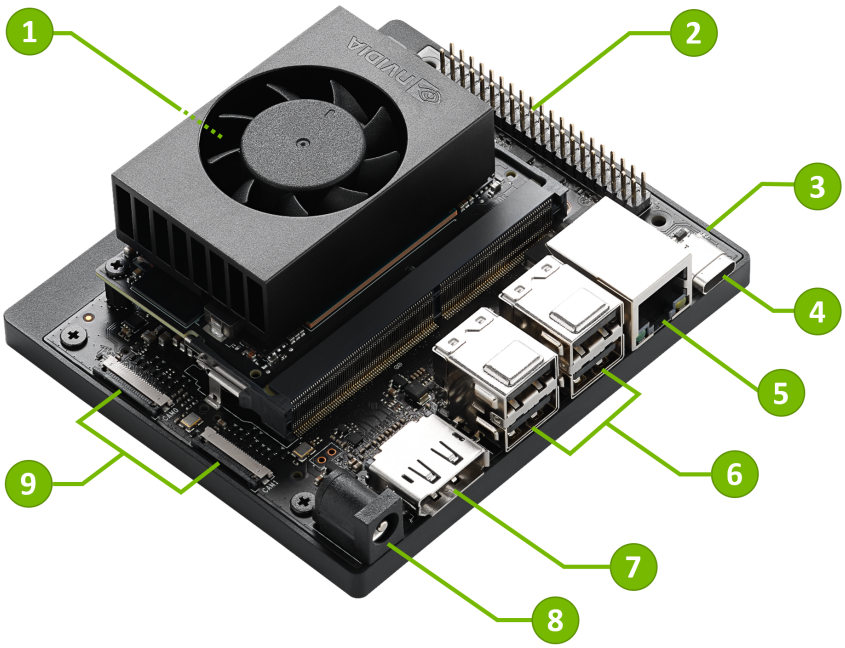


Figure 5.2: NVIDIA Jetson Orin Nano development kit. Board has a SD card slot (1), 40-pin expansion header (2), power indicator (3), USB-C (4), Ethernet (5), USB 3.1 type A (6) and DisplayPort (7) connectors. Power input is provided via DC jack (8). Camera can be attached to the development kit using interface (9). Source: [22].

5.3 Summary of experimental results

The results measured on the devices mentioned above are presented here. The graph in Figure 5.3 shows comparison of inference times for 1000 input sequences on all three devices. It's clearly obvious that NVIDIA Jetson Orin Nano has the fastest CPU among tested boards. It took only 0.052 s to process 1000 input sequences with the fastest model (**denseae2**) on this device. Even the slowest of LSTM autoencoders executed on the NVIDIA Jetson Orin Nano achieved an inference time below 1.5 s.

We can observe that differences between Raspberry Pi 4B and AMD ZCU104 are not that significant (except the LSTM autoencoder). The interesting fact here is that the fastest model on the fastest device (denseae2 on Orin Nano) is almost 186x faster than the slowest model on the slowest hardware (lstmae2 on AMD ZCU104). The results correlate with measurements made with the Raspberry Pi 4B and TensorFlow Lite models (see Table 4.8), where LSTM-based models have been slowest and Dense autoencoders were quickest.

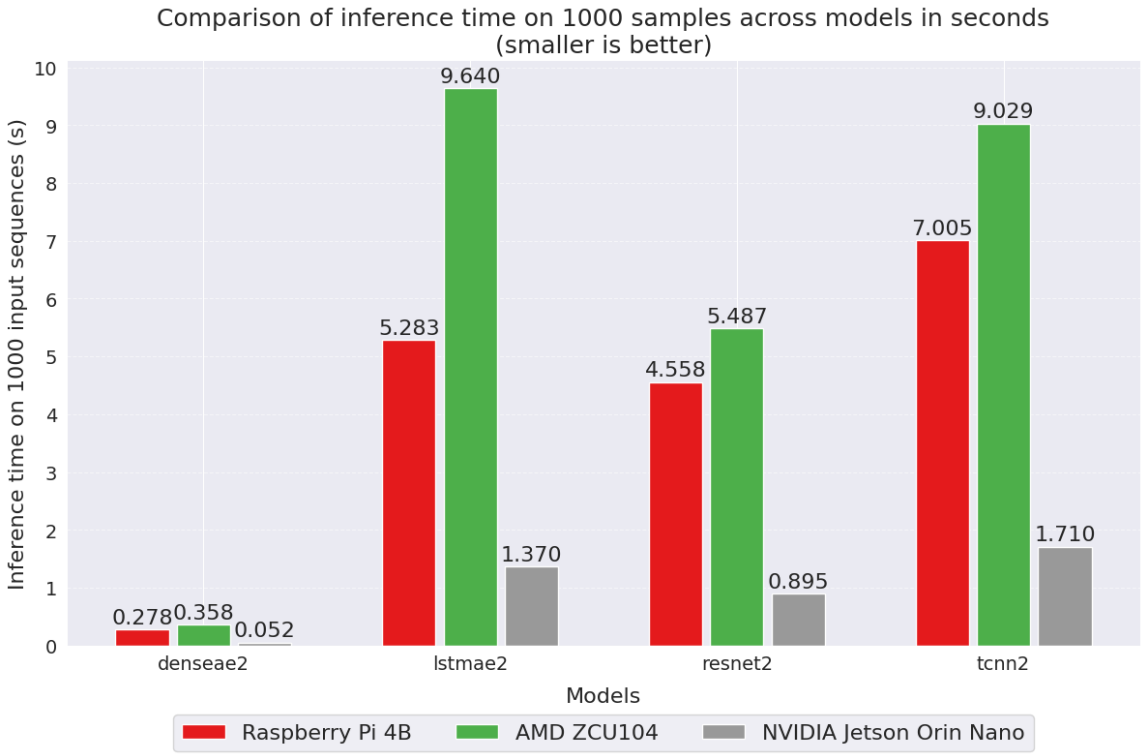


Figure 5.3: Comparison of inference times on CPUs for 1000 input sequences between Raspberry Pi 4B, AMD ZCU104 and NVIDIA Jeston Orin Nano.

The Table 5.1 measured results. Models have been tested on the same full size dataset with 86400 timestamps as the original models. It's obvious that model sizes shrinked with conversion in ONNX format. For example, the largest ResNet model had size of 97.93 MB (see Table 4.2) in the TensorFlow version, now the size of this model is 33.42 MB. Another interesting insight is that for models **resnet2** and **tcnn2** the metric PR AUC increased. This might be the result of different data preprocessing algorithms.

The power consumption of idle AMD ZCU104 board was 12.7 W, while running LSTM model it raised only to 13.04 W. The power consumption of an LSTM autoencoder running on a Jetson device was found to be 8.02 W, while the device's idle power drain was found to be 2.95 W. For the Raspberry Pi 4B, power drain stayed in the range of 3 – 5 W.

Model name	Model size [MB]	PR AUC	Original PR AUC
denseae2	6.35	0.850	0.870
LSTMae2	1.01	0.914	0.946
resnet2	33.42	0.877	0.841
tcnn2	1.85	0.854	0.788

Table 5.1: Metrics of ONNX versions of anomaly detection models. Original PR AUC is taken from full size TensorFlow models.

The results presented demonstrate that converting anomaly detection models to the ONNX format allows them to be deployed efficiently across different edge and embedded

computing platforms while maintaining competitive detection performance. Of the devices tested, the NVIDIA Jetson Orin Nano delivered the best inference performance, significantly outperforming the Raspberry Pi 4B and the AMD ZCU104. Dense autoencoder models were the most computationally efficient across all platforms, whereas LSTM-based models consistently required longer inference times. In addition to improving performance, ONNX conversion further reduced model sizes, which is advantageous for deployment in resource-constrained environments (although models are already quite small).

Chapter 6

Real mission tests – ESA AD

The ESA Anomaly Dataset [15] was selected as a representative of the real world dataset for testing purposes as mentioned in Subsection 3.3.1. Data from this dataset are anonymized – channel names have been replaced with plain numbers. This means that anomaly detection can be performed but one don't really know specifically what anomaly means. Data from Channel 12 have been selected and visualized as an example (see Figure 6.1). It can be seen that there are around 14 years of data from Mission 1 with varying sampling frequency. The dominant sampling frequency for these data is 0.033 Hz.

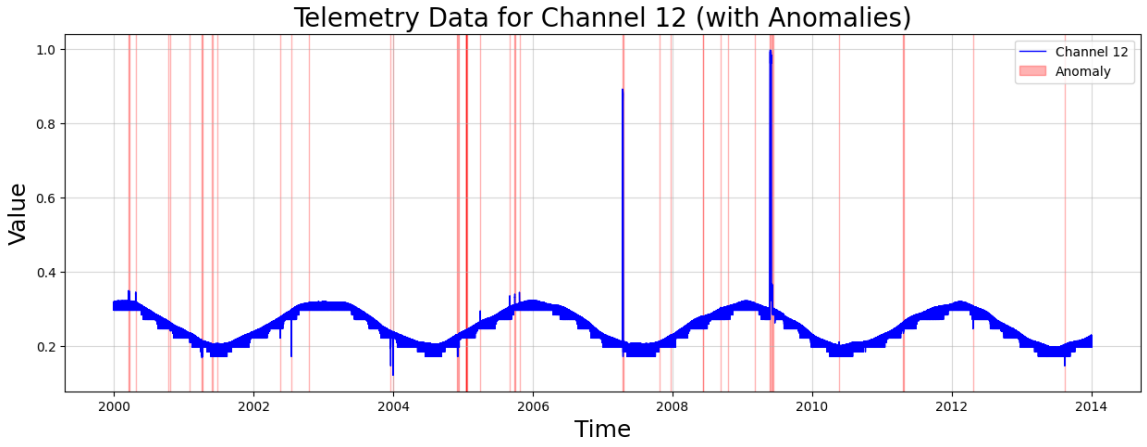


Figure 6.1: Example of data from original ESA AD dataset. The graphs shows Channel 12 from ESA AD with highlighted anomalies. Channel 13 looks similarly.

Because working with the full dataset requires substantial computational resources – approximately 512 GB of free disk space and at least 64 GB of RAM, according to information provided in the official repository[1] – a subset of the data from the Kaggle competition [16] was used in this work.

In this subset, the data are already preprocessed and stored in `.parquet` files, which simplifies loading and handling of the dataset. The training dataset contains approximately

[1] https://github.com/kplabs-pl/ESA-ADB?tab=readme-ov-file

15 million records, while the testing dataset contains 521,280 records representing previously unseen data. Anomaly detection is evaluated across 58 telemetry channels.

The Kaggle competition also allows comparison of results with those of other participants. Although the competition officially ended on June 11, 2025, the platform still allows late submissions, making it possible to obtain a score and compare model performance with previously submitted solutions.

6.1 Dataset preprocessing

The challenge focuses specifically on 58 target channels, which are detailed in the **target_channels.csv** file. The training data features binary labels (0 for nominal, 1 for anomalous) and encompasses three event categories: anomalies, nominal rare events, and communication gaps. Communication gaps are excluded from the test dataset.

It was necessary to perform downsampling of the data to shrink the size of the dataset to a usable size. After a few attempts, the best tradeoff between size and loss of information, the downsampling factor 10 was selected. The mean of each 10 values was kept. Because the model must learn standard operational baselines, only non-anomalous data was retained for training. The preprocessing script generated input sequences with the shape `(number of sequences, sequence length, number of channels)`, where the sequence length was retained from LSTM-based model (250) and the number of channels was 58 (channels from **target_channels.csv** file). The target arrays were formatted as `(number of sequences, next steps, number of channels)`. The *next steps* parameter was initially set to 50 but was reduced iteratively during the development process.

For competition submissions, predictions must be formatted as binary values (0 or 1) per timestamp. The Kaggle evaluation system splits the test dataset into a public leaderboard (approximately 35 %) and a private leaderboard (the remaining 65 %). Because of this separation, each submission is evaluated twice, so each competitor gets two results. Submissions are evaluated using the corrected event-wise F0.5 score for anomaly detection in time series. This metric calculates the harmonic mean of corrected event-wise precision and recall, heavily prioritizing precision by assigning it twice the weight of recall [16].

6.2 Forecasting-based approach

The forecasting-based LSTM autoencoder (**lstmae2 model**) was selected as the primary baseline architecture for the ESA AD Kaggle challenge. This architecture was chosen due to its performance during tests on synthetic datasets. The initial configuration had a two-layer encoder structure with 128 and 64 neurons, respectively, 32-neuron latent space bottleneck. The model was designed to process an input sequence length of 250 timestamps and forecast the next 50 values.

This approach did not achieve notable results when applied to the complex, real-world spacecraft telemetry of the Kaggle dataset. The number of predicted steps was iteratively reduced from 50 to 1 (multi-step forecasting in highly dynamic, multivariate telemetry likely led to compounding prediction errors, making it impossible to establish a reliable threshold for anomaly detection). Even after applying this change and other changes to autoencoder structure (including increasing network depth by adding layers, introducing dropout mechanisms for regularization, and adjusting neuron counts across the hidden layers), results did not get better.

The best performance achieved with this specific architecture was an $F_{0.5}$ score of **0.084** on the public part and **0.191** on the private part. There is a significant contrast between the model's success on synthetic data with 20 channels and its failure on the Kaggle dataset. This emphasizes the complexity, noise, and non-stationarity of actual spacecraft telemetry compared to controlled synthetic environments. Given the consistent lack of progress during this experimental phase, after numerous attempts to improve results, it was decided to move on with reconstruction based model.

6.3 Reconstruction-based approach

Reconstruction-based autoencoders use only input data, they do not require targets (as stated in 2.4). The training data sequences created from the provided training dataset had the following shape: (255180, 250, 58). Using those data, model based on lstmae2_rec from Chapter 2 with slight changes in the architecture (added Dropout layers, LSTM encapsulated into Bidirectional layers – see Figure 6.2) has been trained. Due to a very large dataset, the model found an optimum in a few epochs. The model used Adam optimizer, *mean absolute error* as a loss function, and batch size 256.

The model achieved a $F_{0.5}$ score **0.664** in the public part of the dataset and **0.475** in the private part of the dataset (see Figures 6.3 and 6.4 for the top competition results). These results mean that if the challenge was still open, the created LSTM autoencoder would have achieved a 14. place in the public part and 20. place in the private part.

Performance boundaries have been found for this type of anomaly detector; further improvements would require better data preprocessing, greater training time, and different anomaly detection architecture. The resulting scores of 0.664 and 0.475 represent a fair result considering the size of the test dataset and the rarity of the anomalies in it (around 1 – 2 %).

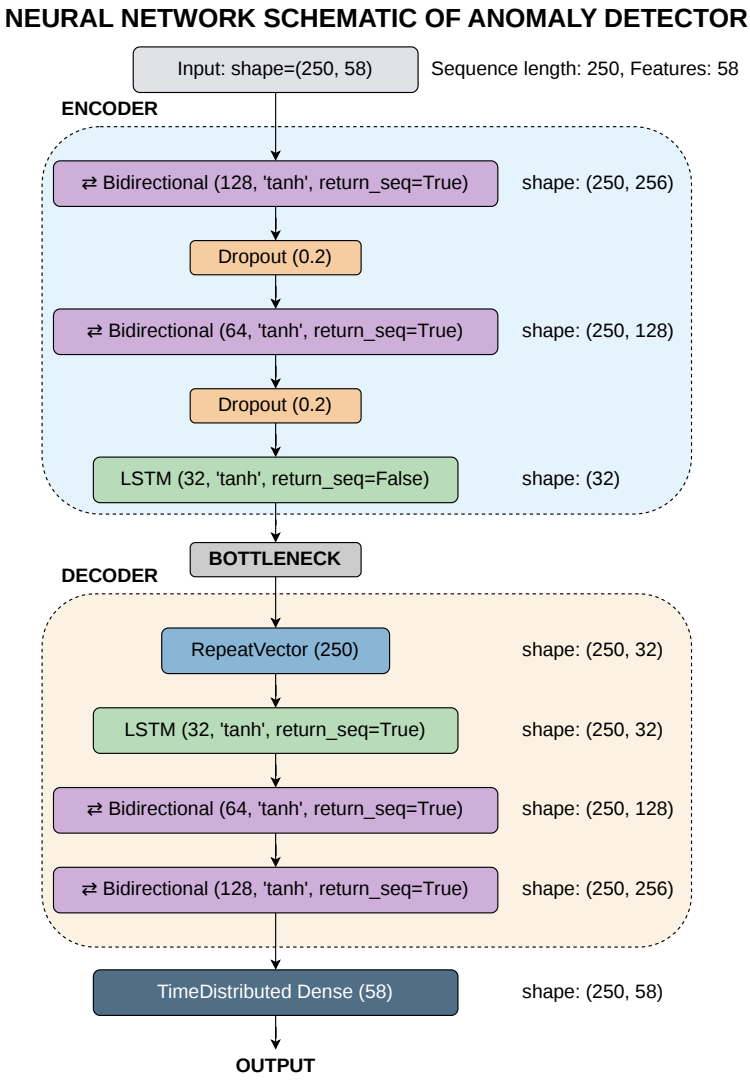
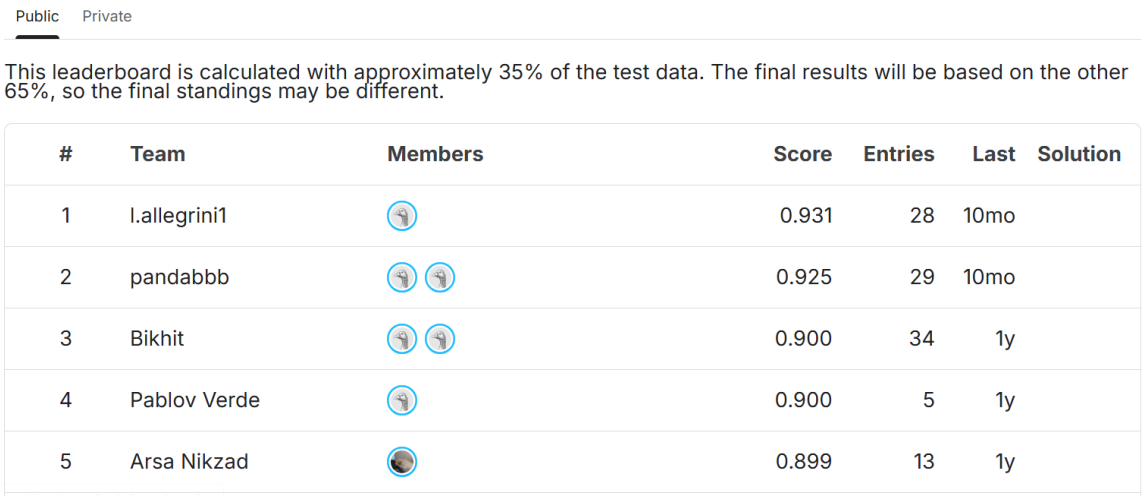


Figure 6.2: Final architecture scheme of best-result LSTM reconstruction based autoencoder in ESA AD Kaggle challenge.

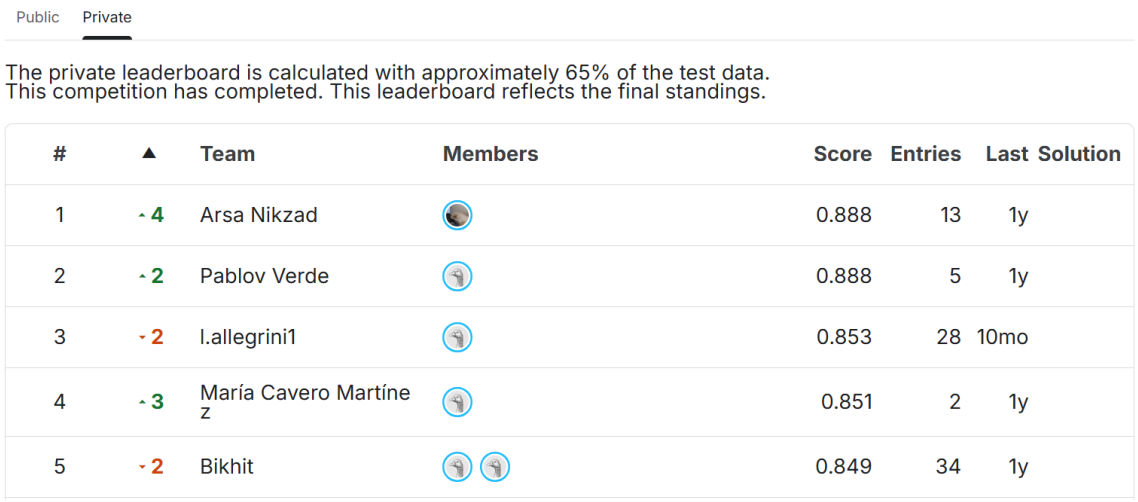


Public Private

This leaderboard is calculated with approximately 35% of the test data. The final results will be based on the other 65%, so the final standings may be different.

#	Team	Members	Score	Entries	Last	Solution
1	l.allegrini1		0.931	28	10mo	
2	pandabbb		0.925	29	10mo	
3	Bikhit		0.900	34	1y	
4	Pablov Verde		0.900	5	1y	
5	Arsa Nikzad		0.899	13	1y	

Figure 6.3: Top 5 results from public part of the leaderboard from Kaggle competition [16]. Created model achieved score 0.664 and would be ranked 14th.



Public Private

The private leaderboard is calculated with approximately 65% of the test data.
This competition has completed. This leaderboard reflects the final standings.

#	▲	Team	Members	Score	Entries	Last	Solution
1	▲ 4	Arsa Nikzad		0.888	13	1y	
2	▲ 2	Pablov Verde		0.888	5	1y	
3	▼ 2	l.allegrini1		0.853	28	10mo	
4	▲ 3	María Cavero Martínez		0.851	2	1y	
5	▼ 2	Bikhit		0.849	34	1y	

Figure 6.4: Top 5 results from private part of the leaderboard from Kaggle competition [16]. Created model achieved score 0.475 and would be ranked 20th.

Chapter 7

Anomaly exploration and reporting

This chapter completes the end-to-end processing pipeline. Building on telemetry data generation (Section 3.4) and the model design and training (Chapter 4), it addresses the final stage – interpreting anomaly detection results to provide meaningful explanations. The chapter focuses on a modular exploration and reporting system for anomaly detectors. The system is designed to load test data and predictions (or calculate them if needed), and provides an interface for custom modules which should attempt to explain why anomalies happened and how similar issues could be avoided in the future. The first part of this chapter (Section 7.1) reviews existing approaches and introduces the theoretical background of anomaly exploaration/explanation. The modules of the proposed anomaly exploration system can be designed based on the findings discussed in this section.

Section 7.2 then describes the implemented anomaly exploration and reporting system. Since this problem is active research field and author of this thesis is not an expert on space and satellite sensors, the proposed system follows a modular architecture that allows developers with better expertise to easily extend it by adding their own report-generation Python scripts (modules).

This chapter does not focus on neural network explainability frameworks such as SHAP (SHapley Additive exPlanations)[1] or LIME (Local Interpretable Model-Agnostic Explanations) [17]. Instead, explainability in this context of exploration refers to identifying why anomalies occurred based on relationships in the data, rather than explaining why the internal components of a neural network classified a specific data point as anomalous.

7.1 Anomaly exploration theory

Before introducing the proposed system, this section first covers the theoretical background and existing approaches to anomaly explanation and exploration. Yepmo at el. [39] propose four categories for anomaly explanation methods:

- **Explanation by feature importance** – anomalies are explained by identifying the features that contributed most to the detection of the anomaly.
- **Explanation by feature values** – there is no single feature that clearly separates the anomalous point from others, but the anomaly can be characterized by specific ranges of feature values.

[1] https://shap.readthedocs.io/en/latest/

- **Explanation by data point comparison** – anomalies are explained by comparing the anomalous data point with other data points, such as normal prototypes or similar instances.
- **Explanation by structure analysis** – the intrinsic structure of the dataset is analyzed (e.g., clusters or groups of normal data), and anomalies are explained by how they differ from this structure.

Based on this categorization, the modules of the proposed anomaly exploration system primarily focus on approaches related to feature values, data point comparison, and structural analysis. These methods are well suited for time-series telemetry data, where anomalies often arise from unusual combinations of values across multiple channels or from deviations from typical patterns.

A different approach was taken by Qi et al. [28]. They used metadata of images to explain anomaly occurrence. Traffic camera images were used as the primary data and additional information (annotations), such as weather conditions or time of day, served as metadata. Their goal was to find conditions on metadata that describe subsets of anomalies (which are rare) with high precision. In their framework, anomaly detection is performed separately and the proposed method focuses only on explaining the detected anomalies. They developed an algorithm designed to learn high-precision rules describing anomalies, for example: (weather = snow) AND (time = rush hour). These rules characterize groups of anomalous events rather than individual anomalies. Their approach can generate human-readable explorations for a large fraction of anomalies. However, the performance of the method strongly depends on the availability of a sufficiently large dataset and informative metadata features, which are critical for producing meaningful explanations.

Explanation systems are used in the space sector as well. Katsube and Sahara [14] proposed a **Telemetry Monitoring System** that uses feature engineering to not only detect but also explain anomalies . The system extracts features based on moving averages, telemetry periods, and waveform differences, and summarizes them using the Mahalanobis distance. Crucially, the system provides explainability by calculating the contribution of each feature to the final anomaly score. This allows operators to identify both the specific type of anomaly (trend, periodic, or waveform) and the exact telemetry parameter causing it. The authors successfully validated their approach using both artificial datasets and real operational data from the Suzaku satellite.

Schefels et al. [30] from the German Aerospace Center (DLR) used causal inference on time-series data to provide explainability for satellite anomalies. They utilized the PCMCI+ and J-PCMCI+ algorithms for causal discovery. The found relationships are presented using causal graphs, which clearly show how satellite components affect each other. The results from those studies help satellite operators to better understand and diagnose these anomalies.

7.2 Anomaly exploration and report system

In this section, a Python-based modular wrapper for the exploration of anomalies and the generation of reports is presented. The general workflow of AERS (**Anomaly Exploration Report System**) is illustrated in Figure 7.1.

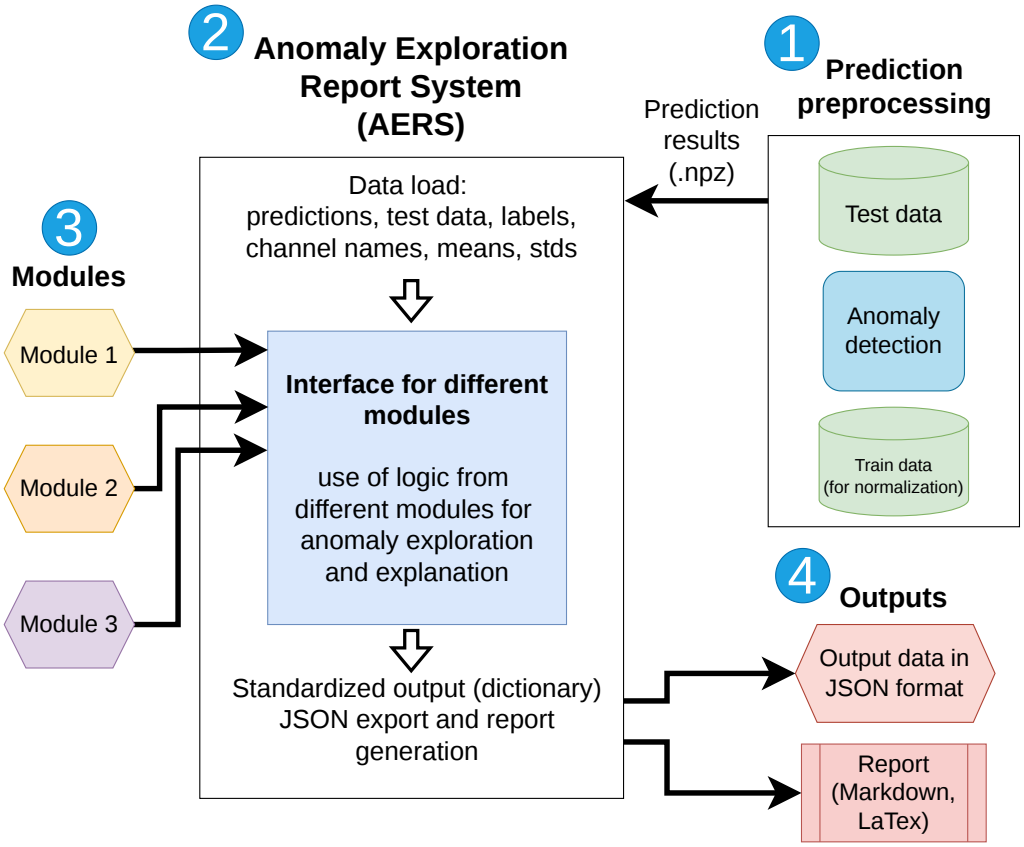


Figure 7.1: The figure shows whole system scheme. (1) Predictions are made by anomaly detection model and passed with other statistics and original data to AERS (2) in `.npz format`. AERS handles data and modules (3) load and runs modules code. In the end, outputs are generated in selected format + JSON (4). Modules receive data in standardized form through **analyze** function they inherit from **base module class**. Their output must be a Python dictionary with given rules. This modular architecture of whole system enables users to create their own modules and to run anomaly detection with any model or method they want.

Unlike traditional explainability methods (e.g., feature attribution techniques), AERS modules focus on identifying interpretable patterns, statistical relationships, and conditions under which anomalies occur. This approach allows for more flexible and domain-specific explorations and anomaly explanations that are not tied to a particular model architecture. AERS is composed of three main components:

- **predictor** – performs inference using a given anomaly detection model, applies data normalization, and generates a `.npz` file required as input for the exploration system. This file contains predictions, test data, labeled anomalies, channel (column, feature) names, and normalization statistics (mean and standard deviation). Formally, the `.npz` file must include those elements (it is expected that feature-related elements all have the same shape):

  - predictions
  - test_data
  - test_labels
  - feature_names
  - feature_means
  - feature_deviation

- **anomaly exploration and report interface** – manages data and module loading, and produces reports in Markdown or LaTeX format, as well as machine-readable JSON,

- **exploration/explanation modules** – each module implements a unified interface via the `analyze` function, which receives test data, predictions, ground-truth labels, and normalization statistics. This design ensures that all modules operate on the same standardized input, enabling consistent and comparable analysis across different explanation strategies. Users can create their own modules with specific logic, provided they inherit from the `BaseModuleExplainer` class. Each module must return results in a predefined format (Python dictionary). The output has mandatory fields such as `title`, `summary`, `details`, and `plots`, allowing seamless integration into the reporting pipeline.

Separation of inference and exploration components allows users to employ any type of anomaly detection method, including neural networks, rule-based approaches, or tree-based models. The use of a standardized `.npz` file as an intermediate representation decouples the prediction and exploration/explanation stages. This enables interoperability between different anomaly detection models and exploration modules, regardless of their internal implementation. The quality of exploration depends on the design of individual modules and the availability of labeled anomalies. Furthermore, the interpretability of results may vary depending on the complexity of the analyzed data. The complete repository containing the exploration system pipeline is available on GitHub [25] under the MIT License.

To demonstrate how modules for AERS could be made, two modules have been created:

- Deviation ranker – computes standard deviation value for all channels in the time of an anomaly. It plots five channels with the highest standard deviation. The AERS output of this module is shown in Figure 7.2.

- Graph ploting – provides visualization of detected anomalies for each sensor (channel) using signal plots. Figure 7.3 shows telemetry signal of **battery voltage** with red sector marking a detected anomaly.

As outlined earlier, users are able to extend the system by implementing their own modules with the required functionality. This flexibility allows the framework to adapt to a wide range of use cases and anomaly detection models without imposing strict constraints on the underlying methods. The proposed system primarily serves as a unifying interface for these modules, providing a lightweight wrapper with minimal dependencies that simplifies their integration, execution, and interaction.

In this way, the system not only standardizes the process of anomaly exploration and report generation but also encourages modularity and extensibility. Users can incorporate custom logic for data processing, explanation and exploration strategies, or visualization

techniques while relying on the core framework to handle data flow and output generation. This design ensures that the system remains both practical for immediate use and adaptable for future extensions.

Overall, this chapter has completed the processing pipeline by addressing the final step – interpreting and explaining detected anomalies. It introduced the theoretical background of data-driven anomaly exploration and presented a modular system that operationalizes these ideas. Altogether, this gives a solid base for creating clear and useful reports that help users understand what caused anomalies and how to avoid similar problems in the future.

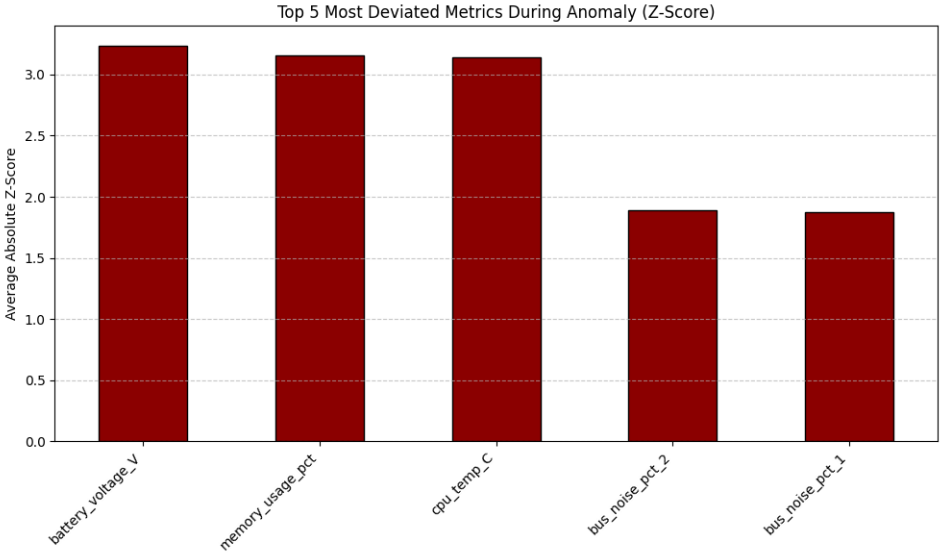


Figure 7.2: Graph output from deviation module used in AERS. Plot shows 5 channels with highest deviation values. This figure is raw output of the module, no postprocessing edits have been applied on it.

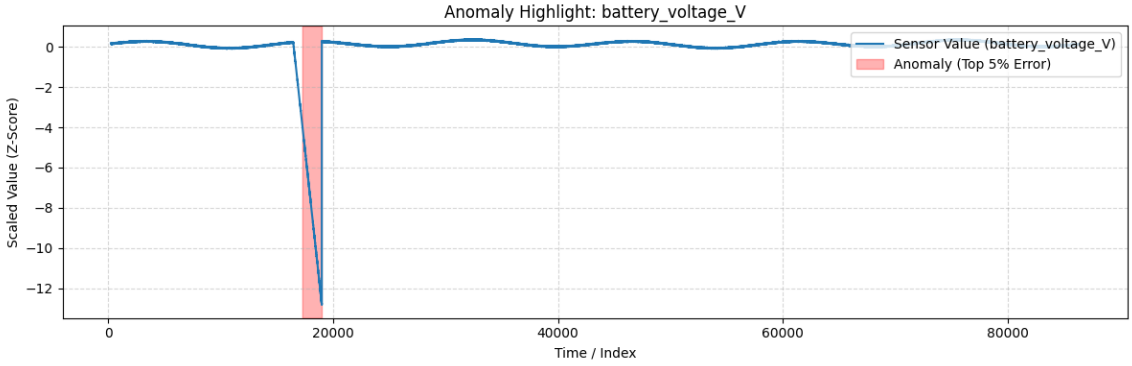


Figure 7.3: Graph of battery voltage from graph plotting module. Anomaly threshold was set to 95th percentile. Detected anomaly is marked red. This figure is raw output of the module, no postprocessing edits have been applied on it.

Chapter 8

Conclusion

The objective of this thesis was to design and evaluate a satellite telemetry anomaly detection system based on neural network architectures. As a prerequisite, existing detectors and state-of-the-art models were reviewed. Publicly available satellite telemetry datasets, such as NASA SMAP & MSL, ESA AD and OPS-SAT, were also examined. Based on this analysis, several new detectors were developed using different neural network architectures (LSTM, TCNN, autoencoders, and models with residual connections).

The existing datasets do not provide sufficient documentation regarding individual signals and sensors. To address this limitation, synthetic (and well-labeled) data were required. For this purpose, a synthetic dataset generator was developed to support both dataset creation and the evaluation of detection models. The generator is can create two categories of anomalies: sensor anomalies (malfunctions) and environmental anomalies (e.g., radiation storms). In total, it simulates 13 types of sensors, generating signals across 20 channels (in default configuration).

Multiple versions of neural network models (based on above mentioned architectures) have been developed and tested throughout this thesis. The main difference between them, apart from the chosen architecture, is in their approach to anomaly detection – either by **predicting** the next N signal values or by **reconstructing** the entire original signal sequence and comparing it with the ground truth (real signal).

The developed LSTM-based autoencoder achieved the highest PR AUC (area under the precision–recall curve), reaching a value of **0.946**. Another strong performing model was the classical (fully connected) autoencoder, which achieved a PR AUC of 0.870. For both architectures, the forecasting variants outperformed the reconstruction-based versions on the synthetic dataset with 20 channels. In contrast, the ResNet-based and TCNN-based detectors achieved lower PR AUC scores and did not perform as well as the autoencoder-based approaches. Experiments conducted on real mission data from ESA AD demonstrated that, for a higher number of channels (58 in this case), the forecasting approach may reach its practical limits. In the Kaggle challenge, reconstruction-based anomaly detection achieved better results, significantly outperforming the models that had performed best on the smaller (lower-dimensional) synthetic dataset.

Inference time is important metric for space sector where computational time can be critical. It was shown that using models created in TensorFlow and then converted to TensorFlow Lite (TF Lite) could have a significant influence on inference speed. Mainly because of how certain neural network block/layers are (not) supported in TF Lite. LSTM-based models performed relatively slowly compared to dense autoencoders. Replacing LSTMs with GRU blocks didn't bring any significant speed improvements.

Conversion of the models to the ONNX format further accelerated the detection process. Experiments conducted on three testing platforms (Raspberry Pi 4B, AMD ZCU104, and NVIDIA Jetson Orin Nano) explored real-world deployment scenarios and provided practical measurements of inference conditions.

To complete the overall detection pipeline, an anomaly exploration and reporting system was introduced in the final chapter. This system provides an interface that enables domain experts to develop custom modules aimed at explaining detected anomalies.

Throughout this work, two projects hosted on GitHub have been created. The first repository addresses the problem of unavailability of well documented satellite telemetry and the second presents a tool for anomaly exploration. Future work may focus on various improvements. The first one is an onboard retraining of the detection models – adaptation of the detection model to a new data (changes in telemetry) directly on satellite. The test of model abilities on real-life scenario data (not only ESA AD) or usage of FPGA to achieve even higher acceleration of model inference speed could be seen as another proposed future work. The final section for improvement is the synthetic anomaly dataset generator. It was demonstrated that models trained on synthetic data did not perform that well on real data from the ESA Kaggle challenge. Further tweaks to sensor signals and generator parameters could improve the performance of the models.

This work was presented at two conferences: Excel@FIT 2026 (poster) and Brno Space Student Conference 2026 (presentation).

Bibliography

[1] AMD. *AMD Zynq™ UltraScale+™ MPSoC ZCU104 Evaluation Kit*. 2026. Available at: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/zcu104.html.

[2] ASTROPAT. *NASA Anomaly Detection Dataset SMAP & MSL*. Oct 2022. Available at: https://www.kaggle.com/datasets/patrickfleith/nasa-anomaly-detection-dataset-smap-msl.

[3] BONIOL, P.; LIU, Q.; HUANG, M.; PALPANAS, T. and PAPARRIZOS, J. *Dive into Time-Series Anomaly Detection: A Decade Review*. 2024. Available at: https://arxiv.org/abs/2412.20512.

[4] CHANDOLA, V.; BANERJEE, A. and KUMAR, V. Anomaly Detection: A Survey. *ACM Comput. Surv.*, July 2009, vol. 41.

[5] CHENG, H.; ZHANG, M. and SHI, J. Q. *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*. 2024. Available at: https://arxiv.org/abs/2308.06767.

[6] COOLS, A.; BELARBI, M. A. and MAHMOUDI, S. A. Benchmarking of Anomaly Detection Methods for Industry 4.0: Evaluation, Ranking, and Practical Recommendations. *Big Data and Cognitive Computing*, 2025, vol. 9, no. 5. ISSN 2504-2289. Available at: https://www.mdpi.com/2504-2289/9/5/128.

[7] FEJJARI, A.; DELAVAULT, A.; CAMILLERI, R. and VALENTINO, G. A Review of Anomaly Detection in Spacecraft Telemetry Data. *Applied Sciences*, 2025, vol. 15, no. 10. ISSN 2076-3417. Available at: https://www.mdpi.com/2076-3417/15/10/5653.

[8] GEEKSFORGEEKS. *Autoencoders in machine learning*. July 2025. Available at: https://www.geeksforgeeks.org/machine-learning/auto-encoders.

[9] GEEKSFORGEEKS. *Introduction to Recurrent Neural Networks*. 2025. Available at: https://www.geeksforgeeks.org/machine-learning/introduction-to-recurrent-neural-network. [Accessed 04-09-2025].

[10] GEEKSFORGEEKS. *Tf.keras.layers.GRU in TensorFlow*. 2025. Available at: https://www.geeksforgeeks.org/deep-learning/tf-keras-layers-gru-in-tensorflow/. [Accessed 30-11-2025].

[11] GHOJOGH, B. and GHODSI, A. *Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey*. 2023. Available at: https://arxiv.org/abs/2304.11461.

[12] Huang, H.; Wang, P.; Pei, J.; Wang, J.; Alexanian, S. et al. *Deep Learning Advancements in Anomaly Detection: A Comprehensive Survey*. 2025. Available at: https://arxiv.org/abs/2503.13195.

[13] Hundman, K.; Constantinou, V.; Laporte, C.; Colwell, I. and Söderström, T. Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding. *CoRR*, 2018, abs/1802.04431. Available at: http://arxiv.org/abs/1802.04431.

[14] Katsube, S. and Sahara, H. Telemetry Monitoring System with Features Explaining Anomalies Based on Mahalanobis Distance. *International Journal of Prognostics and Health Management*, June 2024, vol. 15.

[15] Kotowski, K.; Haskamp, C.; Andrzejewski, J.; Ruszczak, B.; Nalepa, J. et al. *European Space Agency Benchmark for Anomaly Detection in Satellite Telemetry*. 2025. Available at: https://arxiv.org/abs/2406.17826.

[16] Kotowski, K.; Ruszczak, B.; Quarantiello, L.; Shendy, R.; Nalepa, J. et al. *Spacecraft Anomaly Challenge on ESA dataset*. 2025. Available at: https://kaggle.com/competitions/esa-adb-challenge. Kaggle.

[17] Kricheff, S.; Maxwell, E.; Plaks, C. and Simon, M. An Explainable Machine Learning Approach for Anomaly Detection in Satellite Telemetry Data. In: *2024 IEEE Aerospace Conference*. 2024, p. 1–14.

[18] Murphy, J.; Buckley, M.; Buckley, L.; Taylor, A.; O'Brien, J. et al. Deploying Machine Learning Anomaly Detection Models to Flight Ready AI Boards. In: *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2024, p. 6828–6836.

[19] Murphy, J.; Ward, J. and Mac Namee, B. An Overview of Machine Learning Techniques for Onboard Anomaly Detection in Satellite Telemetry*. In:. October 2023, p. 1–6.

[20] Nagel, M.; Fournarakis, M.; Amjad, R. A.; Bondarenko, Y.; Baalen, M. van et al. *A White Paper on Neural Network Quantization*. 2021. Available at: https://arxiv.org/abs/2106.08295.

[21] Neo, M. *A comprehensive guide to neural network model pruning*. Feb 2024. Available at: https://datature.io/blog/a-comprehensive-guide-to-neural-network-model-pruning.

[22] NVIDIA. *Jetson Orin Nano Developer Kit Getting Started Guide*. 2026. Available at: https://developer.nvidia.com/embedded/learn/get-started-jetson-orin-nano-devkit.

[23] NVIDIA. *NVIDIA Jetson Orin*. 2026. Available at: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[24] ONNX. *Tf2onnx – Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX*. 2026. Available at: https://github.com/onnx/tensorflow-onnx?tab=readme-ov-file.

[25] Orava, V. *Anomaly Explanation Report System*. 2026. Available at: https://github.com/vorava/aers.

[26] Orava, V. *Satellite telemetry generator with annotated anomalies*. 2026. Available at: https://github.com/vorava/satellite_telemetry_generator.

[27] O'Shea, K. and Nash, R. *An Introduction to Convolutional Neural Networks*. 2015. Available at: https://arxiv.org/abs/1511.08458.

[28] Qi, D.; Arfin, J.; Zhang, M.; Mathew, T.; Pless, R. et al. Anomaly Explanation Using Metadata. In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2018, p. 1916–1924.

[29] Ruszczak, B.; Kotowski, K.; Evans, D. and Nalepa, J. The OPS-SAT benchmark for detecting anomalies in satellite telemetry. *Scientific Data*, Apr 2025, vol. 12, no. 1, p. 710. ISSN 2052-4463. Available at: https://doi.org/10.1038/s41597-025-05035-3.

[30] Schefels, C.; Ben Salem, B.; Gerhardus, A.; Helmsauer, K.; Lambert, B. et al. Explaining Satellite Anomalies–Causal Inference for Space Operations. In: *18th International Conference on Space Operations (SpaceOps 2025)*. May 2025. Available at: https://elib.dlr.de/218254/.

[31] Schefels, C.; Schlag, L. and Helmsauer, K. Synthetic satellite telemetry data for machine learning. *CEAS Space Journal*, Sep 2025, vol. 17, no. 5, p. 863–875. ISSN 1868-2510. Available at: https://doi.org/10.1007/s12567-024-00589-1.

[32] Sergieieva, K. *Types Of Satellites: Different Orbits & Real-World Uses*. Mar 2025. Available at: https://eos.com/blog/types-of-satellites/.

[33] TensorFlow. *Pruning comprehensive guide*. Available at: https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide.

[34] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L. et al. *Attention Is All You Need*. 2023. Available at: https://arxiv.org/abs/1706.03762.

[35] Wang, F.; Jiang, Y.; Zhang, R.; Wei, A.; Xie, J. et al. A Survey of Deep Anomaly Detection in Multivariate Time Series: Taxonomy, Applications, and Directions. *Sensors*, 2025, vol. 25, no. 1. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/25/1/190.

[36] Wei, L.; Ma, Z.; Yang, C. and Yao, Q. Advances in the Neural Network Quantization: A Comprehensive Review. *Applied Sciences*, 2024, vol. 14, no. 17. ISSN 2076-3417. Available at: https://www.mdpi.com/2076-3417/14/17/7445.

[37] Wikipedia. *Pareto front*. Wikimedia Foundation, Nov 2025. Available at: https://en.wikipedia.org/wiki/Pareto_front.

[38] Wikipedia. *Precision and recall*. Wikimedia Foundation, Oct 2025. Available at: https://en.wikipedia.org/wiki/Precision_and_recall.

[39] Yepmo, V.; Smits, G. and Pivert, O. Anomaly explanation: A review. *Data & Knowledge Engineering*, 2022, vol. 137, p. 101946. ISSN 0169-023X. Available at: https://www.sciencedirect.com/science/article/pii/S0169023X21000720.

[40] Zamanzadeh Darban, Z.; Webb, G. I.; Pan, S.; Aggarwal, C. and Salehi, M. Deep Learning for Time Series Anomaly Detection: A Survey. *ACM Computing Surveys*. Association for Computing Machinery (ACM), october 2024, vol. 57, no. 1, p. 1–42. ISSN 1557-7341. Available at: http://dx.doi.org/10.1145/3691338.

[41] Zhou, J.; Cui, G.; Hu, S.; Zhang, Z.; Yang, C. et al. *Graph Neural Networks: A Review of Methods and Applications*. 2021. Available at: https://arxiv.org/abs/1812.08434.

Appendix A

Structure of files submited to cloud storage

Files uploaded into cloud storage associated with this thesis have the following structure. Files include models, training scripts, data, codes from two GitHub repositories that have been created during the work on this thesis. Directory also includes outputs in forms of graphs and images.

- `ESA-kaggle` – scripts and data for ESA AD Kaggle challenge
- `README.md` – README file describing this structure, installation and usage of provided scripts
- `dataGen` – scripts and data for satellite telemetry generator, more can be found on https://github.com/vorava/satellite_telemetry_generator
- `models` – directory contains all files related to detection models
  - `clean.sh`
  - `graphs` – graphs from model training
  - `jobs` – Metacentrum job scripts
  - `jupyters`
  - `onnx` – ONNX versions of models
  - `results` – CSV files with results
  - `rpi_graphs`
  - `rpi_results`
  - `rpi_results_smaller`
  - `rpi_saves` – TFLite model saves
  - `rpi_src` - TFLite source codes of forecasting-based models
  - `rpi_src_rec` - TFLite source codes of reconstruction-based models
  - `saves` – `.keras` model saves
  - `src` - source codes of forecasting-based models
  - `src_large`

  - `src_rec` - source codes of reconstruction-based models
  - `utils` – training utilities
- `reportSystem` – scripts and data for Anomaly Exploration and Report System (AERS), https://github.com/vorava/aers
- `requirements.txt`
- `text_assets` – scripts for generation of figures and graphs for thesis
  - `anomaly_types.ipynb`
  - `graphs.ipynb`
- `thesis_text` – LaTeX code and PDF of thesis text