Neural Network for temperature monitoring deployed on a low-power CubeSat onboard computer

Rudolf Maununen

TU Delft

Neural Network for temperature monitoring deployed on a low-power CubeSat onboard computer

Thesis report

by

Rudolf Maununen

to obtain the degree of
Master of Science in Aerospace Engineering
at the Delft University of Technology
to be defended publicly on August 30, 2023 at 14:00

Thesis committee:

Chair:	Dr. S.M. Cazaux
Supervisor:	Dr. S. Speretta
External examiner:	Dr. A. Cervone

Project Duration:	January, 2023 - August, 2023
Student number:	4803930

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.
Cover image: iStock. Satellietnetwerk over planeet Aarde.

Faculty of Aerospace Engineering · Delft University of Technology

Preface

This master's thesis is the final part of my 5 year long education as an Aerospace Engineer at TU Delft. This has been an exciting, yet very challenging journey, which made me grow as a person, overcome life's difficulties, and, of course, learn new things about both the world and myself. This journey has also challenged me to reflect deeply upon my life aspirations and become more mindful of my actions and choices, especially now as I am planning my future career.

Late last year, when looking for a topic of my thesis project, I decided to explore the field of machine learning, as it has been of a particular interest to me for the past few years, and I wanted to learn more about it. Eventually I chose the topic of Neural Networks applied to automation of spacecraft operations and deployment of such networks on satellite onboard computers with limited computational resources. I would like to sincerely thank my thesis supervisor, Dr. Stefano Speretta, for helping me to come up with such an exciting thesis topic, for the opportunity to work on such a project, as well as for all of his wise guidance and support that I received throughout the project. Additionally, I would like to thank Ullas Bhat and Gabriele Meoni for their valuable inputs during the midterm review.

I greatly enjoyed working on this project for the past 7 months, and overall, it progressed very smoothly. Nonetheless, multiple times I had to learn the hard way about the importance of being sceptical towards the experimental results that I am getting, properly justifying the design choices that I make, and being able to base the low-level design choices on the system-level requirements. This is definitely something for me to pay more attention to in my future ventures.

The 5 years of the university studies have been a rough ride. I want to thank my family, friends, and everyone else who helped me to stay on track and get through these uneasy times. Special thanks to my parents who made this journey possible, who believed in me and were always there for me during this period of my life.

Rudolf Maununen
Delft, August 2023

Abstract

The applications of deep learning algorithms for automation of spacecraft operations have been of an increased interest to researchers in the recent years as they can potentially reduce operation costs, bandwidth usage and latency of response to detected faults. In the past 20 years, a number of machine learning and deep learning experiments have been conducted onboard flying satellites, which showed promising results of applying deep neural networks to Fault Detection, Isolation and Recovery (FDIR), task scheduling, and payload data processing tasks. Most of these experiments, however, were deployed on particularly powerful satellite onboard computers (OBCs), which provided sufficient computational resources for the inference of reasonably complex neural networks. No such experiments were conducted on low-power onboard computers that are characteristic of small satellites such as CubeSats, as the neural networks generally require extensive RAM and flash memory resources. However, given a sufficiently slow dynamics of the problem to be solved by the neural network and by applying novel techniques for neural network optimization, the model could be made lightweight enough to be deployed even on low-power OBCs. This could potentially result in a small-sized deep learning solution which would be beneficial even for operations of small satellites, e.g. when deployed onboard a CubeSat.

The 7-month Thesis project discussed in this paper is focused on the development of such deep learning solution that would be able to perform certain slow dynamics FDIR tasks, namely the monitoring of the thermal state of a satellite. As a CubeSat case study, the Delfi-PQ satellite developed at the Delft University of Technology is considered. First, an analysis of the possible temperature anomalies is performed in order to establish the desired functionality of the system. Then, a temperature anomaly detecting neural network is designed and deployed on a TI MSP432P401R microcontroller, which is almost identical to the TI MSP432P4111 microcontroller used as the main element of Delfi-PQ's onboard computer. A microcontroller-based testing system is developed in order to allow for more efficient deployment, analysis and comparison of the neural networks. The system relies on the use of TensorFlow Lite Micro embedded C++ library, Python codes for data processing, and a UART communication interface enabled by Python *Serial* library and *UART* driver of Simplelink MSP432p401r SDK software package for the MSP432P401R microcontroller.

The designed temperature anomaly-detecting neural network demonstrator is a 17x96x96x96x4 multi-layer perceptron network that receives temperature sensor readings from 4 different temperature sensors, along with the derivatives of these sensor readings and the estimate of the current time value. The output of the network consists of four values between 0.0 and 1.0 that correspond to the estimated probabilities of anomalies occurring in each of the four temperature sensors at the time of the network inference. The anomaly detection accuracy of the resulting neural network system is evaluated in terms of four accuracy metrics: recall = 0.8588, precision = 0.8988, F1-score = 0.8784, FAR (False Alarm Rate) = 0.0213. The measured time taken by a single inference of the model is 0.119 seconds. The estimated RAM footprint during the test is 59.1 kB and the flash memory usage is 230.5 kB. For a flight software application, the memory footprint could be reduced to as little as 29.5 kB of RAM and 65.2 kB of flash or even less.

Although the resulting model has certain limitations (e.g. in terms of the detection accuracy when multiple anomalies are present simultaneously), and the overall performance could be better (e.g. in terms of the number of false alarms produced), this neural network-based approach could still be more beneficial than some of the traditional thermal monitoring approaches. For instance, the developed neural network system can detect abnormal temperature readings at very early stages, when the magnitude of the temperature deviations is as low as 10°C. This would not be impossible with a simple threshold approach, that can detect anomalies only when the temperature starts exceeding the critical values.

Contents

Preface i

Abstract ii

List of Figures v

List of Tables vi

List of Acronyms viii

1 Introduction 1
- 1.1 Machine learning applied to space operations . . . 1
- 1.2 Operational capabilities of CubeSats . . . 1
- 1.3 The knowledge gap: DL applications for low-power devices . . . 2
- 1.4 Potential DL applications on low-power onboard computers . . . 2
- 1.5 Research formulation . . . 2
- 1.6 Structure of the Report . . . 3

2 Literature Review 5
- 2.1 Machine learning and Deep Neural Networks . . . 5
- 2.2 Deep neural network architectures for fault detection . . . 7
- 2.3 Automation of spacecraft operations using deep learning . . . 8
- 2.4 Neural network deployment on low power devices . . . 9
- 2.5 Target application . . . 12

3 Methodology 17
- 3.1 Resources used . . . 17
- 3.2 Problem analysis and the proposed solution . . . 20
- 3.3 The design approach . . . 24
- 3.4 Verification and validation approach . . . 28

4 Microcontroller Testing System Design 29
- 4.1 Testing system architecture . . . 29
- 4.2 Algorithms used . . . 31
- 4.3 TF Lite test reports . . . 36
- 4.4 Testing system performance . . . 37

5 Preliminary Design of The Neural Network 42
- 5.1 Introducing anomalies into the training data set . . . 42
- 5.2 Training and testing procedures . . . 43
- 5.3 Defining the system architecture . . . 45
- 5.4 Preliminary network performance . . . 58

6 Detailed Design of The Neural Network 61
- 6.1 First iteration of the detailed design . . . 61
- 6.2 Hyperparameters tuning . . . 61
- 6.3 Final design of the neural network . . . 69
- 6.4 C++ code for the time estimation algorithm . . . 70

7 System Integration and Testing 74
- 7.1 System integration . . . 74
- 7.2 Results of the neural network testing . . . 74
- 7.3 Verification and validation of the results . . . 83
- 7.4 Requirements compliance . . . 88

8 Conclusion **90**
8.1 Closing Remarks 90
8.2 Answers to the research questions 91
8.3 Suggestions for future work 92

References **97**

A FUNcube-1 temperature telemetry used for neural network training **98**

B Python and C++ codes **100**

C TF Lite Micro test report of the final neural network deployed on MSP432P401R **101**

List of Figures

2.1 Relation between Artificial Intelligence (AI), deep learning and machine learning. 6
2.2 Sigmoid and ReLU activation functions [8] 7
2.3 Neural network architectures for fault detection 8
2.4 Internet of Things: cloud, edge nodes, edge devices [31] 10
2.5 Pruning for neural network compression [33] 11
2.6 Delfi-PQ design: stack of subsystems [30] 13
2.7 Delfi-PQ processed battery temperature telemetry [49] 14

3.1 Delfi-PQ onboard computer and MSP432P401R board 17
3.2 FUNcube-1 example temperature telemetry (2 orbits) 19
3.3 Delfi-PQ Y+ panel unfiltered temperature telemetry from 2022-01-18 to 2022-03-29 [64] . . 20
3.4 Anomaly free telemetry vs telemetry with 2 outliers 21
3.5 Anomaly free telemetry vs telemetry with 2 flat regions 21
3.6 Anomaly free telemetry vs telemetry with permanent bias 22
3.7 Anomaly free telemetry vs telemetry with two instances of temporary bias 23
3.8 Anomaly free telemetry vs telemetry with clock anomaly 23
3.9 The design process of the software system 25

4.1 Discarded regions of telemetry from FUNcube side +X. 30
4.2 Testing system architecture and the physical setup 31
4.3 Example TF Lite test report 36
4.4 Performance of a simple predictive network 39

5.1 Procedure to introduce artificial anomalies into the training data 43
5.2 Example of a TensorFlow network testing report 45
5.3 Data flow: telemetry, anomaly detecting network and a time estimation algorithm 51
5.4 Anomaly detecting network architecture 55
5.5 Example from the training dataset adm_2_9 55
5.6 Weight importance distribution adm_2_9 58
5.7 Preliminary model outputs example 59

6.1 Training and validation loss evolution 67
6.2 Example temperature anomaly and the corresponding model output 68
6.3 Time Estimation Algorithm performance analysis 72

7.1 Model outputs on dataset_1187_0 (anomaly-free) 77
7.2 Model outputs on dataset_1187_2 (2 outliers) 78
7.3 Model outputs on dataset_1187_9 (2 flat regions) 79
7.4 Model outputs on dataset_1187_21 (2 sensors with permanent bias) 80
7.5 Model outputs on dataset_1187_28 (2 sensors with temporary bias) 81
7.6 Deployed model performance with TEA on dataset_1187_0 (anomaly-free) 81
7.7 An output of the deployed model with σ = 2°C temperature sensor noise 84
7.8 An output of the deployed model with -2 min. time input error 85
7.9 An output of the deployed model with +6 min. time input error 86
7.10 Progressively increasing permanent bias anomaly 87
7.11 Model outputs for +X panel on a progressively increasing permanent bias anomaly 87

A.1 FUNcube-1 temperature telemetry for +X, -X, +Y, -Y solar panels. Received on 04/02/2016 99

List of Tables

2.1 Processor board characteristics of Delfi-PQ vs in-space Machine Learning (ML) missions . 9
2.2 Comparison of various experiments on TinyML inference 12
2.3 Comparison of various boards in terms of processing capabilities and power consumption[32] 13
2.4 Delfi-PQ orbital parameters[48] 14
2.5 Comparison of various machine learning libraries 15
2.6 Comparison of various Machine Learning on Tiny Devices (TinyML) frameworks and libraries 16

3.1 FUNcube-1 orbital parameters[63] 19
3.2 System requirements for the NN-based embedded software system 26
3.3 Output types of a binary classifier network 27
3.4 Experimental results and V&V procedure 28

4.1 TI example project memory specifications (hello_world_MSP_EXP432P401R_nortos_gcc) 30
4.2 Data types received by the microcontroller 32
4.3 Fixed embedded software parameters 34
4.4 Embedded software memory footprint 37
4.5 Embedded software memory footprint when using *MicroMutableOpResolver* 37
4.6 Case 1 test-specific parameters 38
4.7 Case 1 testing system time performance 39
4.8 Case 2 test-specific parameters 40
4.9 Case 2 testing system time performance 40
4.10 Case comparison on time performance 41

5.1 Trade-off 1: neural network output types 47
5.2 Trade-off 2: network's ability to distinguish between anomaly types 48
5.3 Trade-off 3: number of neural networks deployed 48
5.4 Trade-off 4: separate clock value processing vs incorporated in a single network 49
5.5 Trade-off 5: Neural network architectures 50
5.6 Trade-off 6: system's ability to deal with unknown time (wrong clock) 51
5.7 Trade-off 7: Options for processing the output of the time estimation algorithm 52
5.8 Summary of the iterative design process of the network 53
5.9 adm_2_9 model training parameters 57
5.10 adm_2_9 model performance 59
5.11 adm_2_9 model performance 60

6.1 Comparison of the preliminary model vs initial iteration of the final model 61
6.2 Tuning for number of neurons 62
6.3 Tuning for number of layers 63
6.4 Tuning for layer symmetry 63
6.5 Fine-tuning the number of neurons per layer 63
6.6 Fine-tuning the number of input layer neurons 64
6.7 Choosing hidden layer activation function 65
6.8 Comparing different learning rates 66
6.9 Comparing different training batch sizes 66
6.10 Fine-tuning the number of training epochs 67
6.11 Choosing the loss function 68
6.12 adm_3_8 classifier performance with different detection thresholds 69
6.13 Final neural network design specifications. adm_3_8 anomaly detector. 70

7.1 adm_3_8 model performance on MSP432 75

7.2 NN detection accuracy performance before and after deployment. 76
7.3 Sensitivity analysis on input temperature sensor noise (modelled as Gaussian noise). . . . 84
7.4 Sensitivity analysis on input time error. 85
7.5 Results of re-training the model with Gaussian noise in the training data 86
7.6 System requirements compliance matrix 89

List of Acronyms

AI Artificial Intelligence
ANN Artificial Neural Network
API Application Programming Interface
CNN Convolutional Neural Network
COTS Commercial Of-The-Shelf
DL Deep Learning
EO Earth Observation
ESA European Space Agency
FDIR Fault Detection, Isolation and Recovery
FF Feedforward neural network
FPGA Field-programmable Gate Array
GRU Gated Recurrent Unit
IDE Integrated Development Environment
IoT Internet of Things
KD Knowledge Distillation
LEO Low Earth Orbit
LSTM Long/Short Term Memory network
MCU Microcontroller Unit
MIPS Million instructions per second
ML Machine Learning
MLP Multi-layer Perceptron
NN Neural Network
OBC Onboard Computer
RAM Random-Access Memory
RNN Recurrent Neural Network
SDK Software Development Kit
SRAM Static Random-Access Memory
TinyML Machine Learning on Tiny Devices
TEA Time Estimation Algorithm
TLE Two-line elements
V&V Verification and Validation

1 Introduction

Cost is a critical aspect of most space missions, as it often defines mission feasibility. In order to decrease the costs related to spacecraft operations, various methods are being proposed, among which a major trend is spacecraft autonomy [1]. Automation of spacecraft operations allows to significantly decrease the amount of data sent to- and from a satellite, thus reducing the workload on satellite operators. The advancements in spacecraft autonomy create a need for new onboard software solutions, including novel data-processing and analysis techniques, among which an emerging trend is machine learning (ML) -based satellite data processing [2]. Machine learning is a collection of mathematical methods that allow creation of complex computational models which rely not on explicit definitions of computational rules, but rather on the ability of these models to learn these computational rules by themselves. In the recent years, the great interest has been gained by deep learning (DL), a subset of machine learning that deals with computational models that mimic the way biological neural networks work. The resulting models are called Artificial Neural Networks (ANN) or more commonly, simply Neural Networks (NN). The 7-month Thesis project presented in this report, focuses on machine learning (ML), and more specifically, deep learning (DL) applied to satellite telemetry data and how it can be implemented not only on the ground, but also onboard a satellite itself, which will further increase the level of autonomy.

1.1. Machine learning applied to space operations

Thanks to the high computational capabilities, efficient development, and particularly high performance in classification-, pattern recognition- and data reconstruction tasks, the potential applications of ML algorithms to space operations have been extensively studied. Such algorithms have been widely applied in Earth Observation for image processing, including filtering, pattern recognition, terrestrial features classification and analysis tasks. Other applications of ML in space technology include Fault Detection, Isolation and Recovery (FDIR) algorithms, task scheduling, instrument data processing, etc.

The traditional use of ML in spacecraft operations is limited to running ML-based software to process telemetry or satellite payload data on the ground, and then using the outcomes of the data processing to control the satellite in a desired way. However, thanks to modern advancements in electronics, especially, the increased processing power of onboard computers, ML is currently being implemented not only on the ground, but also onboard satellites themselves, which allows to perform the same operations, but without having to downlink all the telemetry/payload data all the time. The decision making is delegated to a trained ML model uploaded to the spacecraft's onboard computer (OBC), so that the tasks related to payload usage or bus operations, are performed autonomously by the spacecraft. This approach has a number of advantages, among which are reduced operations cost, lower workload on satellite operators, improved data security and faster response to faults. [3]

1.2. Operational capabilities of CubeSats

CubeSats, also known as nanosatellites, are small spacecraft which consist of standardized 10x10x10 cm structural modules, or units. The compact size of these satellites allows the development of low-cost space missions that are affordable even by universities and a wide range of research institutions. These satellites, however, are generally characterized by severely constrained mass, volume and power budgets. Therefore, these satellites have to work with low-power and low-data rate instruments and low-power

onboard computers, whose capabilities are limited in terms of both the processing power and the available memory.

The operations of these satellites, however, are conceptually the same as those of larger satellites, which means that CubeSats operators have to deal with the same operational limitations, which include limited spacecraft visibility, limited bandwidth, time delays in spacecraft's response to detected faults, etc. This creates a need for similar highly efficient computational solutions that are being explored for the conventional satellites.

1.3. The knowledge gap: DL applications for low-power devices

Machine learning -based models deployed on low-power devices, such as microcontrollers, can be very helpful in all kinds of applications where low response latency and high level of autonomy of so-called edge devices are of high value. Such ML-based solutions are currently being widely explored for applications in various industries, from healthcare and civil engineering to automotive engineering [4]. A big challenge in this field of research is highly limited processing power of edge devices.

The same challenges hold when it comes to in-space applications of Deep Learning (DL). These challenges are particularly pronounced for CubeSats, as they are typically characterized by highly limited computational resources. While larger satellites can have onboard computers with available Random-access Memory (RAM) in order of hundreds of megabytes or even gigabytes, the onboard computers of CubeSats are often based on microcontroller integrated circuits, which only provide a RAM in order of hundreds of kilobytes.

All of the space missions that have deployed DL-based experiments onboard satellites, had particularly powerful onboard computers designed specifically to handle such computationally-intensive experiments. When compared to less powerful onboard computers, such as those found in CubeSats, it becomes apparent that the deployment of ML and Neural Network (NN)-based software on such low-power onboard computers has not been properly explored yet.

1.4. Potential DL applications on low-power onboard computers

It is obvious that the highly limited computational resources are the exact reason why such in-space experiments have not yet been performed: neural networks are generally quite computationally-intense and require a lot of RAM and flash memory to perform their specific tasks (inference). However, it should be noted that the size of a neural network is heavily dependent on the complexity of the tasks that it has to perform.

Certain spacecraft operations, such as temperature sensor- or battery capacity data monitoring, are characterized by slow dynamics of the problem resulting in low data rates. A neural network designed specifically for such tasks could have a much smaller size than a neural network for more complex and computationally-demanding tasks such as spacecraft attitude control. Additionally, the novel neural network optimization methods can be explored in order to make such small neural network as lightweight as possible, so that it could be deployed on a low-power onboard computer. In this Thesis, the application of deep learning for temperature data monitoring will be explored as temperature telemetry data is generally more abundant compared to other telemetry types characterized by slow dynamics.

1.5. Research formulation

Based on the identified knowledge gap, the research questions and objectives of the research project are formulated. These research questions and objectives are driving the entire 7-month-long Thesis work presented in this report.

1.5.1. Research questions

The main research question of the Thesis project can be formulated as follows:

Research Question

How effective a deep leaning model deployed on a spacecraft onboard computer with less than 256 kB available RAM can be for satellite thermal state monitoring?

This general question can be subdivided into the following more specific research questions:

1. *How to deploy such deep neural network on a low-power OBC?*
   a) What frameworks/libraries for on-device ML exist?
   b) What framework is most suitable for the implementation of the desired NN architecture on the available board?
   c) How to set up a system to deploy and test neural networks on the available board?
2. *What thermal anomalies should the system be able to detect?*
   a) What thermal anomalies are possible in CubeSats?
   b) What thermal anomalies are critical for CubeSat operations?
   c) What thermal anomalies can a small-sized neural network reasonably detect?
3. *What neural network architecture is best suited for the target application?*
   a) What NN architectures exist?
   b) Which architectures are suitable for on-device implementation?
   c) How do these architecture compare?
   d) How complex should the intended ML model be (in terms of number of nodes, layers and activation functions)?
4. *How such implementation compares to traditional thermal monitoring approaches in terms of performance?*
   a) What is the thermal anomaly detection accuracy of such application?
   b) What is the thermal anomaly detection latency of such application?
   c) What is the flash memory storage space consumption of such application?
   d) What is the RAM usage of such application?

1.5.2. Research objectives

The three main research objectives are as follows:

Research Objective

Develop a system to deploy and test neural networks on a low-power microcontroller board.

Research Objective

Design a neural network for temperature monitoring, that would meet the functional requirements and memory limitations of the available board.

Research Objective

Deploy the designed neural network on the available board and test performance (memory used, inference time, detection accuracy).

1.6. Structure of the Report

The Thesis report is structured as follows. Chapter 2 contains a summary of a literature review on deployment of deep learning models on low-power devices and the Delfi-PQ satellite that is used as a case study for this Thesis. After that, the design problem associated with the research project is further explored in Chapter 3. This chapter also presents an overview of the methodology, available resources, expected results and V&V procedures of these results. Next, Chapter 4 is focused of one of the main parts of the

temperature monitoring system: the microcontroller-based testing environment. Then, the discussion focuses on the design of the temperature anomaly-detecting neural network to be tested on a low-power device. In Chapter 5, the preliminary design phase is discussed, while Chapter 6 is focused on the detailed design of the NN including the network optimization by hyperparameters tuning. The integration of the system and results of the testing of the designed NN are explored in Chapter 7. Finally, the conclusions of the thesis work are drawn in Chapter 8. This chapter also provides recommendations and options for the future research.

2

Literature Review

In preparation for the thesis project, a literature study has been conducted in order to obtain a better understanding of the topic and the knowledge gap to be addressed by the thesis work. This chapter contains a summary of that literature review and its findings. In Section 2.1, a brief review of machine learning and deep neural networks is provided. Then, in Section 2.2, neural network architectures suitable for fault detection tasks are discussed. In Section 2.3, the applications of deep neural networks for spacecraft operations are discussed. Section 2.4 focuses on the methods of deployment of deep neural networks on low-power computing devices. Finally, the case study which is the focus of the thesis project is discussed in Section 2.5.

2.1. Machine learning and Deep Neural Networks

Machine learning is a field of computing which comprises various computational algorithms that can learn patterns and draw inferences without any explicit instructions provided. Deep learning is a subset of machine learning which deals with models that are based on an Artificial Neural Network (ANN) or simply NN, computing systems that imitate the work of biological neural networks. Terms DL and ML are often used interchangeably, however, it is important to understand that the fields of AI and machine learning are not limited to DL only. Figure 2.1 shows the relationships between the areas of deep learning, AI and ML. In Figure 2.1, also the examples of various algorithms are given for each ML type. Although this structure is by far not a complete overview (because of the great variety of existing algorithms), it still shows the general structure and types of ML [5, 6, 7].

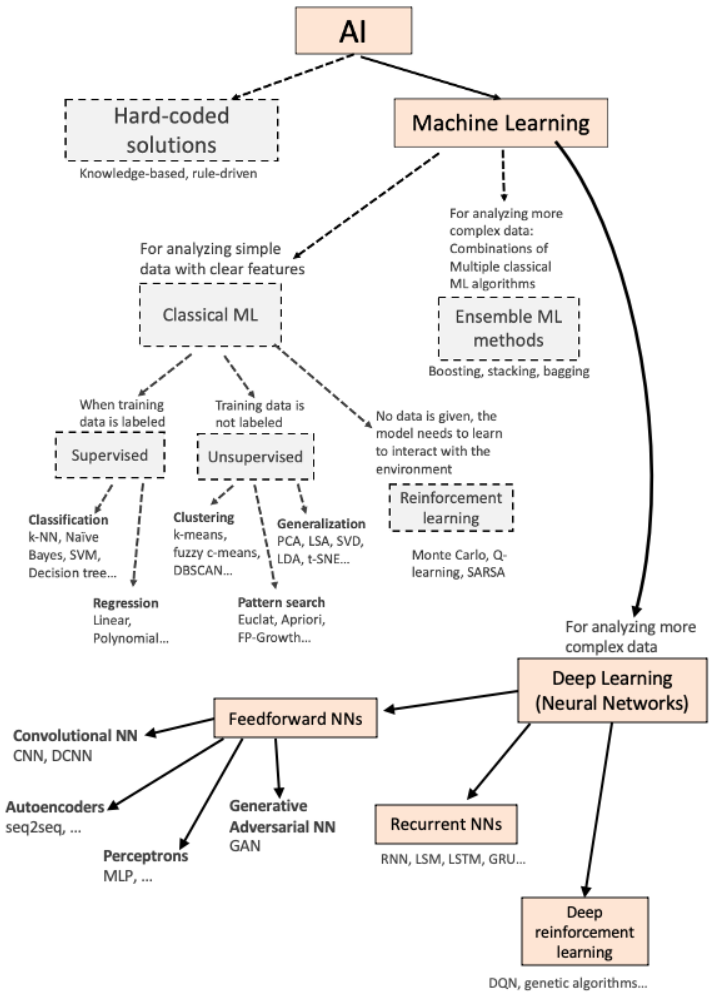


Figure 2.1: Relation between AI, deep learning and machine learning.

While classical ML models (algorithms) can be efficient for applications where there is not much training data, and it has a clear structure, their implementation becomes quite complex and not so efficient when dealing with extensive datasets, e.g. images, video, audio. For applications where the model is to be trained on large sets of data, the implementation of neural networks is easier and can give better results. This is due to the fact that NNs do not rely on formulas or any kind of computational rule or algorithm, but solely on the way the neurons are connected to each other. This makes NNs capable of replacing any of the traditional ML algorithms by imitating their work [6].

At their core, NNs consist of interconnected layers of nodes, or neurons. At each neuron, a weighted computation on input data is performed, and the output is produced as shown in the Equation (2.1):

$$a = f\left(b + \sum_{i=1}^{n} x_i w_i\right) \tag{2.1}$$

Where a is the output of an individual neuron, x is an individual input of the neuron, w is the associated (trainable) weight and b is (trainable) bias of the node. f is the so called activation function, such as the sigmoid or ReLU functions, which introduce non-linearity. The non-linear nature of these functions can be observed in Figure 2.2:

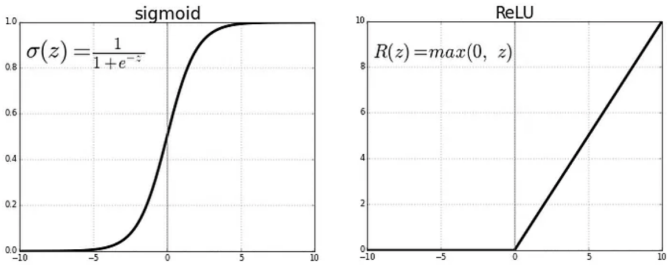


Figure 2.2: Sigmoid and ReLU activation functions [8]

The weights between neurons are adjusted through an optimization process called backpropagation, which minimizes the difference between predicted and actual outputs [9].

Once the neural network is optimised, or *trained*, it can analyse some newly collected data based on the knowledge gained during training. Inference of the NN requires a data input of a relatively small size, and it does not require very much processing power. Training, on the other hand, is much more computationally intensive, it requires a long time and a big data set (thus also the storage capacity for this data), unless the training is done in small batches of sequentially arriving data. [10, 4]

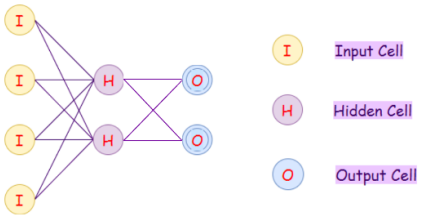
2.2. Deep neural network architectures for fault detection

For the reasons stated previously, deep learning is the preferred solution for a variety computationally-intense applications, including spacecraft operations -related data analysis. Deep learning approaches demonstrate particularly high performance in data classification and feature detection tasks, which makes them suitable for numerous engineering applications where system state monitoring and anomaly detection is involved. Deep neural networks are thus also regarded as a promising technology for spacecraft Fault Detection, Isolation and Recovery (FDIR). [11]

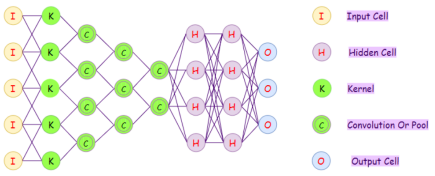
The neuron layers in a network can be arranged in various architectures, which defines suitability of a network for a particular application. For the fault detection/classification purposes, the most suitable architectures include:

- **Multi-layer Perceptron (MLP)**: a very basic Feedforward neural network (FF) architecture consisting of an input layer, one or more hidden layers, and an output layer. They are computationally efficient, easy to interpret, and have low latency when it comes to fault detection. [5]
- **Convolutional Neural Network (CNN)**: mainly used for image classification, but could also be applied to 1D series such as sensor data. By applying convolutional filters, CNNs can learn patterns and identify anomalies in sensor readings. [6]
- **Recurrent Neural Network (RNN)**: processes sequential data by maintaining internal memory. Gated Recurrent Unit (GRU) and Long/Short Term Memory network (LSTM) are popular variations of RNN. They capture temporal dependencies in sensor data and detect faults based on deviations from learned patterns. [6]
- **Autoencoders**: this architecture consists of an encoder and a decoder network. By training the NN on normal sensor data, it learns to accurately reconstruct the input. When presented with anomalous data, the reconstruction error increases, which indicates an anomaly. [5]

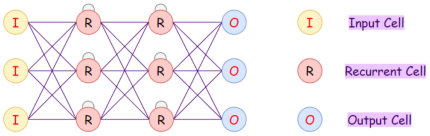
The schematic representations of these four architectures can be seen in Figure 2.3:



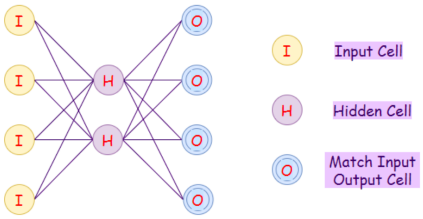
(a) MLP architecture



(b) CNN architecture



(c) RNN architecture



(d) Autoencoder architecture

Figure 2.3: Neural network architectures for fault detection

2.3. Automation of spacecraft operations using deep learning

A crucial part of unmanned space missions is data transfer between a spacecraft and the mission control on Earth. Spacecrafts generate large amounts of data that need to be sent to the ground. This includes telemetry (sensor and status data) and data generated by the payload (imagers, scientific instruments). The costs related to spacecraft operations that enable the data transfer are generally quite high compared to other mission costs. Deep space missions lasting for 10 and more years typically have operations cost that account for about 30% of the total mission cost, while for some missions it can even exceed 50% [12]. Operations costs for Low Earth Orbit (LEO) Earth Observation (EO) satellites with flight duration of 5-7 years can range from 20% of total mission cost for new satellites with considerable level of automation [13], to about 30% for older EO satellites with less automation [14].

Cost is a critical aspect of most space missions, as often defines mission feasibility. In order to decrease the costs related to spacecraft operations, various methods are being proposed, among which a major trend is spacecraft autonomy [1]. Automation of spacecraft operations allows to significantly decrease the amount of data sent to- and from a satellite, thus reducing the workload on satellite operators. The advancements in spacecraft autonomy create a need for new onboard software solutions, including novel data-processing and FDIR techniques among which an emerging trend is NN-based satellite data processing and FDIR [2].

2.3.1. Deep learning for satellite data processing

One of the most common applications of DL for satellites is processing of the data that has been sent from a satellite to the ground. DL is applied for payload data processing, for example, data collected from Earth observation instruments. Neural networks can spot specified patterns and land features, filter out noise, and generate insights that are challenging to extract by manual data processing [15].

DL is also used for on-ground processing of satellite telemetry data, for example, spacecraft attitude [16], thermal data, power level data, navigation data etc. After on-ground processing, the outcomes can be used for task scheduling [17], planning of instrument measurements [18], attitude control, thermal control, FDIR [19] etc. Then, the corresponding commands are sent to the spacecraft. This can be considered a traditional approach, in which both the model training and inference are performed on the ground, and only the outputs of the model are sent to space.

2.3.2. Deep learning in space: onboard inference

Thanks to modern advancements in electronics, especially, the increased processing power of onboard computers, DL is currently being implemented not only on the ground, but also onboard satellites, which allows to perform the same operations, but without having to downlink all the telemetry/payload data all the time. The decision making is delegated to a trained NN uploaded to spacecraft's Onboard Computer (OBC), so that the tasks related to payload usage or bus operations, are performed autonomously by the spacecraft. This approach has a number of advantages [3]:

- Less bandwidth is utilized as less data is sent to the ground (only ML model coefficients and status data is sent).
- Less energy is spent by the spacecraft as the communication system is not used that often.
- Less work is required from the operators on Earth (operations are unsupervised).
- Onboard optimization of operations is possible.

- Operation costs are lower.
- Satellite can perform certain operations even without radio connection with the ground.
- The smaller amount of data sent also makes satellite operations more secure, as there is less opportunity for eavesdropping. Sensitive information can be processed onboard only.
- AI-driven autonomy enables a much faster response to a fault/anomaly, improving the quality of FDIR.

With all the advantages of onboard ML compared to the traditional approach, there is an increased interest towards running inference and potentially even training onboard satellites. However, there are several aspects that make onboard implementation of complex ML models challenging [3, 4]:

- Processing speed of spacecraft onboard computers is highly limited. Inference can take a long time causing response latency.
- Running big AI models requires a lot of electrical power, which affects the power system design requirements and increases the mission cost. The energy lifetime of the spacecraft also has to be taken into account.
- Onboard computers, especially those on CubeSats, have a relatively small Random-Access Memory (RAM), external memory can be used, but accessing this external memory is slow.

Due to these challenges, currently in-space applications of DL and the related research are almost completely limited to only running inference onboard satellites. In this paradigm, the training of the model is performed entirely on the ground by creating a physical model of the spacecraft or/and by using the telemetry data previously obtained (ether from the same satellite or a similar satellite). There has been a number of developments for onboard model inference, mostly in image classification for EO and remote sensing [20, 21, 22, 23, 24, 25, 26], spacecraft task scheduling [27, 28], and FDIR [29]. The only documented in-flight demonstrators up until now were Φ-Sat-1 [20] and OPS-SAT [23] missions by European Space Agency (ESA), and D-Orbit ION SCV-4 mission recently launched, which performed a ML experiment with AWS software [26]. Additionally, there have been two missions by NASA: EO-1 [24] and IPEX [25], however, they used ML algorithms other than neural networks.

All of the missions listed above showed promising results of running NN inference onboard, however, a common feature of all these missions is that all of them were specifically designed to have powerful enough onboard computers to handle the ML-based experimental software. When compared to less powerful OBCs, such as those commonly used in nanosatellites (CubeSats), it becomes obvious that deployment of ML and NN-based software on such low-power OBCs has not been thoroughly explored yet. Table 2.1 makes it clear by providing a comparison between the missions previously mentioned and an example CubeSat mission, for which Delfi-PQ satellite launched in 2022 is taken. Delfi-PQ onboard computer has 10 times less available RAM than the other missions, which raises a question of whether a similar ML-based experiment could be performed on a low computing power board such as the one Delfi-PQ has. Such experiment could result in potential onboard ML applications that can be deployed on lower power but also less expensive boards, enabling higher levels of automation also for lower budget space missions.

Table 2.1: Processor board characteristics of Delfi-PQ vs in-space ML missions

Satellite	Launch year	Board	Processor	Available RAM	CPU Clock
OPS-SAT [23]	2019	Altera Cyclone V SX SoC	ARM Cortex-A9	512 MB	800 MHz
Φ-Sat-1 [20]	2020	?	Myriad 2 VPU	4 GB	933 MHz
IPEX [25]	2013	?	Atmel ARM9	128 MB	400 MHz
EO-1 [24]	2000	?	Mongoose M5	128 MB	12 MHz
Delfi-PQ [30]	2022	MSP432P4111	ARM Cortex-M4	256 KB	48 MHz

2.4. Neural network deployment on low power devices

When it comes to running ML (DL) models onboard a satellite, the main challenge is obviously the computational capabilities of the onboard computer, which may not be able to store a ML model with tens of thousands of parameters, or may not provide enough RAM to run the model. Similar challenges are

also being faced in the fields of embedded systems and Internet of Things, when trying to run NNs on small low-power devices.

2.4.1. Trends in IoT: edge computing

The onboard computer of a satellite can be viewed not only from the spacecraft design perspective, but also from the embedded systems engineering perspective or even as a system similar to an Internet of Things (IoT) edge device. In IoT, computationally-intense tasks and data storage are usually done on the cloud (data center with a server). Via the network edge nodes (routers), the cloud can exchange data with edge devices (small sensors that have limited data processing capabilities or user gadgets with low computational power). This relation is illustrated in Figure 2.4. Edge devices are often deployed in the field, where physical access is challenging or not possible, and only a data transfer channel is available to exchange data with the network.

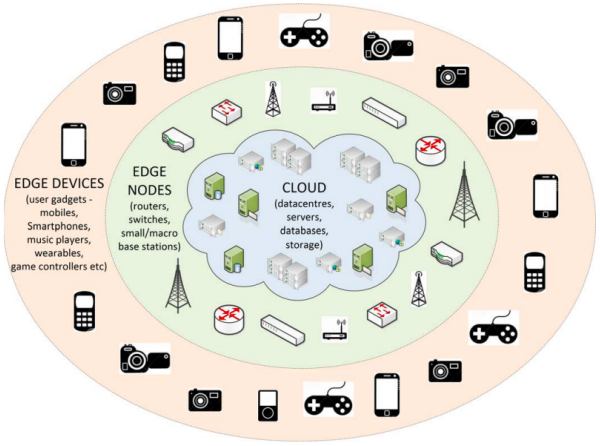


Figure 2.4: Internet of Things: cloud, edge nodes, edge devices [31]

The general trend in IoT is that edge devices are becoming more and more powerful and more computation is performed at the edge of the network. Edge computing is an approach in which the edge devices do not send all the generated data to the network, but rather they process the data locally end extract the valuable insights, so that only the useful data is transferred. This approach allows to decrease the amount of data that has to be transferred through the network, which reduces operations cost, latency and bandwidth usage. It was discussed previously that a very similar trend emerges also in the area of spacecraft operations. In IoT terms, a satellite can be viewed as an edge device, while a ground station is the cloud. [31, 4]

2.4.2. TinyML: machine learning on tiny devices

Machine learning applications in embedded systems and IoT are being widely explored. The area of research that deals with ML implementations on small devices (microcontrollers, Field-programmable Gate Array (FPGA), etc...) is commonly referred to as TinyML (ML on tiny devices). TinyML typically deals with microcontrollers with power consumption in the order of tens to hundreds of milliWatts. [4]

TinyML is gaining popularity and a lot of research is currently being done on on-device inference/training for all kinds of embedded system applications, not being limited to just the topic of onboard ML for satellites. ML in space applications is an inter-disciplinary research field, meaning that in order to develop a satellite ML solution, the related research from other fields, such as computer science and embedded systems, has to be explored. The developments in on-device training for different applications might be later applied also to satellite autonomy. For this reason, the survey performed in the upcoming sections will consider all kinds of TinyML solutions regardless of their intended application.

2.4.3. On-device inference of neural networks

Commercial Of-The-Shelf (COTS) microcontroller boards normally have integrated flash memory in order of 400 kB - 2 MB and RAM ranging from 32 kB to 2 MB [4]. At the same time, even the ML models that are

considered relatively small, can include millions of parameters. Storing and running such models would require tens of MegaBytes flash memory and several MegaBytes of RAM [32]. This means that most of conventional neural networks cannot be directly run on microcontrollers, and certain software-based solutions are required to adapt neural networks for on-device implementation. Such solutions include Pruning, Knowledge distillation, Quantization, Encoding. [33, 34, 32] These four techniques are further explained below.

1) *Pruning*: it is an iterative technique in which connections between certain neurons are removed if the corresponding weights are below a pre-defined threshold (see Figure 2.5). After removing the weak connections, the network is retrained and the process is repeated. Pruning allows for up to 90% size reduction without considerable loss in model's accuracy. [33]

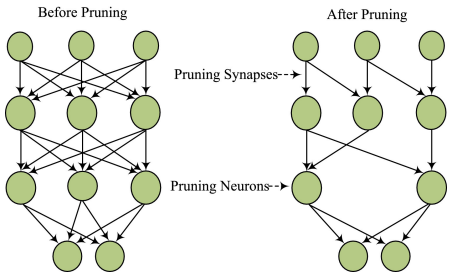


Figure 2.5: Pruning for neural network compression [33]

2) *Knowledge Distillation (KD)*: KD is a compression technique that relies on training an embedded ML model based on a non-compressed ML model. First, a conventional, big size ML model is created and trained to perform the desired functions. This model is referred to as the teacher model. Then, an embedded version of that model is created and trained to imitate the work of the big model, but using much lower number of neurons (the student model). [33, 34]

3) *Quantization*: In this method, the floating-point precision of the model is decreased. This means that the compression is achieved by reducing the number of digits after the decimal point which are used to represent the activation functions and the weights of neural connections. For example, a ML model with 32-bit or 64-bit weight representation can be reduced down to 8-bit representation. [33, 32]

4) *Encoding*: Various encoding can be used. For instance, more frequent weights can be represented with a smaller number of bits, while more rare weights are encoded with higher number of bits. This method can reduce the size of the model by up to 49 times without any loss in accuracy. [33]

2.4.4. Examples of on-device NN deployment

There are numerous papers that present experimental results of running DL model inference on small devices. Most of them use open-source libraries that include various functions that make developing TinyML software much easier. Many of these libraries provide both the possibility to develop an embedded model from scratch and obtain the embedded model by compressing an already existing ML model of a larger size. For the compression, there are built-in quantization and operators, which allow to reduce the ML model to the desired extent. As it was already mentioned, conventional ML models can be tens of MegaBytes in size. Compression techniques allow to reduce the model to a few kB size and consuming a few kB of RAM, which is suitable for running on low-power microcontrollers. For example, in [32], after applying KD and 8-bit quantization to a CNN model, the model size was reduced by a factor of 2356. Such reduced models can still give a high accuracy result when running on a microcontroller. To illustrate this, a number of examples from the existing papers were collected and summarized in Table 2.2. During the literature study, the following criteria were used to determine if a paper will be included in Table 2.2:

- The paper includes information on target application, processor type and chosen NN architecture.
- The processing board is a Microcontroller Unit (MCU) based on Cortex-M or similar.
- Additional information is included: compression techniques used, framework/libraries used, RAM and storage use of the implemented solution, power consumption of the implemented solution, NN test accuracy.

- At least two of the items listed above have to be specified in the paper (otherwise the paper is excluded).

Table 2.2: Comparison of various experiments on TinyML inference

Application	Board/processor	NN Type (N layers, N nodes)	Compression techniques	Framework	RAM use	Storage use	Power consumption	Test Accuracy
People presence detection in thermal images [35]	STM32L476RG (Cortex M4)	CNN (3, 29)	8-bit quantization	CMSIS-NN	6 kB	1.9 kB	16.5 mW	76.7%
Acoustic event detection [32]	STM32L476RG (Cortex M4)	CNN (9, 298), RNN (-, -)	Knowledge Distillation, 8-bit quantization	CMSIS-NN	34.3 kB	30.6 kB	5.5 mW	75%
Object detection in images [36]	STM32H743VI (Cortex M4)	CNN (7, 322)	8-bit quantization	TF Lite	N/A	138 kB	N/A	99.83%
Hand gesture recognition [37]	STM32H743 (Cortex M7)	CNN (-, -), ConvLSTM (-, -)	N/A	N/A	N/A	147 kB	N/A	84.6%
Temperature prediction [38]	Arduino Nano 33 Ble Sense (Cortex M4)	MLP (-, -), CNN (-, -)	N/A	TF Lite Micro	N/A	N/A	170 mW	85%
Classification of organic compounds [39]	Cortex M4	Custom (FF) (4, 41)	8-bit quantization	TF Lite Micro	0.8 kB	8.1 kB	N/A	99.8%
Classification of vehicles based on sensor data [40]	Cortex M4	RNN (-,-)	N/A	TF Lite Micro	N/A	N/A	N/A	98%
Gesture recognition [41]	Arduino Uno (ATmega328P)	FF (3, 11), RNN (3, 10)	N/A	TF Lite	1.44 kB	5.97 kB	N/A	83%
Presence detection [42]	ESP32 (Xtensa 32-b LX6)	CNN (-, -)	8-bit quantization	N/A	27 kB	4.2 MB	N/A	99.9%
Image recognition [43]	STM32F401RE (Cortex M4)	CNN (5, 171)	pruning, quantization	STM32Cube-AI	135.7 kB	668.97 kB	N/A	100%

From the examples shown in Table 2.2, it can be concluded that model complexity does not directly influence its performance, as even very small models show high accuracy. The main driving factors of model accuracy likely include the complexity of the task performed by NN, the quality of training data and its similarity with test data. Also, the choice of neural network architecture supposedly plays an important role. Unlike the accuracy, the RAM and memory use has a strong dependency on the model complexity. Higher number of layers and nodes (neurons) results in higher memory consumption. Among the presented examples, the most often used algorithms are CNN, FF and RNN (including LSTM). The most common applications are classification, detection and pattern recognition. The most used framework for NN deployment is TF Lite Micro.

2.5. Target application

Despite the current limitations that make onboard deployment of NNs challenging for many applications, there are certain applications where the implementation of onboard NN-based software is much more realistic even with highly limited computational resources available. Based on the application, a ML software can be classified as either payload related or spacecraft bus/operations related [15]. Satellite payload (e.g. an imager or other scientific instrument) generally generates data at a very high rate, which is due to temporal- spatial- and other resolution requirements. The associated ML software is therefore more complex and deploying such model onboard is generally costly [44]. However, certain telemetry data, for example, temperature, electric power level, navigation, attitude and other data arrives at a much slower rate, which means that low-complexity networks could potentially be effective at processing these kinds of data. For example, one potential application for DL onboard CubeSats with low-power OBCs could be the prediction of temperature readings in different parts of the spacecraft and detection of thermal anomalies. In case of thermal control subsystem, the changes in temperature sensor readings are relatively slow and the dynamics is predictable. This makes it possible to create a low-complexity ML model that can predict thermal behavior in different components/subsystems of a satellite and detect any anomalies that do not match the predicted behavior. For the number of reasons mentioned above, this particular application will be the main focus of this Thesis.

2.5.1. Case study: Delfi-PQ CubeSat

Three research satellites have been built by the faculty of Aerospace Engineering of Delft University of Technology: Delfi-C^3 launched in 2008, Delfi-n3Xt launched in 2013, and Delfi-PQ launched in early 2022 [45]. The newest satellite, Delfi-PQ (Delfi-PocketQube) features an OBC based on a MSP432P4111 microcontroller, which could be used to explore the applications of onboard machine learning. In this Thesis, Delfi-PQ CubeSat is used as a case study for onboard NN deployment for temperature monitoring. It is, therefore, important to explore the CubeSat's design and operation to obtain a list of system requirements that would influence the design of the ML solution.

Spacecraft onboard computer

Delfi-PQ is designed to have 7 subsystems shown in Figure 2.6:

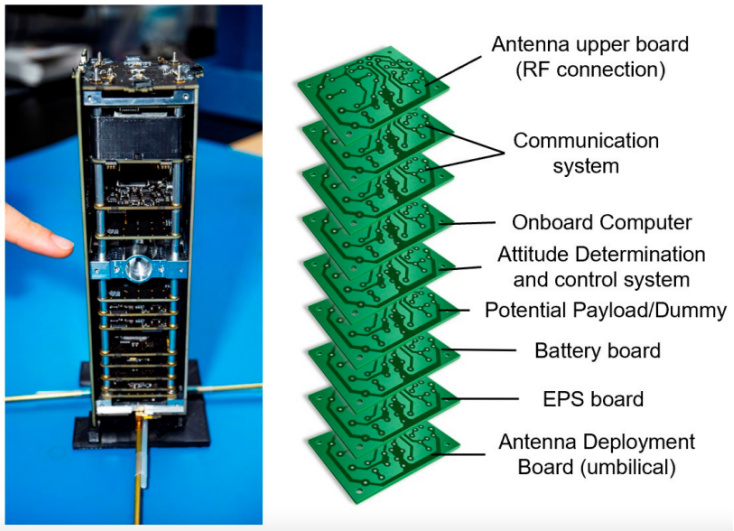


Figure 2.6: Delfi-PQ design: stack of subsystems [30]

Each subsystem board, including the OBC, has a MSP432P4111 microcontroller produced by Texas Instruments. MSP432P4111 is a 48MHz microcontroller with 2MB built-in Flash memory and 256KB Static Random-Access Memory (SRAM). Additionally, there is 2GB external Flash memory for telemetry storage. The OBC can actively send frames over the RS-485 115.2kbps data bus, while other subsystems can only reply passively. RS-485 only allows half-duplex communication where each frame is up to 253 Bytes. [30] MSP432P4111 works with embedded software written in C or C++ [30, 46].

Despite the fact that the suitability of MSP432P4111 for ML applications was previously doubted[47], it was demonstrated by [32] that this board, as well as similar boards produced by TI or STM, are well suited for TinyML applications. In fact, they are winning against other COTS boards that either have too small computational resources or too high power consumption. A comparison of MSP432P4111 to other COTS boards (in terms of memory, RAM, Power and Million instructions per second (MIPS)) is shown in Table 2.3:

Table 2.3: Comparison of various boards in terms of processing capabilities and power consumption[32]

Board name	Flash[KB]	RAM[KB]	Power [mW]	MIPS
Arduino	32	2	60	20
ChipKit uc32	512	32	181	124.8
STM32L476RG	1024	128	26	80
TI MSP432P4111	2048	256	23	58.56
BeagleBone Black	Ext.	524288	2300	1607
Raspberry Pi 3 B+	Ext.	1048576	5500	2800

As it can be seen from the table above, MSP432 has an average memory and storage resources, but also relatively low power consumption.

Spacecraft orbit
The satellite has been launched into a Sun-Synchronous orbit on January 13th, 2022. Table 2.4 shows a summary of the orbital parameters of Delfi-PQ as of June 2023:

Table 2.4: Delfi-PQ orbital parameters[48]

Parameter	Value	Unit
Perigee	457.5	km
Apogee	470.3	km
Inclination	97.5	deg
Orbital period	93.7	min
Semi-major axis	6834	km

Spacecraft temperature telemetry
The temperature telemetry contains the surface temperature measurements from the battery and 4 sides of the satellite: X+, X-, Y+ and Y- (the other two sides, Z+ and Z-, do not have sensors attached to them). The operational temperature limits for the battery are −50°C (min) and 70°C (max), while the temperature limits for the panels are −60°C (min) and 80°C (max). The satellite takes temperature samples every 15 seconds and transmits these samples along with other telemetry data every 60 seconds. [49]

Due to high orbital inclination, the satellite trajectory crosses the polar regions, where no ground stations can receive the telemetry down-linked by the satellite. This results in a significant fraction of the orbit (30% of the orbital period), for which no telemetry data is present. For instance, in Figure 2.7, it can be seen that the battery temperature data is missing for θ (true anomaly) values between 60° and 180°. This missing telemetry data could potentially be extracted from the SD card where all telemetry is stored, however, this is currently not possible due to a software bug.

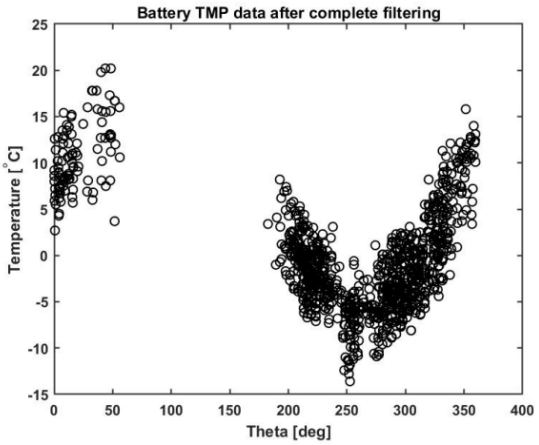


Figure 2.7: Delfi-PQ processed battery temperature telemetry [49]

2.5.2. Selection of the software resources

In order to build, train and deploy a neural network, the use is made of specialized machine learning software libraries. These libraries are collections of pre-written code and software tools that provide frameworks for developing and training NNs. Traditional ML libraries provide support mostly for the training of the neural networks, while TinyML libraries allow to optimize existing pre-trained neural networks and enable their deployment on edge devices. These two types of libraries will be explored in the subsections below. A comparison between multiple existing ML and TinyML libraries will allow to select those that are the most suitable for the training and deployment of NNs on Delfi-PQ and similar CubeSat OBCs.

Machine learning library for NN training

In Table 2.5, a comparison between some of the most commonly used ML libraries is presented. For each ML library, the right-most column of the table includes a comment on the suitability of the library for the defined use-case (training a temperature monitoring network for Delfi-PQ). The cell color of the column corresponds to the assessed suitability of a library (yellow = less suitable, green = more suitable). [50, 51]

As the Delfi-PQ operation team members mainly have Aerospace Engineering background and might not be experienced with machine learning, it is important that the developed software would not only be functional, but also intuitive when it comes to inspection and maintenance of this software. Therefore, the most critical factors that define the suitability of a library for the given application, are its functionality, user-friendliness and community support.

Table 2.5: Comparison of various machine learning libraries

Library	Programming language	Description	Community support, based on 2019 Github research [51]			Comment on suitability
			Stars	Forks	Contributors	
Keras	Python	Front-end ML framework. (API). Works with TensorFlow and Theano.	10,875	10,875	327	Intuitive API, extensive functionality, state-of-the-art optimizers, activation functions, etc. Excellent community support.
TensorFlow	Python, C++	Back-end ML framework. Works with Keras.	4505	667	573	Developed and promoted by Google, thus the good community support. Good speed performance and functionality.
Theano	Python	Back-end ML framework. Works with Keras.	5352	1868	271	More limited functionality and lower speed. Community support has decreased over time.
Torch / PyTorch	C, Lua, Python	Back-end ML framework + API.	6163	1793	113	Excellent speed performance. intuitive programming style. Good functionality. However, the community support is more limited and spread over multiple programming languages.
Caffe	C++, Python, MATLAB	Back-end ML framework + API.	15,057	9,338	222	Excellent speed performance, functionality is more limited than in TensorFlow. Hyperparameters tuning is more tedious than in other frameworks. Extensive community support which, however, is spread over multiple programming languages
MXNet	Python, R, Julia, Scala, C++	Back-end ML framework + API.	7471	2764	250	Framework with a good speed performance and functionality. Allows development using both imperative and symbolic programming paradigms, thus it is suitable for a wide range of applications. Intuitive front-end API. Community support is not as extensive as for other frameworks.

From the comparison, it is concluded that the combination of TensorFlow (as back-end environment) and Keras (as front-end environment) is the optimal choice for the defined application. This combination wins over other options thanks to the extensive functionality, user-friendliness and excellent community support.

TinyML library for on-device deployment
Various TinyML libraries for small devices (microcontrollers in particular) are compared in Table 2.6. In this overview, only TinyML libraries for C/C++ were included. Besides these libraries, there also exist so called TinyML code generation software platforms, such as Edge Impulse [52] and Imagimob [53], which have tools that enable automatic creation of TinyML code for a desired application. These tools however, make use of already existing libraries, such as TF Lite. TinyML code generation platforms are only suitable for a limited number of applications, for example, they cannot generate code that will perform on-device training of a ML model. For these reasons, code generation tools were left out from the overview in Table 2.6. The right-most column color corresponds to the assessed suitability of a TinyML library for deployment on MSP432P4111 (red = not suitable, green = suitable).[33, 4, 54, 55]

Table 2.6: Comparison of various TinyML frameworks and libraries

Framework	Source	Compatible devices	Supported NN types	Comment on suitability
TF Lite	Keras (TensorFlow)	Mobile devices, MCUs that can run C++	Basic NN types. RNN models need to be converted to TensorFlow Lite's fused LSTM operators.	Extensive support available, many examples. Model compression is not as high as in TF Lite Micro.
TF Lite Micro	Keras, TF Lite	Any 32-bit MCU	Basic NN types. Recurrent NNs have limited support.	Extensive support available, many examples. Has been tested extensively on Arm Cortex-M. Good model compression. However, only basic architectures are supported.
STM32Cube-AI	Keras, TF Lite, ...	STM32 only	All basic architectures	Only STM32 MCUs are supported.
MicroAI	Keras, PyTorch	Any 32-bit MCU	MLP, CNN, ResNet	Only one paper describes it, very limited support.
Microsoft Embedded Learning Library (ELL)	-	Devices that can run Python or C++ (RPi, Arduino etc.)	Basic NN types: FF, CNN	Very low level, not many examples are available.
CMSIS-NN	-	Arm Cortex-M based	Pretty much all basic NN types. For certain recurrent layers, workarounds are to be used.	Much better performance compared to other frameworks when deployed on Cortex-M. Downside: does not support certain recurrent layer architectures, workarounds required.
Neural Network on Microcontroller (NNoM)	Keras/CMSIS-NN	Any 32-bit MCU	All basic architectures (including recurrent layers) and some advanced: Inception, ResNet, DenseNet, Octave convolution, etc.	Can be used as an extension to CMSIS-NN. A list of examples is available.
DeepC	TensorFlow, PyTorch, LLVM, ONNX	Various ARM processors	All basic architectures	Limited support and not many examples

TensorFlow-Lite library is derived from classical TensorFlow, it provides a variety of functions, including various layer and node architectures for small-sized NNs. TF Lite can also compress existing large scale ML models into small models suitable for embedded applications. TF Lite Micro is a TF Lite variant optimized specifically for low-power devices, such as microcontrollers. Although it has more limited functionality compared to TF Lite, this framework is the optimal choice for the target application for the following reasons:

- It still has a much better community support much more examples than, for example, CMSIS-NN and NNoM.
- TFLite Micro is optimized for microcontrollers by reducing dependencies and unnecessary functionality. It therefore, achieves a significantly smaller code and memory footprint compared to TF Lite. TFLite Micro typically requires a few tens to hundreds of kilobytes of memory for the core runtime. [56] At the same time, the core runtime and dependencies of TF Lite can occupy a few hundreds of kB to a few MB of microcontroller memory. [57] This is a very critical selection criteria, as the available memory is highly limited in the case of MSP432P4111. Choosing TF Lite Micro over the regular TF Lite thus increases a chance of successful NN deployment on MSP432 and similar boards.

3

Methodology

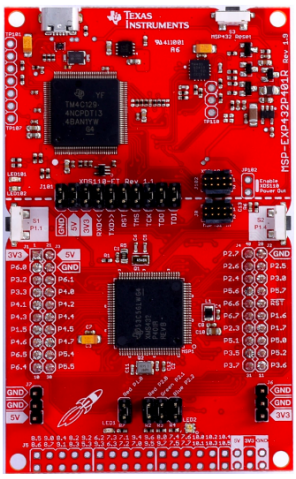
In this chapter, the methodology used to answer the research questions is discussed. In section Section 3.1, the software, hardware and data resources used to achieve the research objective are discussed. After that, the design problem to be solved will be discussed in more detail in Section 3.2. Finally, in Section 3.3 the design procedure applied to the development of the NN-based software is discussed. This section also provides an overview of the expected project results along with an explanation of the verification and validation procedures applied to these results.

3.1. Resources used

The research questions cannot be answered exclusively by further exploring the literature - a physical setup is required. The developed ML software has to work on the demonstrator OBC (Delfi-PQ) and the development boards that are available. For this project, the available development board features a TI MSP432P401R microcontroller. This microcontroller is almost identical to the TI MSP432P4111 used in Delfi-PQ, with the only difference being the available flash memory and RAM. MSP432P401R has only 256 kB of flash memory and and 64 kB of SRAM. TI MSP432P4111 is based on the Cortex-M processor core architecture developed by ARM and works with embedded software written in C or C++ [46], and the same goes for MSP432P401R. In fact, most microcontrollers that are used as spacecraft onboard computers, are programmed in C/C++, so the developed software will be quite universal and can be deployed on a variety of similar devices.

Figure 3.1 (a) shows the OBC of Delfi-PQ based on TI MSP432P4111 microcontroller. In Figure 3.1 (b), a TI MSP432P401R Launchpad microcontroller development board is shown:

(a) MSP432P4111-based Delfi-PQ OBC [46]



(b) MSP432P401R Development board [58]

Figure 3.1: Delfi-PQ onboard computer and MSP432P401R board

In addition to the software and hardware aspects, there is also the aspect of data resources. The neural network has to be trained on satellite telemetry data which is representative of the sensor readings that can be obtained in a real-life scenario.

The target application then necessitates the use of a variety of hardware, software and data resources, which will be further discussed below.

Hardware resources
The following hardware resources are used in the Thesis work:

- Microcontroller development board TI MSP432P401R Launchpad with power/data cords. A bread-board with some jumper wires and basic electronic components are also available to test the proper functioning of the board and try different functionalities.
- Personal laptop (or any other computer that can run the necessary software tools). For this project, MacBook Air (MacOS BigSur 11.6) is used.

Software resources
The software resources used in the project include Integrated Development Environment (IDE) for code development, libraries and add-ons, which are listed bellow:

- *Pycharm IDE* for all code development related to the testing environment design. Codes that process raw spacecraft telemetry, send this telemetry to the microcontroller, and process the testing results, are written in Python programming language (Python 3.8). Additionally, the neural networks that are deployed on the microcontroller, are initially written, trained and tested in Python on a development computer. This makes it possible to efficiently train, test and compare various NNs before deploying them to the microcontroller. To perform these tasks, Python is chosen over other programming languages for its extensive libraries for machine learning and data analysis, simplicity, readability, and strong community support.
- *Tensorflow* [59] Python library for machine learning. This library is an optimal choice for this project as it provides a variety of DL tools, examples, and an extensive community support, facilitating rapid development and deployment of such deep learning projects.
- TI *Code Composer Studio* (12.2.0) IDE to develop and deploy C/C++ embedded software for the MSP microcontroller. This IDE is an optimal choice as it is designed to work specifically with MSP and other microcontrollers produced by Texas Instruments.
- *Simplelink MSP432P401R SDK* [60] (3.40.01.02), a Software Development Kit (SDK) which provides various drivers for the MSP432P401R microcontroller, which significantly improves the efficiency of the software development process.
- *GCC Toolchain* [61] with *GCC* compiler and debugger is used to compile and upload the firmware for the microcontroller.
- *TF Lite Micro* [56] C++ library for embedded ML software development. The choice of this library is a result of a trade-off between multiple TinyML frameworks for C/C++ based microcontrollers. This trade-off can be seen in Table 2.6.

Data resources
As it was previously discussed in Subsection 2.5.1, the temperature telemetry received from Delfi-PQ is missing for a substantial part of its orbit. In addition to that, it is known that the Delfi-PQ telemetry very often corrupt and contains numerous wrong entries, which are attributed to hardware-, data storage-, timing- and transmission errors. This makes working with the satellite's telemetry particularly challenging, as preliminary filtering operations are required. These factors in combination with the highly limited amount of the recorded data, make the the Delfi-PQ telemetry a less favourable option when it comes to producing a data set for neural network training.

A feasible solution to the discussed problem is to use not the telemetry from Delfi-PQ, but a set of telemetry data from a similar satellite. One such satellite, for which the temperature telemetry is available and comparable to Delfi-PQ, is FUNcube-1 CubeSat. FUNcube-1 is a 1-unit (10x10x10 cm) amateur CubeSat launched on November 21, 2013. The satellite continuously transmits its telemetry to the ground via radio waves, which allows students and enthusiasts to receive and decode the telemetry [62]. FUNcube does not

have the same missing data problem that Delfi-PQ has. Its telemetry is also error-free, which apparently is due to more reliable onboard data handling and transmission. As can be seen from Table 3.1, the orbit of FUNcube-1 (as of June 2023) is quite similar to the orbit of Delfi-PQ in terms of size and inclination:

Table 3.1: FUNcube-1 orbital parameters[63]

Parameter	Value	Unit
Perigee	579.4	km
Apogee	651.0	km
Inclination	97.7	deg
Orbital period	96.9	min
Semi-major axis	6986	km

The temperature telemetry of FUNcube includes readings from 8 temperature sensors attached to different panels of the spacecraft (Black Chassis, Silver Chassis, Black Panel, Silver Panel, and 4 solar panels: +X, -X, +Y, -Y), where each sensor reading is sampled every 60 seconds. Figure 3.2 shows an example temperature profile measured by one of the temperature sensors for the duration of about 2 orbital periods:

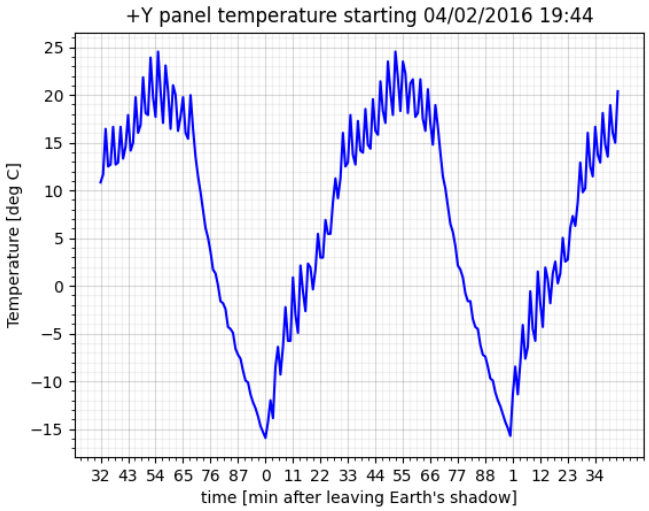


Figure 3.2: FUNcube-1 example temperature telemetry (2 orbits)

As it can be seen from the figure above, there is a notable effect of spacecraft tumbling, which introduces approximately +-3°C oscillation in temperature, with a period of about 170 seconds. This effect is present in all other parts of the available telemetry as well.

Due to a database problem that is still being solved by the FUNcube team, not all of the telemetry can be extracted. The available telemetry set, which is used for this Thesis project, contains a total of 1394 minutes of telemetry (that is 1394 temperature measurements from each sensor) obtained on February 4, 2016. The complete telemetry set plotted for some of the temperature sensors (those that are used to make the data set for NN training), can be found in Appendix A. In the telemetry set shown, one can easily identify the regions where all of the sensors simultaneously ”get stuck” at a particular value, which demonstrates one of the possible real-life scenarios where a fault occurs either in one of the sensors or in all of the sensors simultaneously, and they can no longer provide reliable measurements.

As the available telemetry spans only one day, and more data cannot be extracted at the moment, the dataset for this Thesis project will consist of both the original- and artificially modified telemetry data, which will allow to use more data for training.

3.2. Problem analysis and the proposed solution

Once the available resources are identified, the problem and the proposed solution can be specified in more detail. This includes defining the exact anomaly types that the neural network system should be able to detect, and the main parts of which this software system will consist.

3.2.1. Anomaly types

In order to design an effective model for anomaly detection, a clear distinction has to be made between the nominal and anomalous behavior in the satellite temperature sensor readings.

The analysis of the available FUNcube-1 telemetry (example of which can be found in Figure 3.2) has shown that at a particular moment in time all temperature sensors have roughly the same temperature, which can vary in the range of approximately +-3°C due to tumbling. The analysis of one of the Delfi-PQ sensor readings collected over a period of about 2.5 months, as shown in Figure 3.3, reveals that the vast majority of the sensor readings fall within the range of approximately +-7°C from the general trend line.

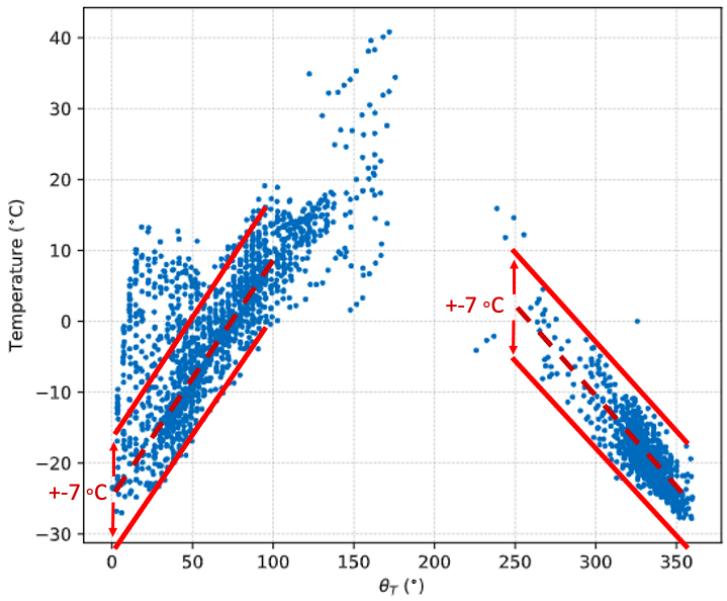


Figure 3.3: Delfi-PQ Y+ panel unfiltered temperature telemetry from 2022-01-18 to 2022-03-29 [64]

Following this analysis and using a safety factor of 1.4, a temperature anomaly can be defined as any temperature reading that deviates by more than 10°C from the expected value. Such expectation (prediction) is based on the current orbital position or system time, temperature profiles registered over previous orbits or extrapolation of the previous N-points of the temperature readings. A NN-based system could potentially be able to detect such small-sized anomalies in the temperature sensor readings long before they grow into anomalies that exceed the safe operating range and cause damage to the spacecraft. This approach would thus be more beneficial than the traditional approaches that rely on using thresholds on minimum and maximum allowable temperature values. Such threshold approach can only detect anomalies when the temperature has already exceeded the thresholds that are set beyond the nominal operating range. In contrast, the NN-based approach could detect faults at early stages of their development, which would allow for more time to isolate the faults.

Based on the definition previously mentioned, any individual reading that exceeds 10°C threshold condition is considered to be an anomaly. However, these readings can be classified into various types of anomalies, which may have different origins and consequences. The four main types of anomalies are identified: outliers and flat regions, permanent bias, temporary bias, and clock anomaly. The current approach to managing these anomalies relies on using thresholds for the minimum and maximum allowable temperature. This means that the fault isolation and recovery procedures can only be applied if the value read from a temperature sensor is not within the allowable range. However, it is common for various anomalies to occur also within the allowable temperature range, where they cannot be detected by the current approach.

With a more advanced approach to FDIR, any faults could be timely detected and isolated before they cause any damage to the spacecraft.

The mentioned anomaly types will now be further discussed below.

Outliers and flat regions
Outliers (spikes) and flat regions in the sensor readings are anomalies related directly to the functioning of the temperature sensors, as the temperature itself cannot be realistically changing at such a fast rate that outliers would imply, nor can it be constant as in flat regions given the dynamic environment of the low Earth orbit.

Two instances of outlier anomaly manually introduced to FUNcube telemetry can be seen in Figure 3.4:

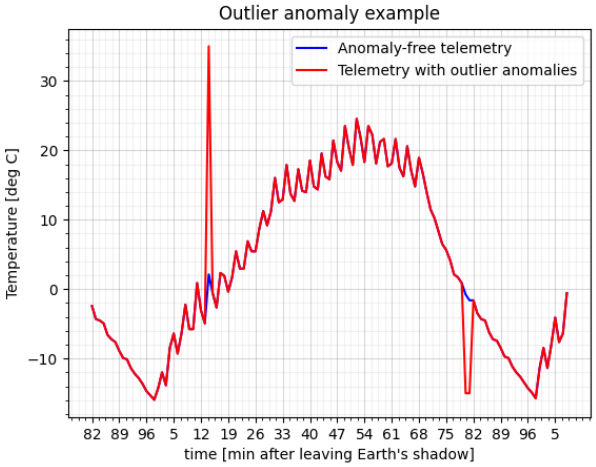


Figure 3.4: Anomaly free telemetry vs telemetry with 2 outliers

Outliers can result from individual errors in the temperature measurements, which can be attributed to either signal attenuation due to noise sources or computational errors. As individual outliers are present for just 1 or 2 consequent measurement points, they are unlikely to cause significant effect on the satellite operations and damage the satellite in any way.

Two examples of flat regions are shown in Figure 3.5:

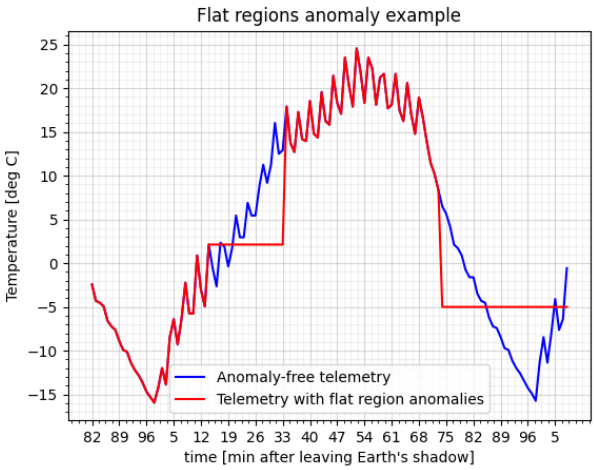


Figure 3.5: Anomaly free telemetry vs telemetry with 2 flat regions

Unlike individual outliers, flat regions persist for long periods of time, and are attributed to either temporary (but consistent) errors in temperature readout, or by permanent sensor- or readout failure. This type of anomaly can potentially compromise the operation of the satellite, as it prevents the OBC from receiving the correct temperature readings, rendering it unable to detect the dangerous conditions where the temperature is actually too high or too low.

Permanent bias
Permanent bias is another type of anomaly, where the measured temperature profile is uniformly offset from the nominal temperature profile. It can be caused either by a fault in the temperature sensor, such as a persistent error in the readout or sensor degradation, or by a uniformly increased temperature on one of the satellite panels. This could indicate that there is a change in the satellite attitude dynamics, which makes one of the satellite sides exposed to sunlight more (or less) than other panels. Additionally, this anomaly could be a sign of excessive heat being generated by spacecraft subsystems or instruments.

An example of constant bias is shown in Figure 3.6:

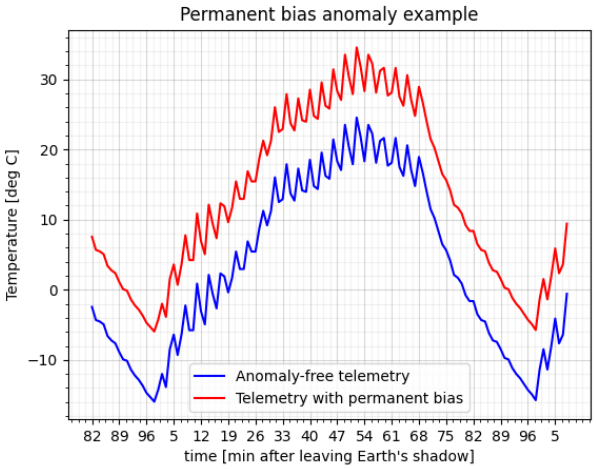


Figure 3.6: Anomaly free telemetry vs telemetry with permanent bias

It is crucial to detect this anomaly and identify its cause. In case the permanent bias is caused by decreased or increased internal heat generation, it would indicate that one of the spacecraft subsystems or payloads does not function properly. Then the fault needs to be isolated in order to restore the normal operation of the affected component or at least to prevent the anomaly from spreading and affecting other components. In the case the error is caused by a sensor problem, the permanent bias could be estimated and subtracted from the measured temperature in subsequent operations.

Temporary bias
Temporary bias anomaly is a noticeable deviation of the measured temperature from the expected one, which however, disappears after some time. This behaviour could be caused by one of the satellite components rapidly heating up or cooling down to abnormal temperatures.

Two instances of temporary bias anomaly can be seen in Figure 3.7.

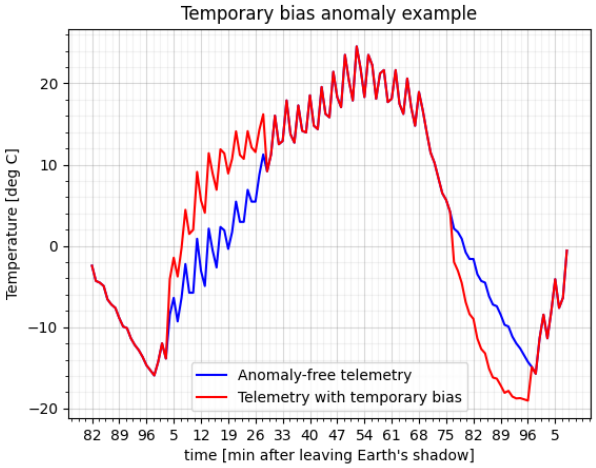


Figure 3.7: Anomaly free telemetry vs telemetry with two instances of temporary bias

In this example, each of the anomalies lasts for approximately 20 minutes, after which gets resolved by itself. However, it could also be that the temperature would keep increasing/decreasing, i.e. drifting away from the nominal temperature, thus progressively increasing the risk of component failure. Therefore, it is important that these anomalies can be detected quickly and correctly attributed to potential faults in specific components.

Clock anomaly
Clock anomaly happens when incorrect timestamps are assigned to the measured temperature profile data points. Then, the measured temperature profile would appear shifted with respect to the temperature profile expected for the given time series. The cause of this anomaly lies solely in the clock reading of the satellite. In Delfi-PQ such anomaly is caused by a periodic reboot of the satellite due to power drops. The satellite reboot causes the clock to reset, and incorrect timestamps get assigned to the temperature measurements.

Example of a clock anomaly is shown in Figure 3.8:

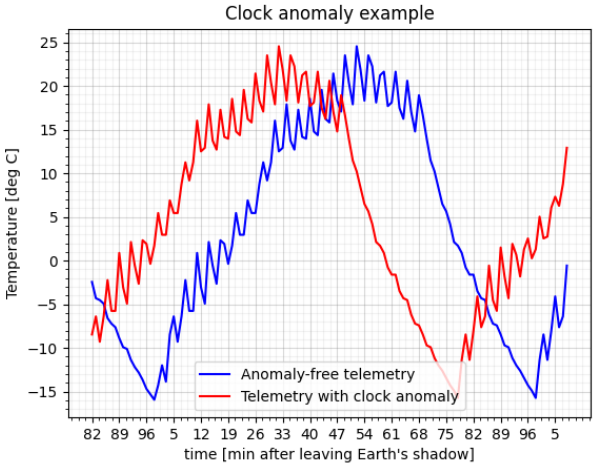


Figure 3.8: Anomaly free telemetry vs telemetry with clock anomaly

Delfi-PQ OBC clock gets reset once every orbit due to a software error, which reboots the satellite when it is in the Earth's shadow and the voltage at the spacecraft's battery drops. This makes it impossible

to reliably attribute a particular temperature reading to a particular time stamp, without applying a data processing algorithm, which estimates the series of the time values from other knowledge sources, for example, the temperature readings.

In principle, this anomaly is harmless to the satellite as long as the temperature does not cross the minimum and maximum allowable temperature thresholds. This anomaly, however, makes it very challenging to implement more advanced temperature monitoring algorithms, which would enable faster detection of anomalies. The same goes for the neural network -based anomaly detecting model designed during this Thesis project. This anomaly type would therefore need to be addressed along with the other sensor- and temperature anomalies.

3.2.2. Concept for the NN-based embedded software system

As it was stated in Chapter 1, the objectives of the research project include developing a NN-based system for temperature monitoring and a testing environment for such system, and then, testing the performance of the developed system. Conceptually, this means that the Thesis work is split into three main parts, namely:

- **Creating the testing environment**, which includes:
    1. Compiling one of the TinyML libraries (TF Lite Micro) on a microcontroller board, which is representative of that used as the OBC of Delfi-PQ.
    2. Setting up a temperature telemetry processing system on a computer, where this telemetry is stored.
    3. Setting up a communication system between the microcontroller board and a computer, where the NN model- and testing data is stored.
    4. Developing a system which will automatically test NNs on the microcontroller, then record and store the testing results.
- **Developing a neural network**, which will be able to:
    - Receive the temperature sensor readings as an input.
    - Produce an output, which will indicate the probability of anomaly or anomaly status (e.g. 1/0) at a particular sensor or a combination of sensors.
- **Deploying the NN on the board using the developed testing environment and measuring its performance**. This is the integration and testing part of the project.

3.3. The design approach

As it was said earlier in Subsection 3.2.2, the thesis work generally consists of three main parts: Creating the testing system/environment for the MCU, developing a neural network for temperature monitoring, and finally, performing integration and testing. These parts, in their turn, further break down into a number of steps, which define the general workflow of the thesis project. This workflow consists of 5 steps, which are explained below:

0. To begin with, a definition of system requirements, expected results, and Verification and Validation (V&V) procedures for these results has to be performed.
1. Once the requirements and V&V procedures are defined, the environment for the MCU testing is developed. At this step, the TF Lite Micro library is deployed on the microcontroller and it is tested with a few simple NN models. Then, an interface between the MCU board and the development computer is established and algorithms for the processing of the testing results are implemented.
2. After that, begins the phase where the neural network which will later serve as a main demonstrator of the system's capabilities is designed. This part of design is focused mainly on defining the necessary inputs and outputs of the network.
3. Then, the neural network is designed in detail, which among other activities includes tuning the hyper-parameters of the network and optimizing the network for size, so that it can be deployed on the MCU board.

4. Finally, the designed network is deployed on the board using the testing environment prepared earlier, and the performance of the network is measured according to the results collection procedures defined in step 0. At last, the V&V procedures are applied to confirm the quality of the obtained results.

This workflow description is summarized in Fig. 3.9:

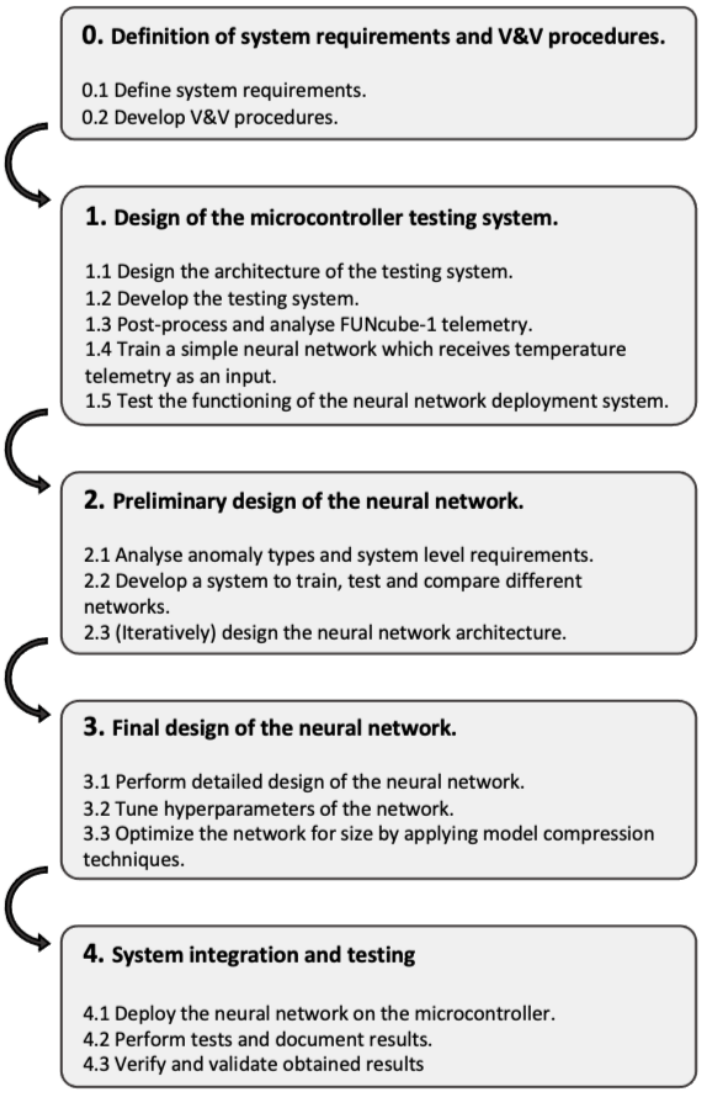


Figure 3.9: The design process of the software system

The following two sections will be focused on the design phase 0, which includes the definition of requirements, expected results and V&V procedures. These definitions lay the foundation of the design process and the methodology that is used to arrive at the project results.

3.3.1. System requirements for the embedded software system

After an inspection of the available Delfi-PQ telemetry parts, the following important assumptions have been made, to produce more specific and realistic system requirements that will drive the design process:

- There is less than 1 anomaly happening within an orbital period (93.7 minutes). The network should be able to detect at least 1 anomaly within 1 orbital period, if any are present.
- A sensor reading is considered indicative of an anomaly if its value deviates from the expected value by more than 10 degrees. Such expectation (prediction) is based on the current orbital position or

system time, temperature profiles registered over previous orbits or extrapolation of the previous N-points of the temperature readings.

- The samples of the temperature sensor readings are all equally spaced in time. That is, the clock frequency is stable over a long time and obtaining the temperature samples always takes the same time period, which is currently equal to 15 seconds.

These assumptions, together with the obtained knowledge on the Delfi-PQ system design and the knowledge of the MSP432P401R development board specifications, result in a number of system requirements, which the NN-based software system designed in the course of this Thesis project should be able to meet. These system requirements are summarized in Table 3.2:

Table 3.2: System requirements for the NN-based embedded software system

Category	ID	Description
Neural network performance	SW-PERF-01	The neural network shall be able to detect all anomaly types that are identified (outliers, flat regions, temporary and permanent bias)
	SW-PERF-01-A	The neural network shall detect any sensor reading that deviates by more than 10 degrees from the expected value.
	SW-PERF-01-B	The neural network shall be able to perform 1 inference in less than 15 seconds.
	SW-PERF-01-C	The neural network shall be designed to handle at least 4 temperature sensors and be scalable to handle more in the future.
	SW-PERF-01-D	The neural network shall be able to detect at least 1 anomaly every 93.7 minutes.
	SW-PERF-01-E	The neural network shall be able to specify the sensors which indicate an anomaly and those that do not.
Model deployment	SW-DEPL-01	The software shall be compatible with TI MSP432P401R microcontroller.
	SW-DEPL-02	The software shall be developed in C/C++ language (to be compatible with other Cortex-M based microcontrollers).
Memory management	SW-MEM-01	The software shall take up less than 256 kB of flash memory storage space.
	SW-MEM-02	The software shall use less than 64 kB of RAM in any operation mode.
Safety	SW-SAF-01	The software shall not cause harm to the satellite or other satellites.
	SW-SAF-01-A	The designed software shall not compromise the operation of other embedded software.
	SW-SAF-01-B	The software shall not compromise the working of other satellite subsystems.
	SW-SAF-01-C	The software shall not cause unintended reboot of the microcontroller it is deployed on.
Maintainability	SW-MNT-01	The software shall be easy to update, modify, and troubleshoot as needed.
	SW-MNT-01-A	The code shall be easily readable (properly structured and commented).

3.3.2. Experimental results

In order to answer the main research questions, the performance of the developed ML software needs to be assessed in terms of the accuracy of anomaly/failure detection, i.e. how many % of anomalies can the ML model correctly identify. Also, does the model wrongly interpret some nominal behavior as a failure/anomaly (yes/no) and if yes, then in how many % of the cases this happens? The choice of metrics that are applied to quantify the accuracy of the model's predictions varies based on the nature of the output of the network.

The output of the preliminary versions of the model (before hyper-parameters tuning) is expected to be a float value between 0 and 1. This float value translates into the estimated probability of an anomaly happening at the moment a particular inference of the model is performed. In this case, it is reasonable to use the following two metrics:

Accuracy of anomaly detection in Equation (3.1) shows in how many % of cases the model detects an anomaly when it is supposed to do so, that is when it performs inference on data which is labeled as anomalous.

$$A_a = \frac{\sum m_a}{N_a} \tag{3.1}$$

Where m_a is an individual output value of the model when tested on anomalous data, ranging between 0.0 (no anomaly) and 1.0 (anomaly). $\sum m$ is a sum of all of such output values. N_a is the number of times the model is tested on anomalous data.

Accuracy on NO anomalies in Equation (3.2) shows in how many % of cases the model does NOT detect an anomaly when it is NOT supposed to detect one, that is when it performs inference on data which is labeled as correct or NON-anomalous.

$$A_{na} = \frac{\sum(1 - m_{na})}{N_{na}} \tag{3.2}$$

Where m_{na} is an individual output value of the model when tested on NON-anomalous data, ranging between 0.0 (no anomaly) and 1.0 (anomaly). $\sum m_{na}$ is a sum of all of such output values. N_{na} is the number of times the model is tested on NON-anomalous data.

The output of the final version of the model is, however, expected to be binary: a threshold is applied on the anomaly probability estimated by the model, thus there is either an anomaly, or there is NO anomaly. This implies that any output of the network falls into one of the four categories shown in Table 3.3:

Table 3.3: Output types of a binary classifier network

		Model output	
		1	0
Actual data label	1	True positive (TP)	False negative (FN)
	0	False positive (FP)	True negative (TN)

Then, the following metrics are applied to quantify the anomaly detection effectiveness based on the numbers of true/false positive/negative outcomes:

Recall, given by Equation (3.3), shows how well the model detects faults/anomalies when those are actually present.

$$\text{recall } = \frac{TP}{TP + FN} \tag{3.3}$$

Precision, given by Equation (3.4), shows how many of the positive outputs were true positive.

$$\text{precision } = \frac{TP}{TP + FP} \tag{3.4}$$

F1-score, given by Equation (3.5), is a combination of recall and precision, and thus can be viewed as a general effectiveness measure of the classifier network.

$$F1 = \frac{2 * \text{ precision } * \text{ recall}}{\text{precision } + \text{ recall}} \tag{3.5}$$

False Alarm Rate (FAR), given by Equation (3.6), shows how often the model produces a false alarm output.

$$FAR = \frac{FP}{TN + FP} \tag{3.6}$$

Additionally, there are other performance parameters, which need to be measured to assess the feasibility of deploying the designed software system onboard a spacecraft. These parameters include:

- **Time per one inference** of the network, in seconds.
- **RAM usage** in kB.
- **Flash memory usage** in kB.

3.4. Verification and validation approach

For each variables that quantitatively represent the experimental results, as was discussed in Subsection 3.3.2, a specified measurement method and the associated results verification and validation methods are defined. These definitions, being the outcomes of phase 0. of the design project, can be found in Table 3.4:

Table 3.4: Experimental results and V&V procedure

Experimental result	Units	Measurement technique	Verification of results (was the method applied correctly?)	Validation of results (are the results correct?)
Accuracy metrics A_a, A_{na}, recall, precision, F1, FAR	[-]	Select test data sets, run the embedded software and count each outcome type. Compute metrics.	Code verification by unit testing. Hardware verification by ensuring that the wiring was done correctly.	Evaluate accuracy on various sets of data samples. Perform sensitivity analysis.
Anomaly detection time delay	s	Run software on a number of test data samples, measure delay with inbuilt microcontroller clock or an external clock.	Code verification by unit testing. Hardware verification by ensuring that the wiring was done correctly.	Confirm clock readings with a different internal or external clock, or analytically.
Memory consumption	MB	Measure using IDE inbuilt memory monitor.	Code verification by unit testing. Check for correct data reception/ transmission via the interfaces.	Use built-in memory monitoring tools of the IDE used. Confirm results with another method, e.g. analytically or with a memory profiler software tool or by using a memory monitor peripheral of the microcontroller (if available).
RAM consumption	MB	Measure using IDE inbuilt memory monitor.	Code verification by unit testing. Check for correct data reception/ transmission via the interfaces.	Use built-in memory monitoring tools of the IDE used. Confirm results with another method, e.g. analytically or with a memory profiler software tool or by using a memory monitor peripheral of the microcontroller (if available).

4

Microcontroller Testing System Design

The first phase of the main design work of this Thesis project is the development of an environment for the MCU testing. Such environment is necessary as it allows to efficiently deploy neural networks on the development board, and test them automatically by feeding test telemetry data sets and registering the model outputs. As the first step at this phase, the TF Lite Micro library is deployed on the MCU and it is tested with a few simple NN models. After that, an interface between the MCU board and the development computer is established and algorithms for the processing of the testing results are implemented. The general architecture of the developed testing system is discussed in Section 4.1. The algorithms used to build the necessary (embedded) software will be explained in Section 4.2. Then, the documentation of the testing results will be described in Section 4.3. Finally, the performance of the testing system will be discussed in Section 4.4.

4.1. Testing system architecture

The testing system architecture consists of the TF Lite Micro library deployed on MSP432P401R microcontroller, as will be explained in Subsection 4.1.1, Python codes deployed on the development computer as will be briefly discussed in Subsection 4.1.2, and the communication interface between the development computer and the MCU board, described in Subsection 4.1.3. All the Python and C++ programs discussed further in this chapter can be found in the GitHub repository of the Thesis project, which can be accessed via the link is provided in Appendix B.

4.1.1. TF Lite Mirco deployment on MSP432

TensorFlow Lite Micro C++ library is deployed on the MSP432 microcontroller with the aid of the procedure provided in an example TF Lite Micro project created by Texas Instruments [65]. In this example, the TF Lite Micro library is built to work specifically on TI devices. In order to deploy this library in the development environment (Code Composer Studio) and upload it to the MSP432 board, the use is made of Simplelink MSP432P401R SDK and GCC toolchain (GCC compiler and debugger).

SimpleLink SDK for MSP432 microcontrollers is designed to simplify the development of IoT applications by providing a comprehensive set of software components and tools. One of the benefits of using Simplelink SDK is that it provides a number of pre-built peripheral drivers for various MSP432 hardware peripherals, such as GPIO, UART, SPI, I2C, etc. With these drivers, communication between the board and external devices can be implemented more efficiently.

Another software package previously mentioned, the GCC Toolchain, includes the GCC compiler and GNU debugger. The GCC compiler is a widely used open-source compiler that supports C and C++ programming languages. It is used to convert the C/C++ source code into machine code that can be executed on the MSP microcontroller. Another software component provided in this toolchain, the GNU Debugger, is an open-source debugger that allows to examine the execution of the embedded code on the microcontroller board. Among the features this debugger provides are breakpoints, stepping through code and variable inspection.

The example project provided by TI, includes a simple neural network which is trained to construct a sine wave, $y = sin(x)$, meaning that it predicts a single y value based on a x value in range from 0 to π radians. The input value is initially of type $float32$, but it gets quantized to $int8$ type before it is put into the

input tensor. Similarly, the output of the network is dequantized from $int8$ to $float32$. Table 4.1 shows the memory used by the entire project, the RAM and flash memory size it uses once deployed on the microcontroller, and the size of the actual neural network, stored in the flash memory of the board:

Table 4.1: TI example project memory specifications (hello_world_MSP_EXP432P401R_nortos_gcc)

Parameter	Value	Units
Project storage space	50.8	MB
NN model size	2312	B
Flash use on MSP432	238k	B
RAM use on MSP432	35k	B

It can be seen that the embedded software project occupies 238k out of available 262k bytes, or about 90%, of the flash memory of the microcontroller and 35k out of 65.5k, or 53%, of available RAM bytes. At the same time, the actual network is just over 2.3k bytes in size, which means that the greatest consumer of flash memory in this case is the actual TF Lite Micro library, and in particular, its operations resolver (*AllOpsResolver*), which supports numerous mathematical operations needed for neural networks functioning. At the same time, there still remains a significant fraction of both flash and RAM, to accommodate for larger neural networks. In case the NN model data will be stored in flash, it can occupy up to 26.3k bytes, and if it is stored in RAM, it can be up to 30.5k bytes in size, assuming the rest of the code remains similar in size, i.e. the same *AllOpsResolver* is used.

The code for the actual NN testing system developed in this Thesis project, however, differs in multiple ways from the TI example code, even though it makes use of the same TF Lite Micro library and the software packages previously discussed. The structure and functioning of this embedded code will be further discussed in Subsection 4.2.1.

4.1.2. Python codes for telemetry processing

The primary function of the Python codes is to process raw FUNcube-1 spacecraft telemetry and convert it into data sets, which can be used to train and test the temperature monitoring neural network designed in this Thesis project.

From the FUNcube temperature telemetry, only the readings from the four sensors attached to solar panels are selected, these are +X, -X, +Y, and -Y panels. This selection is done to match the placement of the temperature sensors of Delfi-PQ, which are also located on +X, -X, +Y, and -Y sides. With the Python codes, this telemetry is first filtered to remove the regions with the constant sensor values (this problem was discussed under **Data resources** in Section 3.1) and split into separate text files. An example of the discarded and the remaining parts of the telemetry is shown in Figure 4.1. It can be seen, that the telemetry gets split into 7 regions (parts) of 'clean' telemetry.

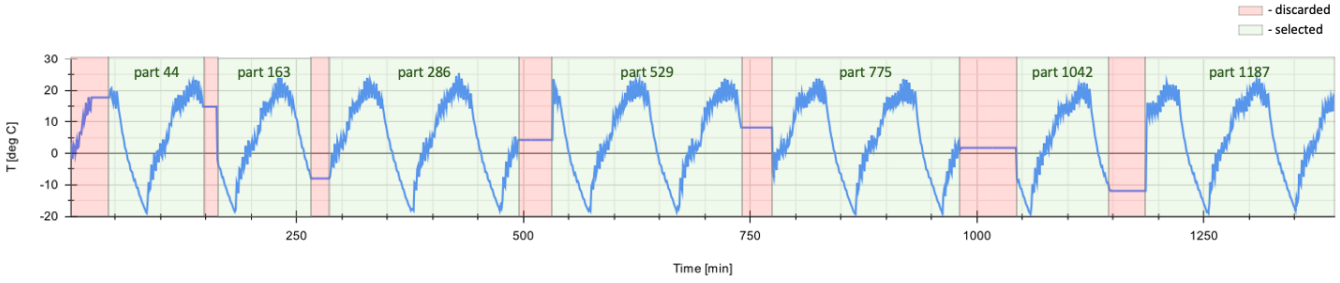


Figure 4.1: Discarded regions of telemetry from FUNcube side +X.

After splitting the files, individual files of filtered telemetry are made for each sensor and each part of the telemetry. Having the telemetry processed in this way, allows to efficiently introduce artificial anomalies into any sensor for any specific telemetry part.

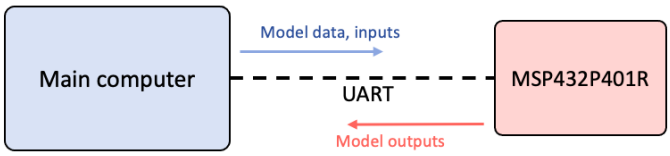
Then, both the 'clean' data and the data with artificially introduced anomalies (anomalies will be further discussed in Section 5.1) are stored in the working directory as text files, where every line contains one normalized (to the range from 0.0 to 1.0) temperature sensor value. These files are thus ready to be read and the temperature values can be sent one-by-one to the microcontroller over the communication interface discussed in the following subsection.

4.1.3. Computer-to-board communication interface

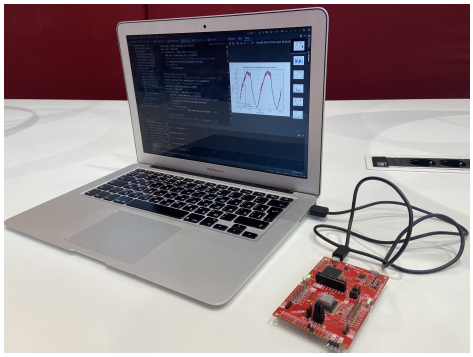
The communication between the development computer and the microcontroller board is achieved by setting up a UART interface. On the Python side, this is achieved by using the *Serial* module, which allows to read from and write data into the serial port. On the side of the microcontroller board, the same reading and writing functionality is implemented using one of the drivers conveniently provided by Simplelink SDK, called *UART*. This *UART* driver then becomes the main data exchange channel used by the embedded software, which allows to remove the "Display" driver and "micro_error_reporter" functionality, which was originally used in the examples provided by TI. Thus the size of the program can be slightly reduced.

During the testing of a particular neural network, the data exchange over the UART interface connecting the development computer and the MCU board, happens as follows: first, the computer sends the size of the NN model data, which will be used to correctly initialize the model when the actual model data will be transmitted. Then, the input and output sizes of the model are sent in order to initialize model input and output tensors of the corresponding size. After that, the actual neural network model data is sent via UART as a C-array and the model is initialized at the MSP board. Then the input data, for example, temperature sensor readings (telemetry data points) are sent one by one via UART, and the MSP performs inference with the initialized model and sends back neural network outputs one by one. The Python code in its turn, reads the data from UART and produces Matplotlib plots with the results. Then the Python code puts all the resulting plots into an HTML page report for readability of the results.

A simplified schematic representation of this system and the data exchange within it is shown in Figure 4.2 (a). The physical setup of this system can be seen in Figure 4.2 (b).



(a) Testing system architecture. UART interface.



(b) Physical setup

Figure 4.2: Testing system architecture and the physical setup

4.2. Algorithms used

Although the resulting software cannot be shown and explained entirely in this Thesis report, the general principles and algorithms behind its operation can be briefly discussed. Subsection 4.2.1 focuses on the embedded software developed for the MSP432 microcontroller. The operation of the Python code deployed on the main computer, is discussed in Subsection 4.2.2.

4.2.1. TF Lite Micro C++ on MSP432

As it was previously explained, all the model- and testing data is received by the MSP board via the UART driver. One downside of this driver is that it does not support all data types, but only unsigned integer and $char$ types (while the temperature telemetry data is of type $float$). This means that $float$ temperature values cannot be received directly, but should either be quantized to int or converted to $string$ before they are transmitted. Another issue which complicates the UART communication is the fact that the

microcontroller should not only receive the model- and input data, but also certain commands which tell the microcontroller what to do with this data. These commands must never be confused with the actual data, which might actually happen if both the commands and data are integer numbers. It should also be noted that all the received data should be of the same type, as it would not be possible for the *UART_read()* command of the UART driver to distinguish between the data types. The chosen solution to this problem is to always use $string$ ($char$) type for the transferred data and commands. In this way, the commands can be encoded as alphabetical characters, and the actual data in numerical, characters. The code would first check for any of the commands represented by a set of known alphabetical characters, and if they are not found, the conversion to other types can be performed. Table 4.2 shows all the data types received by the microcontroller and the type conversions it performs to convert the received data to their actual types:

Table 4.2: Data types received by the microcontroller

Received data	Example	Actual type	Received as type	Function(s) to convert to actual type
Model data	0x1c, ...	byte	char buffer	strtok(), strtoul()
Model size, input/output size	1000 10/1	int	char buffer	strtoul()
Input data	10.1234	float32	char buffer	string_to_float() (custom)
Commands	'i', 'f'	char	char buffer	-

Program setup
Before the microcontroller enters the main loop of the program, where it performs model inferences on the input data, a setup part of the code is run first. In this part, the microcontroller first initializes the UART communication at the specified baud rate, after which it can receive the NN model size (how many bytes does model data consist of), the input and output sizes (how many neurons are there at input and output layers) of the model. These are integer numbers, which are, however, initially formatted as $string$. In order to convert them to int, the *strtoul()* function of the standard C library is used.

Then, the MCU reports on the received variable values over UART, after which is starts receiving the model data. The model data consists of individual $byte$ values, which are, however, formatted as $string$. These values are each converted to $byte$ using *strtok()* and *strtoul()* functions from the standard C library and stored in an array of the size received previously. To illustrate this, an example model data array is shown below:

Example model data

```
unsigned char model_data[model_size] = {
0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, //…
}
```

Once the model data is received, it is mapped into a TF Lite model data structure to be passed into the TF Lite interpreter, which is the main component that handles the execution of the model's operations.

Program main loop
In the main loop of the program, the embedded code reads the string messages received over UART interface and first checks whether the latest received string contains a command character, which can be either 'i' or 'f' character. The functions of these two command characters are as follows:

- **'i'** command character indicates that the model should perform network inference with the input tensor that it has assembled by the current loop iteration. The inference is not performed until this command is received.

- **'f'** command character indicates that a particular testing data set has ended, and the input tensor should be cleared to prepare it for the next data set.

If the string received at a particular iteration contains no command characters, the code converts each such string to a float value. In order to implement this conversion functionality, additional functions were introduced to convert received string to float and float to string. Once an individual float value is received, it is quantized into int8 (8-bit quantization). This quantization technique is the most commonly used one in TinyML, and as was shown in Table 2.2, it is an optimal choice for almost every microcontroller application. The reduction of memory footprint, increased memory size and good accuracy preservation also make this quantization technique a good choice for the anomaly detecting system discussed in this Thesis. The 8-bit quantization includes the following mathematical operations:

1. Normalization: first, float values are normalized within a certain range. The range is determined by the minimum and maximum values that the TF Lite model has been trained to work with.
2. Scaling: next, the normalized float values are mapped to the target range of int8 values (-128 to 127).
3. Rounding: the result of scaling is rounded to the nearest integer.
4. Clipping: in case, the resulting value is beyond the -128 to 127 range, the value is clipped to fit within the this range.

In the code, this looks as:

8-bit quantization example

```
float input_float = 3.14;
int8_t quantized_input = static_cast<int8_t>(input_float /
                         input->params.scale + input->params.zero_point);
```

Here, *params.scale* and *params.zero_point* are values that are obtained automatically from the model configuration data.

After the quantization has been performed, the quantized value is passed into the model input tensor.

A circular buffer is used as the model input tensor. The size of this buffer is set to be equal to the "input size" received via UART at setup. The circular buffer is initialized when the first model input value is received. At the initialization, all of the entries of the circular buffer are set to the same value, which is exactly the first model input value. As the subsequent model inputs are received, the values in the circular buffer get shifted by one position every time, and at a certain point, the values which the buffer was initialized with vanish.

This code architecture allows the system to work with two kinds of input tensors (arrays of data):

- **Case 1:** If the model input consists only of **sequential readings from a single (temperature) sensor**, the input values are sent to the microcontroller and put into the circular buffer one by one. In this case, inferences can be performed every time the new value is received. This means that the Python code sends the 'i' command flag after each float value.
- **Case 2:** If the model input consists of **multiple concatenated sequences or non-sequential data**, the input values are also sent to the microcontroller and put into the circular buffer one by one. However, this time, the inferences are performed only when a certain number of input values has been received. In this case, the values in the circular buffer should all change to the values of the new input series before the next inference is performed. Only after that, the Python code sends the command character 'i' and network inference is run.

The code architecture has been designed specifically to work with these two types of inputs. This makes the entire system suitable not just for the desired temperature monitoring application, but potentially for other kinds of applications as well. When there is a single series input, the testing process can be sped up by the fact that the previous values of the series get saved in the circular buffer, and only the next point is needed to perform the next inference. When the input is not a series of a single sensor readings, the system still works, but the testing time is longer.

Every time the 'i' flag is received and model inference is performed, an output tensor is produced of the size determined in the setup. The program first de-quantizes the entire output tensor to $float32$ type. Then it loops through the elements of the output tensor, coverts each element to a string using a custom written function *float_to_string()*, and sends the output element over UART with an accompanying 'Output' $string$ label.

A pseudo-code (simplified version of the actual code) that demonstrates the structure of the program discussed earlier, including the setup, loop and main parts, is presented below:

Pseudocode for MSP432P401R embedded software

```
//main_functions.c
#include <ti/drivers/UART.h>
#include "tensorflow/lite/micro/…"

void setup() {
  UART_receive_model_data_size();
  UART_receive_model_input_size();
  UART_receive_model_output_size();
  UART_receive_model_data();
  initialize_TF_Lite_interpreter();
}
void loop() {
  UART_receive_input_data();
  if("f"_flag_read == true){
    clear_input_tensor();
  }
  make_input_tensor();
  if("i"_flag_read == true){
    run_tensorflow_inference();
    UART_send_output_data();
  }
}
int main(){
  setup(); //run once
  while(true){ //run forever
    loop();
  }
}
```

Overall, this embedded software is designed to avoid as much as possible re-uploading of the program to the microcontroller. This is achieved by ensuring that the variable parameters, including model specifications and data, are received externally and not hard-coded in firmware. However, there are still certain parameters which have to be hard-coded as it is technically impossible to initialize them with values received during the program run time. These fixed embedded parameters, and their values specified in the latest version of the testing system, are shown in Table 4.3:

Table 4.3: Fixed embedded software parameters

Paraeter	Value	Unit
Tensor arena size	4000	Bytes
Baud rate	115200	bit/s
Model data read buffer size	100	Bytes
Model input read buffer size	100	Bytes

The tensor arena is the RAM space where the input tensors, intermediate tensors created during computation, and the output tensors are allocated. Experimentally, the optimal value for the desired kind of neural networks was determined to be 4000.

Baud rate of the UART communication could be increased to achieve faster data exchange, but it was determined that 115200 baud is optimal for the given application in terms of speed and reliability.

Model data- and input read buffer size varies depending on the length of the strings to be received. 100 Bytes proved to be sufficient for reading float values rounded to at least 4 decimals.

4.2.2. Python code to test TFLite Micro on MSP432

Once the telemetry data sets are prepared and stored as described in Subsection 4.1.2, a different Python program sends the data from these data sets to the MSP board. Besides this processed input data, the program requires TF Lite model data, which has to be stored as a C style array. This model data is obtained after first converting a trained Tensorflow model with extension *.h5* to a TF Lite model with extension *.tflite*, and then extracting the bytes from this *.tflite* model and storing them in a C array in a *.h* file. The creation, training and conversion of such models will be further discussed in Chapter 5.

Configurations file

In addition to the input- and model data, the program requires a configurations file, which contains all parameters specific to a particular test and the TF Lite model being tested. These parameters include:

- Test-specific parameters: number of separate data sets used, input data type (sequential/non-sequential), the specific tasks to be performed during the testing sequence (plot the results (y/n), create the report (y/n), etc.).
- Report parameters: directory where an HTML report with test results will be stored, test report ID, date and time.
- Model parameters: model name, location, type, description, size, input and output sizes, number of subsequent sensor readings used at the input, the minimum and maximum temperature values used to normalize/de-normalize sensor values.
- Results processing parameters: apply anomaly detection threshold?(y/n), detection threshold value, apply limits on the output values? (y/n).
- Serial port parameters: serial port name, baud rate, timeout.
- Input data parameters: data sets directory name, a list of individual data sets to be used (all data sets used if empty).

Main file

The main program file first imports the configuration parameters from the configurations file previously discussed. Then it initialized the serial port interface with the specified baud rate and time out, now the program is ready to exchange data with the microcontroller.

First, three integer values are sent as strings: the model data size, model input- and output size. After sending each of these parameters, the code waits for a response from the board to confirm the receipt. The associated waiting time is set to 20 [ms]. This value is not critical for the general operation of the testing system, and thus, not included in the configurations file.

Then, the model data bytes are retrieved from the *.h* file and transmitted one-by-one. After that, the code again waits for the confirmation from the MCU, however, in this case, the waiting time is set to 1000 [ms] to ensure the successful initialization of the TF Lite interpreter, which takes more time than just reading values.

At the next step, the program iterates over the data sets and sends each data set line-by-line to the microcontroller. Depending on the nature of the model's input, the 'i' command tags for inference are put either after each line, or after each value in a line. The model outputs for each data set are temporarily stored in numpy arrays and plotted using matplotlib library afterwards. The resulting plots are arranged in HTML-based test reports, which will be further explained in Section 4.3.

To illustrate the program structure discussed in this section, a pseudo-code of main file algorithm is shown below:

Pseudocode for Python test system software

```
#main.py
from config_file import *
import matplotlib.pyplot as plt
import serial

initialize_serial_port()
send_model_data_size()
send_model_input_size()
send_model_output_size()
send_model_data()

for file in telemetry_test_files:
  for line in file:
    send(line)
    receive_output()
  plot_outputs()
make_html_report_with_plots()
```

4.3. TF Lite test reports

Besides the Python programs that prepare and send testing data to the microcontroller, there are also other routines that retrieve the model outputs from the microcontroller and post-process them. Python code plots the recorded model outputs against the expected values (e.g. true sensor anomaly labels) using Matplotlib library. Simultaneously, the plots of the corresponding input data (sensor readings) are generated and stored.

After that, the code places all the generated plots into an HTLM page, where all the tests can be viewed by the system user. This HTML report includes the plots of the model outputs produced for every data set, information about general model performance (in terms of time and accuracy), as well as test and model specifications. The first two pages of an example test report are shown in Figure 4.3. An example of a complete HTML report can be found in Appendix C.

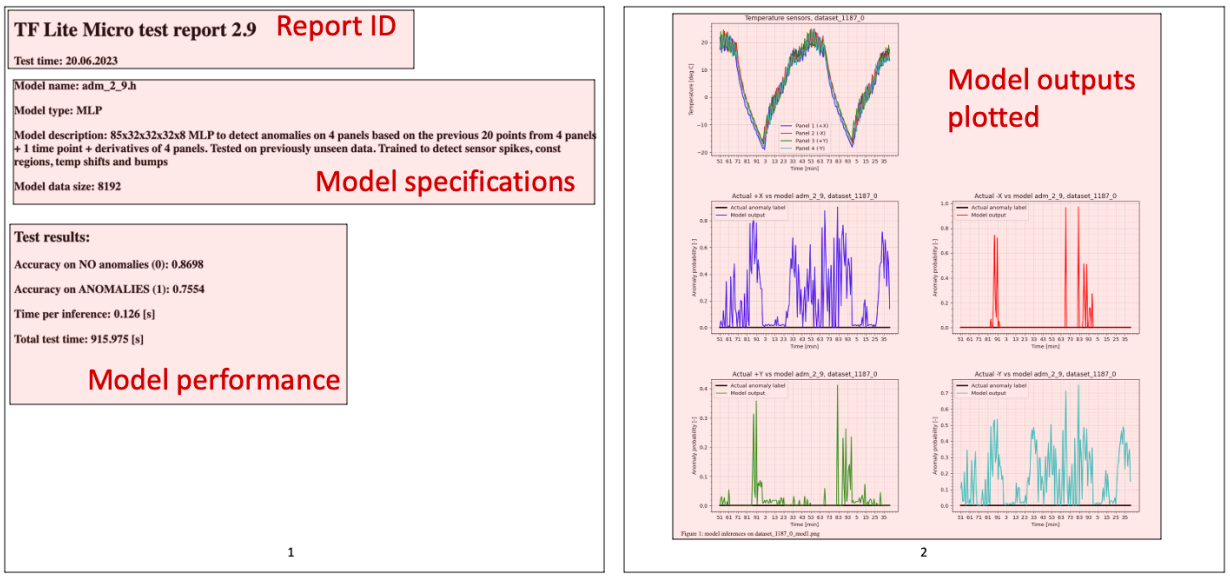


Figure 4.3: Example TF Lite test report

4.4. Testing system performance

Once the design of the microcontroller testing system is finalized, its performance can be evaluated. The most important metrics of performance in this case, are the memory usage of the software system, and the speed at which it is able to conduct the tests. These two aspects of the testing system's operation will be discussed in the following two subsections.

4.4.1. Memory usage

Table 4.4 shows the memory footprint of the embedded software, measured using the built-in memory allocation monitor of Code Composer Studio IDE:

Table 4.4: Embedded software memory footprint

Paraeter	Value [B]	Limit [B]	Fraction [%]
FLASH	236k	262k	89
SRAM	35.9k	65.5k	54

The values in the table provide only an initial estimate of the memory used by the system. This is because the memory allocation monitor can only account for the memory space that is known to be allocated to specific variables before the program is run. The variables for which the memory space is allocated during the run time, are not a part of this initial estimation as the actual memory footprint of these variables will only be known during the run time. It is, however, possible to estimate the memory footprint analytically.

It is known that during run time, no data is put into the flash memory of the microcontroller. This is because all received data, i.e. model size, input/output size, model- and input data, are stored in arrays that are declared as a local variables within a functions or code blocks. This means that all these data will be allocated in the stack memory of the board, which is part of the RAM. This leads to the conclusion that the flash memory footprint remains fixed at 236k bytes, and only the RAM usage is subject to change.

Although 236k bytes is a substantial part of the microcontroller flash memory, there is still space available (26k bytes) for anything that might need to be added to the testing system. This flash memory footprint would also not be a problem if a modified version of this system would be deployed onboard Delfi-PQ satellite, which has 2 MB built-in flash in the OBC, a 2 GB SD-card, and a spare microcontroller board which also provides 2 MB of unused built-in flash memory. Having said that, there are also possibilities for further optimization of the testing system. According to the official TF Lite Micro manual [56], the size of the TF Lite Micro library deployed on a microcontroller can be reduced to as little as a few tens of kilobytes by changing the list of operations supported by the operations resolver. This can be done either manually by deleting all the unused operations from the *AllOpsResolver*, or by switching to a resolver which has a much shorter, but still sufficient list of supported operations. Using one of such optimized resolvers, *MicroMutableOpResolver*, can result in a much more lightweight code for the embedded software. In case of *MicroMutableOpResolver*, however, the user has to manually specify the list of operations that need to be registered with the resolver. For example, if the neural network is a MLP or similar network consisting solely of fully-connected layers, the operations corresponding to fully-connected layers have to be registered in with the resolver. In this MLP example, the memory footprint is significantly reduced, as can be seen in Table 4.5:

Table 4.5: Embedded software memory footprint when using *MicroMutableOpResolver*

Paraeter	Value [B]	Limit [B]	Fraction [%]
FLASH	42.4k	262k	16
SRAM	30.2k	65.5k	46

While the memory footprint reduces dramatically, there is actually no noticeable effect on the inference execution speed, as the operations used to perform inference are still all the same. The only difference is that there are no longer other operations, which might be needed for other NN architectures.

It is obvious that the optimized operations resolver with only the specified operations would be the preferred choice when it comes to the actual flight software for Delfi-PQ or a similar satellite. However, this would be different for the implementation of the testing system, as it has to support as many architectures and operations as possible, as long as there is enough space remaining to accommodate for the model data and tensors. This makes the *AllOpsResolver* the preferred choice for now, even though it consumes much more flash memory.

For the microcontroller testing system using the *AllOpsResolver*, the RAM space pre-allocated to the run time variables is 35.9k bytes. This leaves 29.6k bytes for the memory allocated during run time, to which the main contributors are, of course, the model data size and the model input tensor (circular buffer). Luckily, the model data array is only used in the setup part to initialize the TF Lite interpreter, while the input and output tensors are initialized in the main loop, which is a different scope. In C/C++, when the function or block of code containing a certain variable ends, the memory occupied by that variable is automatically reclaimed. Since the model data array is declared as a local variable in the setup, it will be automatically deleted from RAM once the setup scope ends. The memory can then be reused by other parts of the program. This means that all of the available RAM (29.6k B) can be safely used to accommodate the model data, and the three integer model parameters, for which the read buffers are 100 bytes each, as was mentioned earlier in this chapter. This leaves **29.6k** bytes for the model data. This value serves as the baseline for the maximum allowable size of the temperature monitoring model designed within this project.

4.4.2. Testing speed

The second important metric that defines the performance of the testing system is the speed at which it can send, receive and process the temperature data. This also includes the time interval taken to perform a single inference of a neural network, time taken to transmit the model data and initialize the model, and other time-related parameters. It is expected that such parameters do not depend only on the system itself, but also on the complexity of the model. Therefore, the system's performance in terms of speed will be assessed for two cases with different model complexity. In both cases, the speed measurements are performed using the *time* Python module [66] that allows to track time using the CPU clock of the development computer.

Case 1: Single-sequence 30 inputs - 1 output MLP

The model in this first case is a simple multi-layer perceptron neural network, similar to the one in the TI example, but with a larger input tensor. This particular network is trained to predict the next temperature measurement of a temperature sensor based on the previous 30 temperature points. This implies that the model works with sequential data from a single sensor, and therefore, the testing procedure is applied as in Case 1 in 'Program main loop' part of Subsection 4.2.1.

Table 4.6 contains some test- an model-specific parameters of this network:

Table 4.6: Case 1 test-specific parameters

Parameter	Value	Unit
Model name	ptm_1_1.h	[-]
Model type	MLP	[-]
Layer structure	30x16x16x16x1	neurons
Model size	3952	B
Number of data sets used in the test sequence, N_{sets}	32	[-]
Average number of time points per data set	157	[-]
Number of plots to be produced per data set	2	[-]

This model is now tested on the microcontroller to measure its performance in terms of speed. The testing

system produces a test report with plots of model's outputs for each data set, an example of which can be found in Figure 4.4:

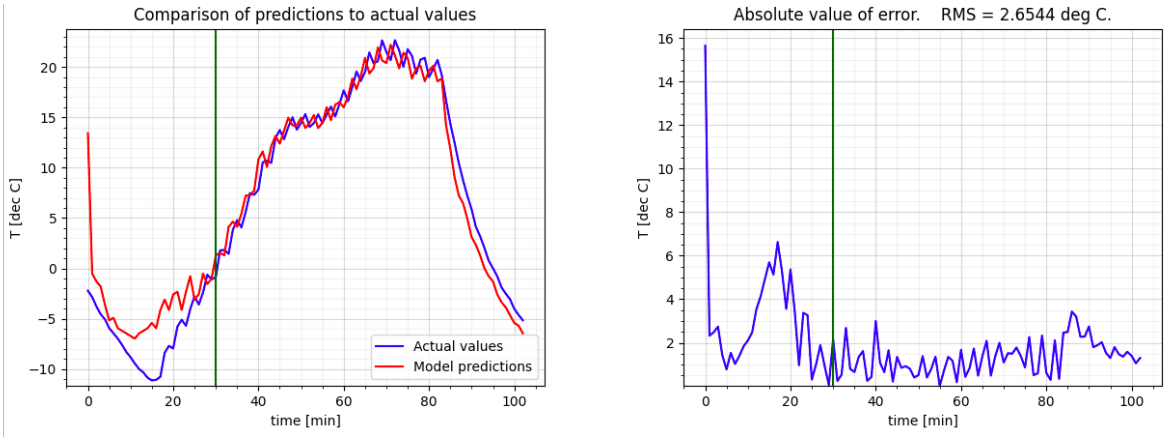


Figure 4.4: Performance of a simple predictive network

From the plot on the right, which demonstrates the absolute value of the model's prediction error, it is clearly seen that the accuracy of the NN prediction gradually improves until the first 30 points are received. This illustrates the fact that the circular buffer needs time to fill up with the actual temperature values rather than the initially assumed ones. For an in-space application, this issue would also play a role in case the circular buffer would be filled with streaming sensor data stored in RAM. In case of a reboot of the satellite, the previously allocated data would be lost and the buffer would have to be refilled. The issue could be avoided by extracting the continuous telemetry data from the flash memory of the OBC every time a reboot happens.

Once the testing sequence is complete, the time statistics are calculated and also put in the test report. Table 4.7 provides a summary of these timing results:

Table 4.7: Case 1 testing system time performance

Time parameter	Value [s]
Time to receive model size, input and output sizes, T_s	0.11
Time to transmit model data and initialize the model, T_m	2.61
Average time per data set, T_{set}	8.13
Average time per inference, T_{inf} (including input and output data transfer)	0.039

The measured values of the time parameters demonstrate that transmitting the model parameters and model data, including the interpreter initialization ($T_s + T_m$), does not take significant amount of time, while testing the model on each of the 32 data sets takes on average 8.13 seconds. This time per data set, T_{set} is comprised of multiple parts, as defined by Equation 4.1:

$$T_{set} = T_{inf} \cdot N_{inf} + T_{plot} \tag{4.1}$$

Where T_{set} is the time taken to run a test on a data set, T_{inf} is the time per inference, N_{inf} is the number of inferences (same as number of time points) for the data set, and T_{plot} is the time it takes to produce and save the Matplotlib plots of the test results. In this case, the average time taken by the actual inferences

is $T_{inf} \cdot N_{inf} = 0.039 \cdot 157 = 6.12[s]$, which means that the average time to produce a set of two plots of the results is $T_{plot} = 2.01[s]$. It can thus be concluded, that a substantial part of the total testing time is attributed to the creation and storage of the results plots. When testing the model on 32 data sets, this considerably affects the total testing time, given by Equation 4.2:

$$T_{tot} = T_s + T_m + T_{set} \cdot N_{sets} = 262.88[s] \tag{4.2}$$

Case 2: Multiple sequences 85 inputs - 8 outputs MLP
In this other case, the model is one of the preliminary versions of the actual temperature anomaly detecting model which is the main focus of this Thesis project. More details about its design will follow in Chapter 5. This model is also an MLP, however, unlike the first model, this one takes as input multiple sequences of temperature readings, as well as additional input parameters, such as the slope of each sensor and the current clock reading. This requires the testing procedure as in Case 2 described in 'Program main loop' part of Subsection 4.2.1.

Some test- an model-specific parameters of this model, are shown in Table 4.8:

Table 4.8: Case 2 test-specific parameters

Parameter	Value	Unit
Model name	adm_2_9.h	[-]
Model type	MLP	[-]
Layer structure	85x32x32x32x8	neurons
Model size	8192	B
Number of data sets used in the test sequence, N_{sets}	32	[-]
Average number of time points per data set	157	[-]
Number of plots to be produced per data set	5	[-]

It can be seen that this model is roughly two times the size of the model in Case 1. It is then expected that some of the timing parameter values will be about twice as high in this case. At the same time, the number of data sets and time points (inferences to be made) per data set is the same as for the first case, to allow for a fair comparison.

After performing the microcontroller test sequence, the time-related parameters are measured with the results presented in Table 4.9:

Table 4.9: Case 2 testing system time performance

Time parameter	Value [s]
Time to receive model size, input and output sizes, T_s	0.11
Time to transmit model data and initialize the model, T_m	4.48
Average time per data set, T_{set}	28.48
Average time per inference, T_{inf} (including input and output data transfer)	0.126

This time, for each data set, the average time taken by inferences only is $T_{inf} \cdot N_{inf} = 19.78[s]$, and the time to make the 5 plots is $T_{plot} = 28.48 - 19.78 = 8.7[s]$. The total testing time (for 32 data sets) is $T_{tot} = 915.95[s]$.

Case comparison
A comparison between the measured time parameters in the two cases is shown in Table 4.10:

Table 4.10: Case comparison on time performance

Parameter	Notation	Case 1	Case 2	Difference
Model size	-	3952	8192	+107%
Time to receive model size, input and output sizes	T_s	0.11	0.11	+0%
Time to transmit model data and initialize the model	T_m	2.61	4.48	+72%
Average time per inference (including input and output data transfer)	T_{inf}	0.039	0.126	+223%
Average time per data set	T_{set}	8.13	28.48	+250%
Total inference time per data set	$T_{inf} \cdot N_{inf}$	6.12	19.78	+223%
Plots to make per data set	N_{plot}	2	5	+150%
Time to plot results per data set	T_{plot}	2.01	8.70	+333%
Total testing time	T_{tot}	262.88	915.95	+248%

As it can be seen, in the second case the model size is twice as big as the model size in the first case. The model initialization also takes roughly twice the time. When it comes to the actual testing part, however, it now takes more than thee times the time. This is not only due to the increased model size, but also fact that to perform each inference, much more data has to be transferred, namely, 85 times the data which is transferred per single inference in the first case. Additionally, the number of result plots to be produced also affects the total testing time.

From the analysis of the results, it is concluded that the speed performance of the testing system gets considerably worse for non-sequential or multi-sequential inputs, when compared to single sensor series inputs. In addition to that, increased model complexity and the number of output plots to be produced also increase the total testing time.

Despite the fact, that the performance reduces for more complex models, the designed system can conduct the testing sequences within a time period that is reasonable for such applications. Having a possibility to deploy any NN of an appropriate size on the microcontroller and test it on tens of orbital periods of telemetry within a few minutes, significantly aids the design process of such on-device deployed neural networks. This system enables a testing procedure that is incomparably much more efficient than manually placing the model- and input data into the embedded code and re-uploading the firmware every time a test has to be conducted.

5

Preliminary Design of The Neural Network

This part of design is focused primarily on defining the necessary functionality of the neural network, its inputs, outputs, and the general architecture. While the focus of Chapter 6 will be on the detailed design, including the selection of the exact values for hyperparameters, this chapter rather focuses on more high-level aspects of the design. In Section 5.1, the implementation of temperature anomalies in the training dataset will be discussed. Then, the training and testing procedures used in the network design process, will be explained in Section 5.2. Next, the design approach used to define the NN-based system architecture is presented in Section 5.3. Finally, the deployment and performance testing of the preliminary version of the neural network of the chosen system architecture is discussed in Section 5.4.

5.1. Introducing anomalies into the training data set

As it has been determined earlier in Subsection 3.2.1, there are four main types of anomalies: outliers and flat regions, permanent bias, temporary bias (including temperature drift), and clock anomaly. Initially, the FUNcube telemetry pre-processed with Python is completely free of anomalies. All anomalies that will be present in the training dataset are introduced artificially in a controlled manner. This enables full control of the types, magnitude and number of anomalies to be included in the dataset.

Despite the importance of the clock anomalies, they are introduced into the dataset, and there are no functions that can synthesize this anomaly. This is because the main neural network of the anomaly detecting system will not be trained to detect time anomalies, the reasoning for this will be provided in Subsection 5.3.2, along with a solution for dealing with clock anomalies. Individual low-level Python functions have been written to generate anomalies of each of the other three types (outliers/flat regions and two types of biases). Each of these functions receives a list of sequential temperature values that form a single telemetry part. Additional arguments may include the index at which an anomaly starts, the time period for which it lasts and the magnitude of this anomaly. The outputs of these functions are also lists of sequential temperature values, but the anomalous values that have replaced the initial nominal values are accompanied with the corresponding anomaly labels, e.g. *'a'* string label concatenated with the temperature value.

The three anomaly generating functions work as follows:

- *make_const_sensor_anomaly()* function introduces outliers and flat regions starting at a list index n, lasting for k data points and with a magnitude of m (in $^{\circ}C$).
- *shift_temp()* function introduces the permanent bias anomalies with an offset of $m\ ^{\circ}C$ from the original temperature profile. The function simply adds the offset value to each list element.
- *bump_temp()* function introduces the bell-shaped temporary bias anomalies using a formula similar to Gaussian distribution. The function arguments are: the starting index of the anomaly n, the duration of the anomaly k, and the peak magnitude of the bell curve m.

The actual procedure of anomalous training data generation begins with a high-level block of code, where the exact types, numbers, combinations and magnitudes of the anomalies to be included in the dataset are specified. In this block of code, the low-level anomaly-generating functions are called and the outputs of these functions are temporarily stored in new text files that are meant to contain the anomalous telemetry parts.

Then, a different high-level function is used to read the anomalous telemetry text files and use the read data to assemble the training dataset entries, which could include not only the temperature readings, but also other values, such as the computed derivatives of the sensors, the clock readings, and the corresponding desired outputs of the neural network, which is normally a binary anomaly status (0 = no anomaly present, 1 = anomalous entry). At this step, the temperature values are also normalized by mapping each temperature reading into the range from 0.0 to 1.0. Normalization is necessary for more reliable quantization, and better convergence during training. Normalized inputs prevent extremely small or large gradients during backpropagation, resulting in more stable gradient updates and faster convergence.

Finally, all the generated dataset entries, that are normally stored in multiple temporary text files, are combined into a single dataset text file, which could either be directly used for training, or could be modified, for example by removing any duplicate entries or other similar operations.

This procedure is summarized in a form of a block diagram, which is shown in Figure 5.1:

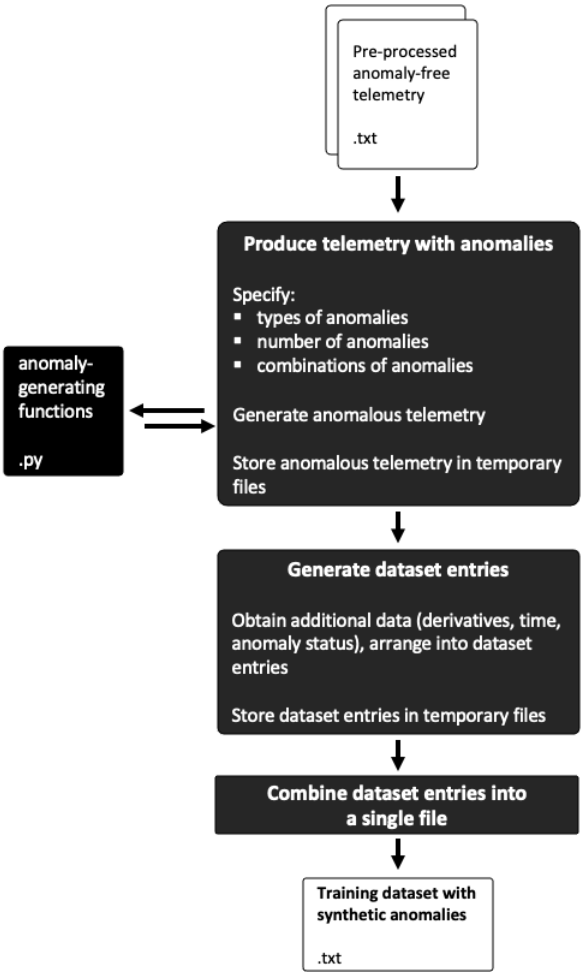


Figure 5.1: Procedure to introduce artificial anomalies into the training data

5.2. Training and testing procedures

Python programs are used to create, train and test neural networks using the TensorFlow library. The code for these programs can be found in Appendix B. In this section, these procedures will be explained in more detail.

5.2.1. Training networks using TensorFlow

Once the the desired network functionality is established and an appropriate training dataset is designed, the process of training the NN with Tensorflow generally consists of the following steps:

1. Determination of the type and the number of neuron layers as well as the number of neurons in each layer. While the final architecture is determined through an iterative procedure, an initial value can be chosen based on the complexity of the problem, dataset size, and the available computational resources. At this step, also the activation functions are chosen, for which an experimentation approach is required.
2. The model is initialized by using TensorFlow's *tf.keras.Sequential* class, which allows to sequentially stack the layers previously defined. Layers can also be appended by using the *add()* method.
3. The model is compiled using the *compile()* method with a specified loss function and the optimizer.
4. Next, the compiled model is trained by using the *fit()* method with the prepared training and validation data, a specified number of training epochs (iterations) and training batch size. The effectiveness of the training process is tracked by monitoring the loss and accuracy metrics before and after the training. To measure the model performance in terms of accuracy, the model inference needs to be performed using the *predict()* method.
5. If the results are not satisfactory, either the training dataset or the training parameters (hyperparameters) can be changed and the model is re-trained. This implies an iterative design and optimization process.

The following set of parameters, also known as the hyperparameters of a neural network, need to be specified to run the NN training code:

- **configuration of neuron layers**: types of layers and number of neurons in each layer.
- **activation functions** (per layer): functions applied to outputs of each layer neurons.
- **model input and output data**: processed data arranged into one dataset, which is split into training and validation parts.
- **number of training epochs**: parameter that defines how many times the model will be exposed to the same training data.
- **training batch size**: parameter that defines how the input training data is structured during each of the training epochs.
- **optimizer**: function which iteratively changes the weights and biases of the model during the training.
- **learning rate**: parameter that sets the step size at which the the weights and biases of the model are updated during the training.
- **loss function**: function that measures the deviation of the model predictions from the expected outputs (actual data labels).

5.2.2. Testing the network performance

When a neural network is trained with TensorFlow, its performance can be tested and compared to similar networks to come up with an optimal network design. In the preliminary design phase discussed later in this chapter, neural networks are not yet compressed and deployed on the MSP432 board. Instead, each trained network is stored as a $.h5$ model file, and the performance is evaluated and compared for these non-compressed models. This procedure will now be discussed in more detail.

First, a trained $.h5$ NN model is loaded using the *tf.keras.models.load_model()* command. Then the inference is performed using the TensorFlow's *predict()* method. This also requires using a part of the dataset that the model has not seen during training. This part of the dataset (more specifically, the data derived from telemetry part 1187 shown in Figure 4.1) has intentionally been removed from the training dataset prior to the training to make sure there will be some remaining data which the model has not yet seen. The *predict()* method returns an array containing the model outputs. These outputs can then be extracted and plotted using *Matplotlib*.

Once the model output plots have been produced, also the accuracy metrics can be calculated. In the preliminary design phase discussed in this chapter, no detection threshold is applied on the model outputs yet. This means that the most suitable metrics in this case are the accuracy on anomalies A_a and accuracy on no anomalies A_{na} which were introduced in Chapter 3. In addition to that, further insight into the model's functioning can be obtained through an investigation of the input layer weights. Evaluating an importance of each of the input neurons can show which inputs are more important for the model's operation than

others. This importance score of an individual input layer neuron can be calculated by taking the absolute value of the sum of the weights of all connections that the neuron has with the first hidden layer neurons. The result can be plotted using a heat-map and stored along with the other plots.

Finally, a test report similar to the one described in Section 4.3 can be produced by placing all the produced *Matplotlib* plots in an HTML page. The calculated values of the accuracy metrics are printed in the beginning of each report. An example of such model test report is shown in Figure 5.2:

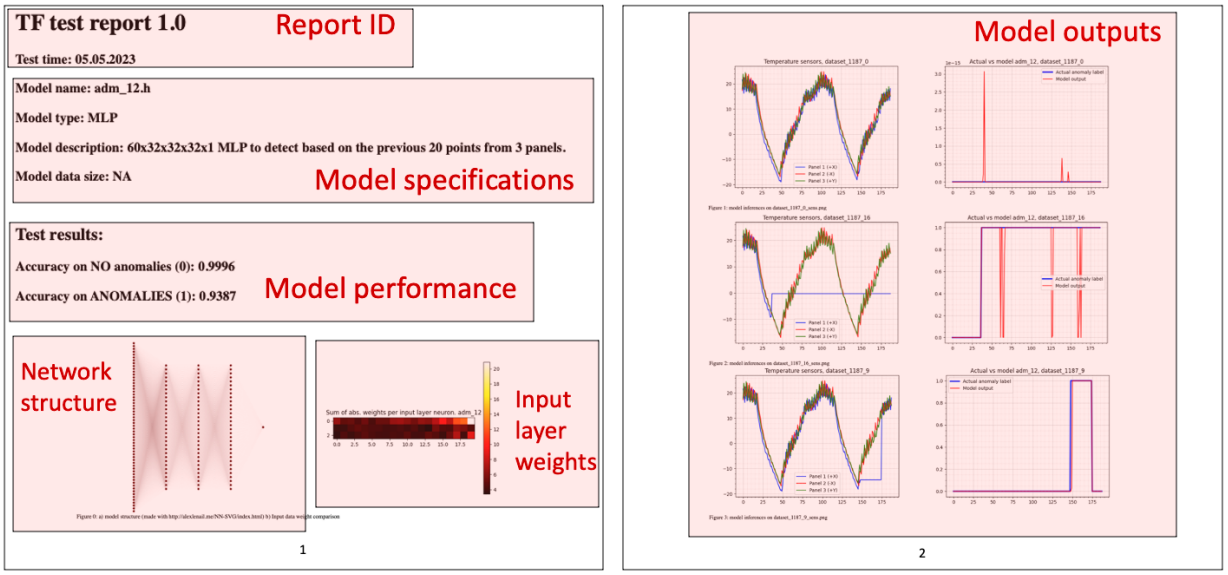


Figure 5.2: Example of a TensorFlow network testing report

5.3. Defining the system architecture

The process of defining the architecture of the neural network system involves identifying the available sources of information that the network can access, performing trade-offs to come up with the conceptual view of the neural network, and applying an iterative approach to refine the conceptual design and arrive at a more specific architecture. The exploration of the available information sources will be presented in Subsection 5.3.1. During the conceptual design phase, a number of important design decisions are made, which lay the foundation of the system. These will be discussed in Subsection 5.3.2. After that, some of the parameters that define a more specific system design are chosen through an iterative process discussed in Subsection 5.3.3. The result of the iterative design will be explained in Subsection 5.3.4.

5.3.1. Knowledge sources

In order to be able to detect anomalies, the network needs to receive information from relevant sources, which have connection to the thermal behaviour of the satellite. The most obvious and relevant sources of information are:

- **temperature sensor readings**: provide the knowledge of the current and previous temperature measurements.
- **temperature sensor derivatives**: provide knowledge of the shape of the temperature graph.
- **clock**: allows to link specific temperature readings to specific moments in time that define specific orbital positions.

Other sources of information that could potentially be used to enable even more educated network predictions include:

- **physics model of the satellite**: could provide knowledge necessary to establish more intricate relations between temperature readings in different parts of the satellite. However, its implementation would be too complex for the selected application (demonstrator) and the potential effects are unpredictable.

- **solar flux measurements**: could provide more reliable knowledge of the temperature, which would free of sensor noise. However, such measurements are not available for the target application.
- **orbital position data**: could potentially provide more precise estimates of the relations between temperature and satellite position than the clock readings alone. However, to obtain this kind of data, the use has to be made of the so-called Two-line elements (TLE) that include satellite tracking data and orbital propagation algorithms, as the telemetry is not accompanied by the position data. It would therefore be too complex for the selected application (demonstrator).

To conclude, for the target application of an anomaly detecting network demonstrator, the only suitable knowledge sources that can be used are the temperature sensor measurements, the computed derivatives of these measurements, and the clock data in form of time stamps that can be assigned to the elements of any temperature telemetry sequence. The preliminary design phase of the anomaly-detecting neural network will therefore revolve around these three knowledge sources and how they can be integrated into the ML model.

5.3.2. Anomaly detecting network conceptual design

Despite the fact that the neural network system has only one important function, that is to detect anomalies, the implementation of this functionality can be achieved in many ways. For instance, instead of a single neural network, the system might consist of multiple neural networks connected together. The type of the neural network(s), the inputs that it uses, the types of outputs it produces, and some other aspects could also vary drastically in different implementations.

In order to come up with a general view of the neural network system, a conceptual system design is performed, which is based on a number of engineering trade-offs. These trade-offs are performed to determine the type of the outputs the system produces, the number of neural networks used, the architecture of the neural network(s) to be used, and the approach that the system uses to determine the current time when the clock value is not correct or not available. In these trade-offs, different implementations are compared based on criteria that are most relevant to this Thesis project, given the constrained time frame of the project and highly limited memory resources of the microcontroller. These criteria impose certain limits on the allowable complexity and memory footprint of a particular implementation, which affects the maximum allowable size of the neural network, difficulty of implementation, etc. Each of the trade-offs will now be described in more detail.

System output type

The output of the NN-based system that is meant to indicate whether there is an anomaly in the current temperature readings or not, can be implemented in three different ways. The first way would be to make the network produce a binary anomaly label as output, for example, 1 indicates anomaly and 0 indicates no anomaly, in other words, a standard binary classifier network. Another way would be not to make the network choose between two options, but instead make it produce an anomaly probability output. This anomaly probability would indicate how certain the network is that an anomaly is currently being present. Finally, the output can be not a binary value or a probability, but a value that corresponds to a magnitude of an anomaly that is estimated by the network. This could be a value measured in $^{\circ}$C, which would show how by many degrees the current temperature value deviates from the expected temperature.

The main factors that influence the choice between these three options include the difficulty of implementation and post-processing, and the extent to which a particular output type could provide insights into the decision-making process that the network applies. The latter criterion is of a particular importance as neural networks are generally characterized by their black-box nature, which makes it challenging to gain insight into how the network actually comes up with the outputs that it produces.

Table 5.1 shows a trade-off between the three output types made based on the described selection criteria. The trade-off is performed as follows:

1. Each selection criterion is assigned a weight in %, which can be seen under each of the selection criteria in the top row of the trade-off table. These weights are chosen after a thorough consideration of the importance of each selection criterion in the context of the Thesis projects, which is driven by the availability of resources, time, system requirements, etc.
2. An evaluation of each design option (network output type in this case) is performed against each of the selection criteria. This results in a number of worded comments in the middle part of the trade-off

table. Each of these comments reflects on how well a particular option performs in a particular selection aspect.

3. A suitability score is given for each option evaluated against each criterion based on the comment previously given. The suitability score is a number from 0 to 3, where 0 is excellent suitability that exceeds the requirements and 0 corresponds to an unsuitable option. These scores are represented as colour codes, which are explained under every trade-off table.
4. For each option, the scores that it got when evaluated against each of the criteria are multiplied with corresponding criteria weights. After that, the results of the multiplication are summed to obtain a single resulting suitability score for each option.
5. The option with the highest suitability score wins the trade-off and thus will be incorporated in the further system design.

The method just described applies to all trade-off tables that will be presented later in this section.

Table 5.1: Trade-off 1: neural network output types

TO1: NN output types	Provides insight into certainty of an output [40% weight]	Post-processing required [30% weight]	Difficulty of implementation in the training data set [30% weight]	Resulting score
Binary anomaly label	No	Can be directly used for FDIR	Relatively low	2,2
Anomaly probability	Yes	A threshold needs to be applied	Relatively low	2,7
Anomaly magnitude	Yes, but might be more difficult to interpret	A threshold needs to be applied depending on the anomaly type	Relatively high	1,7

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

The anomaly probability network output type wins this trade-off. The main advantages of the network that produces an anomaly probability as a float number between 0.0 and 1.0 is the fact that this output directly indicates the certainty of the network's output at each inference. This knowledge can be used to perform a more educated comparison between different network versions, which is highly beneficial for the design process. With the certainty of the network as an output, one can develop insight into the factors that influence the decision-making process that the network uses, to arrive at conclusions that can be used to develop a better performing network at the next design iteration. This type of output will be the most important one during the preliminary design of the network and no post-processing is applied to it yet. For the final design of the network, however, a detection threshold is applied on the probability output to evaluate its performance in terms of multiple insightful accuracy metrics.

Another aspect of the network's output is whether or not it contains information which would allow to distinguish between anomaly types, that is to tell outliers and flat regions from permanent or temporary bias. Although it is not specified by the requirements that the system should differentiate between the anomaly types, this could add value to the fault detection system by providing knowledge about the cause of a specific anomaly. While this functionality could be beneficial in terms of added value or the network's ability to learn more intricate patterns within the input data, its implementation could be not feasible in terms of network size or the effort and time it would take. To clarify this and determine which option performs better in the combination of these criteria, a trade-off is performed as shown in Table 5.2:

Table 5.2: Trade-off 2: network's ability to distinguish between anomaly types

TO2: NN distinguishes between anomaly types?	Small size [30% weight]	Feasible within project duration [30% weight]	Network learns more complex features [20% weight]	Is a necessary part of the demonstrator [20% weight]	Resulting score
NN can distinguish between anomaly types	No	Unknown	Yes	More complexity than needed	2,1
NN CANNOT distinguish between anomaly types	Yes	Yes	No	Yes, required for the demonstrator	2,6

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

The result of the trade-off indicates that the network with more limited output functionality would be preferable. Even though distinguishing between the anomaly types would be a useful feature for FDIR and would likely require the network to learn more complex patterns in the temperature dynamics, it might result in a larger sized solution, which will likely not fit into the memory of the microcontroller, nor it would be feasible to implement within the limited time frame of the Thesis project.

Single network vs multiple networks

As it has been briefly mentioned previously, the NN-based system might potentially consist not of one neural network, but of multiple network, each with its special functionality. For instance, there could be one network per each of the four panels where the temperature is measured. This configuration could result in higher anomaly detection accuracy for any individual panel, but also for the overall system as these networks could be exchanging their outputs and benefit from establishing the relations between different sensor readings. However, also such factors as system complexity, size and the effort required for implementation play an important role when it comes to making design choices. A trade-off is thus performed to determine whether a single neural network would be sufficient or if a system of multiple NNs would be more beneficial while still being feasible. The results of the trade-off are shown in Table 5.3:

Table 5.3: Trade-off 3: number of neural networks deployed

TO3: Number of NNs	Can be deployed on a single MSP board [30% weight]	Feasible within project duration [30% weight]	Is a necessary part of the demonstrator [20% weight]	Detection accuracy for a single panel [20% weight]	Resulting score
There is one NN which receives inputs from all sensors and produces outputs for all sensors	Yes	Yes	Yes, required for the demonstrator	Lower	2
There are multiple NNs, each receives inputs from all sensors and produces output for one sensor	Unknown	Unknown	More complexity than needed	Higher	1,8

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

The trade-off results in choosing a single neural network as the main part of the anomaly detection system. The main factors that support this design choice are again the complexity and size of the system, as well as the feasibility of its implementation given the time constraints. Deploying multiple neural networks on the same microcontroller would not only affect the RAM consumption in a way that is difficult to predict (as it would require creating multiple TF Lite interpreter objects in the same scope), but would also require adjustments to the microcontroller testing system software, which would lead to additional testing procedures of unknown duration. Therefore, a single neural network is chosen, which will receive outputs from all four sensors and produce outputs for all of the four sensors. As was defined by the system requirements, the network should be able to identify which of the four panels experience anomalous temperature behaviour and which do not. This implies that the network output will consist of at least four neurons, one per each of the four temperature sensors.

Another design solution that would result in using multiple neural networks was also considered. In this system, there is one neural network that produces the outputs for all panels, however, there is also a separate network that produces a temperature prediction based not on the temperature inputs, but solely on the clock readings. This time-based prediction would then be fed into the main network. An alternative would be to stick with the one-network-for-all-functions approach and design a network which would detect anomalies based on both temperature and clock readings fed directly into this network. For these two options, the selection criteria are again the added value, system complexity (memory consumption) and the feasibility in terms of time required for development. The resulting trade-off is shown in Table 5.4:

Table 5.4: Trade-off 4: separate clock value processing vs incorporated in a single network

TO4: Separate NN for a time-based prediction?	**Can be deployed on a single MSP board [40% weight]**	**Feasible within project duration [30% weight]**	**Is a necessary part of the demonstrator [30% weight]**	**Resulting score**
There is only one NN which takes both temperature series and clock reading as inputs	Yes	Yes	Yes, required for the demonstrator	2,2
There are two NN's. One to make a time-based prediction, the other to estimate anomaly based on that prediction and temperature sensors	Likely yes (effect on RAM performance is hard to predict)	Unknown	More complexity than needed	1,7

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

The trade-off result indicates that the single network approach is again more favorable, mostly thanks to its simplicity and smaller risk of not being able to deploy it on the MSP microcontroller.

Neural network architecture

When it comes to designing neural network -based systems, one of the most important design choices to be made is the selection of the NN architecture, i.e. the way the neurons, neuron layers and the connections between them are arranged. As it was previously mentioned in Section 2.2, the most suitable neural network architectures for anomaly detecting tasks are MLP, RNN (LSTM), Autoencoder and CNN.

In an ideal case, a number of neural networks of multiple types would need to be developed and compared in terms of size and performance, which is however, not possible given the limited scope and the time frame of the Thesis project. Instead, the selection of the network type is again achieved by performing a trade-off based on selection criteria such as the potential network size, interpretability, difficulty of implementation and the complexity of the patterns that the network would be able to learn. The results of this trade-off are shown in Table 5.5:

Table 5.5: Trade-off 5: Neural network architectures

TO5: Network architecture	Network size [25% weight]	Difficulty of implementation [25% weight]	Interpretability [25% weight]	Network learns more complex features [25% weight]	Resulting score
MLP	Relatively small size	Straightforward	Network size can be linked directly to number of layers and neurons, near linear scalability	Learns local patterns only	2,5
RNN (LSTM)	Potentially smaller than MLP due to increased detection accuracy	Increased difficulty	Due to recurrent cells, the overall size is difficult to estimate analytically, unknown scalability	Learns long-term patterns, but performs worse on short-term patterns	2,25
CNN	Convolutional layers and filters result in greater size	Increased difficulty	Less straightforward than MLP, unknown scalability	Can learn more complex features than MLP	1,75
Autoencoder	Potentially smaller than MLP due to increased detection accuracy	Increased difficulty	Less straightforward than MLP, unknown scalability	Can learn rich representations, better than MLP	2,25

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

The most suitable network architecture for the anomaly detection system demonstrator is MLP. As this architecture is based solely on fully-connected layers, its scalability can be easily understood and applied to the network design, which would allow to efficiently design the network for the available memory size. Another advantage of this network architecture is the simplicity of implementation, which is of a high priority in this project given the limited time resources. For an anomaly-detecting system that would be integrated into the flight software of a real flying satellite, it would be necessary to explore also the other types of the neural networks to ensure the best possible performance, however, for this demonstrator, an MLP network is deemed sufficient to demonstrate the potential capabilities of the system.

Clock anomaly problem

Another important design choice to be made is related to the system's ability to deal with unknown or wrong time, which is characteristic of the clock anomaly introduced in Chapter 3. As the clock readings are actually used to increase the accuracy of the temperature anomaly detection, the clock anomalies should not only be detected, but also fixed by estimating the current time based on other sensor knowledge, such as the shape of the temperature data series.

One way to address the clock anomaly is by designing the neural network to produce not only the anomaly probability outputs for the temperature sensors, but also an estimation of the current clock value. This implies that the network would have to be trained to handle both the anomaly detection and time estimation tasks at once. This design option would, however, require a much larger network, as the length of the input temperature series needs to be longer to properly reconstruct the time series. This option is also associated with an increased complexity of the training dataset development. For these reasons, other design options also need to be considered.

Alternatively, the function of time reconstruction could be delegated to a separate algorithm, which is not a part of the main neural network. This algorithm would use the temperature sensor data as input to produce an estimate of the current clock value. This value would then be passed into the neural network along with the temperature sensor readings to perform the anomaly detection.

One other option would be to not use the clock data at all and base the anomaly detection solely on the temperature sensor data. This approach, however, significantly reduces the anomaly detection performance and the network's ability to distinguish between anomalies on different panels. Still, this option is included in the trade-off, the results of which are shown in Table 5.6:

Table 5.6: Trade-off 6: system's ability to deal with unknown time (wrong clock)

TO6: Dealing with unknown time	Effect on performance [50% weight]	Feasible within project duration [50% weight]	Resulting score
NN estimates current time and uses it at the next inference for temperature anomaly computation	More information sources, better performance	Network complexity increases significantly, as longer time series or a RNN would be required. In both cases, this complicates the design of the dataset and drastically increases the training time.	2
There is a separate algorithm or a network to estimate current time, NN uses the output of that meachanism	More information sources, better performance	Yes	2,5
Time is not used as input of NN	Less information sources are used, thus reduced performance	Yes	1,5

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

Due to sufficient performance and moderate complexity of implementation, the design option of a separate algorithm for time estimation outweighs the other two options and will thus be implemented in the software system. As it was already said, this algorithm, from now onward called the Time Estimation Algorithm (TEA), receives a long series of temperature data as input and produces a time estimation as the output. Figure 5.3 shows the principle of the system's operation with the Time Estimation Algorithm:

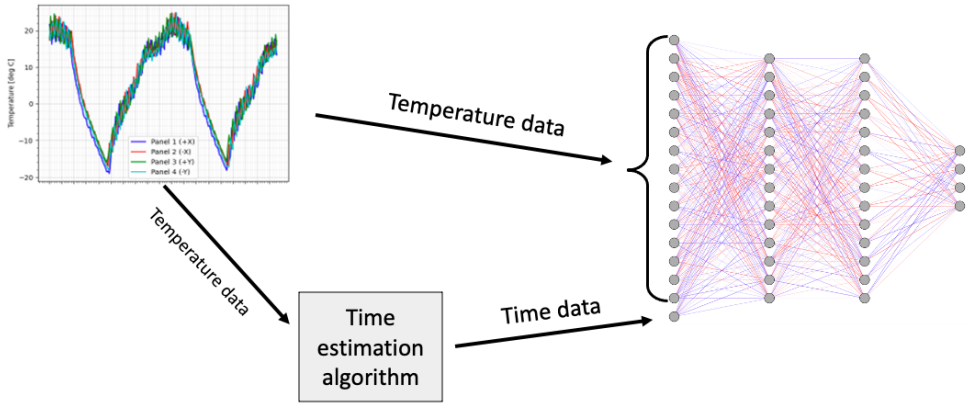


Figure 5.3: Data flow: telemetry, anomaly detecting network and a time estimation algorithm

The options for the TEA implementation include:

- Neural network based approach: a smaller neural network could be designed specifically for the time estimation task.
- Other ML algorithm: ML algorithms other than DL algorithm can be used.
- Non-ML algorithm: a custom algorithm can be designed using a traditional coding approach.

Non-ML algorithm is preferred in this case, as the solution to the time estimation problem is actually quite straightforward, and could look as follows:

1. The lowest temperature value in the last orbital period is found.
2. It is confirmed that the minimum repeats after one orbital period.
3. The reference point is set at the time of this lowest detected value, which would correspond to the moment the satellite leaves the Earth's shadow and starts heating up.
4. The current time is calculated by counting the time elapsed from the reference point.

At this preliminary design stage, TEA is not designed in detail and left as more or less a black-box algorithm concept. The implementation of its functionality will be explained in more detail in Section 6.4.

While TEA outputs a time estimate, it is possible that this estimate could be deemed erroneous or not accurate enough. In this case, the output of the TEA could be passed into the neural network, and the network could be designed to produce a correction for the TEA output. This design option could potentially improve the anomaly detection performance, however, it is associated with an increased network size and difficulty of implementation. Alternatively, the output of TEA could be assumed to always have sufficient accuracy. These two design options for processing the output of TEA are compared against each other in the trade-off as shown in Table 5.7:

Table 5.7: Trade-off 7: Options for processing the output of the time estimation algorithm

TO7: Processing the output of the time estimation algorithm	**Network size [35% weight]**	**Difficulty of implementation [35% weight]**	**Is a necessary part of the demonstrator [30% weight]**	**Resulting score**
NN checks the output of the TEA and produces a correction. The output of the NN can be used to correct the time estimation by TEA at the next inference.	Larger	Increased	More than required, but does not add significant value. Therefore, would be consume time that could be spent on the development of more useful functionalities.	1,7
The output of TEA is assumed to be correct.	Smaller	Nominal	Yes, required for the demonstrator	2,7

	Color code	Score
Excellent, exceeds requirements	green	3
Good, meets requirements	blue	2
Correctable deficiencies	yellow	1
Unacceptable	red	0

As in multiple trade-offs previously performed, the more conservative option receives a higher suitability score thanks to smaller network size and moderate difficulty of implementation.

Temperature measurements sample rate

The final high-level design decision to be made in order to define the concept of the neural network system, is the sample rate of the temperature measurements that the neural network will receive as input.

As was mentioned previously, Delfi-PQ performs the temperature measurements every 15 seconds, while FUNcube-1 samples temperature every 60 seconds. As there is no available FUNcube telemetry with a higher sample frequency, and only the FUNcube telemetry is complete enough to be used for the training dataset, the 60 second spacing between the temperature points is the lowest value that can be chosen for the anomaly detecting system. While it would be possible to apply interpolation to generate more data points, it is actually beneficial to use a more sparse temperature series, as it can provide better insight into the long-term thermal behaviour.

The network will thus work with the 60 second temperature step size. At the same time, one of the system requirements specifies that the anomaly detecting system should be able to perform at least one inference every 15 seconds. For the Delfi-PQ this basically means that the network will be inferred every 15 seconds, but at every inference, not all of the recent temperature measurements will be used, but only those that are 60 seconds apart from each other. This means that at a particular inference, the network would use the current temperature readings, those obtained 60 seconds ago, 120 seconds ago, and so on. This schema would ensure that the network inputs are always sampled with 1 minute period, even though the Delfi-PQ temperature readings are incoming as fast as every 15 seconds.

5.3.3. Iterative approach to the preliminary design

Once the high-level conceptual design solutions are defined, the neural network design can be made more detailed by applying an iterative design procedure. During this procedure, some of the high-level model parameters and functionalities are selected and iteratively adjusted to achieve the design stage, where the model already complies with the most important system requirements. The goal of this iterative process is to gradually implement all of the desired functions while studying the effect of every introduced change and maintaining high accuracy of the anomaly probability outputs.

It is important to note that not all of the hyperparameters are designed at this design stage. Most of the adjusted parameters are actually not hyperparameters but parameters related to the structure of the training dataset. The hyperparameters of the actual neural network, such as hidden layer structure, activation functions, etc., will be designed in Chapter 6. The list of the main parameters that are iteratively adjusted at this stage is as follows:

- number of sensors used for input
- length of the temperature series per panel
- presence of other types of inputs (derivatives, clock)
- number of sensors for which the anomaly probability output is produced
- presence other types of outputs (predictions, derivatives)
- types of anomalies detected

At every iteration, a new dataset is created and a new model is trained with this dataset. To support the design changes applied to every new iteration, the two accuracy metrics are used: A_a and A_{na}. Some of the model iterations are created solely to improve the accuracy performance, while some are trained driven by the need to implement the functionality outlined by the system requirements, regardless of whether the accuracy increases or not. A summary of the design iterations is shown in Table 5.8. Only the iterations with the most significant changes are included in the table.

Table 5.8: Summary of the iterative design process of the network

	Iteration 1	**Iteration 2**	**Iteration 3**	**Iteration 4**	**Iteration 5**	**Iteration 6**	**Iteration 7**	**Iteration 8**	**Iteration 9**	**Iteration 10**
Model name	adm_12.h	adm_13.h	adm_20.h	adm_21.h	adm_22.h	adm_25.h	adm_26.h	adm_27.h	adm_28.h	adm_29.h
Hidden layers	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32	32x32x32
Inputs										
Receives temperature readings from [how many] panels?	3	3	3	3	3	3	3	3	3	4
Temperature series length (per panel)	20	20	20	20	20	20	20	20	20	20
Derivative as input (in total)	-	-	-	-	1	3	3	3	3	4
Time as input (in total)	-	-	-	20	20	20	20	20	1	1
TOTAL INPUTS	60	60	60	80	81	83	83	83	64	85
Outputs										
Produces output for [how many] panels?	1	1	1	1	1	3	3	3	3	4
Anomaly probability as output (per panel)	1	1	1	1	1	1	1	1	1	1
Sensor derivative as output (per panel)	-	1	1	1	-	-	-	-	-	-
Temperature prediction as output (per panel)	-	1	1	1	1	1	1	1	1	1
TOTAL OUTPUTS	1	3	3	3	2	6	6	6	6	8
Functions										
Detects outliers and flat regions	X	X	X	X	X	X	X	X	X	X
Detects permanent bias	-	-	X	X	X	-	X	X	X	X
Detects temporary bias	-	-	X	X	X	-	-	X	X	X
Accuracy performance										
Accuracy on anomalies, A_a	0.9387	0.9307	0.9573	0.9547	0.9601	0.705	0.8946	0.9171	0.9259	0.8195
Accuracy on NO anomalies, A_na	0.9996	0.9974	0.9945	0.9961	0.9987	0.991	0.9766	0.9927	0.9861	0.9699

The first design iteration is an MLP network that receives only the temperature sensor data, which is comprises 20 consecutive temperature points from +X, -X and +Y panels. 20 is chosen as the initial number of the data points per panel in order to allow the network to ”see” more long-term dynamics in the temperature readings, which could potentially result in better anomaly detection performance. Preliminary experiments with simple time series predicting models showed that a time series length of 20-40 points is sufficient to closely predict FUNcube telemetry by 1 point into the future (error is consistently less than 5°C, which is well within the 10°C limit on deviations that can be regarded as non-anomalous). For this application, the value on the lower side is selected, as the network size is of high priority. The initial network output consists of a single neuron which corresponds to the anomaly probability at panel +X. For the hidden layer size, a common rule-of-thumb used to design simple MLP neural networks is to initially choose less layers which have a number of neurons which is in between the number of input neurons and output neurons. In this case, the average between input and output neurons would be roughly 32. To make the model able to learn more complex patterns right away, not one, but three hidden layers are included. As a result, there are three hidden layers, each consisting of 32 neurons. The hidden layer structure and the choice of other hyperparameters will be fixed for all iterations of the preliminary design. The summary of these hyperparameters will be shown in the next subsection.

It can be seen that the network of the first iteration does not yet match the requirement of the output produced for four panels, nor does it use all of the available knowledge sources. The reason for this is that the iterative is intentionally started with a highly simplified version of the model, which would not be difficult to analyse and understand. Over the iterations, the complexity of the training dataset and the functionality of the model are gradually increased.

Adding the temperature sensor derivative to the output at iteration 2 did not improve the performance, therefore the derivative was moved to the input at iteration 5, which did improve the performance. Adding the next point prediction output did not improve the performance either, but this output type was kept for debugging purposes.

Up until iteration 6, the model was trained to produce outputs only for one panel. Once other panel outputs were added, the functionality was intentionally limited to detecting fewer anomaly types to study the behaviour of the new model on simple cases first. After that, the functionality was gradually extended to detect all the required anomaly types.

Starting from iteration 4 up until iteration 9, the clock input was used as a series of 20 consequent clock values. It was then determined that the system requirements allow to assume equal spacing of the time points, which makes having 20 of these points redundant. At iteration 9, the time series input was replaced with a single clock value input, which has slightly improved the accuracy performance.

At the final iteration (10), the network was finally adapted to handle not 3 but 4 panel sensors. The anomaly detection accuracy has decreased compared to the previous iteration, which, however cannot be avoided as the requirement on the number of panels is stronger than the preference for higher accuracy.

5.3.4. Final iteration of the preliminary design

The result of the final iteration is an MLP network with 85 input neurons, 8 outputs, and 3 hidden layers where each layer has 32 neurons (the design of the hidden layers will be further explained in Chapter 6). The inputs include: 4 panel sensor derivatives, 1 clock reading (time at the current inference), and for each of the 4 panels, there are 20 latest temperature sensor readings in °C. The outputs include 4 anomaly statuses (probabilities) and 4 predictions of the next temperature point (1 for each sensor), which are used for debugging. The network architecture is shown schematically in Figure 5.4:

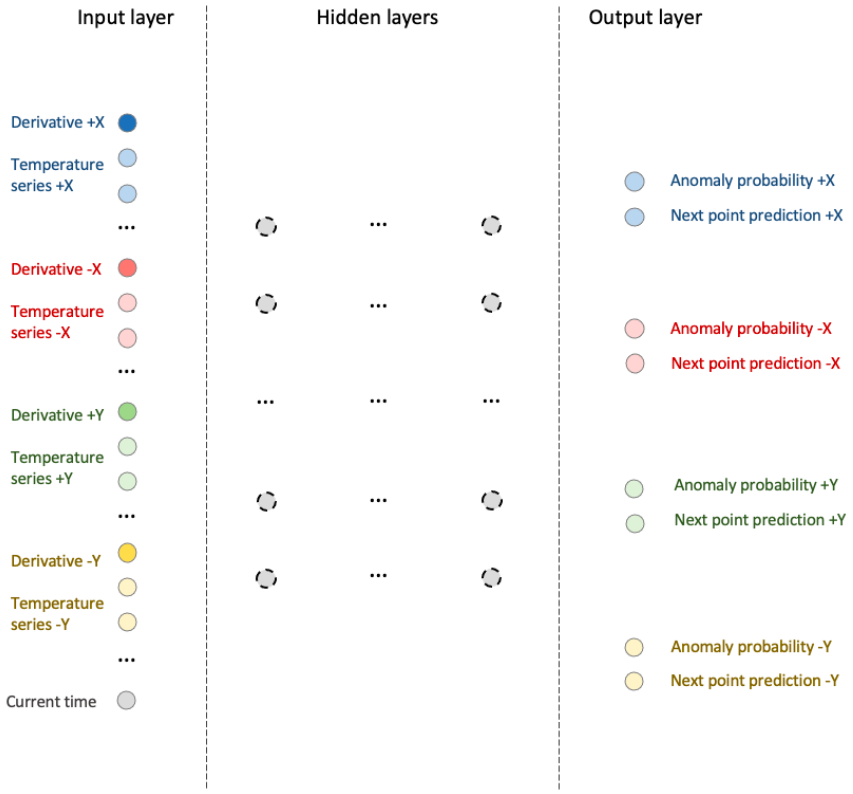


Figure 5.4: Anomaly detecting network architecture

Data set structure

The dataset used to train the final iteration of the preliminary model consists of 26499 lines. Each line contains a set of inputs and the corresponding outputs, which are organized as follows:

4 anomaly outputs| 4 prediction outputs| 4 derivative inputs| 1 time input | 4 x 20 temperature inputs

2 versions of this dataset have been tested, for one of which all duplicate lines were removed, which resulted in only 10718 lines. However, as the effect of removing duplicated on the performance was deemed inconclusive after a few tests (it varies for different data sets), the original dataset with 26499 lines is chosen as the preferred one for further network optimization. The line structure is demonstrated by a snippet of the actual dataset shown in Figure 5.5:

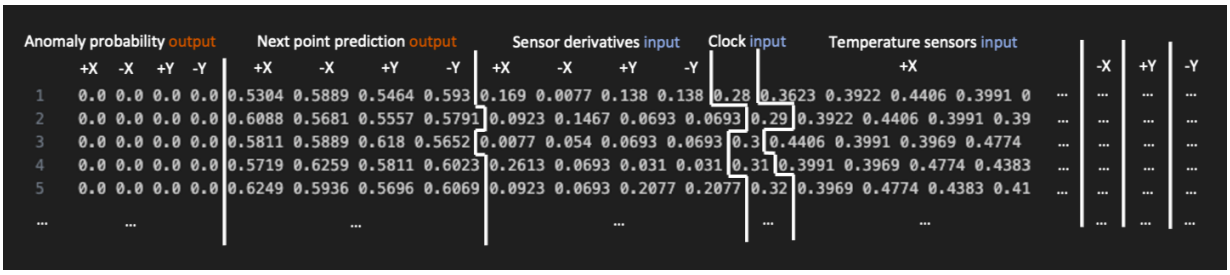


Figure 5.5: Example from the training dataset adm_2_9

It can be seen that all the temperature values are normalized, that is mapped into the range from 0.0 to 1.0. The minimum and maximum values for normalization are chosen to be -40 °C and +50 °C. This range is determined by adding a margin of 30 degrees to the limits of the dynamic range observed in the FUNcube data, which is roughly from -10 °C to +20 °C. The margin of 30 degrees is chosen as it is allows to represent not only the nominal data, but also the artificial anomalies with a magnitude ranging from 10

to 30°C. Sensor readings that are higher than 50 or lower than -40°C are obviously anomalies that are approaching the limits of the allowable temperature. These anomalies could be detected by a traditional threshold approach. Even though they could be added into the training dataset as well to show that the model can also detect these large anomalies, this would not be feasible for this demonstrator due to an increased dataset size and training time. At the same time, the margin of 30°C is chosen as it is not too high to start compromising the performance by decreasing the resolution of the temperature values.

As discussed previously in Chapter 4, the FUNcube telemetry has been split into 7 anomaly-free parts. For each of these parts, 32 variations are produced, where each variation of the telemetry part includes a unique artificially introduced anomaly. Each anomaly has a magnitude between 10 and 30°C, which can be either a positive or a negative number. The deviations of less than 10 degrees are not introduced in order reduce the size of the dataset and the associated training time. For the bias anomalies, the maximum magnitude of the introduced anomalies is reduced to 20°C in order to train the model on lower values of deviations and potentially make it more sensitive to these anomalies. This is done in order to ensure that the model will rely less on the derivative input when encountering these anomalies. The 32 variations of or each of the 7 telemetry parts are as follows:

Outliers:

- An outlier on 1 random panel (random magnitude from 10 to 30°C, random location)
- Outliers on 2 random panels (same random magnitude from 10 to 30°C, same random location)
- Outliers on 3 random panels (same random magnitude from 10 to 30°C, same random location)
- Outliers on all 4 panels (same random magnitude from 10 to 30°C, same random location)
- Outliers on 2 random panels (different random magnitude from 10 to 30°C, same random location)
- Outliers on 3 random panels (different random magnitude from 10 to 30°C, same random location)
- Outliers on all 4 panels (different random magnitude from 10 to 30°C, same random location)

Flat regions with an offset with respect to the last non-anomalous value:

- A flat region with an offset on 1 random panel (random offset magnitude from 10 to 30°C random location and duration of $\geq$20 min)
- Flat regions with same offset, location and duration on 2 random panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)
- Flat regions with same offset, location and duration on 3 random panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)
- Flat regions with same offset, location and duration on all 4 panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)
- Flat regions with different offset but the same location and duration on 2 random panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)
- Flat regions with different offset but the same location and duration on 3 random panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)
- Flat regions with different offset but the same location and duration on all 4 panels (random offset magnitude from 10 to 30°C random duration of $\geq$20 min)

Flat regions with no offset wrt. the last non-anomalous value:

- A flat region on 1 random panel (random location, lasts until the end of time series)
- Flat regions with same location on 2 random panels (random location, lasts until the end of time series)
- Flat regions with same location on 3 random panels (random location, lasts until the end of time series)
- Flat regions with same location on all 4 panels (random location, lasts until the end of time series)

Permanent temperature biases:

- A permanent bias on 1 random panel with a random magnitude from 10 to 20°C

- Permanent biases on 2 random panels with the same random magnitude from 10 to 20°C
- Permanent biases on 3 random panels with the same random magnitude from 10 to 20°C
- Permanent biases on all 4 panels with the same random magnitude from 10 to 20°C
- Permanent biases on 2 random panels with different random magnitude from 10 to 20°C
- Permanent biases on 3 random panels with different random magnitude from 10 to 20°C
- Permanent biases on all 4 panels with different random magnitude from 10 to 20°C

Temporary temperature biases:

- A temporary bias on 1 random panel at a random location and duration of $\geq$10 min., with a random magnitude from 10 to 20°C
- Temporary biases on 2 random panels at same random location, with same duration of $\geq$10 min., and same random magnitude from 10 to 20°C
- Temporary biases on 3 random panels at same random location, with same duration of $\geq$10 min., and same random magnitude from 10 to 20°C
- Temporary biases on all 4 panels at same random location, with same duration of $\geq$10 min., and same random magnitude from 10 to 20°C
- Temporary biases on 2 random panels at same random location, with different random duration of $\geq$10 min., and different random magnitude from 10 to 20°C
- Temporary biases on 3 random panels at same random location, with different random duration of $\geq$10 min., and different random magnitude from 10 to 20°C
- Temporary biases on all 4 panels at same random location, with different random duration of $\geq$10 min., and different random magnitude from 10 to 20°C

Applying this recipe results in a total of 224 individual anomalies, which are spread over 7533 data points. It can be noticed that some of the possible anomaly combinations are not included in the above list, which is because they were either deemed redundant (not differing fundamentally from other combinations included), or resulting in a too large dataset, the use of which would not be feasible because of a too long training time.

Training process

The NN training-related hyperparameters that were introduced in Subsection 5.2.1 have been the same during the entire preliminary design. As these parameters are not being optimized at this design stage, for now they are only starting points, which were determined through experimentation by tuning each of the parameters one-by-one until the loss function is minimized. In addition to the experimental approach, the initial selection of some hyperparameters relies on practices most commonly used in machine learning, such as using ReLU activations and ADAM optimizer. A summary of the discussed parameters can be found in Table 5.9:

Table 5.9: adm_2_9 model training parameters

Parameter	Value
Number of samples	26499
Fraction of samples allocated for validation	0.3
Number of training epochs	400
Batch size	100
Hidden layer activations	ReLu
Output layer activation	Not specified (Linear)
Optimizer function	ADAM
Learning rate	0.01
Loss Function	MAE

A more detailed exploration of these hyperparameters will be presented in Chapter 6.

Weight importance distribution
The design of the neural network can be further improved by inspecting the input layer weights of the network. This provides information about which inputs play more important role in the network inference than others. Less important or not important inputs can be eliminated in order to improve the overall performance of the model. In this analysis, the importance of an input layer neuron is calculated by taking the absolute value of the sum of the weights of all connections that the neuron has with the first hidden layer neurons. The distribution of these calculated importance scores for adm_2_9 is shown in Figure 5.6. In the figure, score '0' means that the input is not used at all.

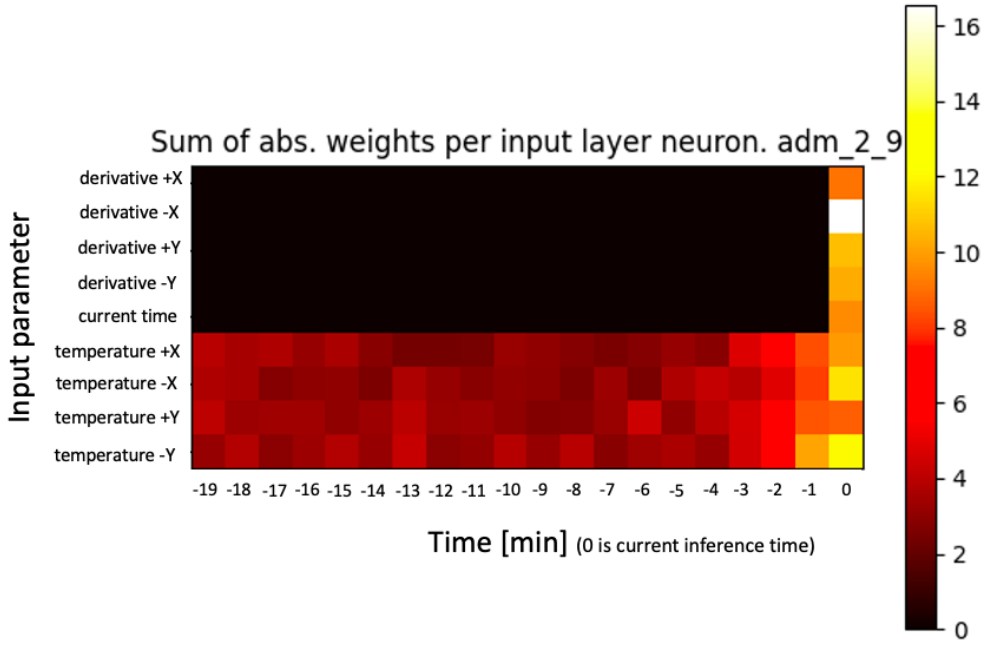


Figure 5.6: Weight importance distribution adm_2_9

From the analysis of the figure, it is concluded that the most important inputs are the derivatives, the time, and the last 4 temperature sensor readings. This information can be indicative of the fact that the number of temperature points used for each sensor is too high and can be safely reduced. This will be a part of the optimization process during the detailed design phase of the network.

5.4. Preliminary network performance

Once the preliminary design of the neural network is obtained, its performance can be tested. First, the original TensorFlow model stored in *.h5* format is tested. The results of this testing will be briefly discussed in Subsection 5.4.1. Then, the model is compressed and optimized using the so-called TF Lite converter. The operation of this converter is discussed in Subsection 5.4.2. Finally, the performance of the compressed model is explored in Subsection 5.4.3.

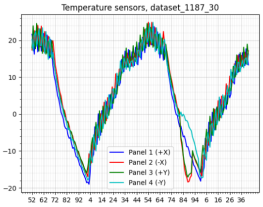
5.4.1. Performance before compression

The performance of the original model is tested using the procedure outlined in Subsection 5.2.2. The result of the testing is a calculation of two accuracy metrics: accuracy on anomalous data and accuracy on anomaly-free data. These results are summarized in Table 5.10:

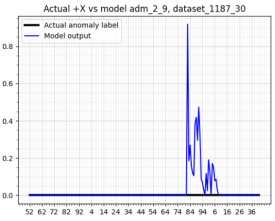
Table 5.10: adm_2_9 model performance

Parameter	Value	Unit
Model size	102	kB
Accuracy on anomalies, A_a (before deployment)	0.8195	[-]
Accuracy on NO anomalies, $A_n a$ (before deployment)	0.9699	[-]

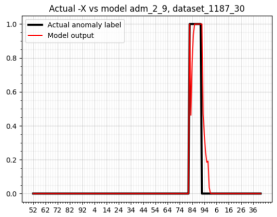
The accuracy on anomalies is quite high, which shows that the model is able to detect anomalies most of the time. However, further optimization for accuracy is required, as there is still a significant portion of anomalous data points that are not properly detected by the network. The accuracy on anomaly-free data is noticeably higher than the accuracy on anomalies. This shows that the model is more prone to not detecting existing anomalies than to producing false alarms. To illustrate this, the following example from the test report is provided. Figure 5.7 (a) shows an example input data including a temporary bias anomaly. In Figures 5.7 (b) - (e), the outputs of the model are plotted:



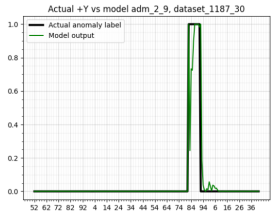
(a) Temperature input data



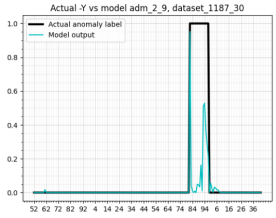
(b) Preliminary NN out. +X



(c) Preliminary NN out. -X



(d) Preliminary NN out. +Y



(e) Preliminary NN out. -Y

Figure 5.7: Preliminary model outputs example

As it can be seen from the above example, the model produces false alarms on panel +X and does not properly detect the anomaly on panel -Y (output probability temporarily drops almost to 0). This indicates that the model optimization for accuracy is required to arrive at the final design (will be discussed in Chapter 6). It can also be seen that the model size is larger than could be fit into the memory of the microcontroller (102 kB model vs 26 kB available space). This shows that an optimization for size also is required to reduce the memory footprint of the model. This optimization procedure will be applied by using the TF Lite converter, as discussed in more detail in the following subsection.

5.4.2. TF Lite converter

TF Lite converter is a Python Application Programming Interface (API) that allows to convert pre-trained TensorFlow $.h5$ models into smaller sized $.tflite$ models optimized for on-device deployment [67]. The following three parameters need to be specified in order to perform conversion:

- **Supported operators**: The TensorFlow Lite Converter is designed to analyze model structure and apply optimizations in order to make it compatible with the directly supported library operators. *TFLITE_BUILTINS_INT8* flag is set, which means that only TensorFlow Lite built-in operators are supported during conversion.

- **Representative dataset**: The representative dataset is needed for the converter to define the types of inputs the model works with, including the number of decimal places, dynamic range of values and the typical magnitude of the input values. For the anomaly detecting network, the
- **Optimizer**: The choice of a TF Lite converter optimizer defines a set of optimizations that are applied to the model. In the current version of TF Lite converter, a set of different optimizers that existed before is replaced by a single *DEFAULT* optimizer. The optimizations that are contained in the DEFAULT optimizer include:
  - *Operator fusion*: To improve efficiency of the model inference, TF Lite converter can combine multiple operations into a single operation. This reduces the number of separate operations to be executed, thereby improving the performance of the model. Other similar optimizations include eliminating unnecessary (redundant) operations.
  - *Quantization*: The DEFAULT optimizer enables post-training quantization of the model. During quantization, the converter converts the model's float type weights, biases and activation functions into less memory demanding representations, such as 8-bit integer (as was mentioned in Subsection 4.2.1, 8-bit quantization is chosen for the anomaly detecting model).
  - *Pruning*: During the weight- and bias pruning, the converter identifies and removes parameters and connections of low importance.

Using the TF Lite converter, the model is compressed from 102 kB size to 8 kB size, which is more than a 10-times reduction in the memory footprint. Now the model fits well within the microcontroller's memory (either flash or RAM as in the testing system), which enables on-device deployment.

5.4.3. Deployment on MSP432

For the testing of the preliminary model on the MSP432 microcontroller, the testing system discussed in Chapter 4 is used. The tests are performed on the input data corresponding to a telemetry part, that was intentionally not included in the training dataset (part 1187). The testing system produces a test report which contains important metrics defining the performance in terms of memory usage, inference time and accuracy of anomaly detection. The measured and calculated performance metrics are summarized in Table 5.11:

Table 5.11: adm_2_9 model performance

Parameter	Value	Unit
Model size	8192	B
Estimated RAM footprint (using AllOpsResolver)	44392	B
Estimated RAM footprint (using MicroMutableOpResolver)	38692	B
Time per inference	0.126	s
Accuracy on anomalies, A_a (AFTER deployment)	0.7554	[-]
Accuracy on NO anomalies, $A_n a$ (AFTER deployment)	0.8698	[-]

The model performs well in terms of RAM footprint, as it uses only 67% of the memory when deployed within the standard testing system (with *AllOpsResolver*), and could use even less if properly optimized for integration into the flight software.

The time taken per inference is again in the order of 0.1 seconds, which allows to comply with the requirement on the minimum frequency of the inferences (once every 15 seconds).

The anomaly detection accuracy has degraded slightly after the compression of the model, and the imbalance between the accuracy on anomalous and anomaly-free data still persists. Nonetheless, there is space for optimizations that will be explored during the final design phase, which allows to expect better performance of the final network.

6

Detailed Design of The Neural Network

The focus of this chapter is on the final design of the neural network, which is achieved primarily by fine-tuning the hyperparameters of the existing anomaly-detecting neural network. In Section 6.1, the preparation of the model for the detailed design phase is discussed. Next, the process of hyperparameters tuning is explained in detail in Section 6.2. After that, the resulting ML model characteristics and performance are discussed in Section 6.3. Finally, Section 6.4 contains a discussion of the implementation of the solution for the clock anomaly problem.

6.1. First iteration of the detailed design

The detailed design that will lead to the final version of the anomaly-detecting model is also an iterative procedure. At each iteration a certain hyperparameter value is changed to see what value of this hyperparameter would lead to an improvement in the network performance. This iterative procedure is started with a model that has been obtained through the preliminary design. However, before the hyperparameters of the model can be tuned, the dataset needs a final adjustment which is achieved by removing the next point prediction outputs that were initially used for debugging purposes. This adjustment results in the first iteration of the final model to be tuned and optimized, it is given the name adm_3_0. The comparison of the model performance in terms of anomaly detection accuracy (as defined in Subsection 3.3.2) before and after the adjustment is shown in Table 6.1:

Table 6.1: Comparison of the preliminary model vs initial iteration of the final model

	adm_2_9	**adm_3_0**
Model size [kB]	102	98
Accuracy on anomalies, A_a (before deployment)	0.8195	0.8153
Accuracy on NO anomalies, A_{na} (before deployment)	0.9699	0.978

As it can be seen from the comparison, the model performance stays at roughly the same accuracy values, which demonstrates that the prediction output did not play an important role in the internal functioning of the neural network. The size of the model, however, has reduced, which is beneficial for the subsequent on-device deployment.

6.2. Hyperparameters tuning

In the final design phase described in this chapter, various network hyperparameters are adjusted based on the changes in the network performance in terms of detection accuracy. The network hyperparameters include the variables such as the number of neurons and layers, training epochs, batch size, optimizer function, learning rate and others. During the preliminary design, the initial choice of these training hyperparameters has been made by relying on experimentation, and rules-of-thumb generally used in ML, as was discussed in Chapter 5. From there, these hyperparameters are further fine-tuned one-by-one

starting with the parameters that are related to the architecture of the network (sizes of input and hidden layers, number of hidden layers), and moving towards training process specific parameters. At the end of the final network design, an anomaly detection threshold is chosen by using a similar iterative approach.

6.2.1. Balancing of the dataset

As has been previously explained in Chapter 5, the training dataset consists of subsets, where each subset corresponds to an individual telemetry part with or without applied anomalies. The number of occurrences of anomalies on each of the panels is adjusted in a way that each panel has equal amount of such occurrences. This is to ensure that the training algorithm would not give preference to the inputs received from a particular panel rather than the other panels.

Additionally, the frequency of anomaly occurrences in general is brought to roughly one anomaly per one orbital period, which matches the formulation of the requirements on the minimum frequency of anomaly detection. This allows the network have adequate sensitivity to anomalies and not produce excessive false alarms.

6.2.2. Network layers and neurons

The first step in the hyperparameter tuning process is the tuning of the number of neurons and the neuron layer configuration. This includes the number of neuron layers, number of neurons per layer and the layer symmetry (whether the number of neurons is the same in all hidden layers or not). Additionally, the optimal number of neurons for the temperature series input is determined.

Number of neurons per layer

First, the number of neurons in the hidden layer is tuned, which is achieved by comparing the original 32x32x32 hidden layer configuration to the one with 2, 3 and 4 times as large configurations. For each case, the network is retrained and tested for the anomaly detection accuracy. The results are shown in Table 6.2:

Table 6.2: Tuning for number of neurons

	adm_3_0	**adm_3_1_0**	**adm_3_1_1**	**adm_3_1_2**
Hidden layer structure	32x32x32	64x64x64	96x96x96	128x128x128
Model size [kB]	98	209	369	573
Accuracy on anomalies, A_a (before deployment)	0.8153	0.8649	0.9068	0.8225
Accuracy on NO anomalies, A_{na} (before deployment)	0.978	0.9844	0.9788	0.9764

The model with 96 neurons in each of the three hidden layers, adm_3_1_1, shows the best results in both of the two accuracy metrics. Increasing the number of neurons seems to improve the accuracy until a certain point as the model becomes capable of learning more complex patterns. Beyond certain size, however, adding new neurons does not increase performance anymore, which is likely due to model overfitting. adm_3_1_1 model is now chosen as the baseline for the tuning of the next parameter, which is the number of hidden layers.

Number of hidden layers

Instead of a three-layer configuration, 2- and 4-layer configurations are tried for 96 neurons per layer. To make sure there are no configurations with more (128) or less (64) neurons per layer, that could lead to better performance, some of them are also tested. The results can be seen in Table 6.3:

Table 6.3: Tuning for number of layers

	adm_3_1_1	adm_3_2_0	adm_3_2_1	adm_3_2_2	adm_3_2_3
Hidden layer structure	96x96x96	64x64x64x64	96x96x96x96	96x96	128x128
Model size [kB]	369	270	487	250	373
Accuracy on anomalies, A_a (before deployment)	0.9068	0.7929	0.8767	0.9006	0.8065
Accuracy on NO anomalies, A_{na} (before deployment)	0.9788	0.9672	0.9714	0.979	0.9696

No improvement is achieved in any of the configurations when compared to the original adm_3_1_1 model. adm_3_2_2 shows a decrease in size while keeping the same accuracy, however, after the inspection of the output plots, it was concluded that the output of adm_3_2_2 is less stable and more noisy. Therefore, adm_3_1_1 with three layers is still the preferred option.

Layer symmetry
adm_3_1_1 model has a total of 288 neurons in the three hidden layers, which however, could be distributed differently across the layers, resulting in different layer symmetry. Experiments with layer symmetry are shown in Table 6.4:

Table 6.4: Tuning for layer symmetry

	adm_3_1_1	adm_3_3_0	adm_3_3_1	adm_3_3_2	adm_3_3_3
Hidden layer structure	96x96x96	64x96x128	128x96x64	96x128x64	96x64x128
Model size [kB]	369	336	397	389	319
Accuracy on anomalies, A_a (before deployment)	0.9068	0.7908	0.826	0.8872	0.871
Accuracy on NO anomalies, A_na (before deployment)	0.9788	0.9771	0.973	0.9787	0.9778

No improvement is achieved for any of the modifications of the original model with uniform neuron distribution. It can be seen that the choice of the number of neurons is the most critical for the first layer, and does not result in significant changes for the other layers. Layers with equal number of neurons result in the best performing network, which is still adm_3_1_1.

Finalizing the number of neurons per layer
Finally, once the importance of uniform neuron distribution is established, the number of neurons per layer can be varied slightly in order to see if a configuration with slightly more/less neurons per layer would perform better. For this, two more models are explored: one with 101 and one with 91 neurons per layer. The performance testing results for these two models and the original one are shown in Table 6.5:

Table 6.5: Fine-tuning the number of neurons per layer

	adm_3_1_1	adm_3_4_0	adm_3_4_1
Hidden layer structure	96x96x96	101x101x101	91x91x91
Model size [kB]	369	397	340
Accuracy on anomalies, A_a (before deployment)	0.9068	0.8535	0.831
Accuracy on NO anomalies, A_{na} (before deployment)	0.9788	0.9817	0.9805

Even when varying the number of neurons per layer by as few as 5 neurons, the accuracy on anomalous data decreases. The accuracy on non-anomalous data increases slightly, which, however, does not compensate for the decrease in A_a. At a system level, false alarms are not as critical as false negatives, as they do not involve any risk to the satellite, unlike the undetected anomalies. Therefore, adm_3_1_1 is still the optimal configuration.

Number of temperature inputs
To finalize the design of the structure of the neural network, the length of the temperature series received from each sensor is chosen. Initially, the length of 20 has been used, which however was deemed to be redundant as was explained in Subsection 5.3.4. Now the length of the temperature series is reduced to 10, 5 and 3 temperature points and the anomaly detection accuracy is measured, for which the results are shown in Table 6.6

Table 6.6: Fine-tuning the number of input layer neurons

	adm_3_1_1	**adm_3_5_0**	**adm_3_5_1**	**adm_3_5_2**
Number of temperature points per panel	20	10	5	3
Number of input layer neurons	85	45	25	17
Model size [kB]	369	319	296	287
Accuracy on anomalies, A_a (before deployment)	0.9068	0.8947	0.9392	0.9449
Accuracy on NO anomalies, A_{na} (before deployment)	0.9788	0.9905	0.9853	0.9913

Reducing the number of inputs to 3 temperature points per panel has improved both the size and the accuracy of the model. Further reducing the length of the series to less than 3 inputs is not desired, otherwise the network might not capture more intricate patterns when further optimizations are applied. Therefore, adm_3_5_2 is the model with the most optimal structure of the neurons and layers. Next, the activation functions and training-related hyperparameters are explored.

6.2.3. Activation functions

The activation functions define the behaviour of individual neurons in the network. The activation functions of the hidden layers have to be non-linear functions, so that they can capture complex patterns in the data. At the same time, the choice of the output layer activation is task-specific, and thus has to be made separately from the choice of the hidden layer activations.

Hidden layer activations
The following options are most suitable and commonly used for the hidden layer activations: [68]

- Rectified Linear Unit (ReLU): ReLU is the most common activation function which can effectively handle non-linear relationships. In case the summation of the neuron weights results in a negative value, ReLU sets the output to zero, while for the positive values, it sets the output equal to the sum of weights.
- Exponential Linear Unit (ELU): ELU is very similar to ReLU with the only difference being the way it handles negative values. In ELU, the negative values are smoothly mapped in the range from 0 to -1 using an exponential relationship. In some cases, ELU can in superior performance compared to ReLU.
- Softplus: this is a smooth (unlike ReLU), non-linear function that maps the weight sum values to positive values only. Softplus is often used when when the network works exclusively with positive input values. As normalization is applied to the input data, all inputs of the network are indeed positive, thus Softplus function is also tested.
- Gaussian Error Linear Unit (GELU): while ReLU consists of two linear parts, GELU approximates the same shape using a single (smooth) line. GELU can outperform other functions in certain applications.

The comparison in terms of anomaly detection accuracy between networks with each of these activation functions used is shown in Table 6.7:

Table 6.7: Choosing hidden layer activation function

	adm_3_5_2	**adm_3_6_0**	**adm_3_6_1**	**adm_3_6_2**
Hidden layer activations	ReLU	ELU	Softplus	GELU
Accuracy on anomalies, A_a (before deployment)	0.9449	0.8709	0.6831	0.9053
Accuracy on NO anomalies, A_{na} (before deployment)	0.9913	0.9832	0.9657	0.9782

The testing results demonstrate that ReLU is the optimal choice, while all other options of activation functions result in inferior performance.

Output layer activations
Given the task of producing an anomaly probability output between 0 and 1 with uniform distribution of probabilities between these two values, only the following two options would be suitable for the output layer activation function:

- Linear Function: this is the simplest activation that outputs the weighted sum of the connection weights (plus a bias) without applying any non-linear transformations. This allows the network to output a float value that directly represents the anomaly probability evaluated.
- ReLU: although it is mainly used for hidden layers, in this model, ReLU can also be applied to the output layer. However, ReLU is not designed to constrain the output to the 0 to 1 range, which affect the detection accuracy performance.

As the design goal is to estimate a probability between 0.0 and 1.0 without applying any non-linear transformations, the linear activation function is the simplest and most suitable choice for the anomaly-detecting network's output layer.

6.2.4. Training parameters

Optimizer
The optimizer is a method used to dynamically change the attributes of the neural network, such as weights and learning rate during the training process. This allows to minimize the training and validation losses, thus making the network attributes converge at optimal values.

For the anomaly detecting network application, ADAM (Adaptive Moment Estimation) optimization function is chosen as it combines the advantages of the most popular algorithms used for model optimization, such as Adaptive Gradient Algorithm (AdaGrad), the Root Mean Square Propagation (RMSProp), and the classical stochastic gradient descent method. For this reason, ADAM is commonly considered the best optimizer function for most NN applications. [69]

Learning rate
Learning rate is a parameter that is specified within the optimizer function. This parameter defines how fast the model converges at optimal values of attributes during training. Decreasing the learning rate from 0.01 to smaller values, for example 0.001 (10 times smaller), can help the model converge more gradually and potentially achieve better performance.

On the other hand, increasing the learning rate to values like 0.1 (10 times larger) or even higher, can result in faster and more impactful updates of model weights, which might help the model escape from local minima or achieve faster convergence. The learning rates of 0.001 and 0.1 are tested and the resulting performance is compared to the original model as shown in Table 6.8:

Table 6.8: Comparing different learning rates

	adm_3_5_2	adm_3_7_0	adm_3_7_1
Learning rate	0.01	0.1	0.001
Accuracy on anomalies, A_a (before deployment)	0.9449	0.9241	0.9352
Accuracy on NO anomalies, A_{na} (before deployment)	0.9913	0.9948	0.9924

The model's accuracy on anomalous data degrades by roughly the same magnitude when both increasing or decreasing the learning rate. Although it can be also noted that the accuracy on anomaly-free data is slightly higher for the modified models, the degradation of A_a is more significant in this case. It is thus concluded that 0.01 is actually the optimum value of the learning rate.

Batch size
The batch size represents the amount of training data to which the model is exposed at every training epoch. In the case the model is overfitting, reducing the batch size can be beneficial. With smaller batch sizes, more noise is introduced into the parameter updates, which can help prevent overfitting of the model. The downside of smaller batch sizes is that they might lead to slower convergence.

Larger batch sizes can improve training speed as the model can process more examples simultaneously. However, using larger batch sizes generally results in larger memory footprints during training and can lead to overfitting. For this reason, the optimization will be in the direction of reducing the batch size. The results for various batch sizes ranging from 100 to 25 are presented in Table 6.9:

Table 6.9: Comparing different training batch sizes

	adm_3_5_2	adm_3_8	adm_3_8_0	adm_3_8_1
Batch size	100	75	50	25
Accuracy on anomalies, A_a (before deployment)	0.9449	0.9538	0.9591	0.9312
Accuracy on NO anomalies, A_{na} (before deployment)	0.9913	0.9891	0.9879	0.9843

The batch size of 75 results in an improvement in anomaly detection compared to the size of 100. Even though the batch sizes of 50 and 25 do not show improvement, it is generally true that smaller batch sizes have potential of resulting in a model with higher accuracy, which can be further explored in this case. However, the training time increases drastically with smaller batch sizes, thus it might not be feasible. Therefore, adm_3_8 is now the optimal model.

Number of training epochs
The number of training epochs is another important parameter that influences the training time. Increasing the number of training epochs exposes the model more to the same training data. This allows the model to potentially capture the more complex data patterns that it could not capture during previous iterations. At the same time, training for more epochs can cause the model to shift from generalization to overfitting and simply memorizing the training data.

Decreasing the number of epochs, on the other hand, can speed up the training process as well as prevent overfitting. Training the model for too few epochs, however, can lead to underfitting (not learning the patterns it could learn).

In this optimization process, the number of training epochs is incremented by +- 100, to estimate the general effect of training for longer or shorter time. The results for 300, 400 and 500 training epochs are shown in Table 6.10:

Table 6.10: Fine-tuning the number of training epochs

	adm_3_8	**adm_3_9_0**	**adm_3_9_1**
Training epochs	400	300	500
Accuracy on anomalies, A_a (before deployment)	0.9538	0.9156	0.9516
Accuracy on NO anomalies, A_{na} (before deployment)	0.9891	0.9765	0.9897

The model trained for 300 epochs, adm_3_9_0, does not only demonstrate worse performance, but also produces a considerably more noisy output. Increasing the number of epochs, on the other hand, results in less noise, and the training loss (MAE loss function, explained later in this subsection) approaches the limit value of roughly 0.006. The accuracy of model predictions, however, does not show visible improvement past 400 training epochs. Thus it is still the optimum value. One more test has been performed for 700 training epochs. This test confirmed that the training loss plotted in Figure 6.1 approaches the limit after about 400 epochs.

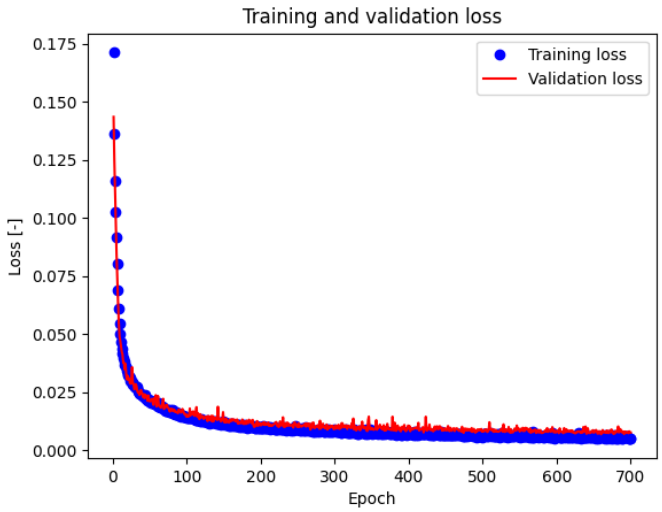


Figure 6.1: Training and validation loss evolution

Loss function

The loss function is a function that measures the deviation of the model predictions from the expected outputs (actual data labels). The optimization process is based on comparing the following loss functions that are most commonly used in ML for regression, anomaly detection and classification tasks:

- Mean Squared Error (MSE): this function is commonly used for regression tasks. When applied to the anomaly-detecting model, it would calculate the mean squared difference between the estimated anomaly probability and the true anomaly label.
- Mean Absolute Error (MAE): also commonly used for regression tasks. It calculates the mean absolute difference between the estimated anomaly probability and the true anomaly label. MAE is less sensitive to outliers in the calculated difference than MSE.
- Binary Cross-Entropy (BCE): mainly used binary classifier networks, but it can also be used for a model that outputs an anomaly probability. BCE is a function that involves natural logarithms, that would be a measure for (dis-)similarity between the anomaly probabilities detected by a network and the actual anomaly labels.
- Mean Squared Logarithmic Error (MSLE): another function that is mostly used for regression tasks.

MSLE calculates the mean squared difference between the natural logarithms of the anomaly probability estimated by the model and the true anomaly label.

The comparison of the results for the networks trained with the different loss functions is shown in Table 6.11:

Table 6.11: Choosing the loss function

	adm_3_8	**adm_3_10_0**	**adm_3_10_1**	**adm_3_10_2**
Loss function	MAE	MSE	BCE	MSLE
Accuracy on anomalies, A_a (before deployment)	0.9538	0.8712	1.0	0.8958
Accuracy on NO anomalies, A_{na} (before deployment)	0.9891	0.9798	0.0	0.989

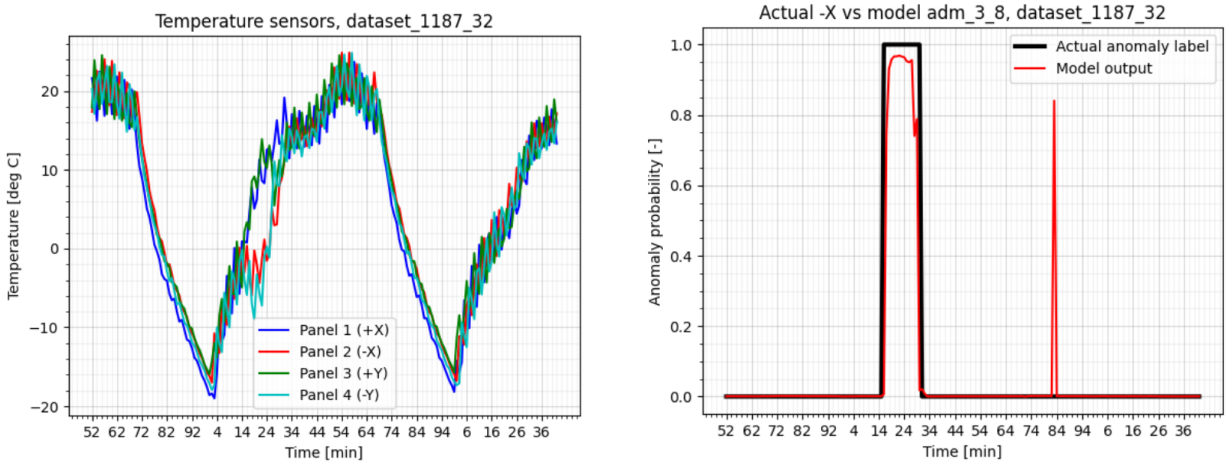
MAE proves to be the best option for the network, while MSE and MSLE result in noticeably lower accuracy on anomalous data. As mentioned previously, the superior performance with MAE is likely due to the decreased sensitivity to outliers. The BCE function resulted in an inadequately performing model, which demonstrated that it is actually inapplicable to this type of networks.

6.2.5. Detection threshold

To finalize the design of the neural network, a detection threshold is applied on the anomaly probability estimated by the neural network. This threshold is a value between 0 and 1, below which all model outputs are classified as 0 (no anomaly), and above which all outputs are classified as 1 (anomaly present).

The choice of the threshold is a trade-off meant to minimize the undesired effects caused by a too low or a too high set threshold. If the threshold is set too low, the system becomes very sensitive to any irregularities in the incoming temperature data and produces false alarms, which is undesired as it may initiate unnecessary FDIR procedures. On the other hand, if the threshold is set too high, the system may not be able to properly detect anomalies when they occur.

In order to quantize the effect of applying the detection threshold, a set of metrics presented in Chapter 3 is used. These metrics are: *recall, precision, F1-score,* and *False Alarm Rate (FAR)*. Although no system-level requirements were initially imposed on the values of these parameters, a judgement can still be made on the values that would be adequate for the given application. Figure 6.2 (a) shows an example of a temporary temperature bias anomaly in panels -X and -Y. The corresponding output of the adm_3_8 model (the best performing model discussed so far) is plotted against the expected model output (actual anomaly label) in Figure 6.2 (b).



(a) Example temperature anomaly

(b) adm_3_8 anomaly detection output

Figure 6.2: Example temperature anomaly and the corresponding model output

It can be seen that the model predicts the anomaly with very high probability (>90%). The analysis of the test report of the adm_3_8 model reveals that this high confidence of detection is the case for almost all anomalies in the testing data set, regardless of their magnitude with respect to the nominal temperature behaviour. This allows to set the value of the threshold high, without a significant risk of an anomaly not being detected.

At the same time, this example is also illustrative of a potential false alarm (a single point spike around minute 84 in Figure 6.2 (b)). By inspecting Figure 6.2 (a), it can be noticed that this false alarm is most likely caused by the slight deviation of the panel -X sensor from the other three sensors. Such individual small deviations are common in the available FUNcube data. The analysis shows that the neural network is quite sensitive to such anomalies and often assigns to them such a high probability, that they can be mistakenly classified as anomalies. One important distinction of such false alarms is the fact that almost always they manifest in the form of individual spikes. This allows to introduce a post-processing algorithm that would eliminate all positive network classifications if they do not hold for 2 or more consecutive time points. This eases the requirement on the threshold height (false alarms can be filtered out, so they are not that critical), however, it should preferable stay as high as possible to reduce the occurrence of false alarms. Table 6.12 shows a comparison of adm_3_8 model performance with different thresholds applied:

Table 6.12: adm_3_8 classifier performance with different detection thresholds

	0.9	**0.85**	**0.8**	**0.75**
recall	0.9224	0.9327	0.9369	0.9468
precision	0.9923	0.9905	0.9819	0.9796
F1	0.956	0.9607	0.9589	0.9629
FAR	0.0016	0.002	0.0038	0.0044

After inspecting the corresponding test reports, it was concluded that there are no significant changes in anomaly detection performance when the value of the threshold is changed. The only noticeable difference is the increased false anomaly rate, which requires more filtering of the outputs to eliminate the false alarms. Based on the F1 score, the most favorable options for the threshold are 0.85 and 0.75, however, given the increased FAR, it is recommended that the threshold of **0.85** is used despite the slightly lower F1 score.

6.3. Final design of the neural network

The hyperparameters tuning process described in the previous section has resulted in the final design of the anomaly-detecting neural network. Now the network is tested using the procedure explained in Subsection 5.2.2. The testing dataset is the same as for the preliminary network: it consists of a nominal two-orbit telemetry part 1187 and 32 variations of this telemetry part with artificially introduced anomalies. As for the preliminary network, the two anomaly detection accuracy metrics are calculated. However this time, also the anomaly detection threshold can be applied and the four accuracy metrics of a binary classifier model are be calculated. The design specifications, including the selected hyperparameters and the measured performance metrics of this final neural network design are summarized in Table 6.13:

Table 6.13: Final neural network design specifications. adm_3_8 anomaly detector.

Parameter	Value
Architecture	
Model type	MLP
Number of input neurons	17
Number of output neurons	4
Hidden layer structure	96x96x96
Hidden layer activations	ReLU
Output layer activation	Linear
Model size [kB]	287
Training	
Number of samples in the train. dataset	26499
Fraction of samples allocated for validation	0.3
Number of training epochs	400
Batch size	75
Optimizer function	ADAM
Learning rate	0.01
Loss Function	MAE
Performance	
Accuracy on anomalies, A_a (before threshold is applied)	0.9538
Accuracy on NO anomalies, A_{na} (before threshold is applied)	0.9891
Detection threshold	0.85
Recall	0.9327
Precision	0.9905
F1	0.9607
FAR	0.002

It should be noted that all the performance-related specifications shown in the table above are obtained for the network before it is compressed and deployed on the microcontroller. The current network with the size of 287 kB obviously cannot be deployed on the microcontroller as is, and model compression is needed. The deployment and testing of the network on the MSP432 board will be discussed in Chapter 7.

In the current state, the model shows a visible improvement in performance compared to the preliminary design. The detection accuracy on anomalous data, A_a has increased by 16.4% and the accuracy on anomaly-free data, A_{na} has increased by 2.0%. When the detection threshold is applied, values of recall, precision and F1 score are quite high (approaching 1.0). When anomalies are present, the model is able to detect them in 93.3% of cases, and 99% of all detected anomalies are true positives (less than 1% are false alarms). These results will be further evaluated in Chapter 7 to produce a judgement on whether they are satisfactory for the target application. The same will be done for the model once it is deployed on the microcontroller.

6.4. C++ code for the time estimation algorithm

As was explained in Section 5.3, the clock anomaly problem is solved by a separate time estimation algorithm, as a NN-based solution would be too large in terms of memory size. In Section 5.3, also a concept of the algorithm has been proposed. A modified version of that algorithm has been developed and written in Python. That modified algorithm has actually been used throughout the preliminary and final design stages of the neural network to produce the time stamps (anomaly-free clock readings) that

the neural network uses as input. This algorithm takes an entire telemetry part of temperature data and generates time stamps for all data points at once, which means that this is a post-processing algorithm that does not work with streaming sensor data. The Python algorithm works as shown in the pseudocode below:

Python algorithm to produce time stamps

```
telemetry_list = read_telemetry_data()
#make a list for time series of the same length:
time_series = [0 for i in range(len(telemetry_list))]
ref_point = find_index_of_the_lowest_temperature()

for data_point in telemetry_list[ref_point::-1]:
  extrapolate_time_backwards()
  allocate_time_into_time_series()
for data_point in telemetry_list[ref_point:]:
  extrapolate_time_forward()
  allocate_time_into_time_series()

write_time_series_into_txt_file()
```

The Python algorithm requires at least one full orbit of temperature data to work correctly. In the real-life scenario, however, the telemetry for the previous orbit might not be available all at once and the time estimation algorithm has to be able to reconstruct the time based on the new incoming data. To achieve this functionality, the algorithm is modified as follows for the embedded C++ code implementation:

1. At initialization, the algorithm assumes time 0 (spacecraft leaves the Earth's shadow).
2. If the average temperature among 4 sensors keeps decreasing, the time is set to 0 at every subsequent time point, where the temperature is lower than the lowest average temperature measured so far.
3. If the temperature is getting higher than the minimum temperature measured within the past orbital period, the time estimation simply adds one minute to the current time estimate.
4. It continues increasing the time value incrementally, until it reaches the new expected minimum.
5. then it sets the time to 0 again and checks whether a correction needs to be introduced.

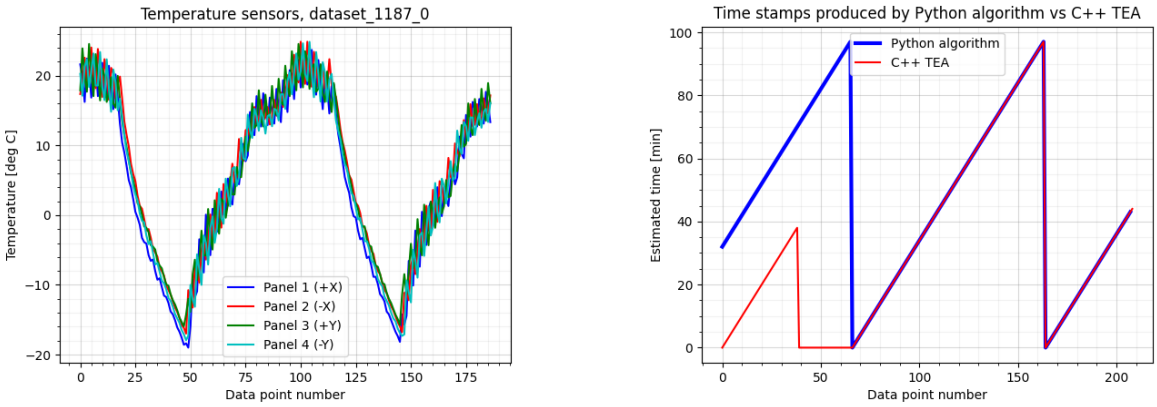
In pseudocode, this looks as follows:

Pseudocode for TEA implemented in C++

```
void setup() {
  int time = 0;
  float last_min_temp = 50.0; //[deg C]
  const int T_orb = 98; //Specify orbital period [min]
}
void loop() {
  receive_temperature_data();
  avg_t = compute_average_temperature();
  //reset time if a minimum is encountered
  if (avg_t < last_min_temp){
      time = 0;
      ast_min_temp = avg_t;
  }
  //otherwise increment time
  else{
```

```
      time ++;
  }
  //reset time when  1 orbital period elapses.
  //If the minimum is not passed yet, this will be corrected at the next iteration.
  if (time == T_orb){
      time = 0;
      last_min_temp = avg_t;
  }
}
int main(){
  setup(); //run once
  while(true){ //run forever
    loop();
  }
}
```

A comparison between the two algorithms is made by running them on the same telemetry part that has been reserved for the testing of the neural network (part 1187), and the results are presented in Figure 6.3:



(a) Sensor data from telemetry part 1187

(b) Comparison of Python and C++ algorithms

Figure 6.3: Time Estimation Algorithm performance analysis

It can be seen that in this test case, the C++ TEA requires around 65 minutes to start producing the correct time value, as the actual temperature minimum is not registered by the algorithm up until minute 65. However, once the temperature minimum is encountered, the algorithm is able to produce correct time stamps for the next orbital period and beyond. These limitations of the developed time estimation algorithm and their effect on the system performance will be assessed along other results in Chapter 7.

Once the time estimation algorithm is integrated into the microcontroller testing system discussed in Chapter 4, the embedded code changes slightly, as now there is an option to run the network with a time estimation from TEA by using a **'t'** command flag. This is illustrated by the following pseudocode:

Pseudocode for MSP432P401R embedded software with TEA

```
//main_functions.c
#include <ti/drivers/UART.h>
#include "tensorflow/lite/micro/…"

void setup() {
```

```
  UART_receive_model_data_size();
  UART_receive_model_input_size();
  UART_receive_model_output_size();
  UART_receive_model_data();
  initialize_TF_Lite_interpreter();
}

void loop() {
  UART_receive_input_data();
  if("f"_flag_read == true){
    clear_input_tensor();
  }
  if("t"_flag_read == true){
    estimate_time_with_TEA();
    add_TEA_output_to_input_tensor();
  }
  else{
  make_input_tensor_with_input_data();
  }

  if("i"_flag_read == true){
    run_tensorflow_inference();
    UART_send_output_data();
  }
}
int main(){
  setup(); //run once
  while(true){ //run forever
    loop();
  }
}
```

7

System Integration and Testing

In this chapter, the results of the final system integration and deployment of the designed neural network on the microcontroller board are discussed. First, in Section 7.1, the deployment of the final neural network design on the MSP432 microcontroller is explained. An overview of the neural network testing results is provided in Section 7.2. Validation and verification of the obtained results is explained in Section 7.3. Finally, a judgement is made on whether the obtained results fulfil the system level requirements, which will be discussed in Section 7.4.

7.1. System integration

As it has been discussed in the previous chapter, the resulting NN has the size of 287 kB, and compression is necessary to deploy it on the microcontroller. For this purpose, again the TF Lite converter is used with the same settings as listed in Subsection 5.4.2. With this settings, the TF Lite converter applies 8-bit integer quantization, pruning and operator fusion optimizations to the model.

One of the important arguments of the converter function is the representative dataset that is needed for the converter to define the types of inputs the model works with, including the dynamic range of values and the typical magnitude of the input values. For the conversion of the final model, a dataset part made with telemetry part 44 is used. This dataset part is chosen as it has been previously included in the main training dataset and is a good representation of the data that the model normally works with. In order to avoid potential overflow problems in the compressed model, the dataset 44 temperature and sensor derivative entries are multiplied by a safety factor of 1.6. This allows to increase the dynamic range of the data to represent also abnormally high/low values that can occur within the entire normalization range (-40°C to +50°C).

Using the TF Lite converter with these settings and the representative dataset, the model is compressed from a size of 287 kB to 24 kB (24368 B), which is over a 10 times reduction. The compressed model data is stored in a testing system directory from where it can be sent to the MSP432 board.

TEA is also integrated into the final version of the embedded software, as was shown in Section 6.4. Then, the firmware is updated one last time, after which the microcontroller is ready for the final testing using the testing environment previously discussed in Chapter 4. The testing is conducted on the part of the dataset that was not included in the training process, the *dataset_1187*, that was used to test the preliminary and the final design of the anomaly detecting model before deployment. This allows for a fair comparison between the results before and after deployment of the model on the microcontroller board. In the next section, an overview and discussion of the obtained testing results is provided.

7.2. Results of the neural network testing

Two test sequences are performed on the final model (compressed *adm_3_8*): one without a threshold applied, and the other with the threshold of 0.85 applied. In both cases, the system is tested on 33 data sets, which include 1 dataset without any anomalies (*dataset_1187_0*), and 32 variations of this dataset with the various (combinations of) anomaly types. Before the threshold is applied, the accuracy metrics are used as in the preliminary model testing. This allows to see how much improvement the final model shows compared to the preliminary one. After the threshold is applied, the recall, precision, F1 and FAR metrics are used, as the model effectively becomes a binary classifier.

Table 7.1 presents a summary of the most critical performance parameters of the final model after it has been deployed and tested on the microcontroller, which among others includes the measured and calculated anomaly detection accuracy performance metrics which were defined in Subsection 3.3.2:

Table 7.1: adm_3_8 model performance on MSP432

Parameter	Value	Unit
Model size	24368	B
Estimated RAM footprint (using AllOpsResolver)	60568	B
Estimated RAM footprint (using MicroMutableOpResolver)	54868	B
Time per inference	0.119	s
Accuracy on anomalies, A_a (AFTER deployment)	0.8934	[-]
Accuracy on NO anomalies, A_{na} (AFTER deployment)	0.9039	[-]
Recall	0.8588	[-]
Precision	0.8988	[-]
F1-score	0.8784	[-]
FAR	0.0213	[-]

In the following subsections, these and some other performance-related results will be discussed in more detail.

7.2.1. Memory footprint

As it can be seen from Table 7.1, the model data size of the final deployed neural network is 24368 B, which is almost 3 times larger than the preliminary deployed model. The total estimated RAM footprint of the model and the test system software necessary to run it is 60568 B (92.5% of the available RAM). This includes 24368 B of model data and 36200 B reserved for the TF Lite library and other operations. In case *MicroMutableOpResolver* would be used as the operations resolver, the total RAM footprint would be 54868 B (83.8% of the available RAM). At the same time, the flash memory usage is 236000 B (89% of the available flash).

It is worth noting again that these results only hold for the microcontroller-based testing system used to study the performance of the model. If the neural network were to be deployed in the flight software of a satellite, the model data would be stored not in RAM, but in the flash memory. It would also be reasonable to use the *MicroMutableOpResolver* for this application. Then, in the flight software system, the total flash usage of the anomaly-detecting network would be around **65.2 kB** (25.5% of the available flash) and RAM usage around **29.5 kB** (46.1% of the available RAM).

7.2.2. Inference speed

As it was stated in Subsection 5.4.3, the time taken per single inference is 0.126 s for the preliminary model. By considering the serial port baud rate and the number of transferred bytes, it is estimated that about 0.045 s (35.7%) of this measured inference time is taken by the data transfer.

This time, the measured time per inference is 0.119 s, which is about 6% faster than for the preliminary model. Because of the smaller input size of the final model, the estimated data transfer time is 0.012 s, which is 73.3% faster than for the preliminary model. Although the final model is more complex and the time taken exclusively by running the neural network is higher, the faster data transfer due to the smaller input tensor results in the decrease in the total inference time.

7.2.3. Anomaly detection accuracy

The measured values of the accuracy metrics are compared to those measured before the deployment of the model on the microcontroller. This allows to study the effect that the model compression has on the accuracy of anomaly detection. Table 7.2 contains the comparison between the performance of the model before and after compression:

Table 7.2: NN detection accuracy performance before and after deployment.

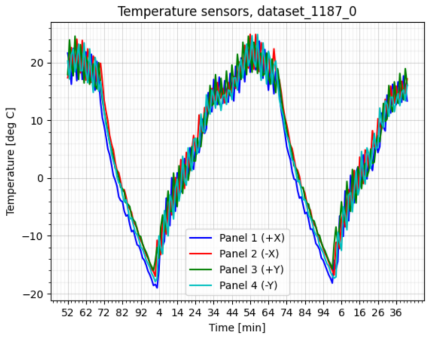
Parameter	Initial model	Deployed model	Difference
Accuracy on anomalies, A_a (before the threshold is applied)	0.9538	0.8934	-6.34%
Accuracy on NO anomalies, A_{na} (before the threshold is applied)	0.9891	0.9039	-8.61%
Recall (threshold 0.85)	0.9327	0.8588	-7.92%
Precision (threshold 0.85)	0.9905	0.8988	-9,26%
F1 (threshold 0.85)	0.9607	0.8784	-8.57%
FAR (threshold 0.85)	0.002	0.0213	+965%

As it can be observed, the accuracy performance noticeably decreases compared to the non-compressed network. While the performance on all metrics except one decreases by roughly 8%, the precision is actually affected more than recall. This means that the model becomes more prone to generating false alarms. This indicates that the output of the model becomes more noisy and some individual spikes of false positive outputs are produced even on anomaly-free data. This is well demonstrated by the False Alarm Rate, that increases almost 10 times.

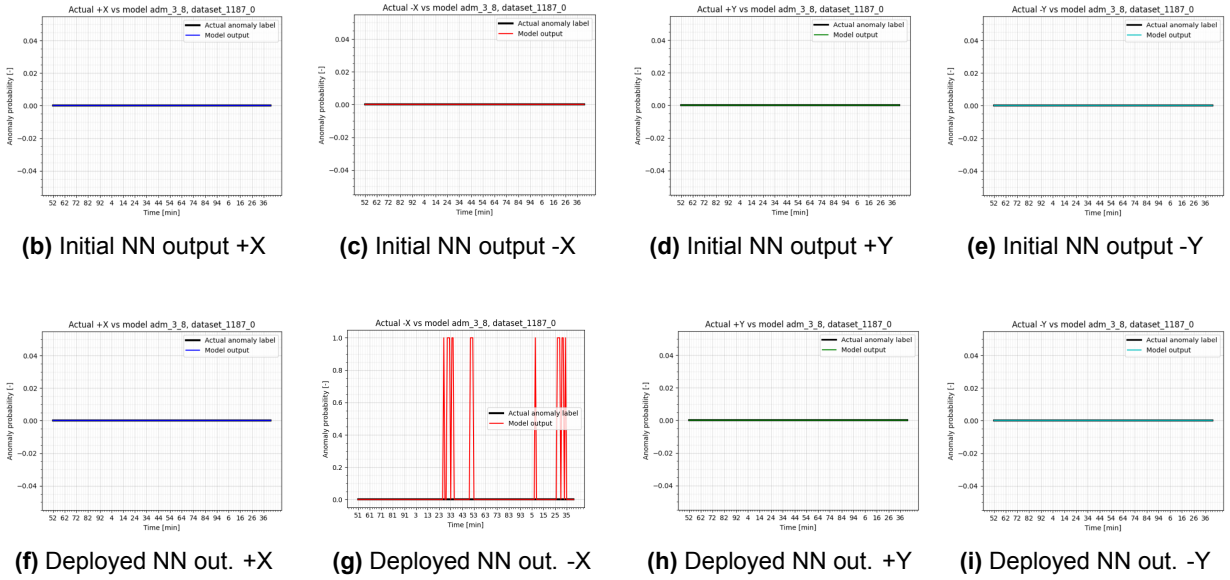
Overall, the values of recall and precision remain quite high: when anomalies are present, the model is able to detect them in 85.9% of cases, and 89.9% of all detected anomalies are true positives. However, in order to make a judgement on whether these results are satisfactory, the model outputs stored in the test report are studied for different anomalies. The different anomaly cases will now be explored. For each anomaly case, again the results of the model before and after deployment will be compared.

Anomaly-free data

First, both the non-compressed TensorFlow model and the compressed TF Lite model deployed on the MSP are tested on anomaly-free data, shown in Figure 7.1 (a). Each of the four outputs of the model should be 0 for the entire duration of the telemetry series. Figures 7.1 (b) - (e) show the outputs of the initial non-compressed model, while Figures 7.1 (f) - (i) show the outputs of the deployed TF Lite model.



(a) Input data



(b) Initial NN output +X **(c)** Initial NN output -X **(d)** Initial NN output +Y **(e)** Initial NN output -Y

(f) Deployed NN out. +X **(g)** Deployed NN out. -X **(h)** Deployed NN out. +Y **(i)** Deployed NN out. -Y

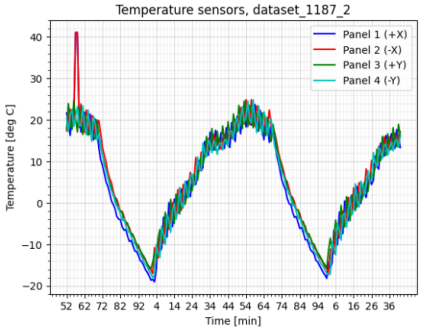
Figure 7.1: Model outputs on dataset_1187_0 (anomaly-free)

By analysing the figures, it can be concluded that the initial model performs entirely as expected, while the compressed model produces a number of false alarm spikes on panel -X, which cannot be attributed to any specific patterns in the -X sensor readings that could produce such noisy output. Although the exact reason cannot be established because of the black-box nature of the neural network, it can nonetheless be concluded that the false alarms result from the compression of the model, during which some of the relatively important weights of neuron connections were pruned, and the model output became less stable for panel -X. Although this behaviour is not dangerous for the system operation, as no real anomalies are present, this can still lead to certain limitations of the system's functionality, which will be demonstrated by the following anomalous data example.

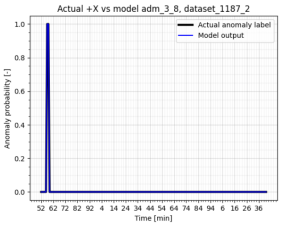
Detection of outliers and flat regions

Next, the model is tested on the variations of the original dataset which include outlier- and flat region anomalies on different (combinations of) panels. As it would not be feasible to discuss the results of all of these tests, only one example per anomaly type will be provided.

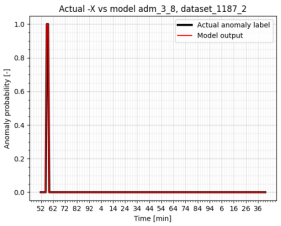
In this example, single-point outlier anomalies are present on two different panels at the same time (+X and -X). Figure 7.2 (a) shows the data with the two outliers. Figures 7.2 (b) - (i) show the corresponding outputs of the model before and after deployment on the microcontroller.



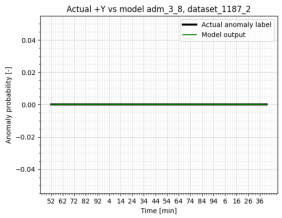
(a) Input data



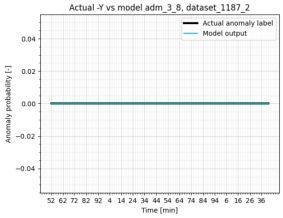
(b) Initial NN output +X



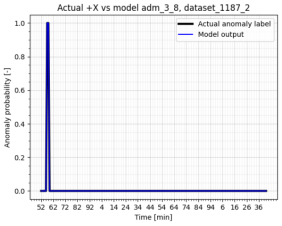
(c) Initial NN output -X



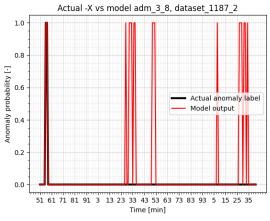
(d) Initial NN output +Y



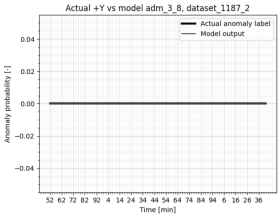
(e) Initial NN output -Y



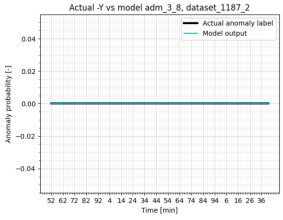
(f) Deployed NN out. +X



(g) Deployed NN out. -X



(h) Deployed NN out. +Y



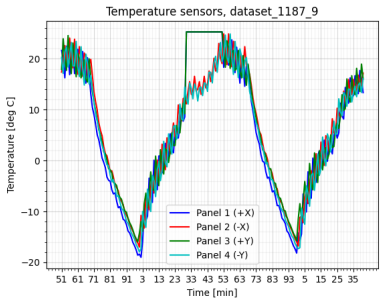
(i) Deployed NN out. -Y

Figure 7.2: Model outputs on dataset_1187_2 (2 outliers)

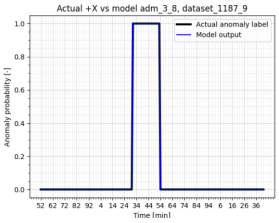
It can be seen that both the initial and the deployed models detect the outlier anomaly correctly and without any time delay (detection speed is limited to data transfer and inference speed), but the false alarms at panel -X persist and can be easily mistaken for anomalies similar to the one that is actually present in this test. In a real-life scenario, this would mean that the satellite FDIR system would not be able to distinguish between an actual outlier anomaly and a false positive output spike produced by the model. Two possible solutions to this problem that do not require model re-training are identified:

1. All individual spikes could be ignored and only the positive outputs that last for more than 2 points would be taken into account. The downside of this approach is that the model would not be able to detect any individual outlier anomalies.
2. The specific false alarm spikes produced for panel -X could be identified and filtered out (the system would ignore only these particular spikes). This approach would probably be unfeasible in a long run, when the conditions change and the model would start producing false alarm spikes at different locations.

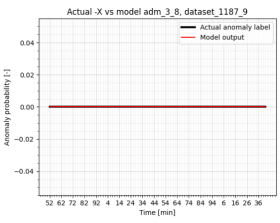
In the following example of flat region anomaly, there are flat regions on panels +X and +Y at the same time, as can be seen in Figure 7.3 (a). As can be seen, the two flat regions are at the same temperature value and have the same duration. Figures 7.3 (b) - (i) show the corresponding model outputs before and after deployment on the microcontroller.



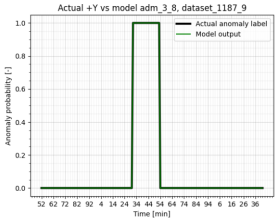
(a) Input data



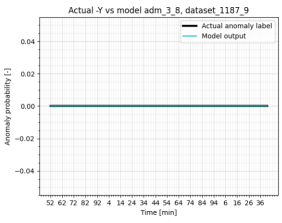
(b) Initial NN output +X



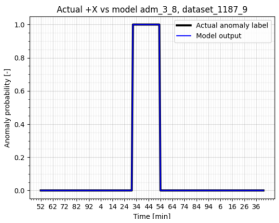
(c) Initial NN output -X



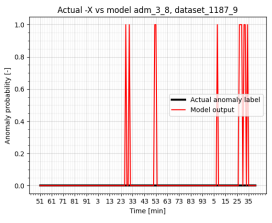
(d) Initial NN output +Y



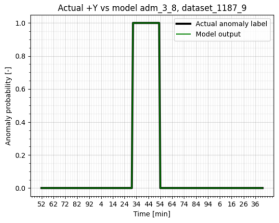
(e) Initial NN output -Y



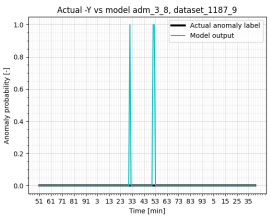
(f) Deployed NN out. +X



(g) Deployed NN out. -X



(h) Deployed NN out. +Y



(i) Deployed NN out. -Y

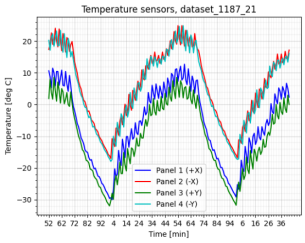
Figure 7.3: Model outputs on dataset_1187_9 (2 flat regions)

As before, both the initial and the compressed models detect the flat regions correctly with no time delay. However, besides the fact that false alarms still persist on panel -X, there are now also some false alarms on panel -Y. This illustrates the fact that due to certain connections getting pruned, the model becomes more sensitive to certain inputs and can sometimes produce more false alarms than the original model. Nonetheless, despite the false alarms, the anomalies that are actually there, get detected as they should. Therefore, the FDIR procedures that might be initiated in response to the model outputs, are actually necessary and justified, even though some of the sensors might not actually be affected by the anomalies.

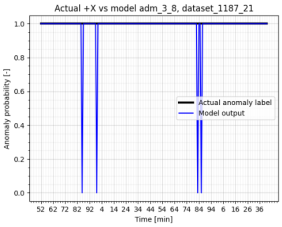
In this example, the false alarm spikes could be removed by using one of the approaches previously suggested. This is possible as all the false alarms manifest in the form of individual spikes, and can be easily distinguished from the positive outputs corresponding to the flat region anomalies.

Detection of permanent bias

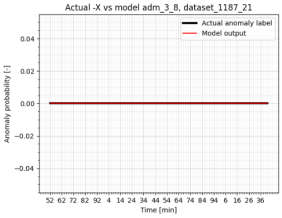
Figure 7.4 (a) shows an example of a permanent bias anomaly from the final testing. In this example, a permanent bias of about -15°C is introduced into the readings of sensors +X and +Y. The corresponding model outputs can be found in Figures 7.4 (b) - (i).



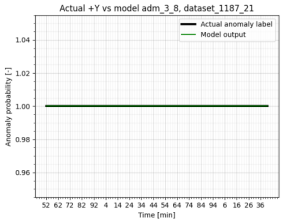
(a) Input data



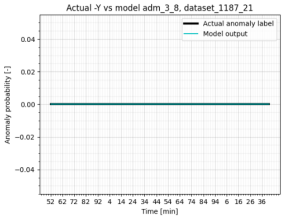
(b) Initial NN output +X



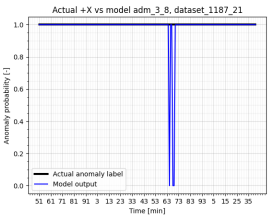
(c) Initial NN output -X



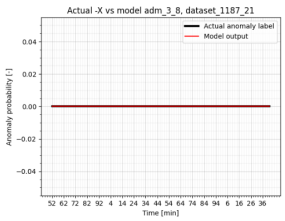
(d) Initial NN output +Y



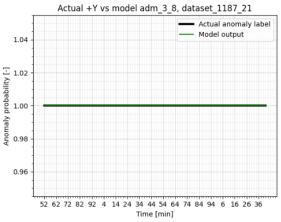
(e) Initial NN output -Y



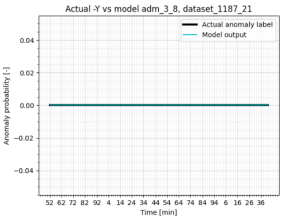
(f) Deployed NN out. +X



(g) Deployed NN out. -X



(h) Deployed NN out. +Y



(i) Deployed NN out. -Y

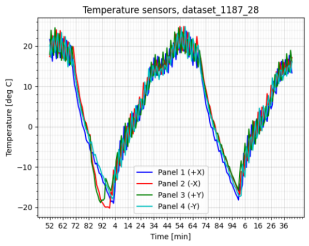
Figure 7.4: Model outputs on dataset_1187_21 (2 sensors with permanent bias)

The original model detects the bias on the two panels that actually have it with no time delay, while it can be seen that there are some false negative outputs for the +X panel. These false negatives are not critical for the satellite operations, as the actual anomaly is detected for the most part before and after these individual false negative outputs occur.

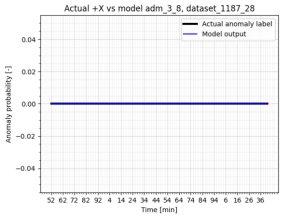
The compressed model demonstrates roughly the same accuracy with no detection time delay, and in certain parts performs even better than the original model. On panel +X there are even less false negative outputs than the original model produces. This can be due to the fact that certain connections that made the original model more sensitive in certain situations, have also been removed. As a consequence, the deployed model can sometimes actually be more stable than the original one.

Detection of temporary bias

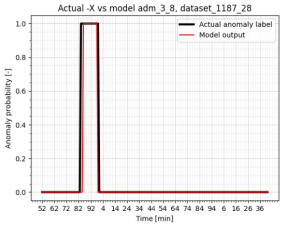
The performance of the neural network on data with a temporary bias anomaly is demonstrated by the example shown in Figure 7.5. In the example, the temporary bias occurs on panels -X and +Y around a local temperature minimum and has a peak magnitude of about -10 degrees. The outputs of the TensorFlow and TF Lite models are plotted in 7.5 (b) - (i):



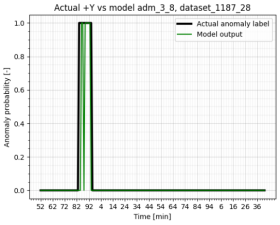
(a) Input data



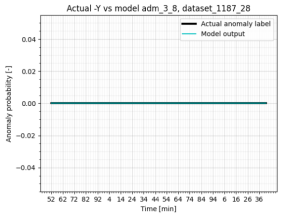
(b) Initial NN output +X



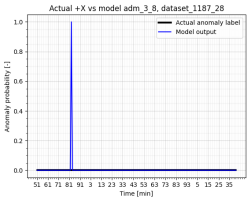
(c) Initial NN output -X



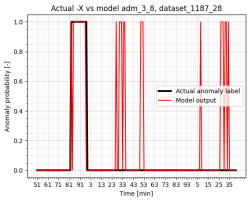
(d) Initial NN output +Y



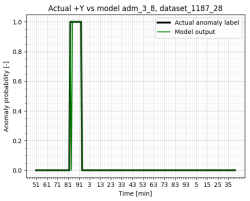
(e) Initial NN output -Y



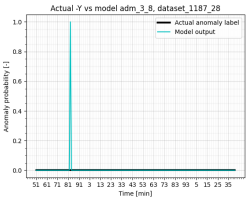
(f) Deployed NN out. +X



(g) Deployed NN out. -X



(h) Deployed NN out. +Y



(i) Deployed NN out. -Y

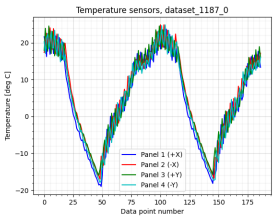
Figure 7.5: Model outputs on dataset_1187_28 (2 sensors with temporary bias)

In this case, the outputs are again the same for the models before and after compression, except for the aforementioned false alarm spikes on panel -X and some more individual false alarms and false negative spikes that can be seen in the outputs for other panels. These individual false positives and false negatives, however, do not compromise the detection of the anomalies that are actually present. Using an approach as discussed previously, these spikes could simply be filtered out.

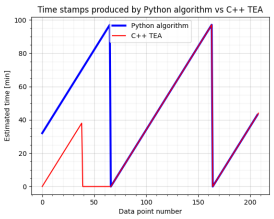
The false negative spikes in panels -X and +Y are actually associated with an anomaly detection delay that is caused by the nature of the temporary bias anomaly (it gradually appears and gradually disappears), which is actually a sign of the system operating as expected.

Performance with the Time Estimation Algorithm

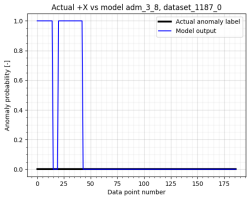
As it was discussed previously in Section 6.4, the time estimation algorithm needs some time to reach the temperature minimum of the orbit to correctly assign the reference point for the time. This results in an incorrect temperature estimate up until the point of the temperature minimum. The effect that this incorrect initial estimate of the time has on the accuracy of the anomaly detection is studied by running the model with anomaly-free temperature readings (dataset_1187_0) and the corresponding time estimates produced by TEA. The resulting model outputs can be seen in Figures 7.6 (c) - (f):



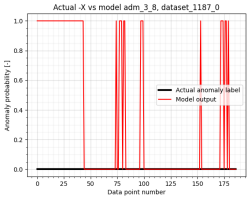
(a) Temperature input data



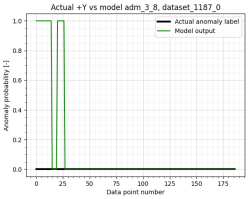
(b) Time input data (red line)



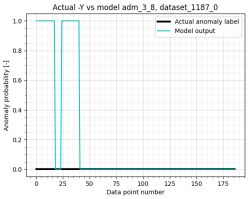
(c) Deployed NN out. +X



(d) Deployed NN out. -X



(e) Deployed NN out. +Y



(f) Deployed NN out. -Y

Figure 7.6: Deployed model performance with TEA on dataset_1187_0 (anomaly-free)

In this test, at the first inference, the time value is effectively unknown (assumed to be 0), and TEA tries to improve an estimate at every subsequent inference. The test results show that the initially incorrect time estimate indeed has a significant effect on the temperature detection. The network tends to produce false alarms on all four panels roughly until the data point 50, which corresponds to the temperature minimum of the orbit. Beyond this temperature minimum, the network starts producing the correct outputs again, except for panel -X false alarms, which have been discussed earlier.

Judgement on the anomaly detection performance
The testing of the final compressed model shows that it does detect anomalies when they occur. However, there is a number of limitations that are determined through the testing:

- More in-depth analysis of the final test report, which can be found in Appendix C, revealed that the model might sometimes not detect some of the anomalies when they occur at multiple panels at the same time. For instance, only 2 out of 3 or 3 out of 4 anomalous sensors are identified. This shows that the model has a limited ability to distinguish between sensors with- and without anomalies. This, however, is not critical for the satellite operations, as the model can identify at least one anomaly whenever they occur. The FDIR procedures can be performed on all panels at once when at least one of the sensor anomalies is detected.
- While detecting certain anomalies, some individual spikes of false negative outputs can be observed. These false negative outputs are not critical for the satellite operations either, as the actual anomalies are detected for the most part for the data points right before and after the false negatives occur.
- Another limitation revealed by testing is the fact that the outlier anomalies can be confused with false alarm spikes that are excessively produced by the compressed model. Several solutions are proposed which hypothetically can lead to less false alarms:
  - As one solution to this limitation, it is proposed that the individual spikes are simply not taken into account by the anomaly detection system. The individual outlier anomalies do not actually cause any damage to the satellite, rather they can simply be indicative of temporary faults in the sensor operations, which can be ignored or observed in a long term perspective to ensure that this behaviour does not result in more serious faults. The final neural network, however, is too sensitive to these types of anomalies to be used for their detection.
  - Adjustments can be made to the way these individual outliers are added to the training dataset, and the model could be re-trained to be less sensitive to individual outliers. For instance, only individual outliers with particularly high magnitude could be given the anomaly label ’1’, or there could be some smaller magnitude outliers with the anomaly label ’0’ assigned.
  - The false alarm problem could be associated with the way the temporary bias anomalies are introduced into the training dataset: currently the these anomalies are given the anomaly label ’1’ even at the data points that correspond to the anomaly just appearing. For these data points, the difference between the measured and the expected temperature has not yet surpassed the 10°C threshold, and thus it should not be referred to as an anomaly. The anomaly label allocation could therefore be adjusted.
  - The increase in the false alarm rate can be attributed to the model compression. Various solutions can be explored to re-train the compressed model without adding any new trainable parameters (weights and biases). This could also be done in the context of exploring the on-device neural network training solutions for space operations.
  - The model could also be re-trained with Gaussian (normally distributed) noise added on top of the temperature data, which would simulate the sensor noise and potentially make the output of the model less sensitive to small deviations in the input data, resulting in more stable outputs.

Overall, the model performs its functions, detects the anomalies when needed. False negatives, when they occur, do not last for long enough to put satellite subsystems at risk. False positives, when they occur, can be filtered out, as they generally exist in the form of individual spikes. Distinction between the panels that have anomalies and those that do not does not always work properly. Detection of individual outliers is complicated as they could be confused with false alarm spikes caused by model outputs’ instability. The speed of the response to anomalies is very high: in many cases anomalies are detected instantly, which is due to the short length of the input time series.

7.2.4. Power consumption

The measurements of the electrical power consumption of the microcontroller board were not conducted within this Thesis project, as it is known that the peak consumption of the onboard computer of Delfi-PQ is 7 mA current at 3.3 V (23.1 mW of power). This means that the OBC based on MSP432P4111 microcontroller can run any embedded software without exceeding the peak power consumption of 23.1 mW. At the same time, it is known that as of July 2023, the orbit average available power of Delfi-PQ is 1.2 W at the nominal

power consumption of the entire satellite of 200 mW. This means that at every orbit, Delfi-PQ has on average 1.0 W of power that can be used to run any additional software or use additional instruments.

The potential NN-based anomaly-detecting software would therefore not consume more than 2.31% of the available power onboard Delfi-PQ. It can thus be concluded that such application would definitely be feasible in terms of usage of the electrical power capabilities of the satellite.

7.3. Verification and validation of the results

Verification of the results has been performed throughout the Thesis project in accordance with the procedures specified in Section 3.4. The correct training, compression, deployment and testing of the model has been ensured by consistent use of unit tests integrated into the main code, which check the data sizes and the correct alignment of the data in each line of the training and testing datasets. The correct operation of the communication interface of the NN testing system is verified by using a system of confirmations on successful data reception once data is transferred to the microcontroller. Consistent hardware checks were also used to ensure correct wiring of the setup throughout testing.

In order to prove that the obtained results and conclusions are correct, the following validation procedures are applied:

- For accuracy calculations - a sensitivity analysis is performed on the output of the model to determine how well the model would respond to variations in the input data, that might be present in the data previously unseen by the model.
- For the measured inference time - the error of the time measurements is estimated analytically.
- For the results on memory consumption - the error of the measured RAM and flash memory usage is estimated analytically.

7.3.1. Sensitivity analysis on anomaly detection accuracy

A sensitivity analysis is applied in order to gain knowledge of how sensitive the model's outputs are to variations in the input parameters. The first part of the analysis is performed on the temperature sensor readings and the sensor derivatives inputs. The other part is focused on the effects of varying the time input of the model. In the sensitivity analysis, the input parameters are varied systematically by progressively increasing the magnitude of the variations. The corresponding model outputs are registered and then analysed.

Temperature sensor readings sensitivity

As one of the main sources of uncertainty in the problem is sensor noise, the sensitivity analysis is performed by modelling this sensor noise and then systematically varying its magnitude to determine the effect it has on the anomaly detection accuracy of the neural network. A common approach used in signal processing is to model the sensor noise as a Gaussian noise, which follows a normal distribution of random signal disturbances.

Introducing the Gaussian noise into the temperature sensor readings automatically affects the calculation of the sensor derivatives. This means that for every variation of the testing dataset and for every data point, once the noise is added on top of the original temperature data, the derivatives are re-calculated just before passing the data into the neural network.

In this sensitivity analysis, the standard deviation (σ) of the noise is initially varied with a 0.5°C increment and then the variation is increased to 1°C increment once it is confirmed that the model accuracy does not drop too rapidly. The results of the analysis are shown in Table 7.3.

Table 7.3: Sensitivity analysis on input temperature sensor noise (modelled as Gaussian noise).

	Noise standard deviation				
Parameter	**0 deg**	**0.5 deg**	**1 deg**	**2 deg**	**3 deg**
Recall (threshold 0.85)	0.8588	0.8434	0.8398	0.7933	0.7332
Precision (threshold 0.85)	0.8988	0.8619	0.7603	0.5385	0.455
F1 (threshold 0.85)	0.8784	0.8525	0.798	0.6416	0.5615
FAR (threshold 0.85)	0.0213	0.0298	0.0583	0.1497	0.1935

The analysis shows that by increasing the magnitude of the noise, more false alarms are generated and it becomes progressively more difficult to distinguish between true and false positive outputs. Precision drops more quickly than recall, indicating that the model becomes more prone to producing false alarms than not detecting anomalies. For 2 degrees standard deviation and more, the false alarms become so frequent, that often they no longer take the form of individual spikes but rather they are series of subsequent false positive outputs that can no longer be filtered out. This is illustrated by an example output shown in Figure 7.7:

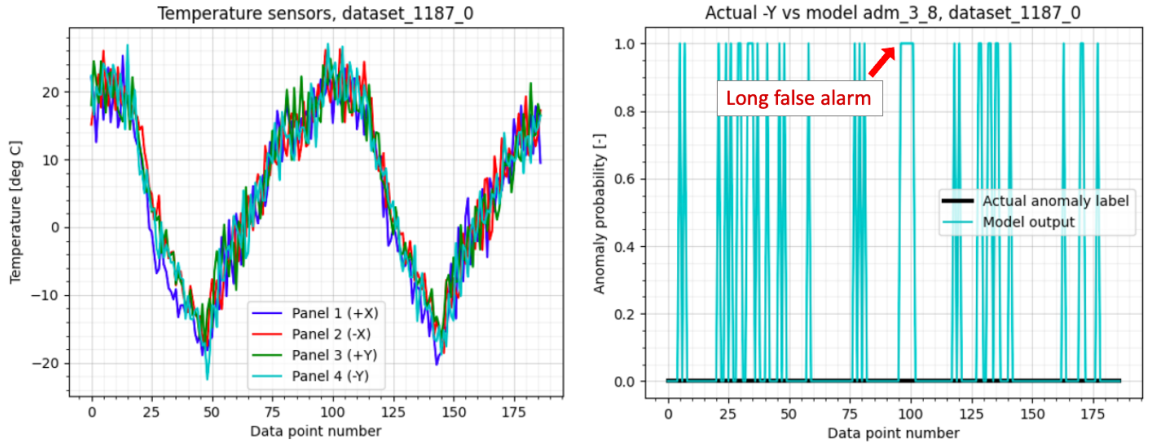


Figure 7.7: An output of the deployed model with σ = 2°C temperature sensor noise

At the same time, the detection of the actual anomalies is still accurate enough to correctly detect the anomalies when they are present. However, the fact that the true and false positives can no longer be distinguished from one another, potentially renders the use of the model unfeasible for a real-life application if the sensor noise exceeds 1°C standard deviation.

Clock readings sensitivity

As it was previously explained, the in-space application for Delfi-PQ would rely on the time estimate produced by an algorithm for time estimation. A potential architecture of such algorithm has been proposed in this Thesis. It is important to note that the operation of this algorithm is associated with an uncertainty of the time output. To better evaluate the effect that this uncertainty has on the model's performance, the uncertainty of the time input is modelled not as Gaussian noise but rather as a constant bias (error) added on top of the actual time estimate used as input.

In the following sensitivity analysis, the value of the time error is first varied with an increment of 30 seconds, which is later increased to 1 minute, once it is confirmed that larger variations are necessary. The results of the sensitivity analysis are shown in Table 7.4:

Table 7.4: Sensitivity analysis on input time error.

	Error of the estimated time										
Parameter	**-2 min.**	**-1 min.**	**-30 s**	**0 s**	**+30 s**	**+1 min.**	**+2 min.**	**+3 min.**	**+4 min.**	**+5 min.**	**+6 min.**
Recall (thr. 0.85)	0.8398	0.8526	0.8573	0.8588	0.8645	0.866	0.8631	0.8669	0.8627	0.8550	0.8398
Precision (thr. 0.85)	0.6426	0.7700	0.8567	0.8988	0.9152	0.9528	0.9722	0.9246	0.8660	0.8250	0.7494
F1 (thr. 0.85)	0.7281	0.8092	0.8570	0.8784	0.8891	0.9074	0.9144	0.8948	0.8643	0.8398	0.792
FAR (thr. 0.85)	0.1029	0.0561	0.0316	0.0213	0.0176	0.0094	0.0054	0.0156	0.0294	0.0399	0.0618

Adding a negative bias to the time values for all data points decreases the accuracy already for error values as small as 30 seconds. This affects both recall and precision, however, the effect on precision is more significant, which results in an increased false alarm rate. False alarm spikes become more frequent when further increasing the negative time error. For the error of -2 minutes and higher, the false alarm spikes start to combine into long false alarms that can no longer be distinguished from the true positive outputs for flat regions or bias anomalies. This can be seen in Figure 7.8:

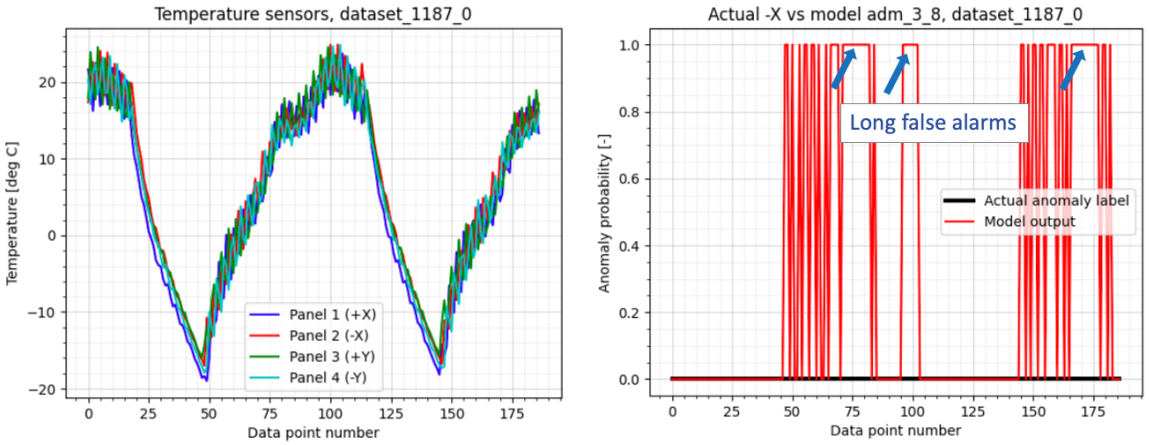


Figure 7.8: An output of the deployed model with -2 min. time input error

As long false alarms cannot be safely filtered out like false alarm spikes can, it is concluded that -1 minute deviation from the way the current time stamps are assigned, is the maximum time error that is safe for the operation of the system in its current state.

Adding a positive error, on the other hand, seems to improve the accuracy and reduce the false alarm rate for certain values of the error. The optimal performance is achieved for the bias of +2 minutes. Further increasing the bias, however, again results in a decreased performance with more and more false alarms. The state where individual false positive points begin combining into long false alarms for the time bias of +6 minutes and more. This can be seen in Figure 7.9:

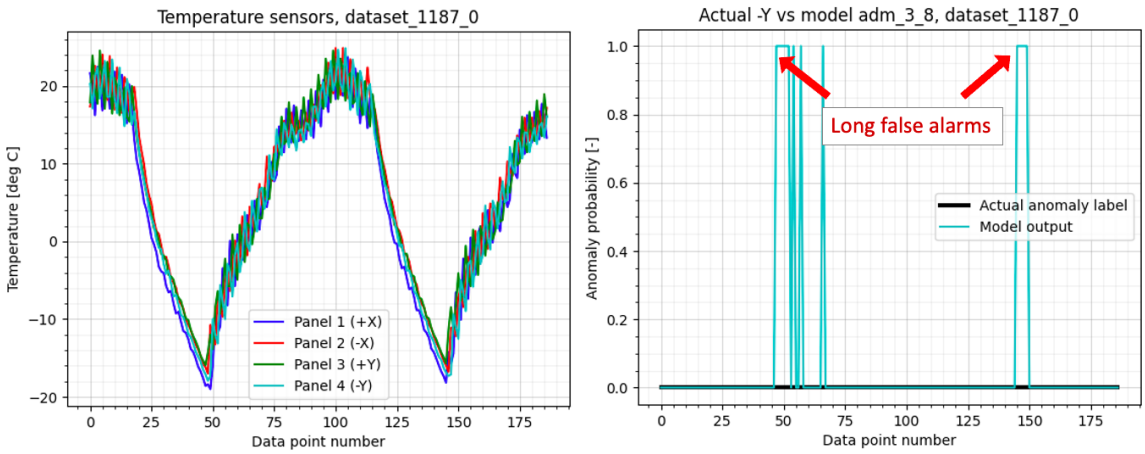


Figure 7.9: An output of the deployed model with +6 min. time input error

The results of the sensitivity analysis show that the anomaly detection accuracy of the neural network can actually be improved by introducing a 2 minute shift into the time inputs provided to the network. This would result in a noticeable decrease in the false alarm rate, which means that also the false alarm spikes in the panel -X output disappear. It is likely that during compression, the TF Lite converter also affected the way the time input is processed by the neural network, which effectively introduced a bias into the time input processing. In case a countering bias of +2 minutes would be added, the model would be able to function reasonably well with the uncertainty of the time input of up to +- 4 minutes.

Further improving the detection accuracy

The sensitivity analysis discussed so far has provided a good understanding of the model's robustness in terms of anomaly detection accuracy. It has also shown that the model outputs can be made even more stable by modifying the time inputs of the model. In this connection, some other potential ways of further improving the model's robustness can be explored.

One such way would be to reduce the model's sensitivity to noisy inputs, by introducing the Gaussian noise also into the training dataset and re-training the model. This could potentially result in better precision and lower false alarm rate. To confirm this, the final model is re-trained several times with Gaussian noise introduced into the temperature sensor data with the sensor derivatives re-calculated accordingly. The resulting datasets are then composed of the original 'noise-free' data and a copy of this data with the same length, but with noise introduced. This allows not only to test the effects of noise, but also obtain a larger dataset, which may also be beneficial for improving the accuracy. A series of tests is conducted for the models re-trained with different magnitudes of the Gaussian noise using the same testing dataset as was used to obtain the results in Subsection 7.2.3. The corresponding detection accuracy performance is measured for each model. The results can be seen in Table 7.5:

Table 7.5: Results of re-training the model with Gaussian noise in the training data

Parameter	Training data noise standard deviation				
	0 deg (adm_3_8)	**0.5 deg (adm_3_11_05_n)**	**1 deg (adm_3_11_10_n)**	**1.5 deg (adm_3_11_15_n)**	**2 deg (adm_3_11_20_n)**
Recall (threshold 0.85)	0.8588	0.9017	0.8781	0.9087	0.7529
Precision (threshold 0.85)	0.8988	0.9624	0.9746	0.9347	0.9605
F1 (threshold 0.85)	0.8784	0.9311	0.9239	0.9215	0.8441
FAR (threshold 0.85)	0.0213	0.0078	0.005	0.014	0.0068

From the testing results it can be seen that with adding more noise, the false alarm rate decreases and the precision improves, which indicates that the model becomes less sensitive to small-sized deviations and the output becomes more stable. At the same time, also the recall increases for certain values of noise magnitude, which means that the model also becomes more sensitive to the actual anomalies. However,

when approaching $\sigma = 1.0°C$ noise, all the performance parameters seem to degrade. In order to find an optimal value of the noise magnitude, one could use the F1 metric, as it corresponds to the overall performance of the model by combining the recall and precision metrics. It can be seen that the F1 score is the highest when $\sigma = 0.5°C$ and decreases afterwards. Although more testing and fine-tuning has to be done to arrive at the optimal design of the dataset, it can still be concluded that introducing noise into the training dataset is actually beneficial and can notably improve the performance.

In order to further improve the performance of the network, also its performance on the very small-sized thermal deviations could be investigated. This would allow to assess how the model performs on the deviations of different magnitude to potentially come up with a more suitable definition of anomalous readings. Currently, the threshold that allows to differentiate between anomalous and anomaly-free data is set to 10°C, however, it could be increased or decreased in case that would benefit the overall system performance. In order to make the first steps in the exploration of this aspect of the model performance, the original final design (adm_3_8 network) is tested on a part of the data which is progressively deviating from the nominal behaviour. For this purpose, a permanent bias anomaly of differing magnitude is generated for one of the sensors as shown in Figure 7.10:

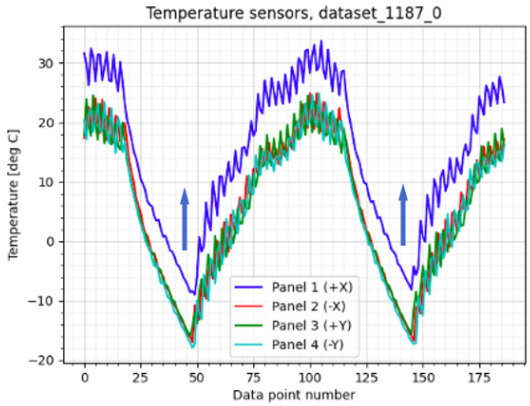


Figure 7.10: Progressively increasing permanent bias anomaly

The resulting outputs for the +X panel are registered for different deviation magnitudes as shown in Figure 7.11:

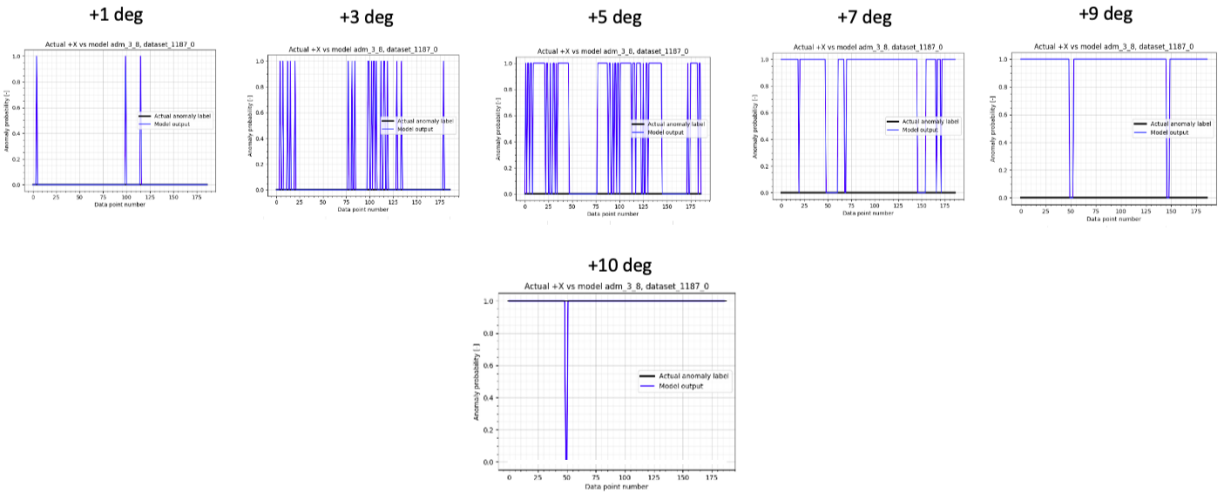


Figure 7.11: Model outputs for +X panel on a progressively increasing permanent bias anomaly

As it can be seen from the results, the system is highly sensitive even to small-sized anomalies, which leads to excessive false positive outputs. With the increasing temperature deviation, more positive outputs

are produced, and for the 5 degree deviation the model reaches roughly 50% accuracy. At 10°C, which is the lowest value for the deviation to be regarded as 'anomalous', the model produces only one false negative output. This indicates that the network was indeed tailored to detect 10°C and larger anomalies with high confidence. However, it is observed that the confidence level is already very high for 9°C and even 7°C deviations, which may suggest that for the future development of such models, a higher threshold for anomalies could be set (by modifying the training dataset) in order to avoid excessive production of false positive outputs for small deviations that are not yet anomalies. This would potentially lead to a higher precision and a reduced false alarm rate.

7.3.2. Error of the measured anomaly detection time

In order to minimize the error introduced by the clock uncertainty when the time per inference is measured, the time measurements are not taken after every single inference, but rather in the end of each dataset. In the final testing sequence each dataset consisted of 187 data entries. In this way, the clock error is distributed among 187 inferences, which allows for more adequate estimation.

As there is a total of 33 tests (1 anomaly-free dataset + 32 anomalous variations) in the testing sequence, the time per inference is measured 33 times and averaged to produce an even more accurate estimation. The testing revealed that the time measurements performed for each test typically vary by up to +- 0.003 seconds. It is thus concluded that the error of the final estimate for the inference time is accurate within $9.1 \cdot 10^{-5}$ seconds. Although, for the clarity of the results, the final estimate of the time per inference (0.119 s) is rounded to 3 significant figures.

7.3.3. Memory usage analysis

Although the memory allocation estimates produced by the TI Code Composer Studio memory allocation monitor are considered to be reasonable for most applications, the exact error associated with the memory estimation is difficult to evaluate. However, as the resulting estimate of the RAM footprint is close to the maximum capacity available on the board and the model deployment has been successfully proven, it can be concluded that the error of the RAM footprint estimate does not exceed 4.8 kB. To test the accuracy of the flash memory usage, an additional test is conducted, in which the model is not sent through the testing system interface, but pre-uploaded to the flash memory together with the firmware. The resulting flash memory usage of the system using the *AllOpsResolver* adds up to 260368 B, which comes close to the maximum capacity of 262k B. Still, the model operates correctly and all the required functionality is preserved. It is thus concluded that the estimate error for the flash memory usage does not exceed 1.6 kB.

7.4. Requirements compliance

Once the results are obtained and the V&V procedures are conducted, it can be determined whether the results meet the system requirements defined in Chapter 3.

The main functional requirements related to the overall performance and functionality of the system are satisfied, with comments/clarifications added to some of the requirements. In general, the system is able to detect anomalies of more than 10°C on all 4 temperature sensors with at least one anomaly being present in one orbital period of the satellite. Therefore, requirements SW-PERF-01-A, SW-PERF-01-C and SW-PERF-01-D are met. The measured time per inference is 0.119 s, which fits well within the requirement of 15 s maximum inference time. This requirement, SW-PERF-01-B, is therefore also satisfied.

Certain limitations of the resulting NN system, such as the limited ability to distinguish between anomalous and anomaly-free panel sensors in case of multiple anomalies and the difficulties with identifying individual outlier anomalies, result in requirements SW-PERF-01 and SW-PERF-01-E being fulfilled only partially.

When it comes to the requirements on the memory usage, the testing has shown that the resulting system does not use more than 92.5% of the available RAM and 89% of the available flash. The requirements on the memory consumption, SW-MEM-01 and SW-MEM-02 are thus also satisfied.

An overview of other requirements and the judgement on the system's compliance to these requirements can be found in the compliance matrix presented in Table 7.6:

Table 7.6: System requirements compliance matrix

Category	ID	Description	Requirement met?	Reasoning
Neural network performance	SW-PERF-01	The neural network shall be able to detect all anomaly types that are identified.	Yes, with clarification	NN detects the identified set of anomalies. However, the detection of individual outliers is complicated due to frequent false alarm spikes. This should be further explored.
	SW-PERF-01-A	The neural network shall detect any sensor reading that deviates by more than 10 degrees from the expected value.	Yes	Testing dataset includes anomalies of the magnitude of 10 degrees and higher.
	SW-PERF-01-B	The neural network shall be able to perform 1 inference in less than 15 seconds.	Yes	Observed inference time including input tensor initialization is in the order of 0.1 seconds.
	SW-PERF-01-C	The neural network shall be designed to handle at least 4 temperature sensors and be scalable to handle more in the future.	Yes	4 sensor channels have been implemented in the final model. Scalability is ensured in the code structure.
	SW-PERF-01-D	The neural network shall be able to detect at least 1 anomaly every 93.7 minutes.	Yes	The final model only takes 3 minutes (3 data points) to escape the influence of any previous anomalies and detect new ones.
	SW-PERF-01-E	The neural network shall be able to specify the sensors which indicate an anomaly and those that do not.	Yes, with clarification	The model does specify the anomalous panels, but the accuracy of distinction between panels is to be further explored.
Model deployment	SW-DEPL-01	The software shall be compatible with TI MSP432P401R microcontroller.	Yes	Proven by successful deployment of the final network on the board.
	SW-DEPL-02	The software shall be developed in C/C++ language (to be compatible with other Cortex-M based microcontrollers).	Yes	Proven by successful deployment of the final network on the board.
Memory management	SW-MEM-01	The software shall take up less than 256 kB of flash memory storage space.	Yes	Proven by successful deployment of the final network on the board.
	SW-MEM-02	The software shall use less than 64 kB of RAM in any operation mode.	Yes	Proven by successful deployment of the final network on the board.
Safety	SW-SAF-01	The software shall not cause harm to the satellite or other satellites.	Yes	Potential risks for the satellite or other satellites are not identified.
	SW-SAF-01-A	The designed software shall not compromise the operation of other embedded software.	Yes	Potential risks for the flight software are not identified.
	SW-SAF-01-B	The software shall not compromise the working of other satellite subsystems.	Yes	Potential risks for the satellite subsystems are not identified.
	SW-SAF-01-C	The software shall not cause unintended reboot of the microcontroller it is deployed on.	Yes	No reboot of the microcontroller was registered during testing.
Maintainability	SW-MNT-01	The software shall be easy to update, modify, and troubleshoot as needed.	Yes	Ensured by an inspection of the code.
	SW-MNT-01-A	The code shall be easily readable (properly structured and commented).	Yes	Ensured by an inspection of the code.

8

Conclusion

8.1. Closing Remarks

The focus of this Thesis project was on automation of satellite operations using novel computational techniques, in particular, Deep Learning (DL). The automation of satellite operations using DL shows promising results in terms of reducing operation cost, increasing the speed of response to certain events onboard the spacecraft, and reducing the data transfer between the satellite and the ground station. Given a small size of the neural network, the slow dynamics of the problem to be solved and various optimizations applied, a deep neural network can be deployed on low-power devices and potentially add value in the field of satellite operations. For this research project, automated temperature monitoring is chosen as one of such slow dynamics problems.

The neural network-based solution designed during the project is meant to detect anomalies in the temperature readings received from four temperature sensors of Delfi-PQ CubeSat. The anomalies include individual outliers, flat regions (constant temperature), temporary- and permanent temperature bias anomalies. A system for the deployment of small-sized neural networks on microcontrollers was designed by creating a communication interface between the main development computer and an MSP432P401R microcontroller board, which is similar to the microcontroller used as the onboard computer of Delfi-PQ. Then, a neural network for temperature anomaly detection was designed based on the system-level considerations such as the available computational- and memory resources, the sample frequency of the temperature measurements, the orbital characteristics, the range of the allowable temperature, and others. Among various possible NN architectures, the Multi-layer Perceptron architecture was selected. Tensorflow Lite micro C++ library has been chosen as the framework for the deployment of such small neural networks on the available microcontroller.

The design process based on trade-offs and iterative optimization, resulted in a design of an MLP network with 17 input layer neurons, 4 output layer neurons, and 3 hidden layers with 96 neurons in each off these hidden layers. The four outputs of the neural network correspond to the probabilities of anomalies occurring on each of the four panels at the time of the network inference. The input of the model consists of 3 latest temperature measurements of each of the four sensors, the 4 derivatives of these sensor readings and a time estimate produced by an external time estimation algorithm. The Rectified Linear Unit (ReLU) activation function is chosen for all hidden layers and the Linear activation is applied to the output layer neurons. This model is trained using TensorFlow library for Python, and then compressed to the format compatible with TF Lite micro using the so-called TF Lite Converter. After an optimization for size with the converter, the resulting model has the size of 24368 bytes.

The resulting model anomaly detection performance is measured using four different metrics for anomaly detection accuracy. The results of testing the model on a dataset that it has not previously seen are as follows: recall = 0.8588, precision = 0.8988, F1-score = 0.8784, False Alarm Rate (FAR) = 0.0213. The analysis of the resulting test reports shows that the model performs its functions, detects the anomalies when needed. False negatives, when they occur, do not last for long enough to put satellite subsystems at risk. False positives, when they occur, can be filtered out, as they generally exist in the form of individual spikes. Distinction between the panels that have anomalies and those that do not does not always work properly. Detection of individual outliers is complicated as they could be confused with false alarm spikes caused by model outputs' instability. The sensitivity analysis of the model revealed that the model

performance could potentially be improved by introducing a +2 minute bias to the time estimate input. This possibility could be further explored in the future research. Additionally, it was demonstrated that the anomaly detection accuracy could be increased by introducing Gaussian noise to the training dataset and re-training the model. Finally, an option to improve the measured accuracy performance would be to increase the threshold that differentiates anomalous data from anomaly-free data, and is currently set to 10°C.

The speed of the response to anomalies is very high: in many cases anomalies are detected instantly, which is due to the short length of the input time series. The measured time taken by a single inference of the model is 0.119 seconds, of which 0.012 seconds accounts for the data transfer time. The estimated RAM footprint during the test is 60.6k B with TF Lite AllOpsResolver. The corresponding flash memory usage is 232k B. For a flight software application, the memory footprint could be reduced to 66.7k B of flash and 30.2k B of RAM or even less. All the requirements derived from the system-level considerations are met, however, for some requirements, clarifications are added based on the observed limitations of the final model.

Overall, the model shows promising results in detecting thermal anomalies at early stages, when they have not yet caused any faults or damage to the spacecraft subsystems. The developed NN testing system could potentially be used for further development and testing of similar small-sized neural networks to be deployed on low-power devices.

8.2. Answers to the research questions

Now when the results of the temperature monitoring neural network have been discussed, the research questions posed in Chapter 1 can be answered. For convenience, the questions are repeated below:

1. *How to deploy a deep neural network for temperature monitoring on a low-power OBC?*
    a) What frameworks/libraries for on-device ML exist?
    Answer: The libraries for ML on low-power devices (TinyML) include TF Lite, TF Lite Micro, CMSIS-NN, Neural Network on Microcontroller (NNoM), STM32Cube-AI and others.
    b) What framework is most suitable for the implementation of the desired NN architecture on the available board?
    Answer: TF Lite Micro is the optimal choice due to good model compression, extensive community support and available examples.
    c) How to set up a system to deploy and test neural networks on the available board?
    Answer: Such testing system has been developed during this Thesis project. It relies on the use of TF Lite Micro embedded C++ library, Python codes for data processing, and a UART communication interface enabled by Python *Serial* library and *UART* driver of Simplelink MSP432p401r SDK software package for the microcontroller MSP432P401R.
2. *What thermal anomalies should the system be able to detect?*
    a) What thermal anomalies are possible in CubeSats?
    Answer: Among the thermal anomalies that can realistically occur onboard CubeSats, the following four are identified: individual (single-point) outliers, flat regions (constant temperature), permanent temperature bias (temperature shift) and temporary bias anomalies.
    b) What thermal anomalies are critical for CubeSat operations?
    Answer: Flat regions, permanent and temporary bias anomalies can be indicative of potential faults in the thermal system, and require FDIR procedures. Individual outliers are not that critical as they are attributed to individual errors in the sensor read-out.
    c) What thermal anomalies can a small-sized neural network reasonably detect?
    Answer: The results show promising performance in detecting flat regions, permanent and temporary bias anomalies. The detection of individual outliers is complicated due to the presence of false alarm output spikes.
3. *What neural network architecture is best suited for the target application?*
    a) What NN architectures exist?
    Answer: There are numerous deep neural network architectures for different applications, however, the most commonly used architectures can generally be classified as either Feedforward

(FF) neural networks (Multi-layer Perceptrons (MLP), Convolutional Neural Networks CNN, Autoencoders etc.) and Recurrent (RNN) neural networks (classical RNN, Long Short-term Memory (LSTM) networks, etc.).

b) Which architectures are suitable for on-device implementation?
Answer: For an on-devise implementation of anomaly detection tasks for temperature monitoring, the most suitable options are: CNN, Autoencoders, MLP and LSTM networks.

c) How do these architecture compare?
Answer: Although the optimal performance could potentially be achieved by employing LSTM or Autoencoder architectures, the optimal choice for the initial exploration of the on-device anomaly detection functionality is MLP architecture. A MLP network allows for efficient development, analysis and scaling, which allows for efficient management of the available time resources.

d) How complex should the intended ML model be (in terms of number of nodes, layers and activation functions)?
Answer: The design trade-offs and an iterative optimization process resulted in the selection of an MLP network architecture with 17 input layer neurons, 4 output layer neurons, and 3 hidden layers with 96 neurons in each off these layers. The Rectified Linear Unit (ReLU) activation function is chosen for all hidden layers and the Linear activation is applied to the output layer neurons. After an optimization for size, the resulting model has the size of 24368 bytes.

4. *How such implementation compares to traditional thermal monitoring approaches in terms of performance?*

a) What is the thermal anomaly detection accuracy of such application?
Answer: The anomaly detection accuracy of the resulting neural network system is evaluated in terms of four accuracy metrics: recall = 0.8588, precision = 0.8988, F1-score = 0.8784, FAR (False Alarm Rate) = 0.0213. It should be noted that although the resulting model has certain limitations (e.g. in terms of the detection accuracy when multiple anomalies are present simultaneously), and the overall performance could be better (e.g. in terms of the number of false alarms produced), this NN-based approach could still be more beneficial than some of the traditional thermal monitoring approaches. For instance, the developed NN system can detect abnormal temperature readings at very early stages, when the magnitude of the temperature deviations is as low as 10°C. This would not be impossible with a simple threshold approach, that can detect anomalies only when the temperature starts exceeding the critical values.

b) What is the thermal anomaly detection latency of such application?
Answer: In most cases the developed neural network can detect the occurring anomalies either almost instantly (after a 0.119 second network inference) or within 1 or 2 minutes (at the second or the third inference).

c) What is the flash memory storage space consumption of such application?
Answer: When deployed in the microcontroller-based testing system, the flash memory usage is 230.5 kB. For an in-space application, this could be reduced to as little as 65.2 kB.

d) What is the RAM usage of such application?
Answer: When deployed in the microcontroller-based testing system, the RAM usage is 59.1 kB. For an in-space application, this could be reduced to as little as 29.5 kB.

8.3. Suggestions for future work

As the aim of this research project was mainly to lay the foundation in the exploration of the in-space application of deep learning on low-power devices, there are numerous research activities that could be carried out to further explore this topic. The following activities are suggested as the possibilities for further research:

- The possibility of deploying multiple neural networks on the same device could be explored. A neural network approach could be used to improve the performance of the Time Estimation Algorithm (TEA).
- LSTM, RNN or Autoencoder architectures for the anomaly-detecting neural network could be explored.
- The current method of anomaly label allocation in the training dataset is not ideal. For instance, the temporary bias anomaly, that starts by gradually diverging from the nominal trend, is always assigned

a ’1’ anomaly label in the training dataset, even when it just starts and its magnitude is so negligibly small that it should not be detected as an anomaly. This makes the model more sensitive to even the slightest deviations from the nominal trend, which results in an increased false alarm rate. An alternative method of assigning anomaly labels for training could be explored.

- The training dataset could be designed differently in order to train a network so that the sensor derivative inputs would be more important. It is suggested that this would improve the recall of the model, especially when detecting flat region anomalies.
- Re-training the model maybe required when working with telemetry from other time periods. Currently the network is trained on temperature readings which are similar in terms of magnitude and curve shape for all panels. This is related to the tumbling of the spacecraft, which makes all panels heat up in approximately the same manner. However, once the spacecraft attitude is more stable, the temperature readings become more separated in terms of magnitude. This may require making changes to the network architecture, as it would need to rely less on the difference between the values of the sensors, and more on the behaviour of each individual sensor. Switching to a RNN network architecture might be necessary.
- Advanced training techniques can be used to increase the overall model performance. For example, dynamic hyperparameter adjustments during training could be applied.
- More of FUNcube telemetry could be obtained to estimate model accuracy degradation with time. Then, the various solutions to the accuracy degradation problem could be analyzed.
- If the performance degradation is significant, on-device re-training of the neural network could be explored. This would allow to dynamically adjust the weights of the neural network, making it adapt to changing environmental conditions. On-device training is currently gaining popularity in the field of TinyML, and in-space applications of on-device training could be explored [4, 70, 71].
- A similar network could be trained on a more abundant telemetry set, e.g. based on a simulation model or from other satellites. e.g. when more of Delfi-PQ telemetry could be obtained.
- The embedded software with the NN-based anomaly detection system can be integrated into the flight software system of Delfi-PQ or a similar satellite. First, the neural network would need to be tailored for the specific operational conditions of the satellite by performing training on a dataset which is more representative of the specific satellite’s telemetry. Then, in-flight tests could be performed.
- The final model relies on the latest temperature measurements rather than on long-term temperature dynamics. This results in a faster response time, but makes the network more prone to generating false alarms when the general trend of the temperature profile changes over time. During training, this model learned a certain shape of the temperature profile, creating a ’mask’ for the temperature data, against which it compares the sensor readings, and if they do not match closely, the system classifies them as anomalies. Alternatively, a network could be trained not to learn the mask, but to learn to predict any long-term temperature behaviour, regardless of the general shape of the profile. This would, however, increase the size of the input tensor of the model and would likely significantly increase the anomaly detection latency.
- A physics-based model of the satellite could be made and combined with the neural network approach. This would ensure that the system can operate even when there are parts of telemetry data missing (which is common for Delfi-PQ). Another advantage of this approach is its ability to filter out certain sources of uncertainty, such as spacecraft attitude dynamics, solar cycles, thermal coupling between different nodes of the satellite, etc. Physics modelling would also increase the exploration ability of the system, which is highly limited for conventional neural networks given their black-box nature.

References

[1] James R. Wertz et al. “Methods for Achieving Dramatic Reductions in Space Mission Cost”. In: 2011.

[2] Massimo Tipaldi et al. “Reinforcement learning in spacecraft control applications: Advances, prospects, and challenges”. In: *Annual Reviews in Control* 54 (2022), pp. 1–23. DOI: https://doi.org/10.1016/j.arcontrol.2022.07.004. URL: https://www.sciencedirect.com/science/article/pii/S136757882200089X.

[3] SmartSat Adelaide Australia. *Machine Learning Onboard Satellites, SmartSat Technical Report no. 010*. 2021. URL: https://smartsatcrc.com/app/uploads/Technical-report-machine-learning-onboard-satellites-PUB-1.pdf.

[4] Norah N Alajlan et al. “TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications”. In: *Micromachines* 13.6 (2022), p. 851.

[5] Jafar Alzubi et al. “Machine learning from theory to algorithms: an overview”. In: *Journal of physics: conference series*. Vol. 1142. 1. IOP Publishing. 2018, p. 012012.

[6] Nitin Kumar Chauhan et al. “A review on conventional machine learning vs deep learning”. In: *2018 International conference on computing, power and communication technologies (GUCON)*. IEEE. 2018, pp. 347–352.

[7] Roberto Iriondo Patrik Shukla. *Main Types of Neural Networks and its Applications - Tutorial*. 2022. URL: https://towardsai.net/p/machine-learning/main-types-of-neural-networks-and-its-applications-tutorial-734480d7ec8e (visited on 11/20/2022).

[8] Towards Data Science. *Activation Functions in Neural Networks*. 2023. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (visited on 01/20/2023).

[9] Ian Goodfellow et al. *Deep learning*. MIT press, 2016.

[10] Yinghan Long et al. “Complexity-aware adaptive training and inference for edge-cloud distributed AI systems”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 573–583.

[11] Yuanyuan Sun et al. “Fault diagnosis for space utilisation”. In: *The Journal of Engineering* 2019.23 (2019), pp. 8770–8775.

[12] The Planetary Society. *The Planetary Exploration Budget Dataset*. 2022. URL: https://www.planetary.org/space-policy/planetary-exploration-budget-dataset (visited on 11/20/2022).

[13] Alejandro Sans Monguiló. “Study of a preliminary value-cost model of Earth Observation (EO) satellites operating in Very Low Earth Orbit (VLEO) and Low Earth Orbit (LEO) for an Ocean Cleanup Organisation.” B.S. thesis. Universitat Politècnica de Catalunya, 2019.

[14] J.R. Wertz W.J Larson. *Space Mission Analysis and Design (SMAD)*. third. Space Technology Library. Microcosm, Inc., 2005.

[15] Vivek Kothari et al. “The final frontier: Deep learning in space”. In: *Proceedings of the 21st international workshop on mobile computing systems and applications*. 2020, pp. 45–49.

[16] Zhenping Li. “A Machine Learning Solution for Satellite Health and Safety Monitoring”. In: 18 (Jan. 2022).

[17] Miguel Ángel Vázquez et al. “Machine learning for satellite communications operations”. In: *IEEE Communications Magazine* 59.2 (2021), pp. 22–27.

[18] WANG Haijiao et al. “Online scheduling of image satellites based on neural networks and deep reinforcement learning”. In: *Chinese Journal of Aeronautics* 32.4 (2019), pp. 1011–1019.

[19] Sara K Ibrahim et al. “Machine learning techniques for satellite fault diagnosis”. In: *Ain Shams Engineering Journal* 11.1 (2020), pp. 45–56.

[20] Gianluca Giuffrida et al. “The Φ-Sat-1 mission: the first on-board deep neural network demonstrator for satellite earth observation”. In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2021), pp. 1–14.

[21] Tianwei Yan et al. “Automatic Deployment of Convolutional Neural Networks on FPGA for Spaceborne Remote Sensing Application”. In: *Remote Sensing* 14.13 (2022), p. 3130.

[22] Emilio Rapuano et al. “An fpga-based hardware accelerator for cnns inference on board satellites: benchmarking with myriad 2-based solution for the cloudscout case study”. In: *Remote Sensing* 13.8 (2021), p. 1518.

[23] Georges Labrèche et al. “OPSSAT Spacecraft Autonomy with TensorFlow Lite, Unsupervised Learning, and Online Machine Learning”. In: *2022 IEEE Aerospace Conference*. 2022.

[24] R. P. Schaffer et al. “Cloud Filtering and Novelty Detection using Onboard Machine Learning for the EO-1 Spacecraft”. In: 2017.

[25] Steve Chien et al. “Onboard Autonomy on the Intelligent Payload EXperiment CubeSat Mission”. In: *Journal of Aerospace Information Systems* 14 (Apr. 2016), pp. 1–9. DOI: 10.2514/1.I010386.

[26] Clint Crosier. *AWS successfully runs AWS compute and machine learning services on an orbiting satellite in a first-of-its kind space experiment*. 2022. URL: https://aws.amazon.com/blogs/publicsector/aws-successfully-runs-aws-compute-machine-learning-services-orbiting-satellite-first-space-experiment/ (visited on 11/20/2022).

[27] Ji Lu et al. “A learning-based approach for agile satellite onboard scheduling”. In: *IEEE Access* 8 (2020), pp. 16941–16952.

[28] Chao Li et al. “Data-driven onboard scheduling for an autonomous observation satellite”. In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. Vol. 2018. AAAI. 2018, pp. 5773–5774.

[29] Maanasa Sachidanand. “AI-driven payload for Miniaturised spacecraft”. MSc thesis. University of Luxembourg, 2022.

[30] Zhuoheng Li. “Use Reinforcement Learning to Generate Testing Commands for Onboard Software of Small Satellites”. MSc thesis. Delft University of Technology, faculty of Aerospace Engineering, 2022. URL: http://resolver.tudelft.nl/uuid:94468343-104c-4964-87dd-8cb2dbe9e01f.

[31] Blesson Varghese et al. “Challenges and opportunities in edge computing”. In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE. 2016, pp. 20–26.

[32] Gianmarco Cerutti et al. “Compact recurrent neural networks for acoustic event detection on low-energy low-complexity platforms”. In: *IEEE Journal of Selected Topics in Signal Processing* 14.4 (2020), pp. 654–664.

[33] Lachit Dutta et al. “Tinyml meets iot: A comprehensive survey”. In: *Internet of Things* 16 (2021), p. 100461.

[34] Visal Rajapakse et al. “Intelligence at the Extreme Edge: A Survey on Reformable TinyML”. In: *arXiv preprint arXiv:2204.00827* (2022).

[35] Gianmarco Cerutti et al. “Convolutional neural network on embedded platform for people presence detection in low resolution thermal images”. In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 7610–7614.

[36] Puranjay Mohan et al. “A tiny CNN architecture for medical face mask detection for resource-constrained endpoints”. In: *Innovations in Electrical and Electronic Engineering*. Springer, 2021, pp. 657–670.

[37] Koen Goedemondt. "μLightDigit: A TinyML System for Contactless Digit Recognition using Ambient Light". MSc thesis. Delft University of Technology, 2022. URL: http://resolver.tudelft.nl/uuid:b8d71229-28ce-41e5-8e81-9fdcb939ef3a.

[38] Maria Francesca Alati et al. "Time series analysis for temperature forecasting using TinyML". In: *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE. 2022, pp. 691–694.

[39] Mohammed Zubair Mohammed Shamim. "TinyML Model for Classifying Hazardous Volatile Organic Compounds Using Low-Power Embedded Edge Sensors: Perfecting Factory 5.0 Using Edge AI". In: *IEEE Sensors Letters* 6.9 (2022), pp. 1–4.

[40] Justin Nguyen et al. "TrafficNNode: Low Power Vehicle Sensing Platform for Smart Cities". In: *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE. 2021, pp. 278–282.

[41] Marcus Venzke et al. "Artificial Neural Networks for Sensor Data Classification on Small Embedded Systems". In: *arXiv preprint arXiv:2012.08403* (2020).

[42] Massimo Pavan et al. "TinyML for UWB-radar based presence detection". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2022, pp. 1–8.

[43] Massimo Merenda et al. "Edge machine learning for ai-enabled iot devices: A review". In: *Sensors* 20.9 (2020), p. 2533.

[44] Gianluca Furano et al. "Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities". In: *IEEE Aerospace and Electronic Systems Magazine* 35.12 (2020), pp. 44–56.

[45] Gunter's Space Page. *Delfi-PocketQube*. 2022. URL: https://space.skyrocket.de/doc_sdat/delfi-pq.htm (visited on 11/20/2022).

[46] Silvana Radu et al. "Delfi-PQ: The first pocketqube of Delft University of Technology". In: *69th International Astronautical Congress, Bremen, Germany, IAC*. 2018.

[47] Zhuoheng Li. "Planning and Reinforcement Learning for Delfi-PQ". MSc thesis literature study. Delft University of Technology, faculty of Aerospace Engineering, 2021. URL: http://resolver.tudelft.nl/uuid:94468343-104c-4964-87dd-8cb2dbe9e01f.

[48] N2YO. *Delfi-PQ tracking*. 2023. URL: https://www.n2yo.com/satellite/?s=51074 (visited on 01/20/2023).

[49] Vincenzo Cinotti et al. "Delfi-PQ Thermal Model Verification". In: *AE4S10: Microsat Engineering Report*. TU Delft. 2022.

[50] Bradley J Erickson et al. "Toolkits and libraries for deep learning". In: *Journal of digital imaging* 30 (2017), pp. 400–405.

[51] Zhaobin Wang et al. "Various frameworks and libraries of machine learning and deep learning: a survey". In: *Archives of computational methods in engineering* (2019), pp. 1–24.

[52] Edge Impulse. *Edge Impulse Software Documentation*. 2022. URL: https://docs.edgeimpulse.com/docs/ (visited on 11/20/2022).

[53] Imagimob. *Imagimob Website*. 2023. URL: https://www.imagimob.com/ (visited on 11/20/2022).

[54] Pierre-Emmanuel Novac et al. "Quantization and Deployment of Deep Neural Networks on Microcontrollers". In: *Sensors* 21.9 (Apr. 2021), p. 2984. DOI: 10.3390/s21092984. URL: https://doi.org/10.3390%2Fs21092984.

[55] Ramon Sanchez-Iborra et al. "Tinyml-enabled frugal smart objects: Challenges and opportunities". In: *IEEE Circuits and Systems Magazine* 20.3 (2020), pp. 4–18.

[56] Tensorflow. *TensorFlow Lite for Microcontrollers*. 2023. URL: https://www.tensorflow.org/lite/microcontrollers (visited on 01/20/2023).

[57] Tensorflow. *TensorFlow Lite*. 2023. URL: https://www.tensorflow.org/lite/guide (visited on 01/20/2023).

[58] Texas Instruments. *MSP432P401R LaunchPad User's Guide*. 2018. URL: https://docs.rs-online.com/3934/A700000006811369.pdf (visited on 02/26/2023).

[59] TensorFlow. *TensorFlow Core Overview*. 2023. URL: https://www.tensorflow.org/overview (visited on 02/21/2023).

[60] Texas Instruments. *SimpleLink MSP432 Software Development Kit (SDK)*. 2023. URL: https://www.ti.com/tool/SIMPLELINK-MSP432-SDK (visited on 02/21/2023).

[61] Texas Instruments. *GCC ToolChain*. 2023. URL: https://software-dl.ti.com/processor-sdk-linux/esd/docs/06_03_00_106/linux/Overview/GCC_ToolChain.html (visited on 02/21/2023).

[62] AMSAT. *AO-73 (FUNcube-1)*. 2023. URL: https://www.amsat.org/two-way-satellites/ao-73-funcube-1/ (visited on 02/10/2023).

[63] N2YO. *FUNcube-1 (AO-73) tracking*. 2023. URL: https://www.n2yo.com/satellite/?s=39444 (visited on 02/20/2023).

[64] Ullas Bhat. "Neural-Network Based Thermal Modeling of Small Satellites". MSc thesis. Delft University of Technology, faculty of Aerospace Engineering, 2023. URL: http://resolver.tudelft.nl/uuid:b56b443a-3097-46b2-ac23-9116d15628bd.

[65] Texas Instruments. *TensorFlow Lite Micro Examples for TI devices*. 2023. URL: https://github.com/TexasInstruments/tensorflow-lite-micro-examples (visited on 02/25/2023).

[66] Python Documentation. *time — Time access and conversions*. 2023. URL: https://docs.python.org/3/library/time.html (visited on 02/21/2023).

[67] Tensorflow. *Conversion of TensorFlow models*. 2023. URL: https://www.tensorflow.org/lite/models/convert/convert_models (visited on 01/20/2023).

[68] Johannes Lederer. "Activation functions in artificial neural networks: A systematic overview". In: *arXiv preprint arXiv:2101.09957* (2021).

[69] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[70] Haoyu Ren et al. "Tinyol: Tinyml with online-learning on microcontrollers". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[71] Han Cai et al. "Tinytl: Reduce memory, not parameters for efficient on-device learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 11285–11297.

A

FUNcube-1 temperature telemetry used for neural network training

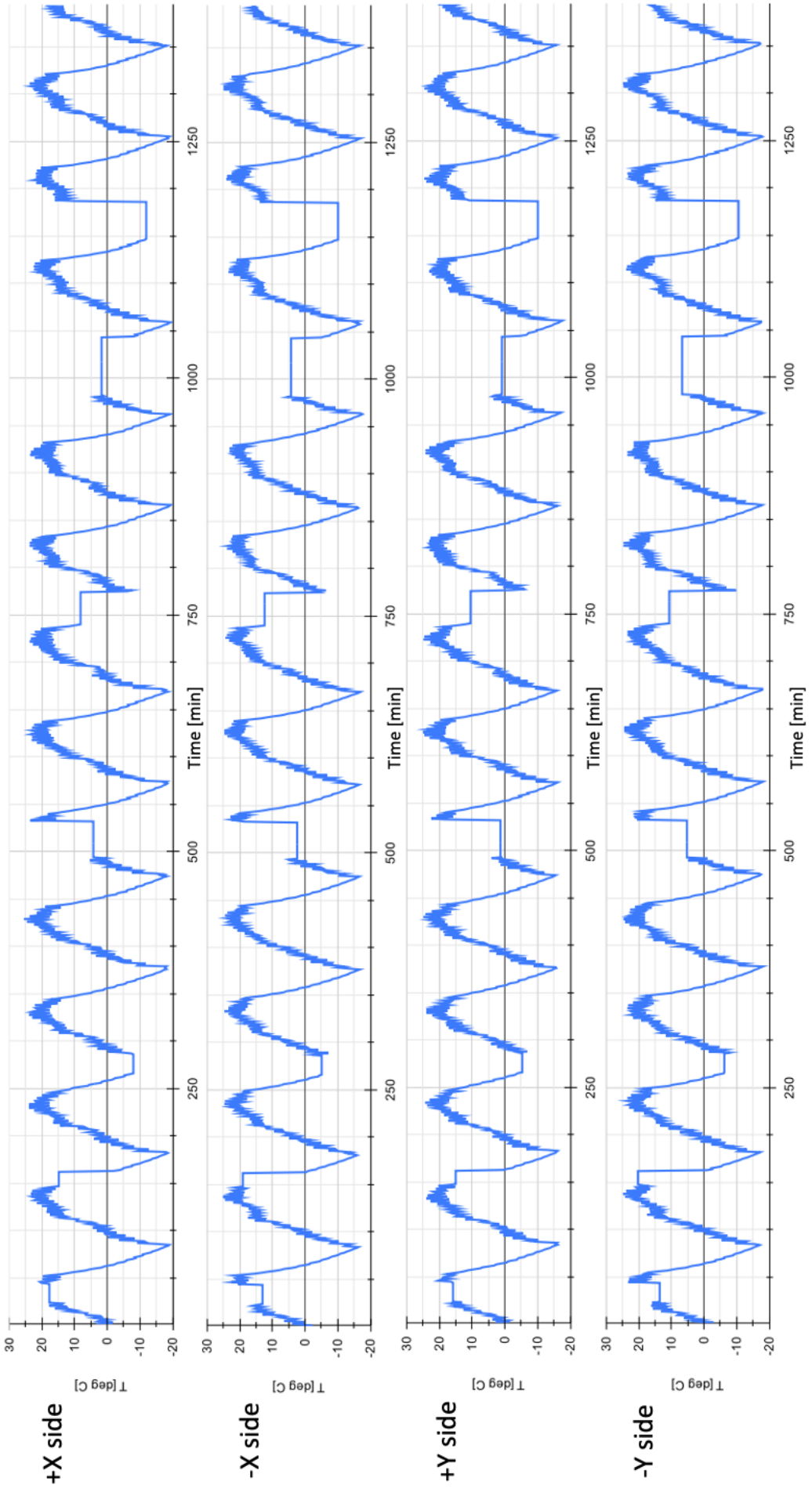


Figure A.1: FUNcube-1 temperature telemetry for +X, -X, +Y, -Y solar panels. Received on 04/02/2016

B

Python and C++ codes

All Python codes for NN creation, training and testing, as well as the C++ embedded codes including the compiled TF Lite micro projects, can be found at GitHub repository:

`https://github.com/rmaununen/Thesis_sat_ML`

C

TF Lite Micro test report of the final neural network deployed on MSP432P401R

TF Lite Micro test report 1.1

Test time: 02.07.2023

Model name: adm_3_8.h

Model type: MLP

Model description: 17x96x96x96x4 MLP to detect anomalies on 4 panels based on 4 x 10 temperature measurements + 4 derivatives + 1 time. Trained to detect sensor spikes, const regions, temp shifts and bumps. Tested on previously unseen data. Detection threshold = 0.85

Model data size: 24368 [Bytes]

Test results:

Accuracy on NO anomalies (0): N/A

Accuracy on ANOMALIES (1): N/A

Time per inference: 0.14 [s]

Total test time: 1023.097 [s]

recall: 0.8588

precision: 0.8988

F1 score: 0.8784

False alarm rate: 0.0213

Anomaly-free telemetry

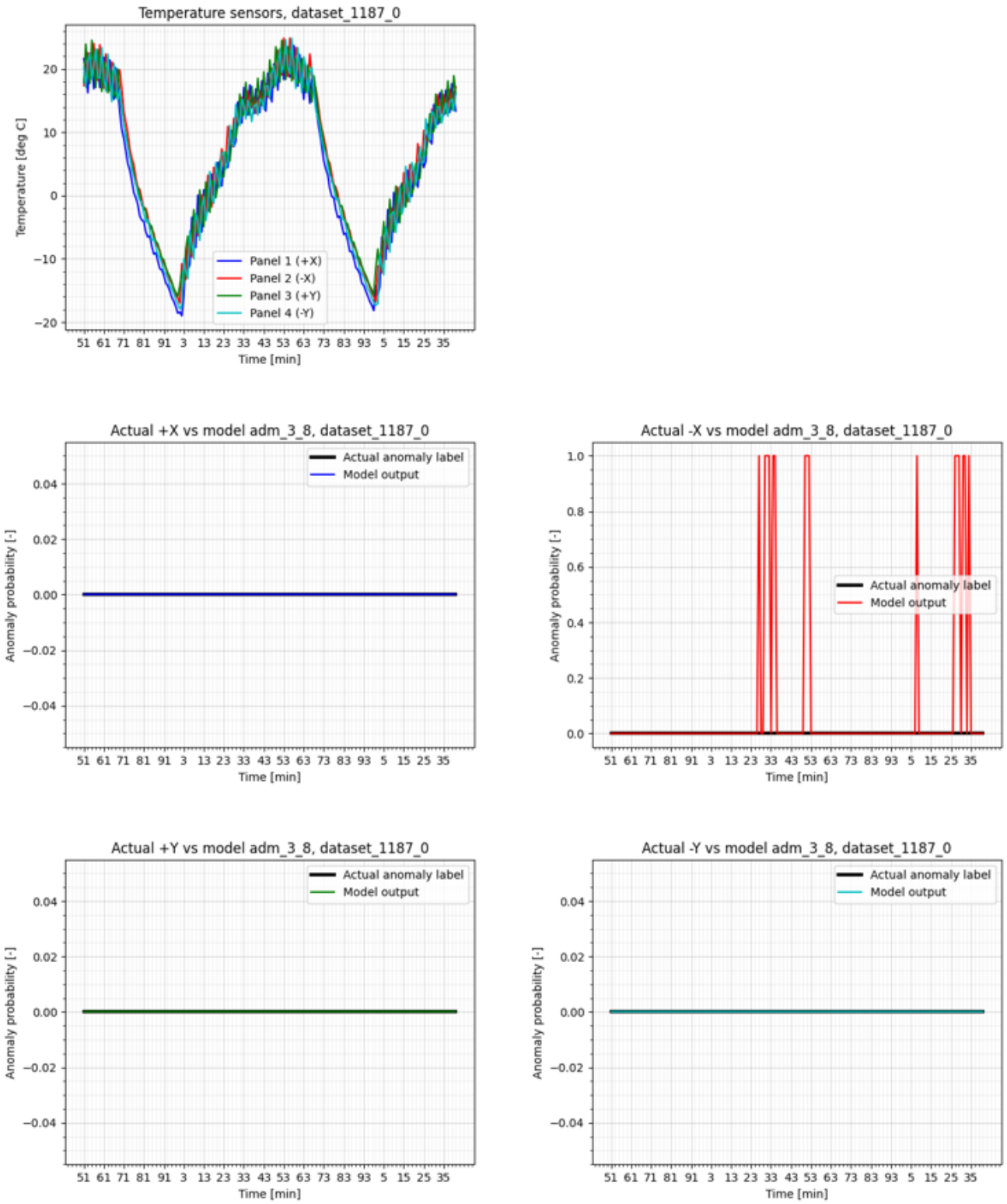


Figure 1: model inferences on dataset_1187_0

Outlier anomalies

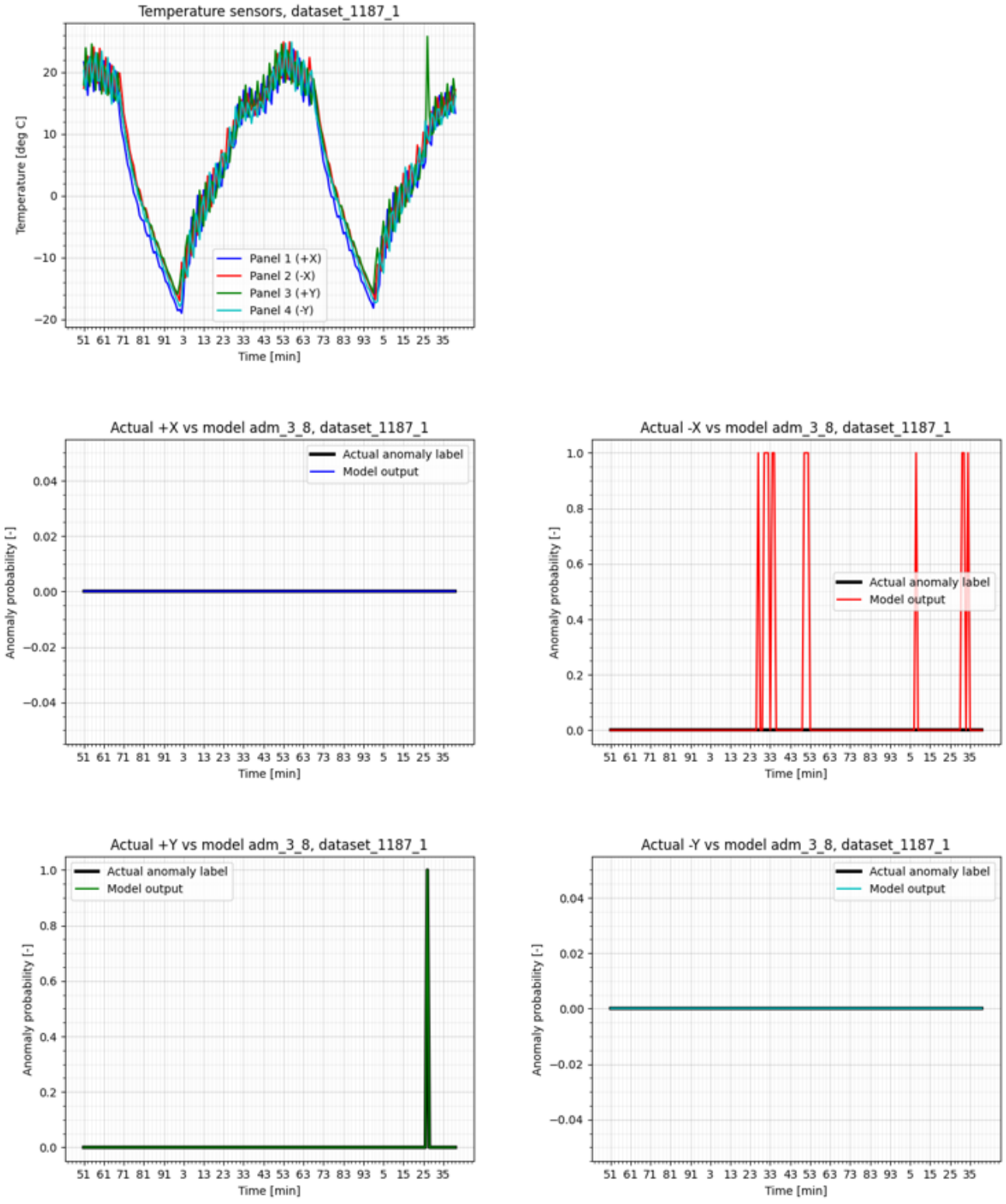


Figure 2: model inferences on dataset_1187_1

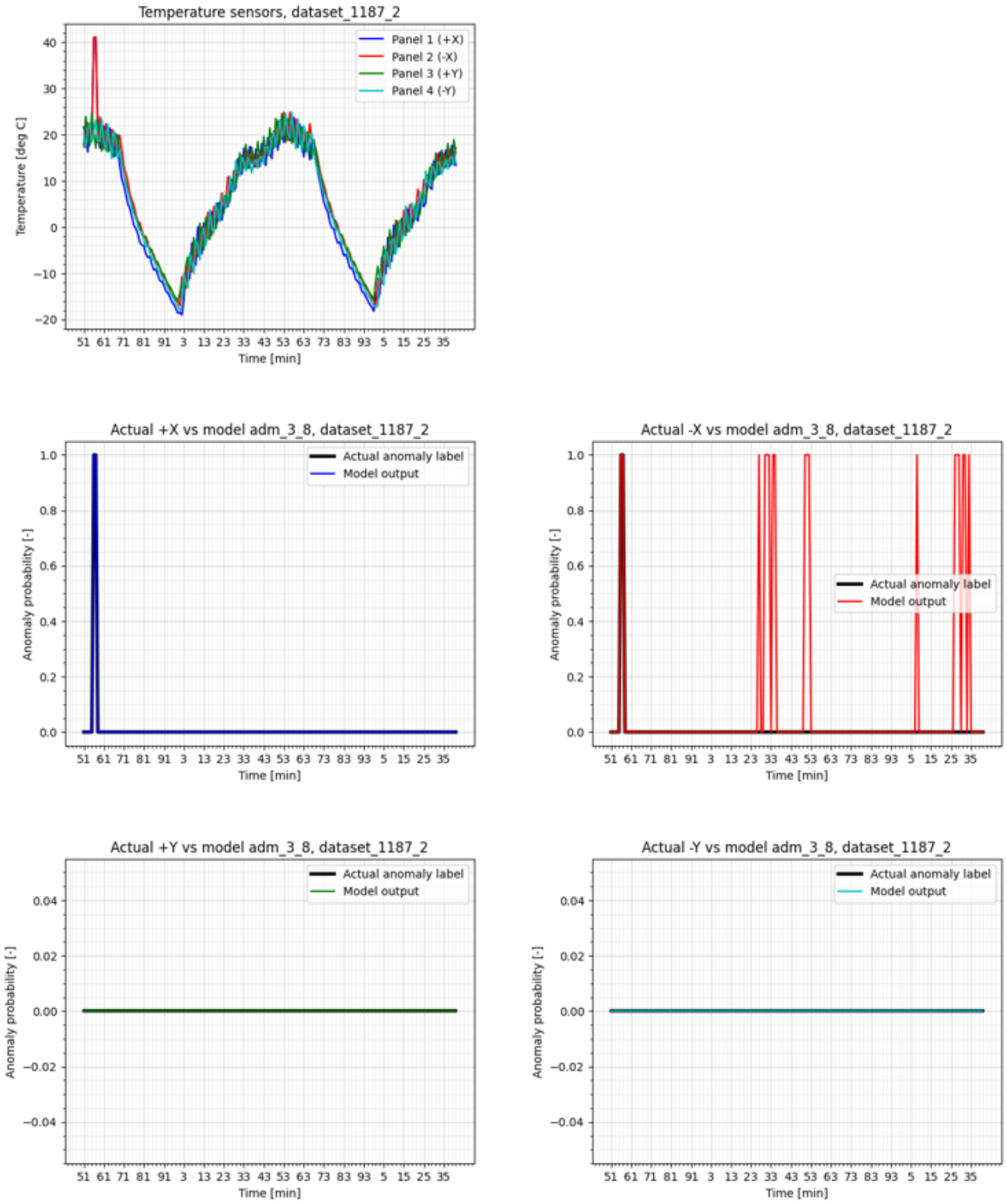


Figure 3: model inferences on dataset_1187_2

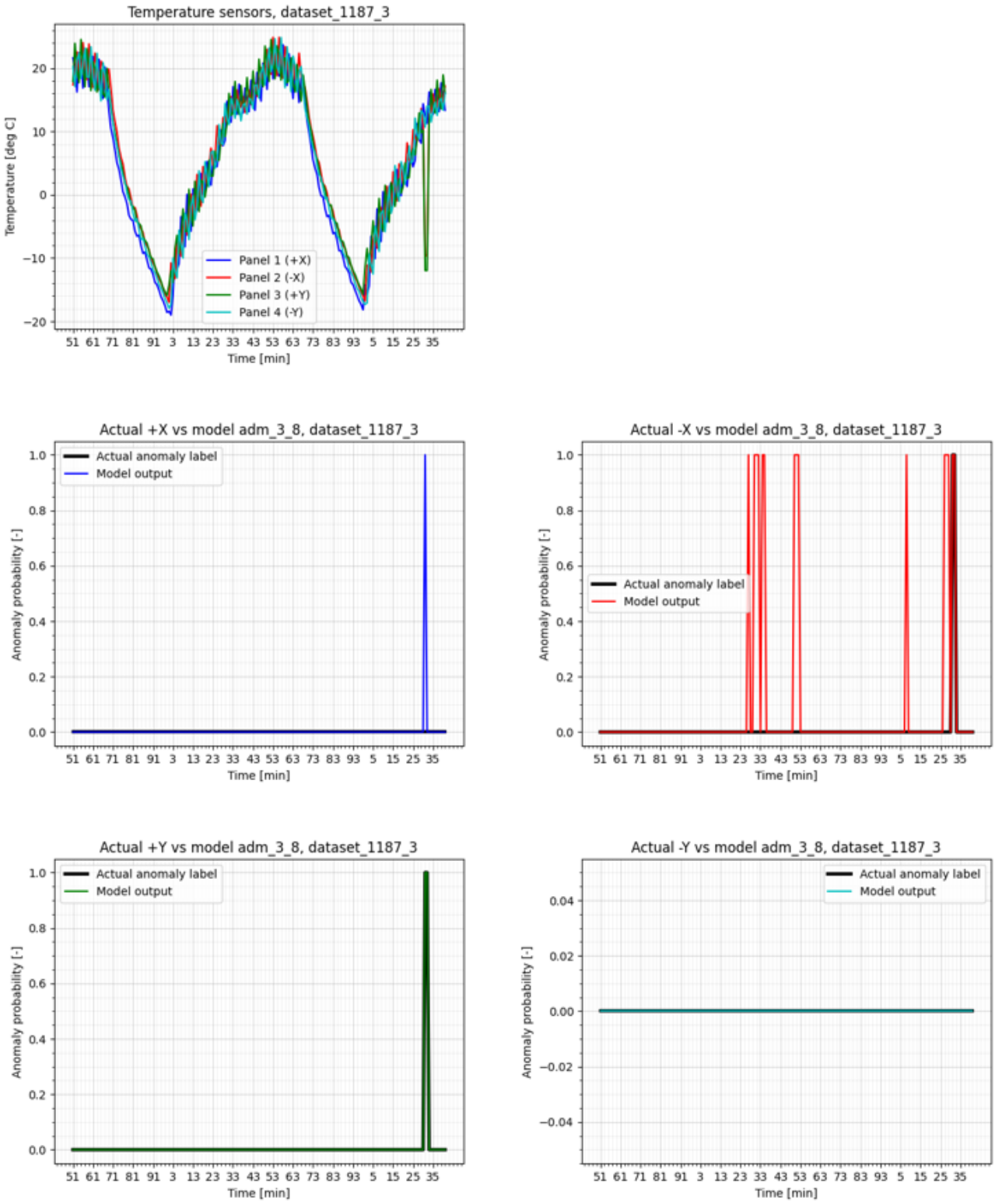


Figure 4: model inferences on dataset_1187_3

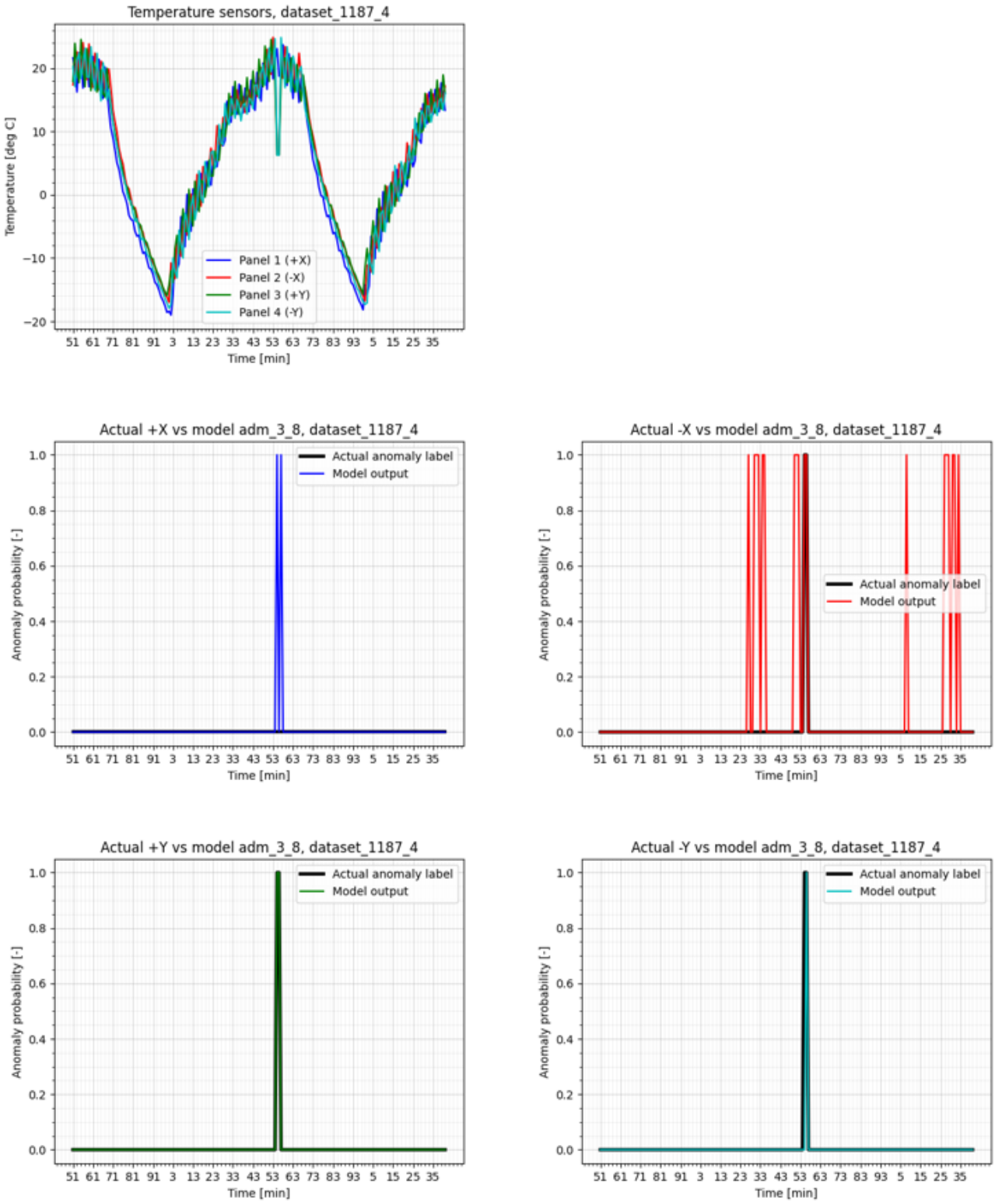
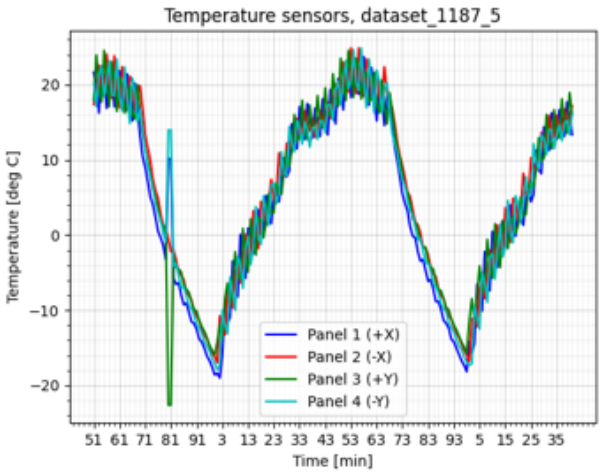
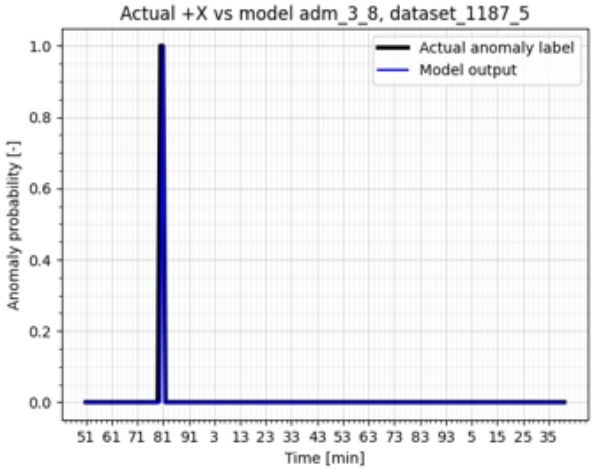
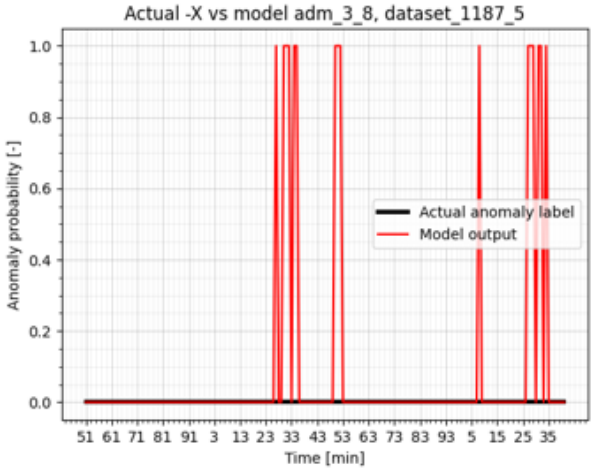
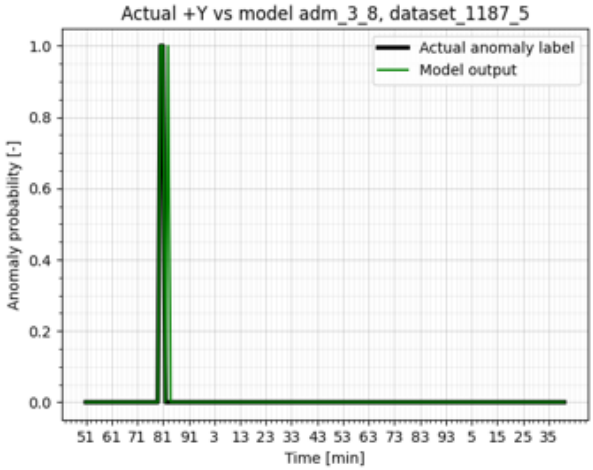


Figure 5: model inferences on dataset_1187_4









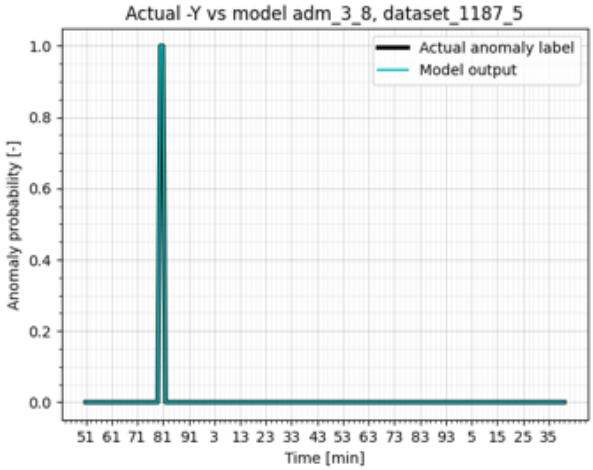


Figure 6: model inferences on dataset_1187_5

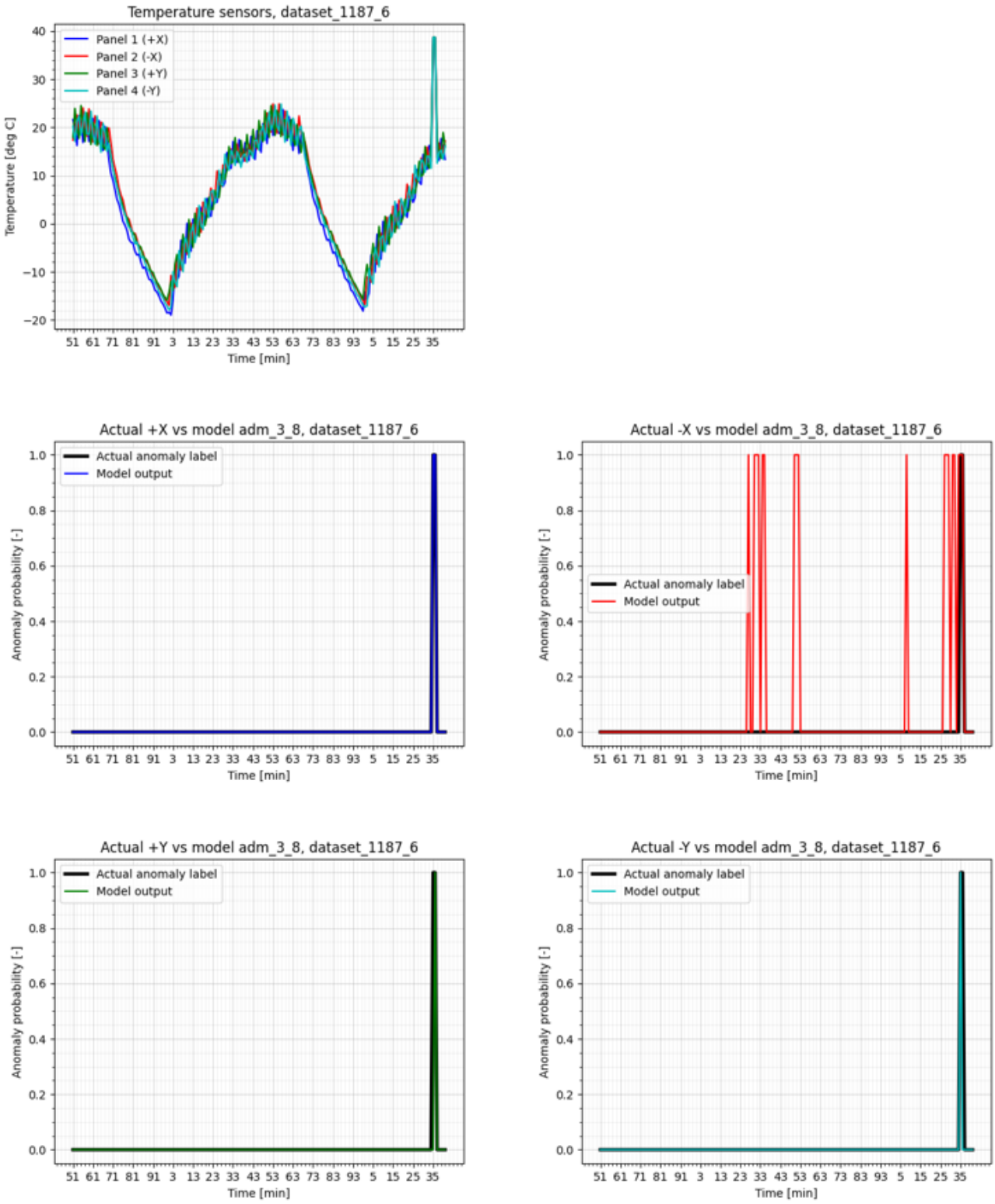
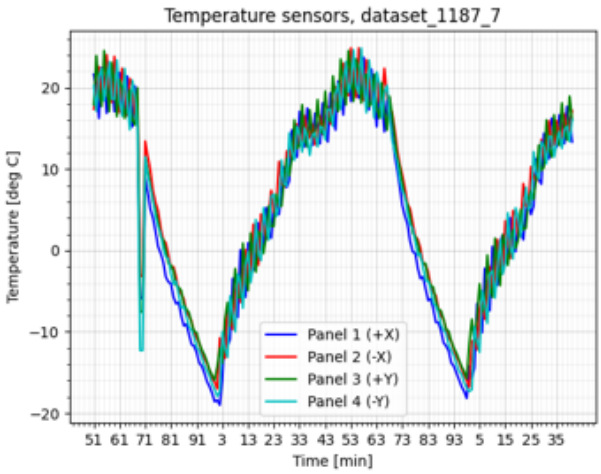
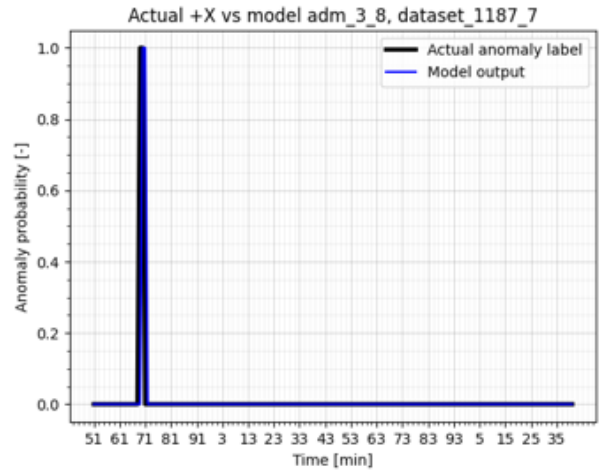
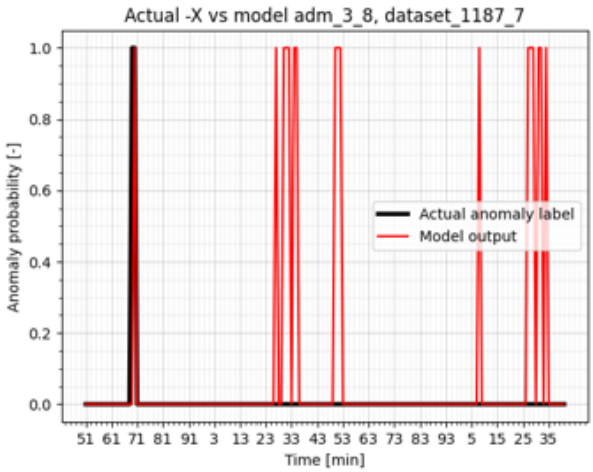
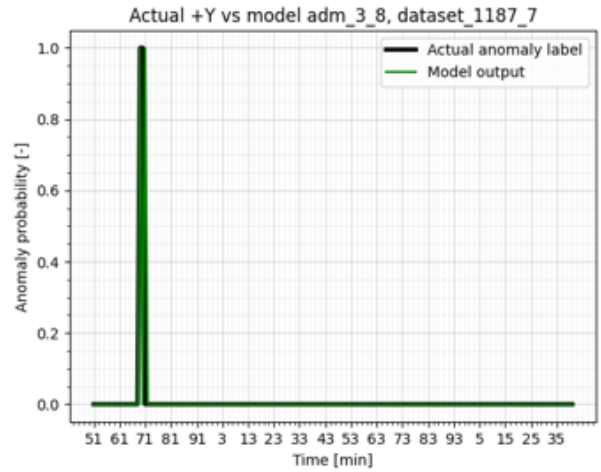


Figure 7: model inferences on dataset_1187_6









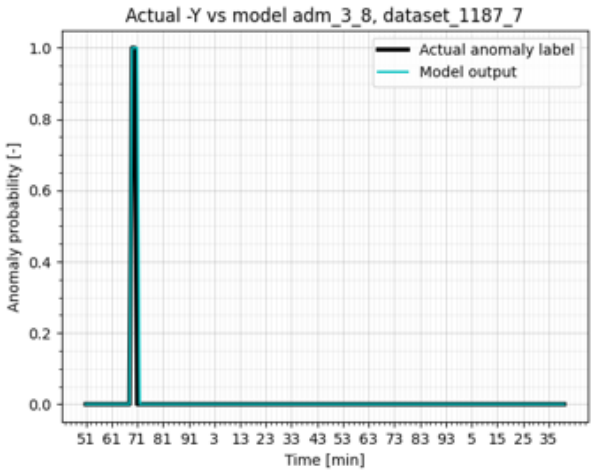
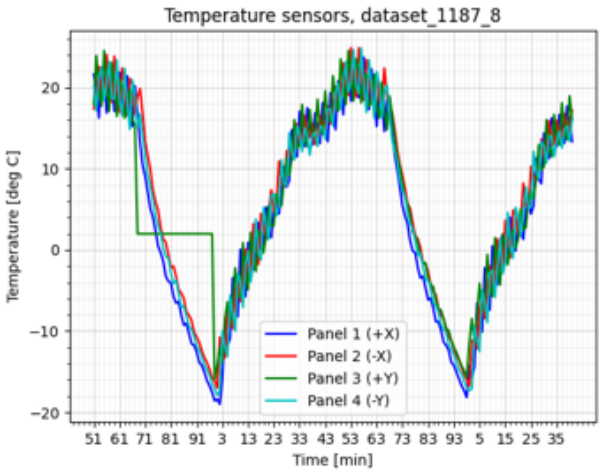
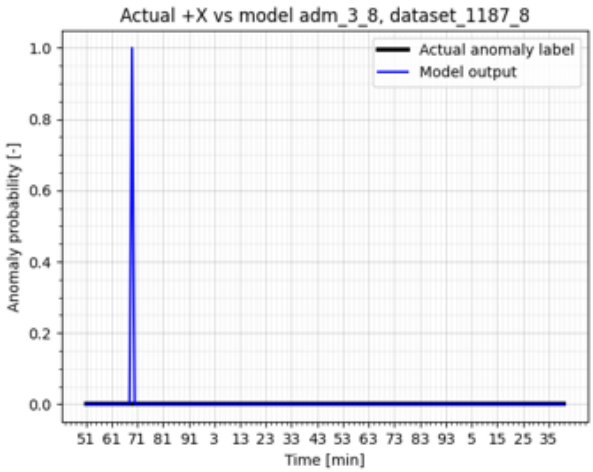
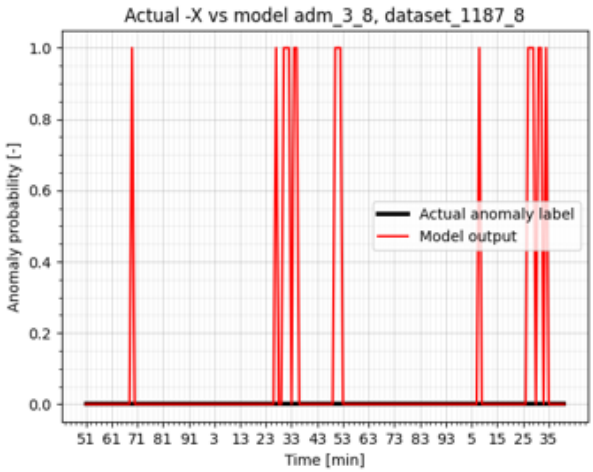


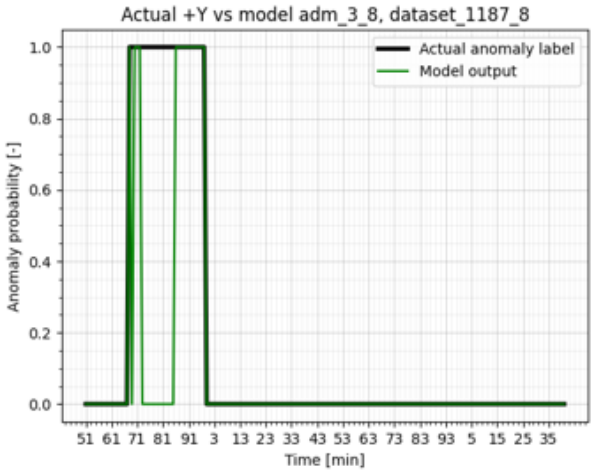
Figure 8: model inferences on dataset_1187_7

Flat regions









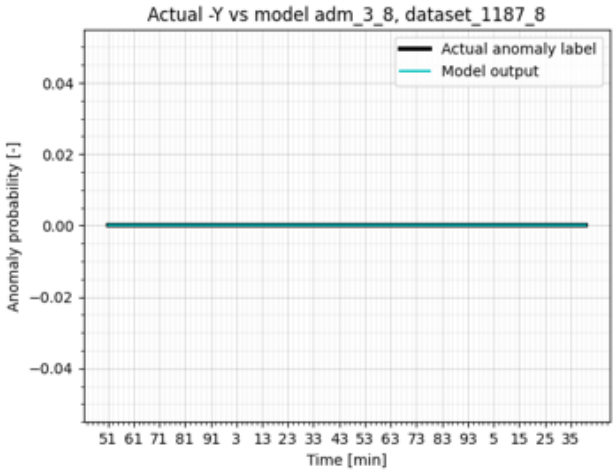
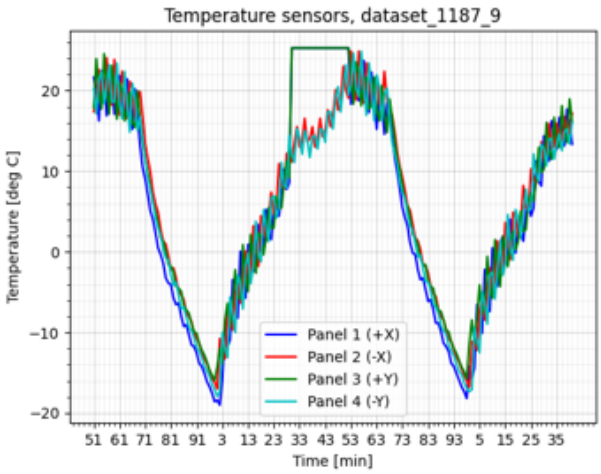
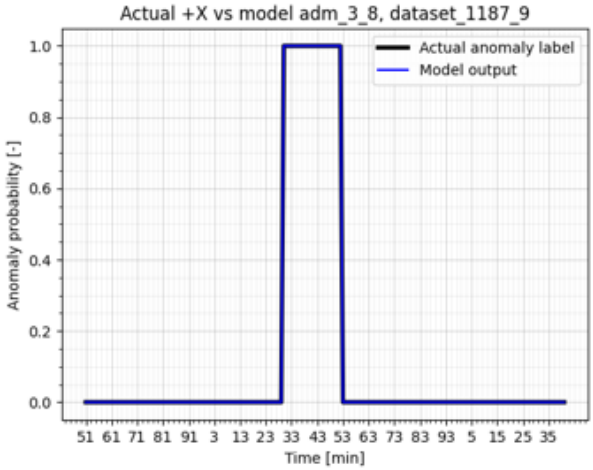
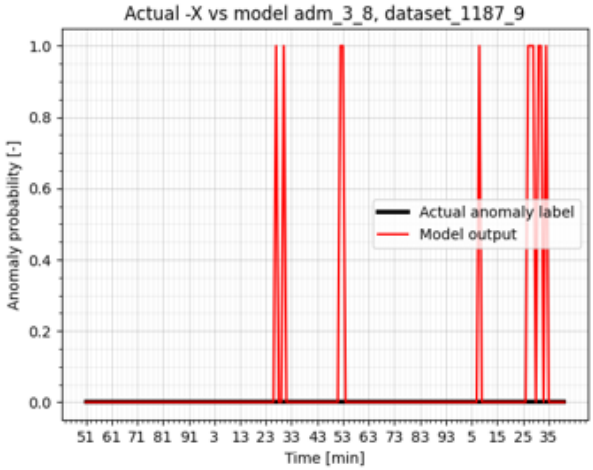
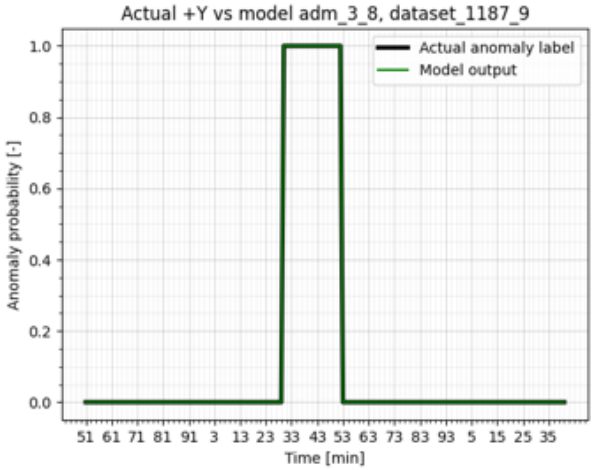


Figure 9: model inferences on dataset_1187_8









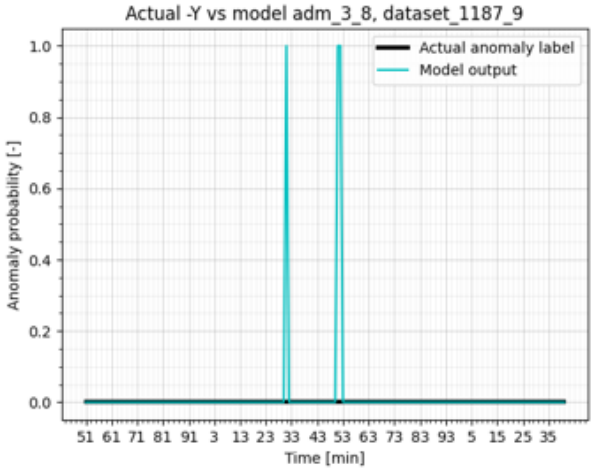


Figure 10: model inferences on dataset_1187_9

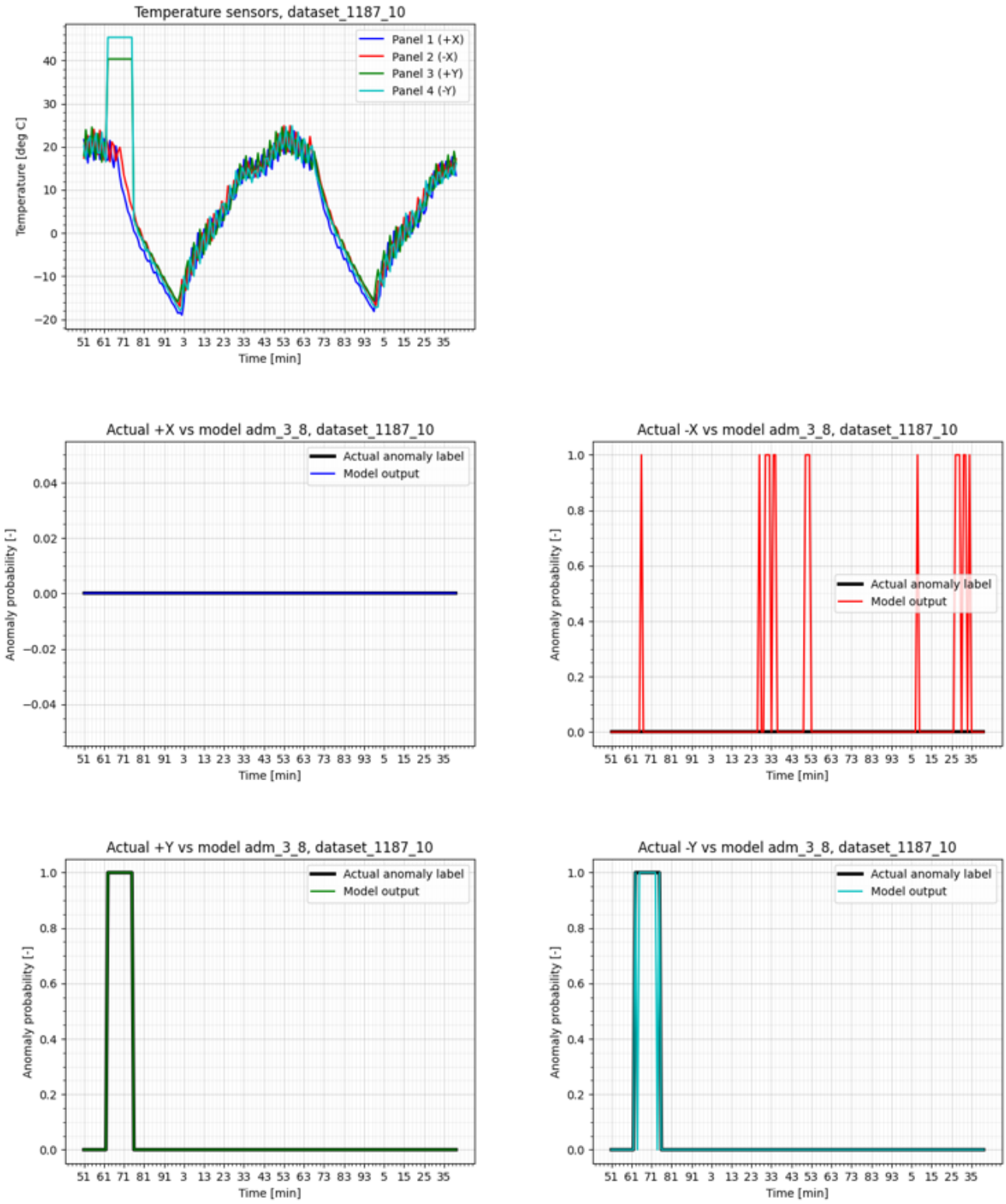


Figure 11: model inferences on dataset_1187_10

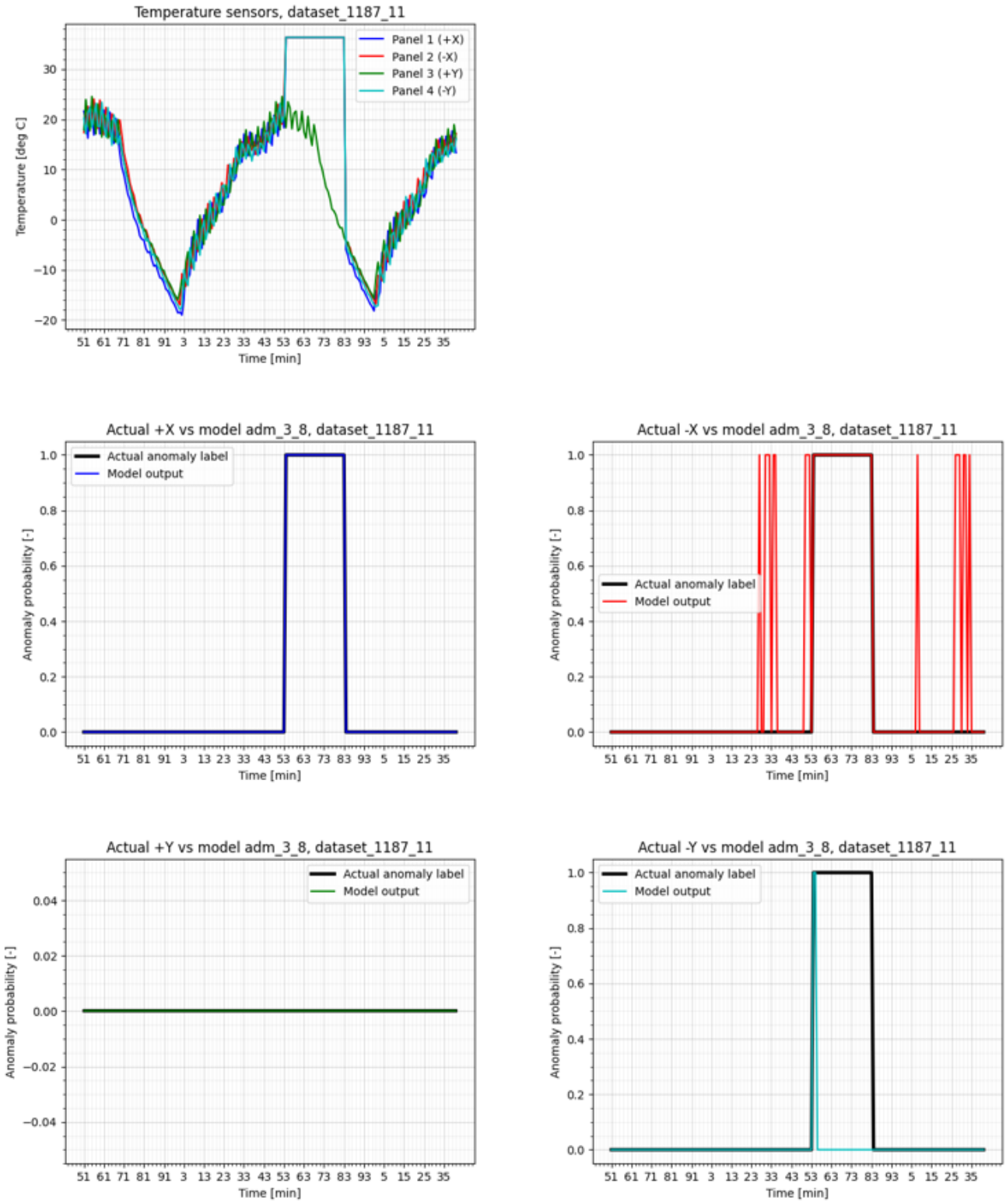
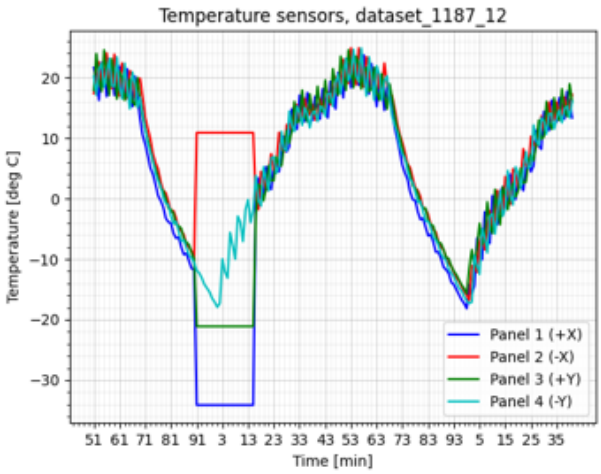
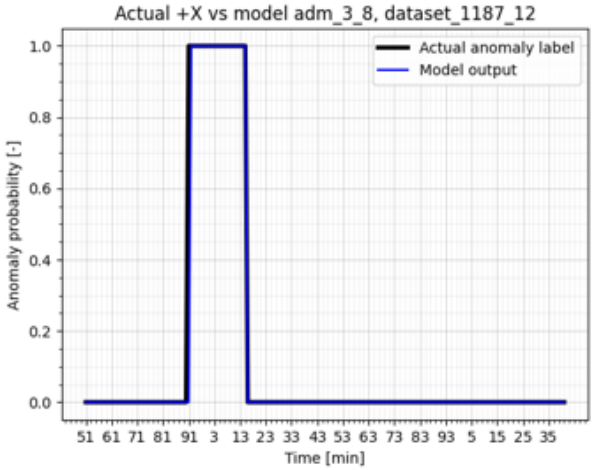
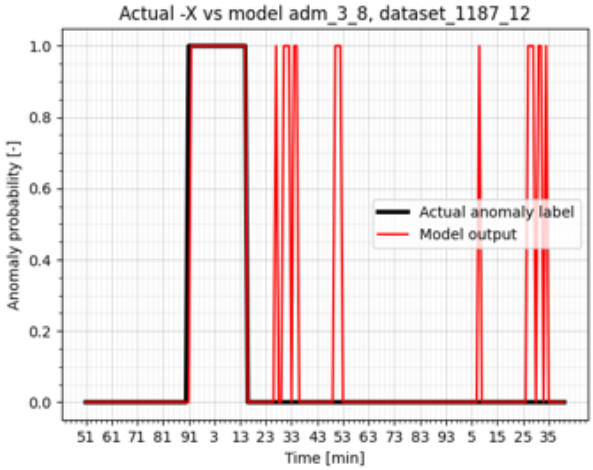
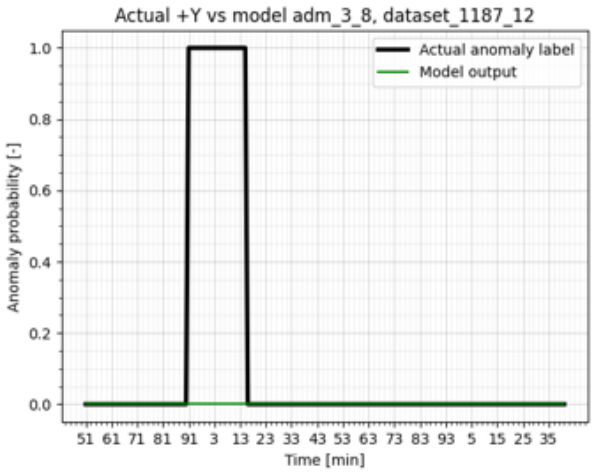


Figure 12: model inferences on dataset_1187_11









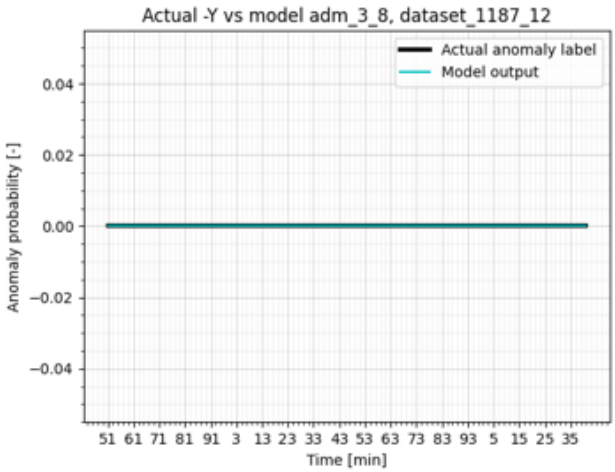


Figure 13: model inferences on dataset_1187_12

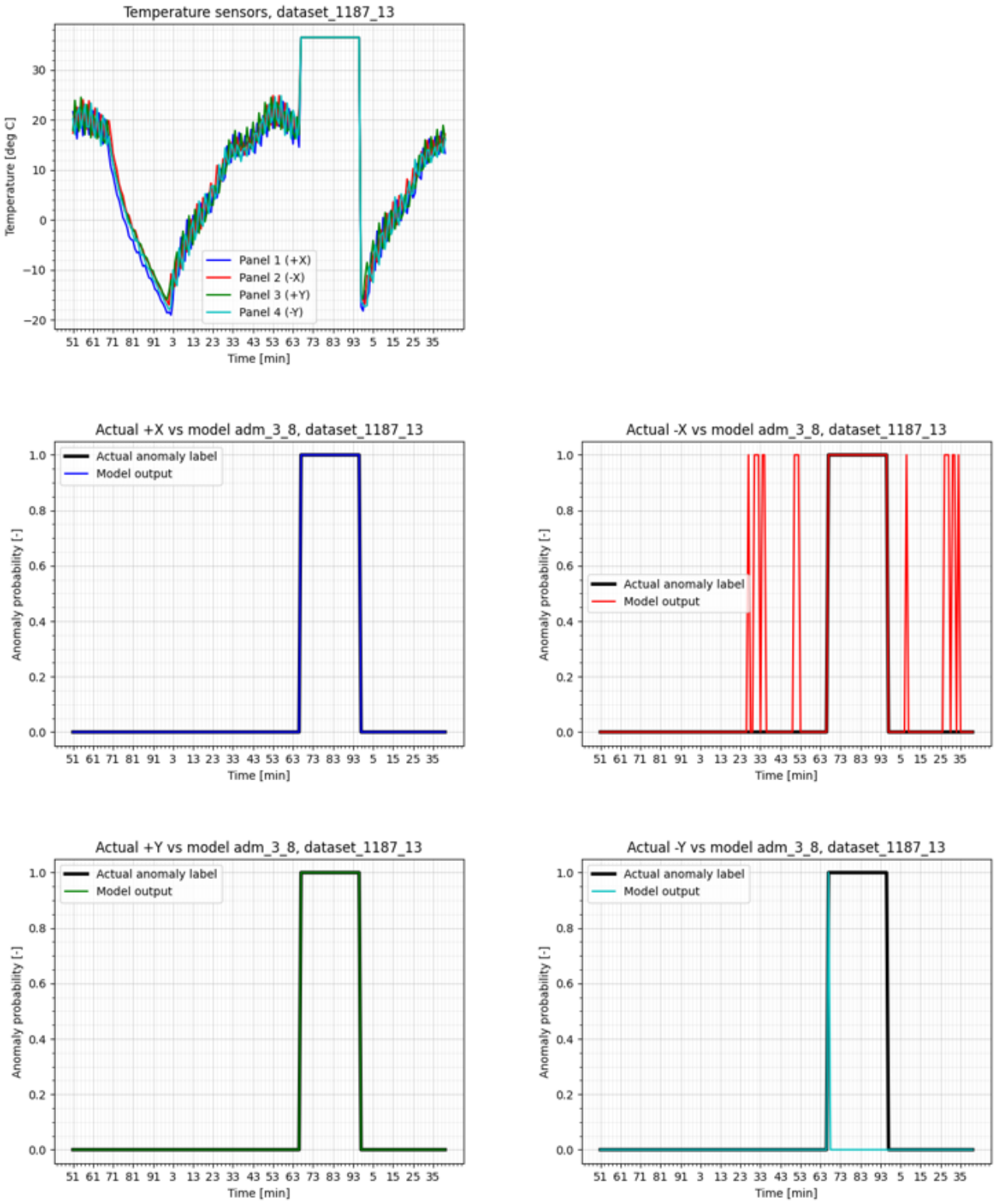


Figure 14: model inferences on dataset_1187_13

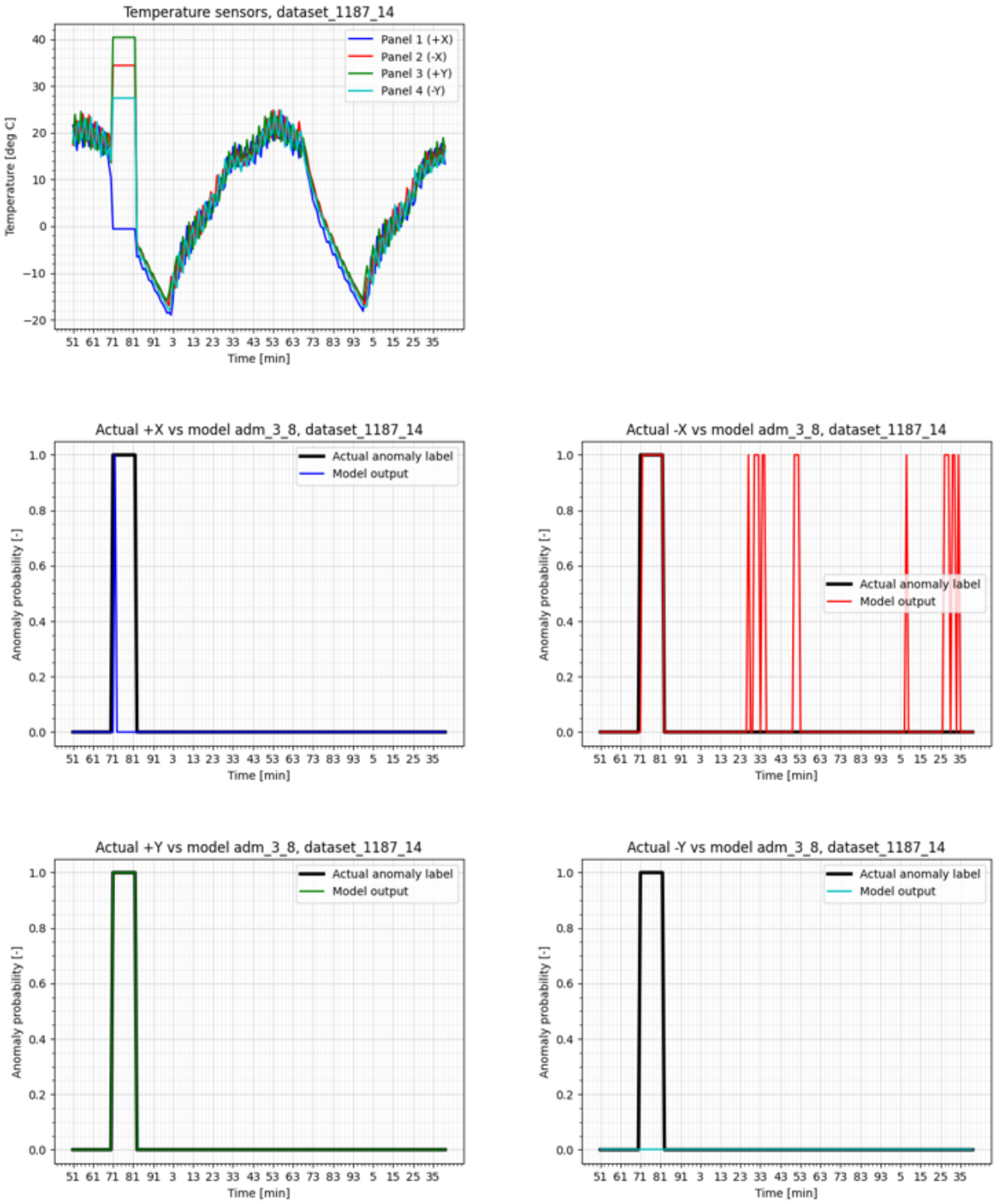


Figure 15: model inferences on dataset_1187_14

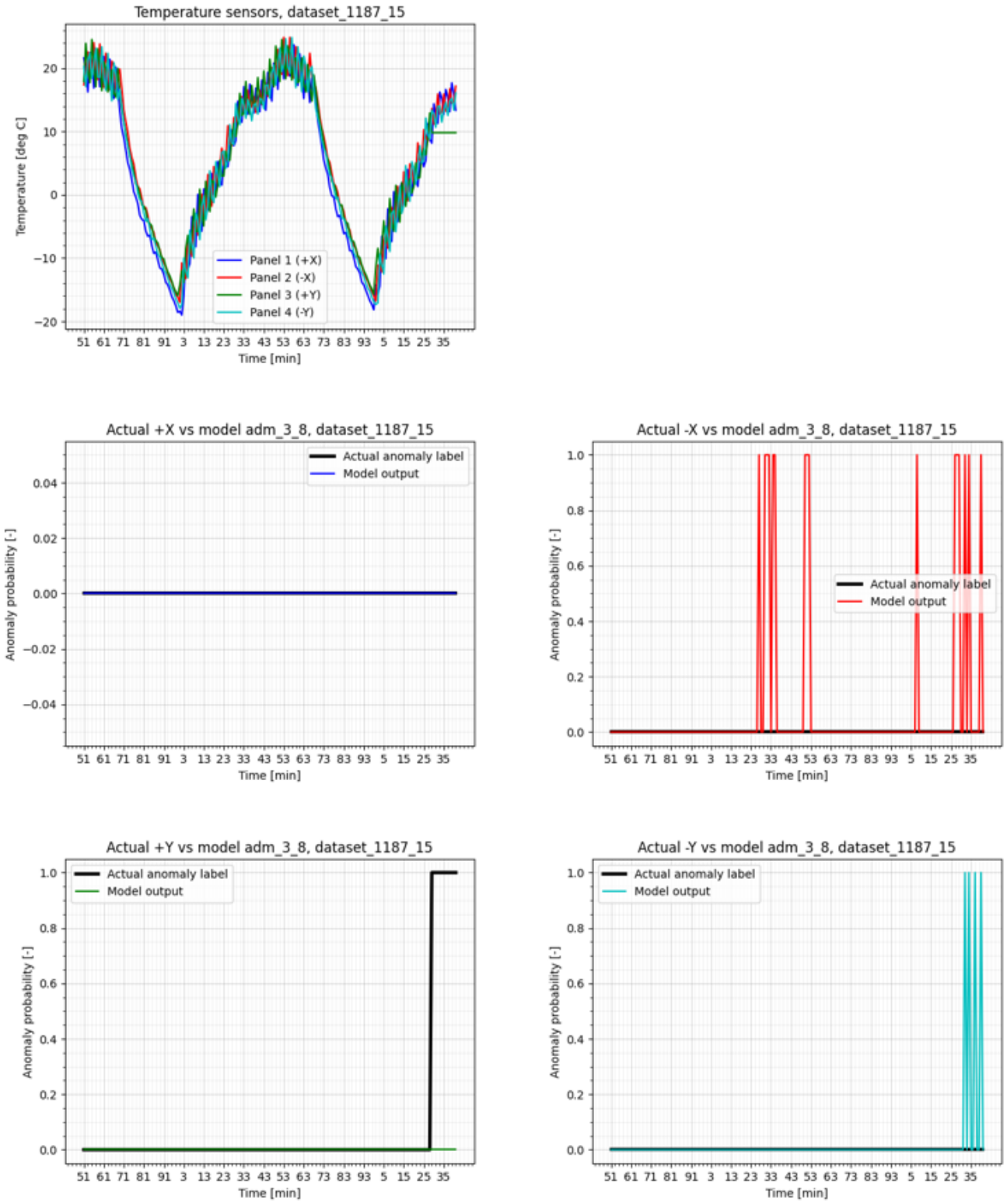


Figure 16: model inferences on dataset_1187_15

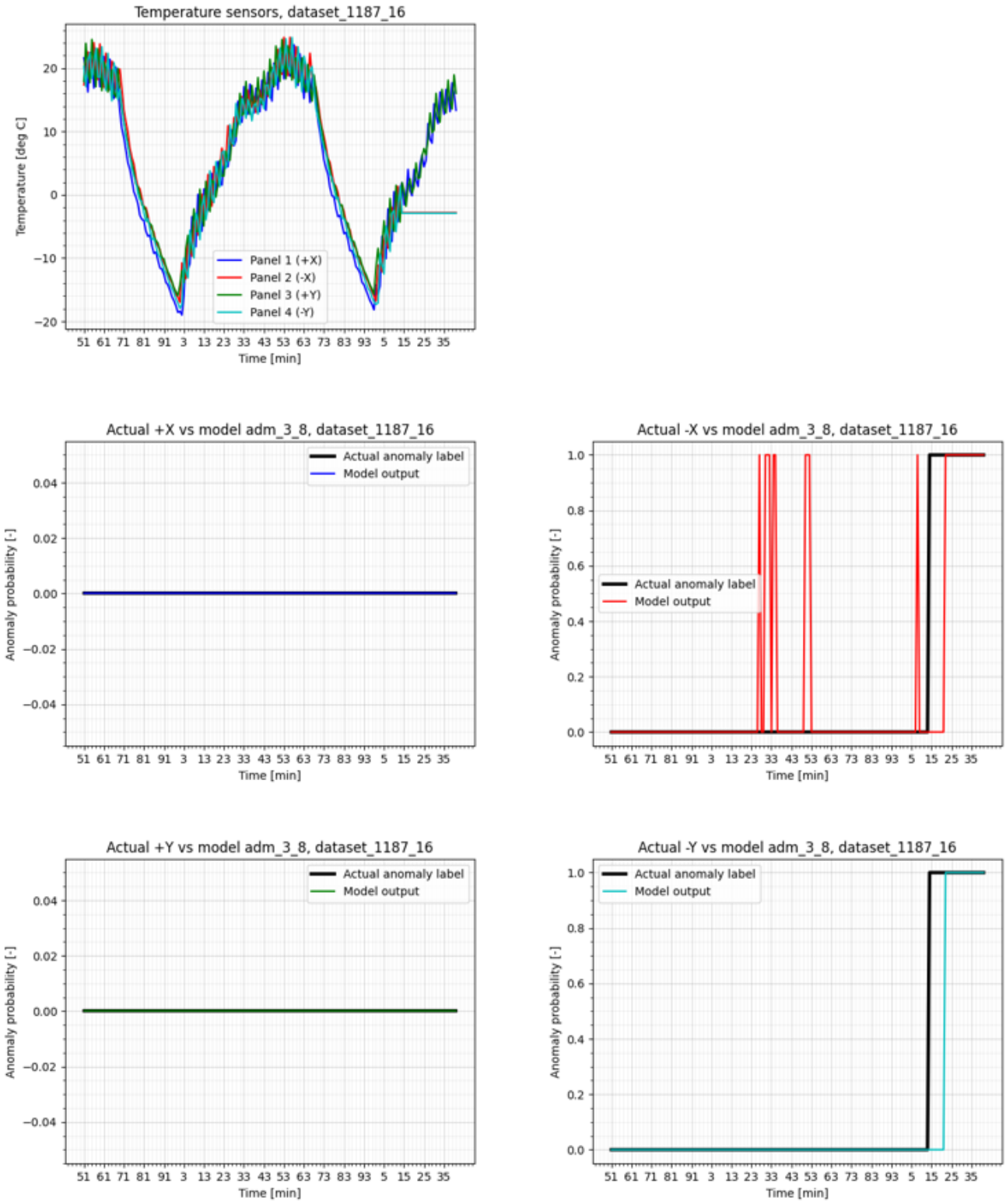
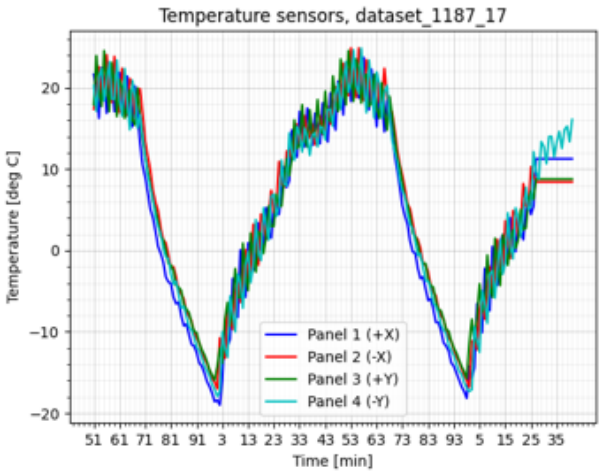
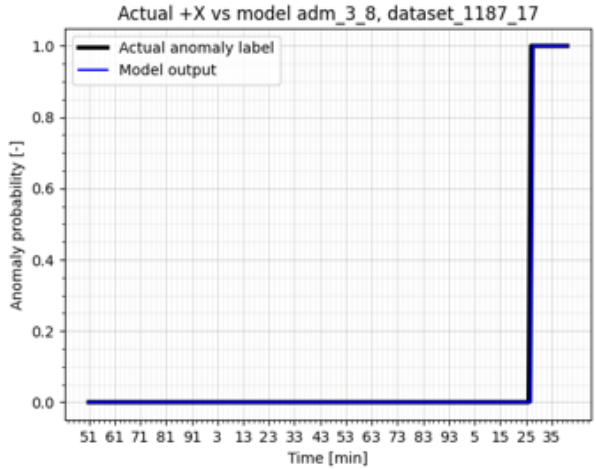
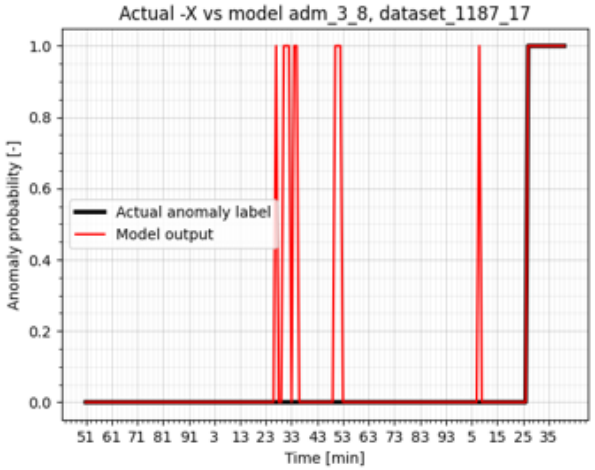
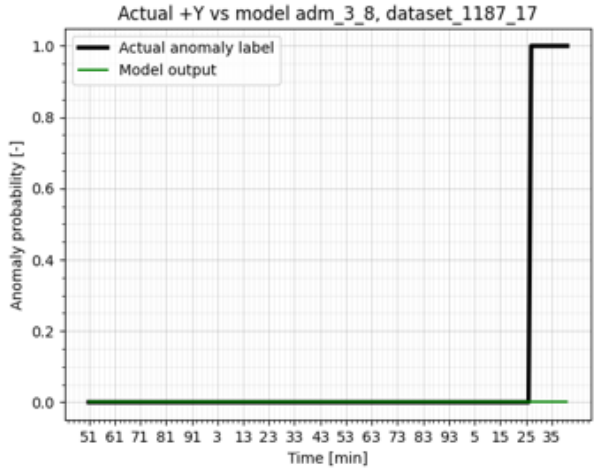


Figure 17: model inferences on dataset_1187_16









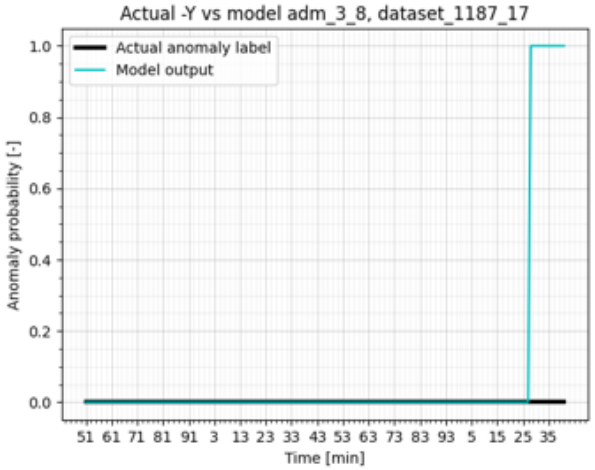
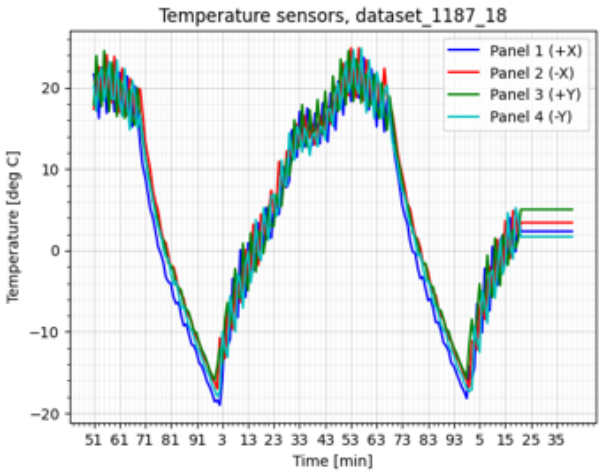
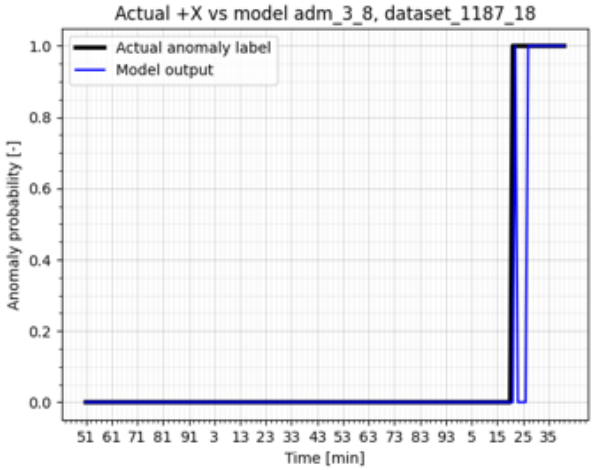
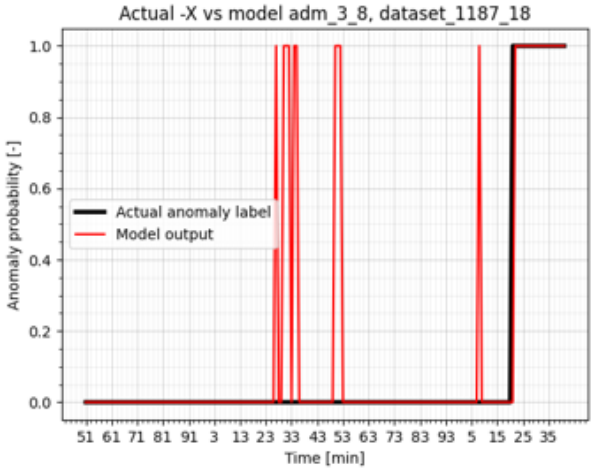
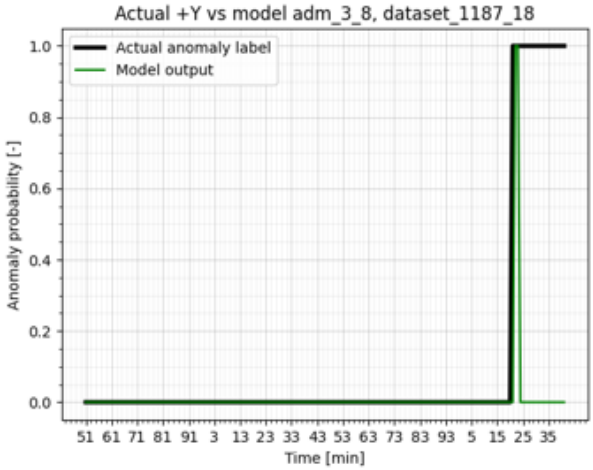


Figure 18: model inferences on dataset_1187_17









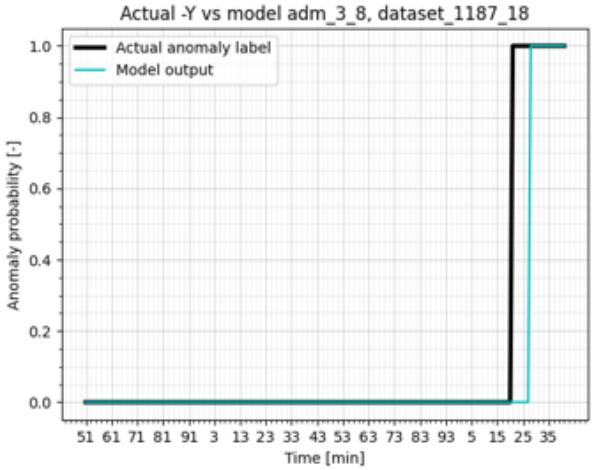


Figure 19: model inferences on dataset_1187_18

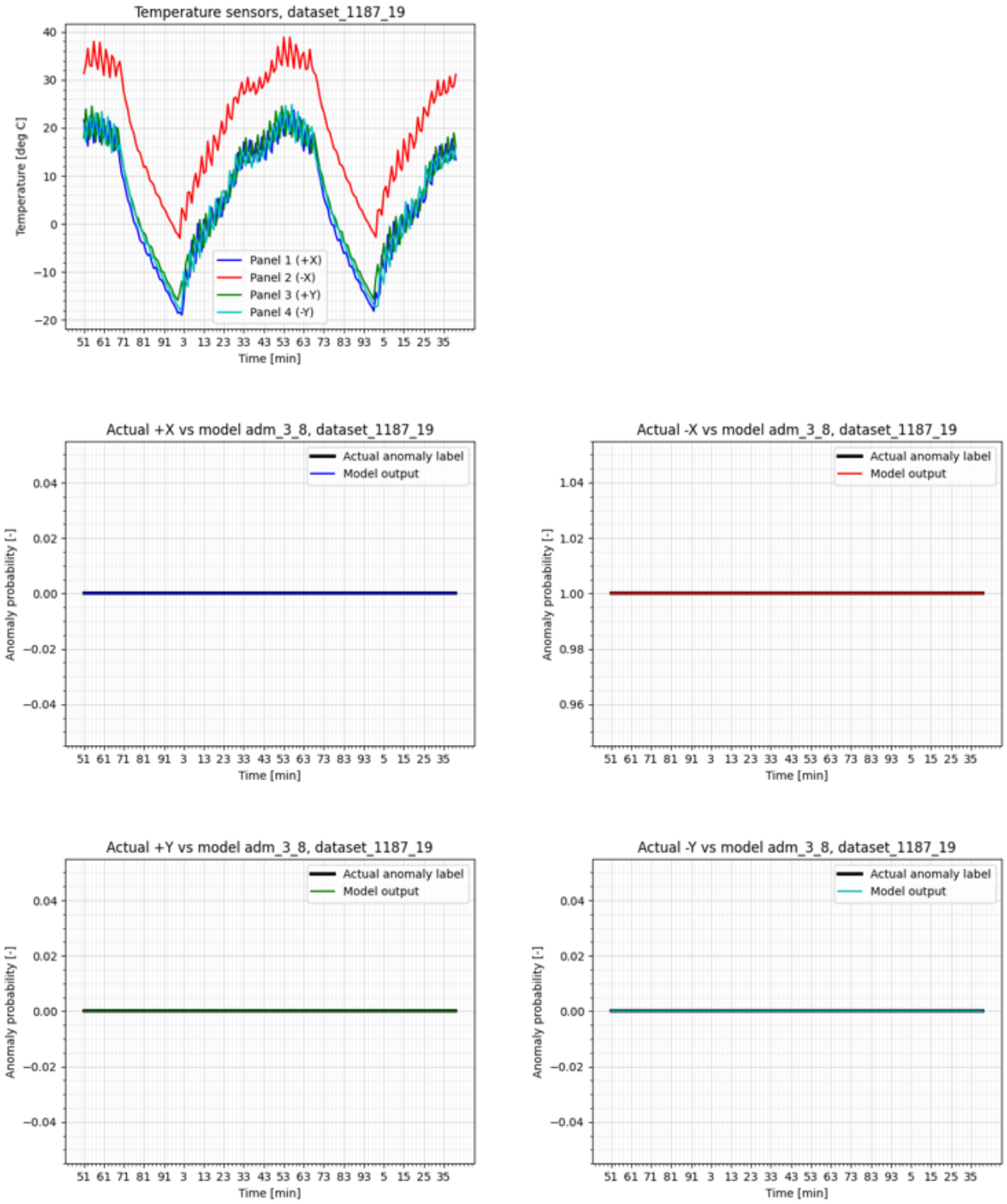
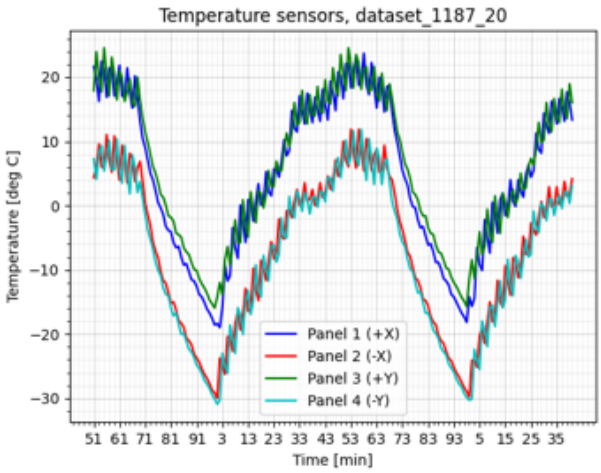
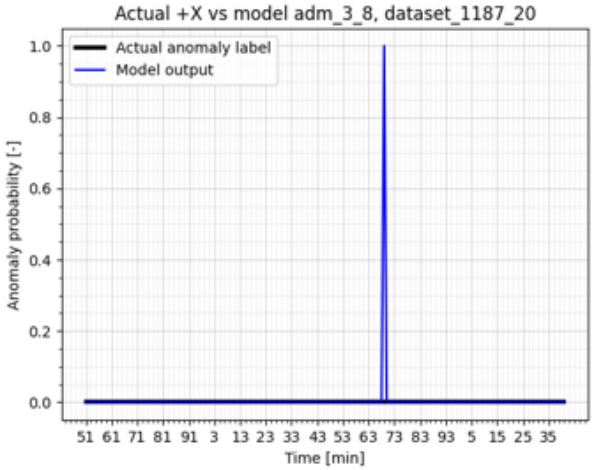
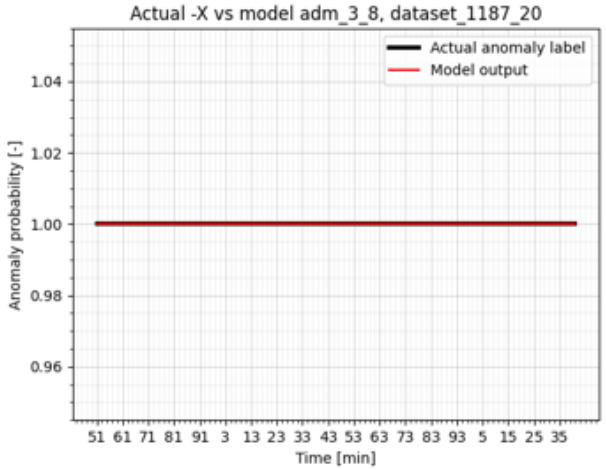


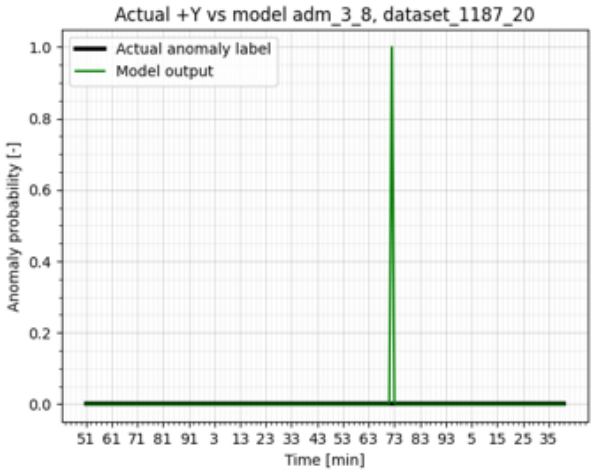
Figure 20: model inferences on dataset_1187_19

Permanent bias









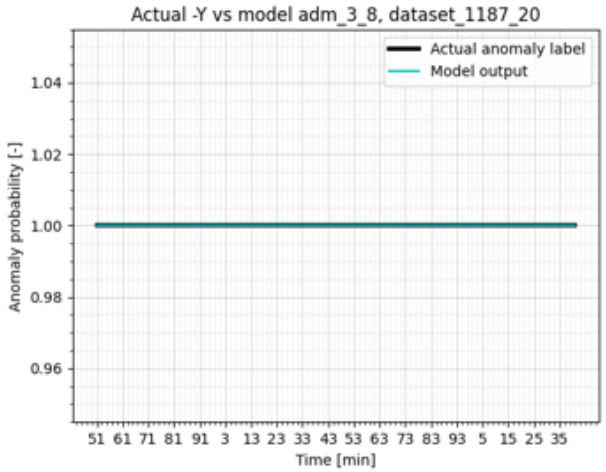
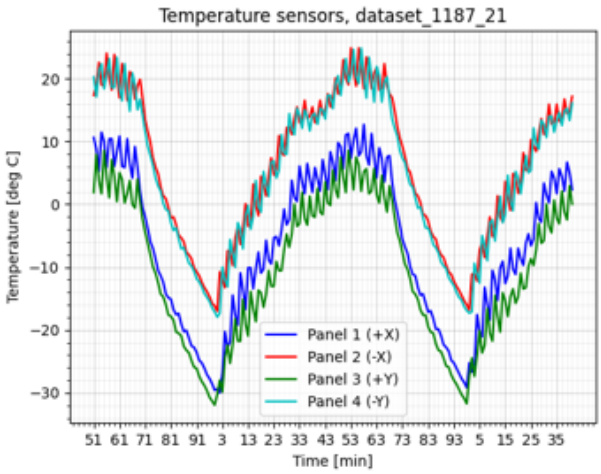
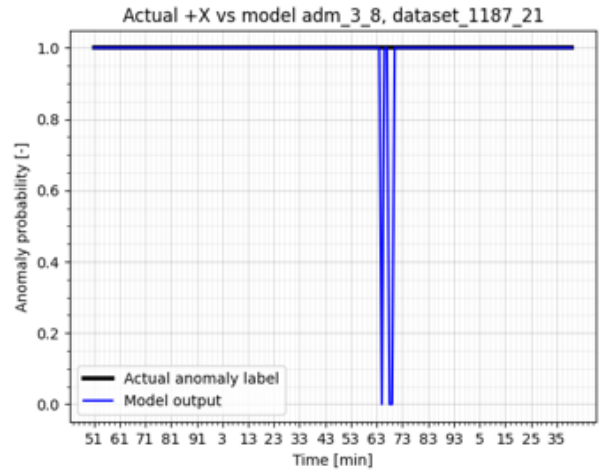
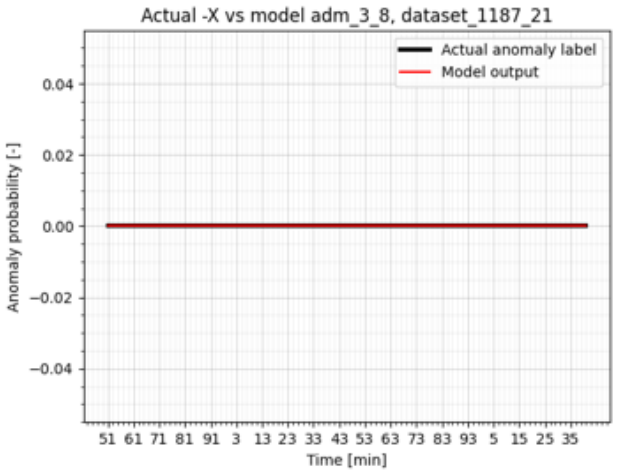
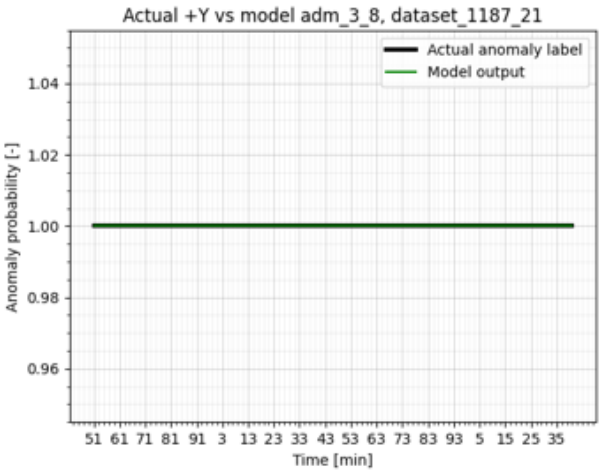


Figure 21: model inferences on dataset_1187_20









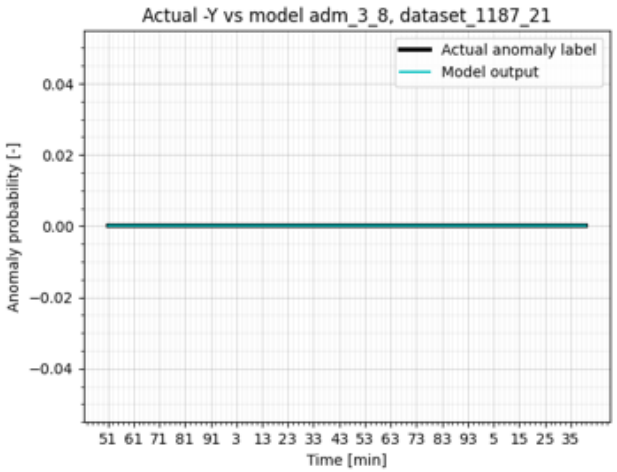
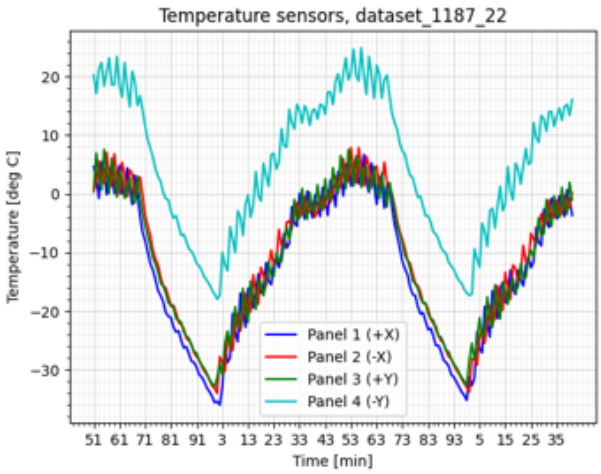
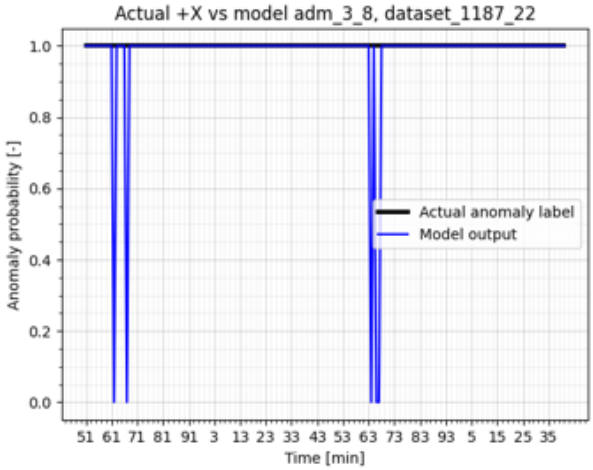
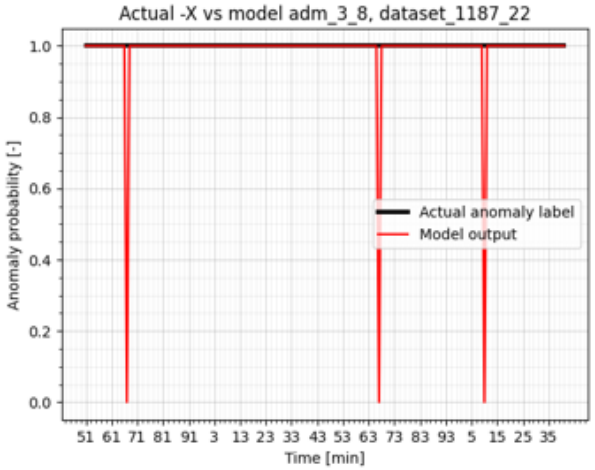
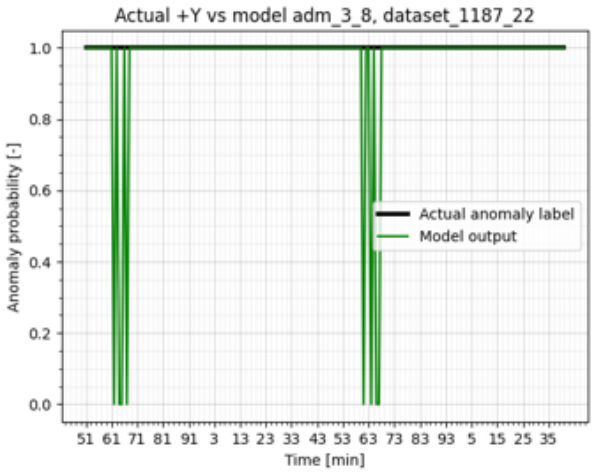


Figure 22: model inferences on dataset_1187_21









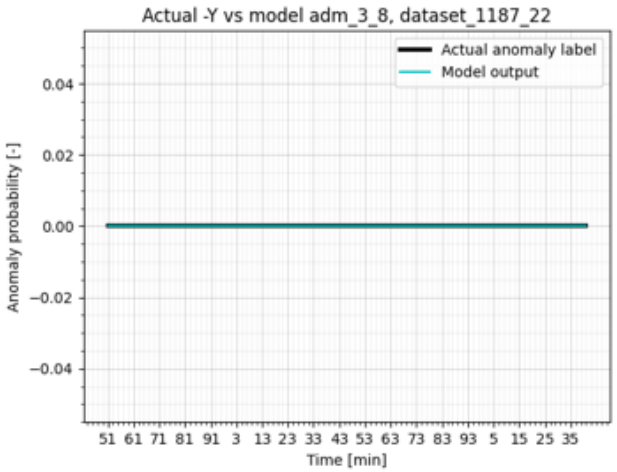


Figure 23: model inferences on dataset_1187_22

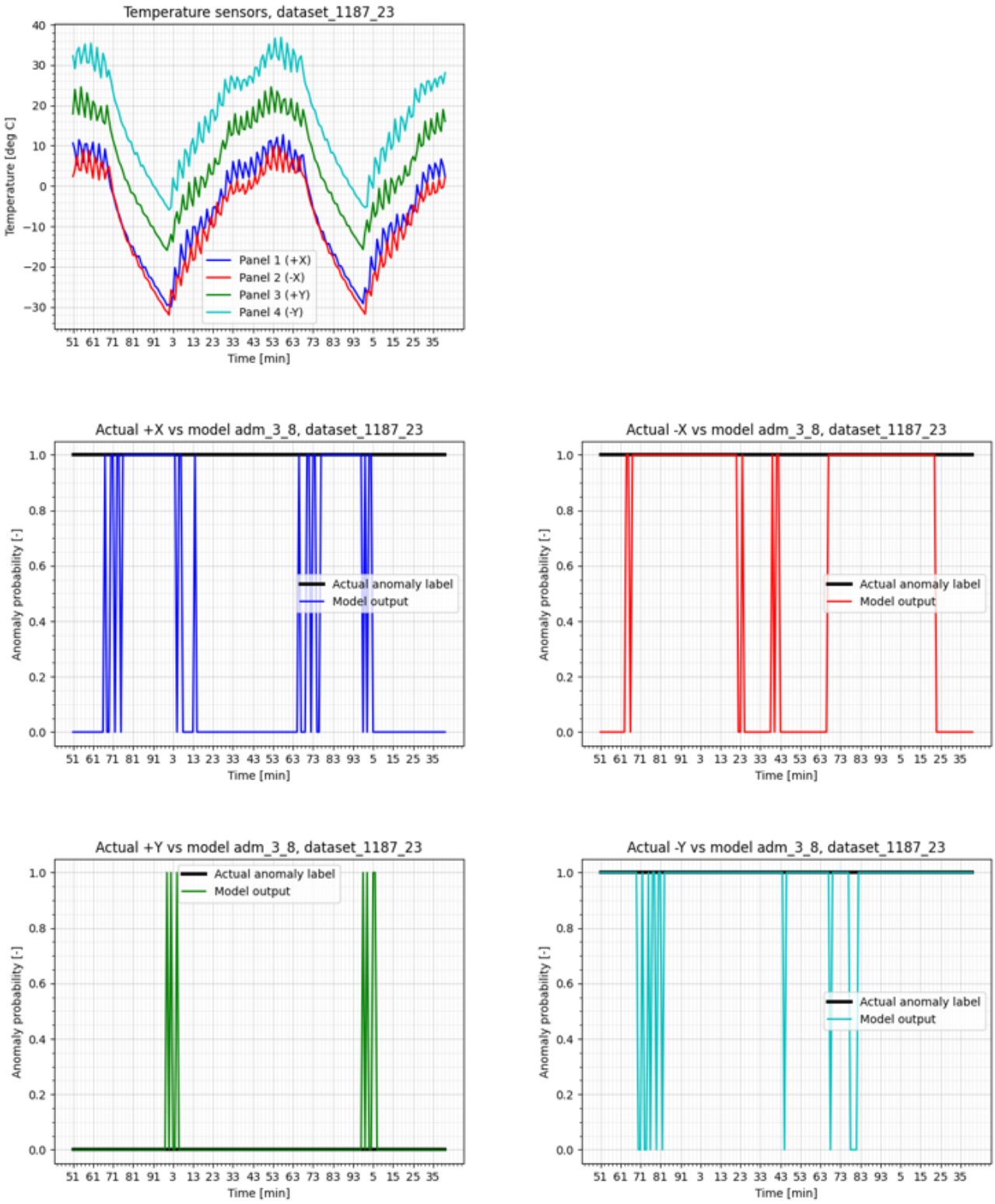


Figure 24: model inferences on dataset_1187_23

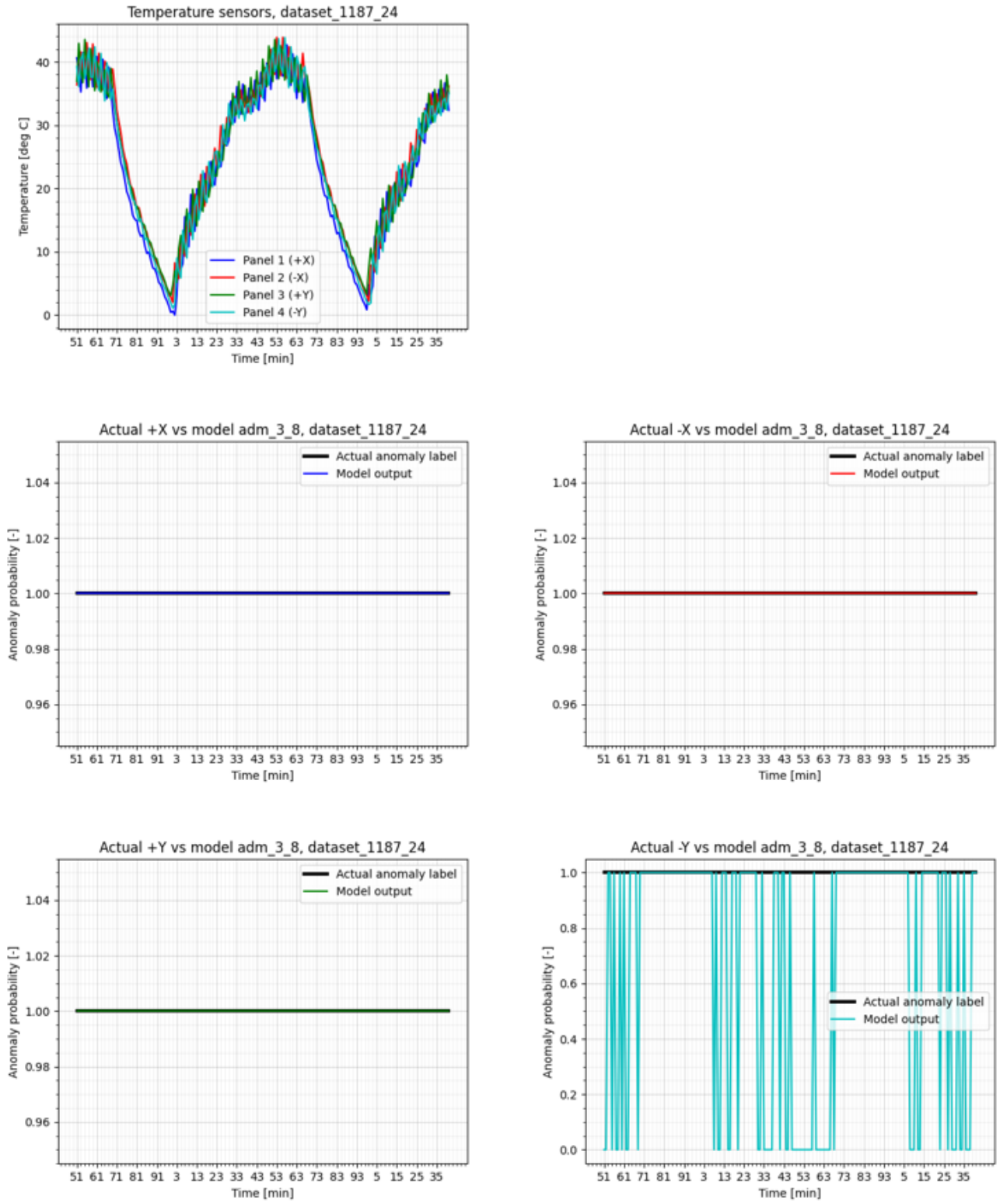


Figure 25: model inferences on dataset_1187_24

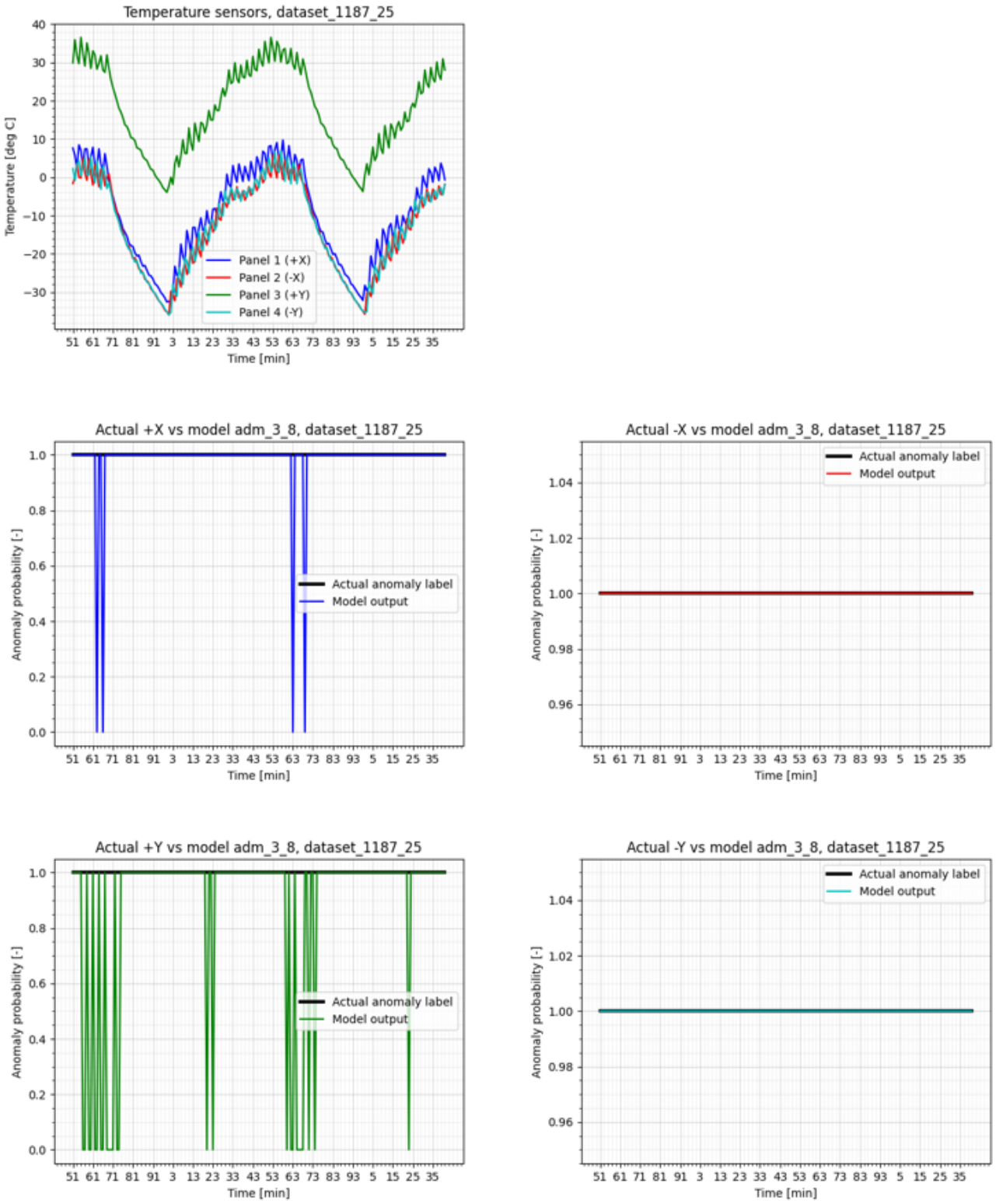


Figure 26: model inferences on dataset_1187_25

Temporary bias

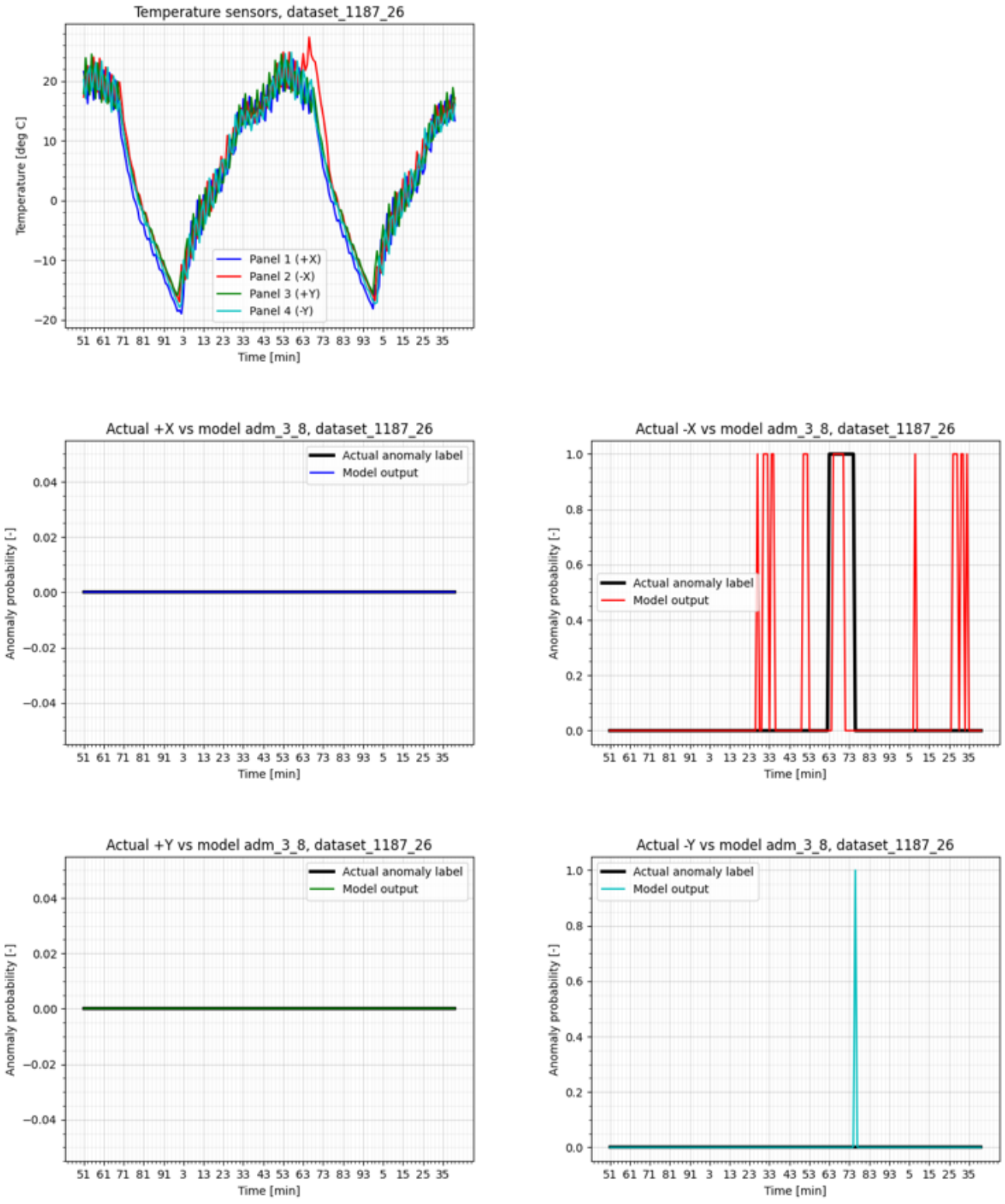


Figure 27: model inferences on dataset_1187_26

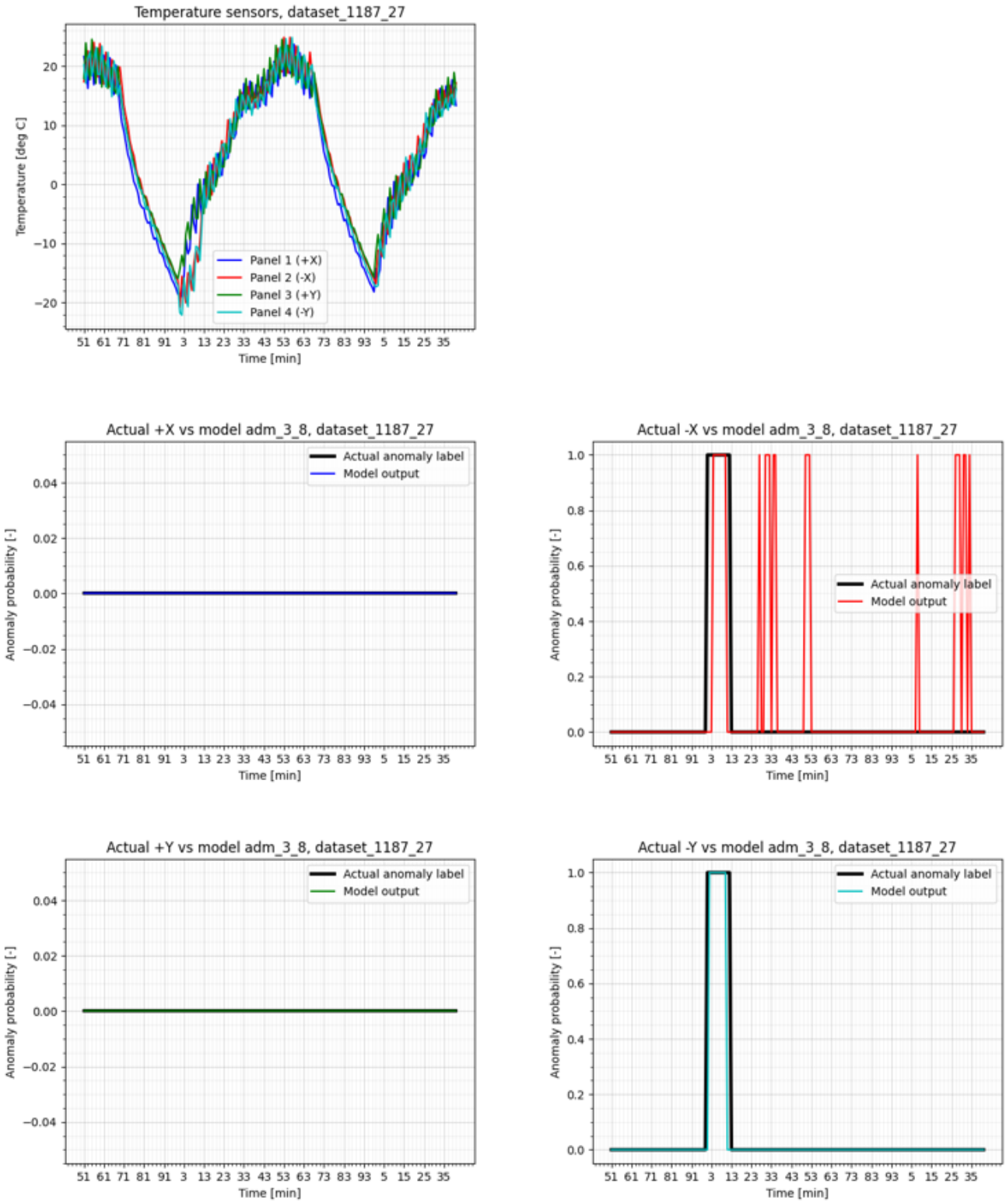


Figure 28: model inferences on dataset_1187_27

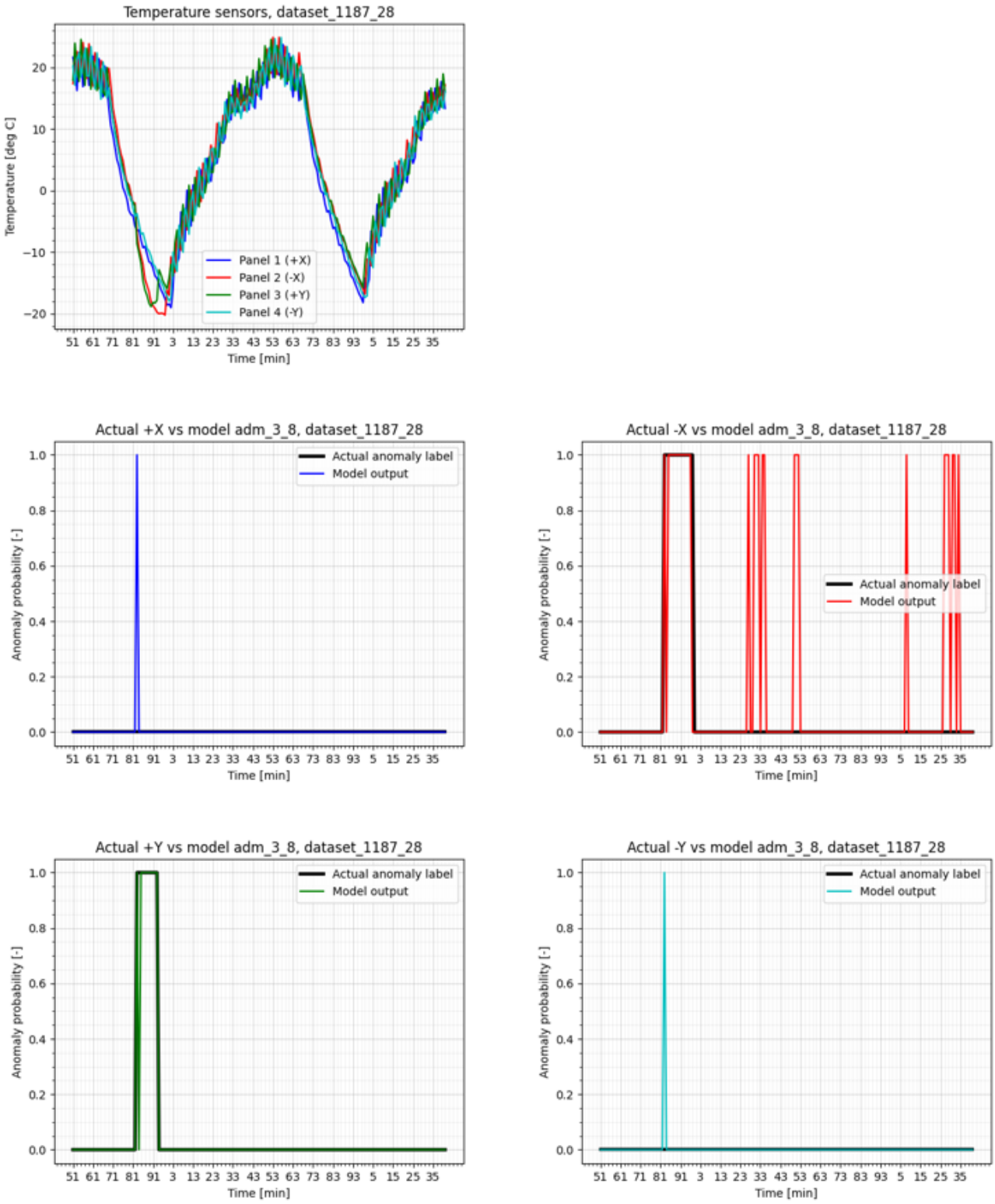


Figure 29: model inferences on dataset_1187_28

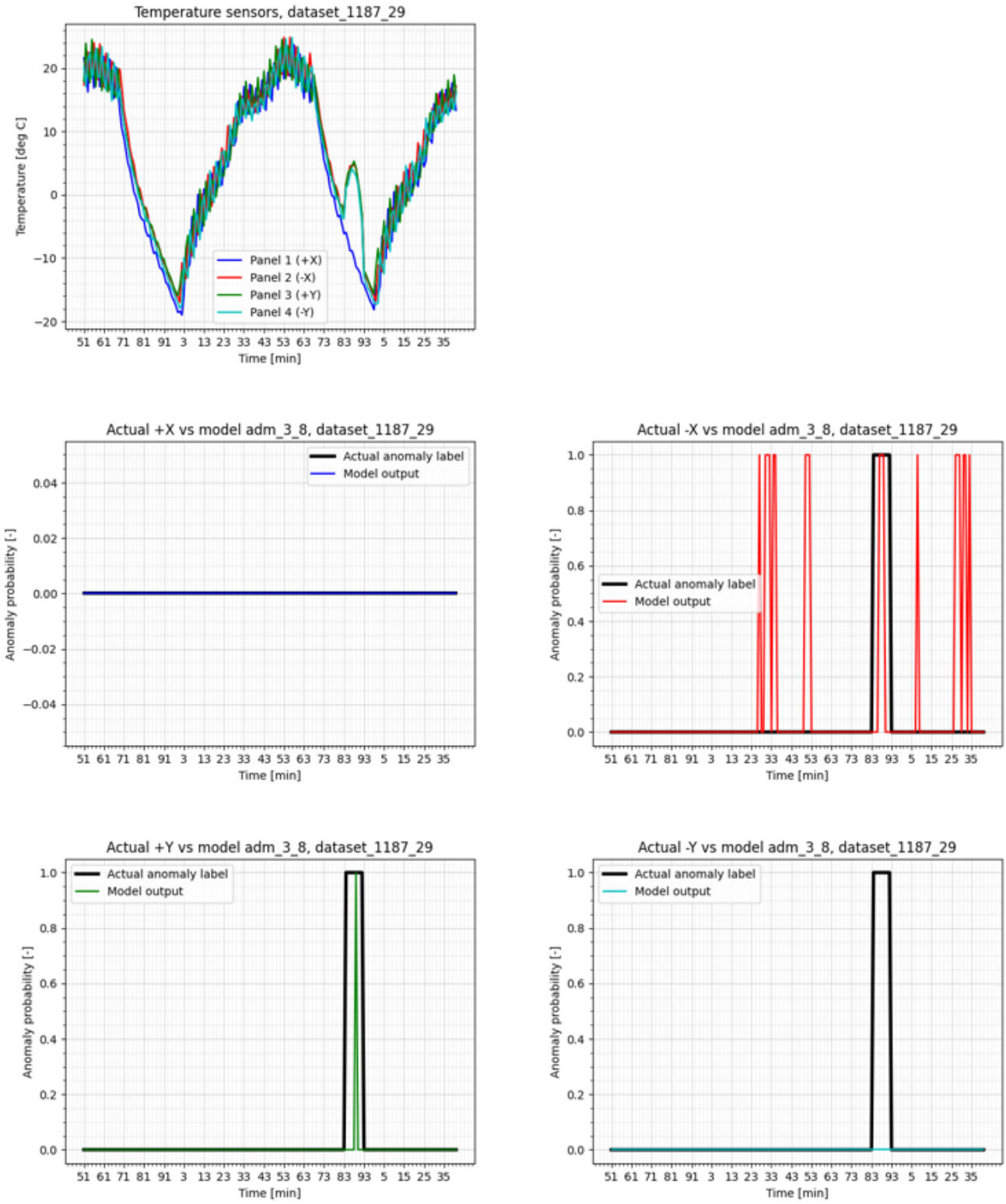


Figure 30: model inferences on dataset_1187_29

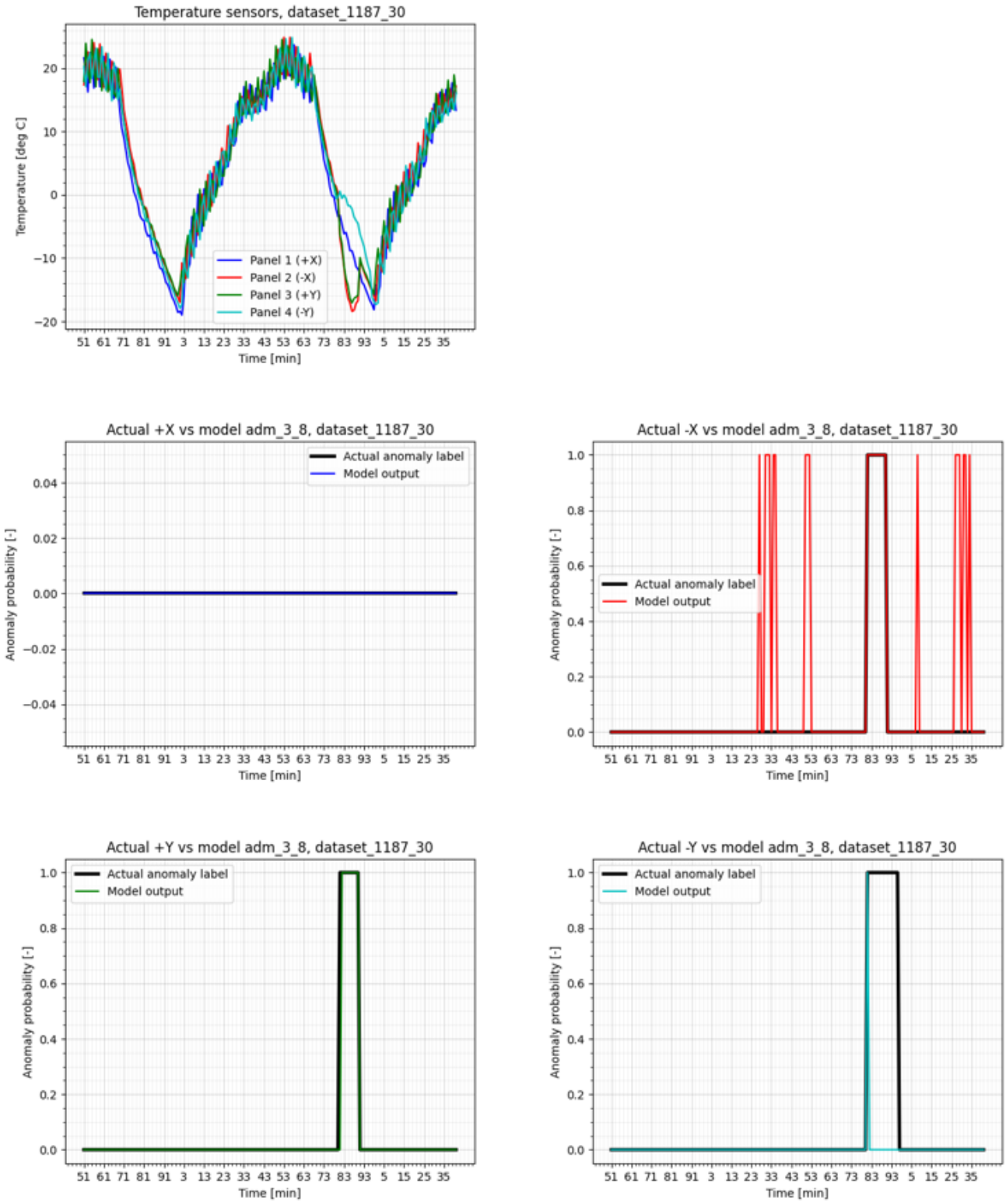
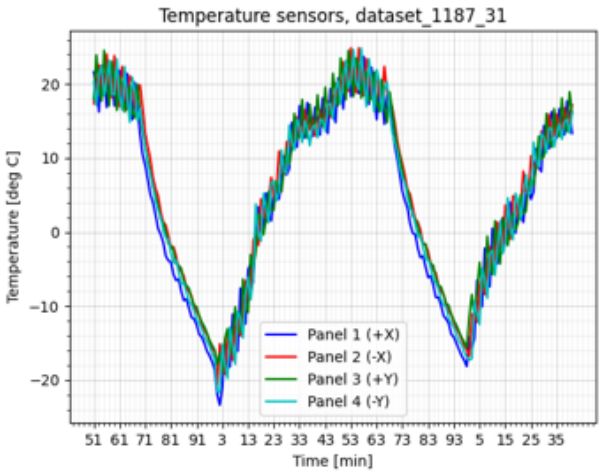
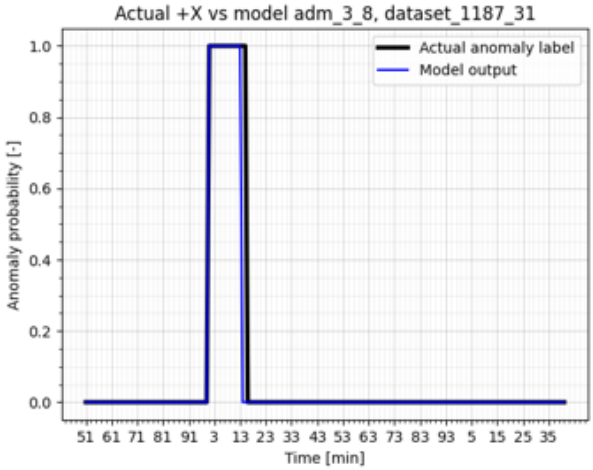
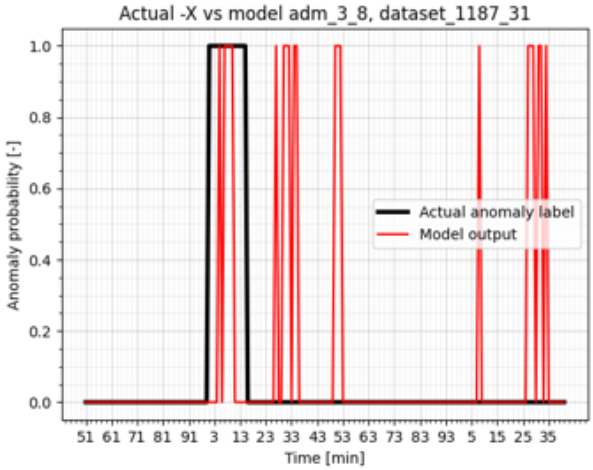
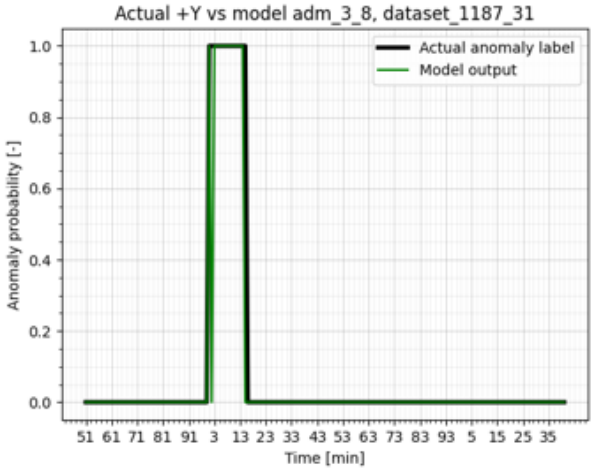


Figure 31: model inferences on dataset_1187_30









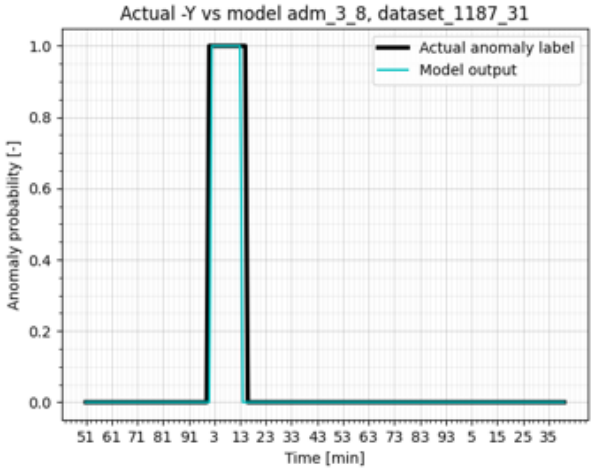
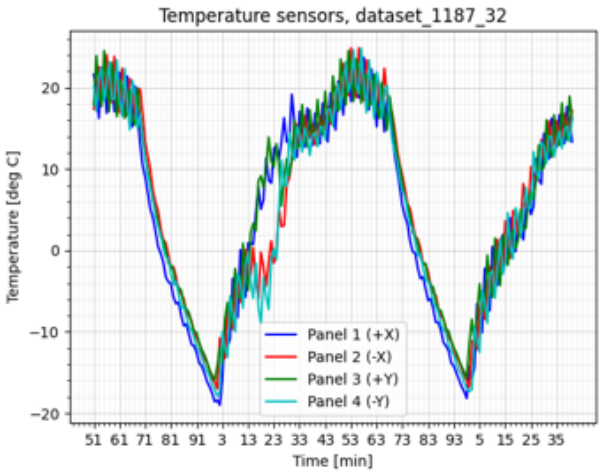
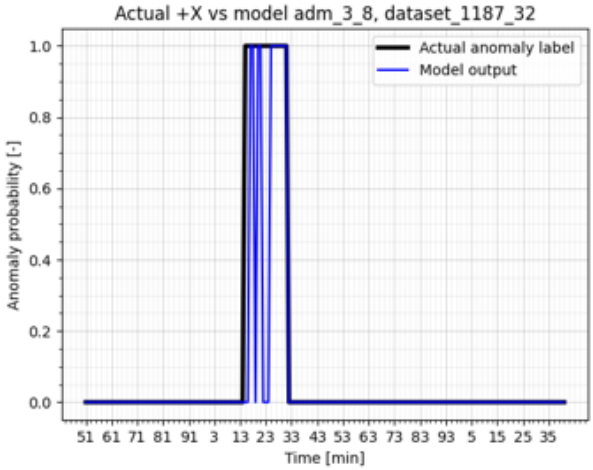
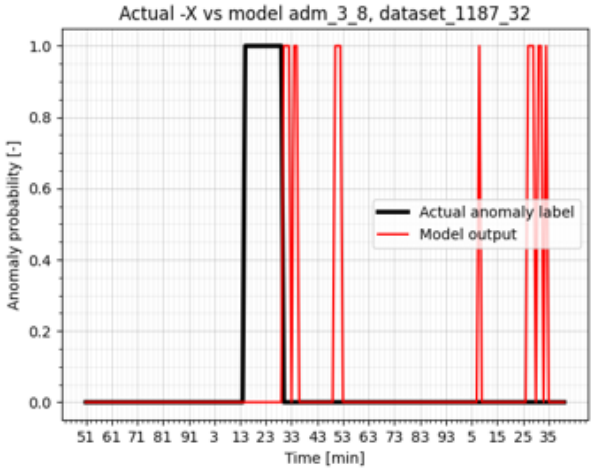
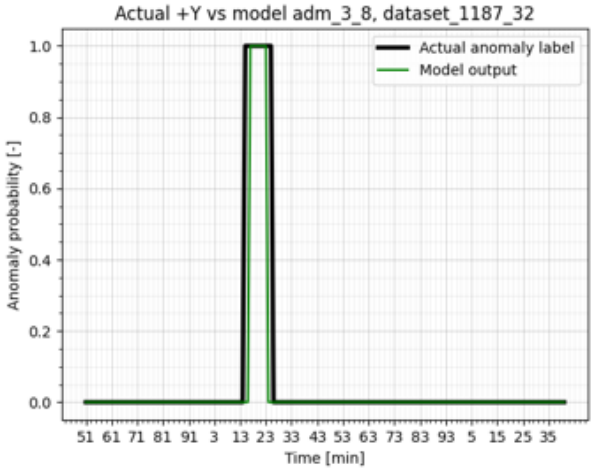


Figure 32: model inferences on dataset_1187_31









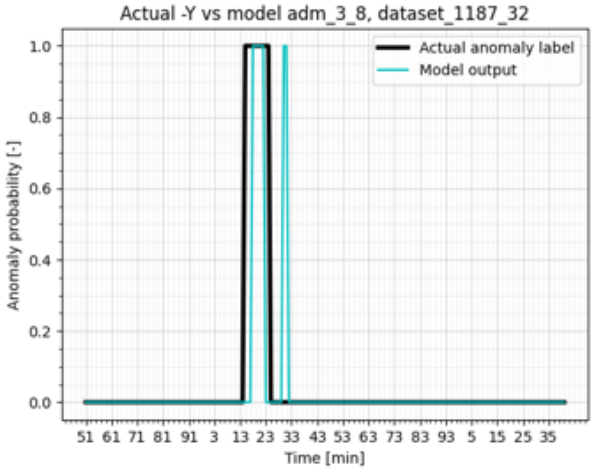


Figure 33: model inferences on dataset_1187_32