POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies

Master's Degree Thesis

Neural Network-Based Classification of Electric Vehicle Acceleration Pedal Signals: From Training to Microcontroller Deployment

Supervisor	Candidate
Prof. LUCA VASSIO	SIMONA CHIURATO
LUCA BUSSI	

October 2023

"Failure taught me things about myself that I could have learned no other way. I discovered that I had a strong will, and more discipline than I had suspected; I also found out that I had friends whose value was truly above the price of rubies. The knowledge that you have emerged wiser and stronger from setbacks means that you are, ever after, secure in your ability to survive. You will never truly know yourself, or the strength of your relationships, until both have been tested by adversity. Such knowledge is a true gift, for all that it is painfully won, and it has been worth more than any qualification I ever earned."

– J.K. Rowling

Summary

In an effort to promote sustainable mobility, electric vehicles have emerged as a crucial innovation in the global transportation sector. This research, undertaken in conjunction with Brain Technologies and its innovative Evergrin project, investigates the use of artificial intelligence in electric vehicles, with a specific focus on the classification of accelerator pedal signals using neural networks. The research explores the application of Tiny machine learning (TinyML) in the automotive industry using the Raspberry Pi RP2040 as the microcontroller of choice.
Key research questions include the ability of neural networks to detect anomalies in pedal signals, performance differences between TinyML models and neural networks, and the trade-off between model latency and accuracy on microcontrollers.
Using throttle position data collected from an internal combustion vehicle's OBD-II system to produce the basic dataset, the study applies several neural network models, including convolutional neural networks (CNNs), long-short term memory (LSTM) and gated recurrent unit (GRU), highlighting their potential in time series classification. The results of the study demonstrate the strong error classification capabilities of neural networks, with all models achieving at least an accuracy of 0.92. The maximum accuracy of 0.96 was reached by the LSTM model with two channels of input data.

While the performance of the TinyML models, obtained through conversion from TensorFlow to TensorFlow lite, is comparable to that of their CNN-based equivalents, the more sophisticated models, such as the LSTM, have problems due to the lack of adequate quantization techniques for their layers. In particular, the

accuracy of the LSTM model dropped to around 0.5. On the other hand, for the two-channel CNN model, the accuracy is the same for both the TensorFlow and TensorFlow lite versions at 0.95.

The complexity of implementing neural networks on microcontrollers was a significant obstacle, especially when considering the safety requirements of the automotive industry. Even the simplest models required more than two seconds for inference, significantly longer than the safety threshold of 100 ms; furthermore, advanced models such as LSTM and GRU exceeded the memory capacity of microcontrollers such as the RP2040, making it impossible to implement said models on them. Given the promising potential for integrating these technologies into the automotive industry, it is believed that neural networks must be implemented on platforms with computational capacity beyond that of microcontrollers for optimal performance and feasibility.

In conclusion, this research emphasises the central role that AI could play in the automotive industry, particularly in the classification of automotive signals. It aims to highlight the need to improve current technologies for optimising and converting machine learning models.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

CNN

Convolutional Neural Networks

cuDNN

CUDA Deep Neural Network

DL

Deep Learning

DSP

Digital Signal Processor

ECU

Engine Control Unit

EV

Electric Vehicle

FCN

Fully Convolutional Networks

GAF

Gramian Angular Field

GAP

Global Average Pooling

GASF

Gramian Angular Summation Field

GP

Gaussian Process

GRU

Gated Recurrent Unit

LSTM

Long-Short Term Memory

ML

Machine Learning

MLP

Multilayer Perceptron

NN

Neural Networks

OBD-II

On-Board Diagnostic

QAT

Quantization-aware training

ReLU

Rectified Linear Unit

RNN

Recurrent Neural Networks

RP2040

Raspberry Pi 2040

tanh

Hyperbolic Tangent

TPS

Throttle Position Sensor

TFLite

TensorFlow Lite

TFLM

TensorFlow Lite for Microcontrollers

TinyML

Tiny Machine Learning

UCB

Upper Confidence Bound

Chapter 1

Introduction

1.1 Scenario

The electric vehicle market is growing and its potential is seemingly endless. Electric vehicles (EVs) have emerged as a beacon of hope for minimizing environmental degradation as the world moves toward sustainable solutions. This thesis explores the technological details of the possibility of using Artificial Intelligence (AI) in electric vehicles, focusing in particular on classifying accelerator pedal signals using neural networks (NNs) and implementing it on a microcontroller through the concept of TinyML.

The worldwide market for electric vehicles is witnessing extraordinary growth. According to the International Energy Agency [1], by 2030, EVs will account for more than 10% of the road vehicle fleet. Total EVs sales will exceed 20 million in 2025 and 40 million in 2030, accounting for more than 20% and 30% of total vehicle sales, respectively (Fig. 1.1). Furthermore, the number of electric car charging stations in Europe has increased significantly, suggesting robust infrastructure development to support the expanding EV ecosystem.
The transition from internal combustion engines to EVs is a visible and ongoing trend in the automotive industry. The European Green Deal, which aims to make Europe carbon neutral by 2050, emphasizes the need for decarbonization. In line

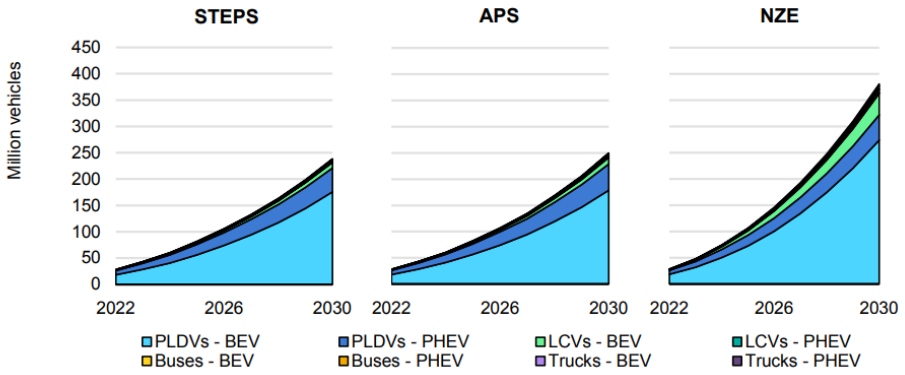


Figure 1.1: Electric vehicle stock by mode and scenario, 2022-2030 [1]
Notes: STEPS = Stated Policies Scenario; APS = Announced Pledges Scenario; NZE = Net Zero Emissions by 2050 Scenario; BEV = battery electric vehicle; PHEV = plug-in hybrid electric; PLDV = passenger light-duty vehicle; LCV = light commercial vehicle.

with this goal, many cities have set limits on cars considered to be major polluters, with a particular focus on diesel engines. The Po Valley in Italy, for example, has already introduced limits on the circulation of Euro 5 diesel vehicles, which were implemented in 2008. Inadvertently, these restrictions have created a huge fleet of vehicles that can be utilized by converting to electric vehicles. Given the size of the fleet and the relatively high cost of brand new EVs, the electric conversion market offers a profitable opportunity.

Enter the *Evergrin* project, led by *Brain Technologies*, an innovative Italian company with which this thesis was carried out. This initiative is further promoted in Italy by Legislative Decree 219/2015 [2], often known as the 'Retrofit Decree'. This legislation not only allows the conversion of internal combustion vehicles to fully electric, but also allows the legal use of these converted vehicles on public roads.

The use of artificial intelligence, particularly through microcontroller implementations, demonstrates the automotive industry's convergence with increasing technological breakthroughs. The conditioning of specific signals that are critical for optimal vehicle performance has changed significantly as electric vehicle systems have evolved. Central to this evolution is the concept of control and functional safety. The need for robust control mechanisms and compliance with functional

safety standards, such as ISO 26262 [3], is paramount as vehicles become more electronically sophisticated. These measures ensure that vehicles operate reliably and safely, even in the presence of potential system failures or malfunctions.
During operation, a modern electric vehicle equipped with sensors and electronic components creates a large amount of data. If properly processed and evaluated, this data can provide insight into vehicle performance, prospective maintenance requirements, and even forecast system faults before they occur. With its data-driven decision-making capabilities, AI is at the forefront of this innovative approach. This thesis focuses on the signal from the accelerator pedal, which will be addressed in more detail in subsequent Chapters.

1.1.1 Functional Safety and Control Mechanisms

In the electric vehicle sector, accurate signal monitoring and control are critical to ensuring both optimal performance and the greatest levels of safety. The concept of functional safety, a standard defined by ISO 26262, is crucial. In the automotive context, functional safety refers to the system's intrinsic capacity to identify, manage, and mitigate possible faults, ensuring that the vehicle works safely even when specific components or systems fail. Adherence to the ISO 26262 standard implies a commitment to stringent safety measures that span the whole life cycle of automotive systems. The importance of functional safety cannot be overstated. As vehicles become more complex electronically, the potential for electronic and software failures increases. These failures can pose significant risks to vehicle occupants and other road users.
The dual-level strategy of primary and secondary controls is an essential part of the functional safety of EVs and is closely related to this research. The vehicle's main interpretive system is the primary control, which immediately translates driver inputs, such as the position of the accelerator pedal, into operational directives. By contrast, secondary control, often referred to as redundant control, is for safety. It runs in parallel, constantly cross-referencing and validating the output of the primary system. Commonly used secondary control technologies include redundant sensors, comparators and in-line monitoring systems. Despite the effectiveness of

traditional approaches, there is growing interest in the use of neural networks for secondary control due to their ability to recognise patterns and detect irregularities. However, the widespread use of NNs in this area is still in its early stages, with challenges related to real-time processing and reliability.

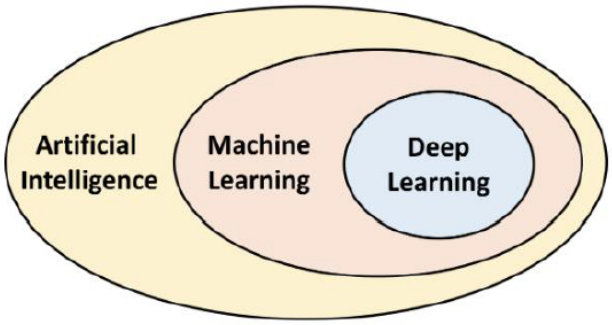


Figure 1.2: Artificial intelligence, machine learning and deep learning paradigm [4]

1.1.2 Machine Learning and Deep Learning

Machine learning (ML), a subset of artificial intelligence, gives machines the ability to learn and adapt. ML algorithms can find patterns, anomalies, and trends in historical and real-time data, enabling proactive intervention to maintain vehicle safety and efficiency. Deep learning (DL), a subset of machine learning, refines the learning process by increasing the depth and complexity of the learning environment [4]. Fig. 1.2 shows the dependencies of AI, ML and DL pictorially.
Deep learning's neural networks are particularly adept at interpreting complicated data structures making them valuable for advanced driver assistance systems and autonomous driving capabilities. Machine learning and deep learning work together to transform computer systems into expert systems. These systems can make decisions and predictions on their own, without the need for human intervention. For example, in the context of accelerator pedal signals in electric cars, these technologies help ensure that the vehicle responds appropriately to driver input while monitoring for anomalies that could indicate potential issues.

In essence, machine learning serves as the framework for implementing decision-making intelligence, while deep learning optimizes and deepens the learning process. As the electric car market expands, the symbiotic relationship between these advanced technologies and automotive engineering will undoubtedly play an important role in determining the future of transportation.

1.1.3 TinyML

In recent years, there has been a paradigm shift in the field of machine learning toward deploying models on edge devices, particularly those with limited computational capabilities. A noteworthy endeavor in this direction was launched in 2014 by a team at Google. Their goal was to run a neural network on a digital signal processor (DSP), which is found in most cellphones. The primary task of this neural network was to recognize the password 'OK Google', a critical feature that allows users to connect to Google's virtual assistant by voice. The difficulty of this task was compounded by a strict constraint: the entire neural network had to fit into 14 kilobytes of memory. To put this in context, typical deep learning networks often require tens of gigabytes, making this a massive task. This constraint is due to the inherent design of DSPs, which typically do not have much internal memory. This Google effort highlights a broader trend in the machine learning community: the search for alternative processors capable of delivering substantial, yet extraordinarily energy-efficient performance. TinyML was conceived as a result of these efforts. TinyML, defined as machine learning models designed for devices with power consumption of less than 1 mW, is an important milestone in edge computing. This seemingly arbitrary power level serves a strategic purpose. Devices that meet this constraint can run for an entire year on a single coin cell battery, representing a breakthrough in energy efficiency [5].
The impact of TinyML goes beyond energy savings. TinyML models improve user privacy by minimizing the need to transmit sensitive information to external servers by processing data locally on the device. In addition, the compact nature of TinyML models results in a smaller overall device footprint, enabling more discrete and portable applications. This feature raises the possibility that TinyML models

could be adopted by edge computing.
TinyML implementations are still in the early stages of development. While progress has been made, the field still faces challenges ranging from improving models to ensuring consistent performance across multiple devices.

1.2 Research Questions

Within the context of AI deployed on microcontroller in the automotive sector, this thesis aims at answering the following relevant points:

- **Effectiveness of neural networks in anomaly detection.** How effective are neural networks in detecting anomalies in accelerator pedal signals?
  - To investigate this, several NNs models were developed and trained. The main objective is to assess their accuracy in classifying accelerator pedal signals, in particular to distinguish signals without errors from those with anomalies.
- **Performance comparison between TensorFlow Lite and larger models.** What are the performance disparities, if any, between TensorFlow Lite models and their more extensive neural network counterparts?
  - After establishing baseline performance with the initial NNs models, these models are converted to TensorFlow Lite models. The aim is to compare the performance metrics of the TensorFlow Lite models with those of the original, larger NNs models, highlighting any notable changes in performance.
- **Balancing latency and accuracy.** Can a balance be achieved between the latency of neural network models on microcontrollers and their performance accuracy?
  - High latency, which can be a problem for large NN models, is an important issue that has emerged. Given the stringent safety criteria of the automotive industry, where real-time responses can be critical, the

question arises as to whether these models, once translated into TinyML models, can meet the specified latency or whether further optimization is required. The problem is to reduce this latency without compromising the accuracy of the model.

1.3 Contributions

The study has made noteworthy contributions to electric vehicle signal processing and TinyML, with a notable highlight being the collaboration with Brain Technologies. In the *Evergrin* project, this research not only advances academia but also ensures that the findings have direct real-world application in the fast-developing EVs conversion industry. Due to data constraints, an innovative method was developed to replicate EVs pedal signals using throttle position data from conventional internal combustion engine vehicles. This pioneering data collection and preprocessing approach can serve as a model for other researchers facing similar challenges.

A substantial portion of this study was dedicated to optimizing advanced NN models for microcontroller implementation. It represents the intersection of advanced machine learning and real-world applications with constrained resources. This thesis emphasizes the feasibility and value of resource-efficient AI solutions in the automotive sector, providing as a testimonial to TinyML's capabilities and clearing the path for its broader adoption in the automotive industry.

The careful selection of appropriate hardware for TinyML in automotive applications is a crucial aspect of this research. Although TinyML has great potential in the automobile sector, the study found that its implementation on certain microcontrollers, such as the RP2040, may fall short of the industry's safety-critical latency requirements. Conversely, analysis using the Edge Impulse's Python SDK reveals that more capable microcontrollers could obtain inference times closer to the desired threshold. This emphasizes the importance of selecting the right hardware platform when deploying TinyML models in safety-critical environments.

1.4 Thesis Overview

This thesis investigates the applicability of artificial intelligence in the context of electric vehicles, with a particular emphasis on the classification of accelerator pedal signals using neural networks. The structure of the thesis is organized as follows:

- **Chapter 1, Introduction**: This first Chapter presents an overview of the electric vehicle market, highlighting its development and future prospects. It explores the possibilities of applying artificial intelligence in EVs and underlines the critical importance of functional safety and control mechanisms in the automotive sector.

- **Chapter 2, Literature Review**: A brief summary of existing neural network architectures for signal classification, including convolutional neural networks and recurrent neural networks, is provided. Furthermore, the Chapter explores into the complexities of implementing these complex architectures on microcontrollers using TinyML.

- **Chapter 3, Data Acquisition and Preprocessing**: the complex method of generating acceleration signals from throttle valve position data is presented. A thorough examination of possible errors that can affect the signal is included.

- **Chapter 4, Neural Network Architectures for Signal Classification**: This Chapter digs into the details of the implemented CNN and RNN models, addressing their structures and applications in time series classification.

- **Chapter 5, Models Optimization**: the optimization of hyperparameters, input signal normalization, and final optimized architectures for neural network models are explored.

- **Chapter 6, TFLite conversion and Deployment on Microcontroller**: This Chapter covers the step-by-step process of converting a TensorFlow model to the TensorFlow Lite format, moving from optimization and quantization, and finally to deployment on a microcontroller.

- **Chapter 7, Discussion and Comparison of Results:** This Chapter rigorously evaluates the performance of the implemented neural network models in identifying errors in accelerator pedal signals. The performance of the TFLite models is compared with the results of the TensorFlow counterparts, showing significant differences. The feasibility of deploying these models on a microcontroller is then investigated. The balance between model latency and accuracy is a crucial challenge, especially given the inference time requirements of this study.
- **Chapter 8, Conclusions and Future Work**: the final Chapter of the thesis summarizes the study findings and proposes new topics for future investigation.

Chapter 2

Literature Review

This chapter provides an overview of the topics explored in this thesis. Neural networks are frequently employed for signal classification, and a plethora of research has been conducted in this domain. The decision to classify the signal stems from the need for the model to determine whether the input is error-free or contains errors. Concurrently, the deployment of machine learning models on microcontrollers represents a nascent yet rapidly evolving domain.

2.1 Neural Network Architectures for Signal Classification

2.1.1 Convolutional Neural Networks

Due to their innate capacity to recognize and capture the complex structures of time series data, convolutional neural networks (CNNs) have become a crucial architecture for time series classification. Traditional feature extraction approaches may fail to capture intrinsic patterns and correlations due to the dynamic nature of time series data, which is generally characterized by high dimensionality, huge datasets, and constant updates. In contrast, CNNs can automatically extract deep features from the input time series through their convolution and pooling operations.

A solid starting point for time series classification using deep neural networks is introduced in research by Zhiguang Wang, Weizhong Yan, and Tim Oates [6]. The study emphasizes the ability of fully convolutional networks (FCNs) to handle time series data. Traditional time series classification methods have frequently relied on distance-based, feature-based, or ensemble-based approaches. These methods often necessitate extensive data preparation and feature engineering. However, the recent shift toward deep neural networks, specifically CNNs, has enabled end-to-end time series classification. On benchmark datasets, the study compares the performance of various models, including Multilayer Perceptrons (MLPs), FCNs, and residual networks.

The architecture and efficiency of the FCN model, in particular, stand out. It works as a feature extractor, with the final output resulting from a softmax layer. The FCN model's fundamental building block, studied by Wang, Yan and Oates, is a convolutional layer, followed by a batch normalization layer and a ReLU activation layer. Three 1-D kernels (without striding) are used to achieve the convolution. The network is built by stacking three convolution blocks, each with a filter of size 128, 256, and 128. Following the convolutional blocks, the features are routed through a global average pooling layer, considerably lowering the amount of weights. A softmax layer then creates the final label. This study makes a strong case for the use of deep neural networks, particularly FCNs, in time series classification. The FCN model, with its own architecture and design, outperforms other state-of-the-art methods, making it a viable option for real-world applications and a platform for future studies in the field.

Another CNN framework specifically designed for time series classification is presented in the research 'Convolutional Neural Networks for Time Series Classification' by Zhao and Lu [7]. Eight real data sets from various application domains and two simulated data sets were used by the authors to rigorously assess the proposed framework. The empirical findings demonstrated that, when compared to contemporary time series classification approaches, the suggested CNN architecture performed better in terms of classification accuracy and noise resistance.

The suggested CNN structure is built around the combination of convolution and pooling processes, which allows for the extraction of deep features from raw data.

These features then communicate with a MLP to do classification. This study's findings not only support the success of CNNs in time series classification, but also highlight the potential of deep learning paradigms to supplement, if not replace, existing feature-based approaches. Zhao and Lu's empirical results imply that their CNN framework holds potential as a benchmark technique for future efforts in time series classification research.

2.1.2 Activation Functions

In neural networks, activation functions are crucial in determining the output of a neuron given a collection of inputs. They bring nonlinearity into the network, allowing it to learn from errors and make corrections, which is necessary for learning complicated patterns. The Rectified Linear Unit (ReLU) and the hyperbolic tangent (tanh) functions stand out among the numerous activation functions due to their specific features and applications.

The ReLU activation function, in particular, has received a lot of attention and is commonly employed in deep learning architectures, particularly convolutional neural networks. ReLU is denoted mathematically as:

$$f(x) = \max(0, x) \tag{2.1}$$

The relevance and efficiency of the ReLU activation function in deep neural networks are comprehensively investigated in the article "Deep Learning using Rectified Linear Units (ReLU)" by Agarap [8]. Because of its ability to add non-linearity without suffering from the vanishing gradient problem, ReLU has traditionally been a popular choice as an activation function in the hidden layers of deep neural networks. Agarap's research dives into the mathematical foundations of the ReLU function, highlighting the qualities that make it suitable for training deep architectures. The paper stresses the mathematical rationale behind ReLU's broad adoption in the deep learning community, as well as its significance in driving current neural network performance.

On the other hand, the hyperbolic tangent function [9], often known as tanh, is a mathematical function that appears in hyperbolic trigonometry equations. The

tanh function is similar to the logistic sigmoid function, however it distinguishes itself by being zero-centric, making it more balanced around the origin. It is mathematically defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

The function translates any real integer to the interval $[-1,1]$, making it particularly useful as a neural network activation function. Its S-shaped curve is centered on zero, which implies that negative or positive inputs will be mapped to strongly negative or strongly positive, respectively, and zero inputs will be close to zero in the output. This feature allows for more balanced activations during training, which can result in faster convergence. Furthermore, the tanh function is always differentiable, which is useful for the backpropagation algorithm. The tanh function, along with the logistic sigmoid function, was widely employed in the early days of neural networks.

The visual representations of the ReLU and tanh activation functions are shown in Fig. 2.1. Both ReLU and tanh activation functions have particular benefits in the setting of neural networks and have proven essential in the evolution of deep learning approaches.

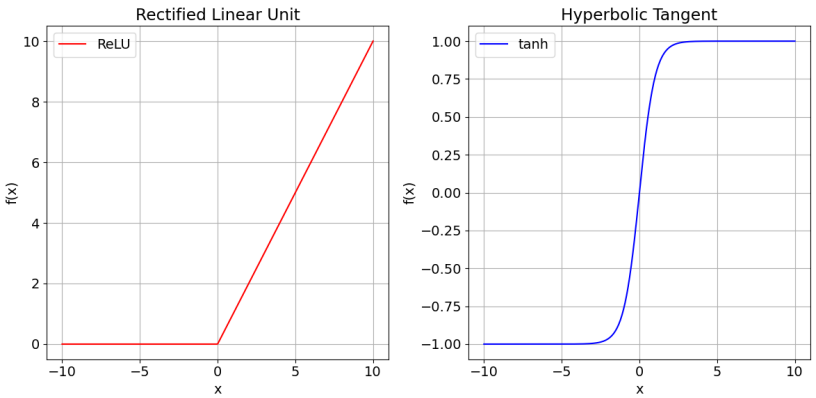


Figure 2.1: ReLU and tanh activation functions.

2.1.3 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is an artificial neural network that identifies patterns in data sequences, such as time series or natural language and it can handle variable-length sequence inputs. Unlike standard feedforward neural networks, RNNs contain connections that loop back on themselves, enabling information to be retained. This is achieved through a recurrent hidden state, which is activated at each time interval and is dependent on the previous time's activation. This looping mechanism enables RNNs to maintain a type of memory of former inputs in their internal state, facilitating the interpretation of data sequences and the identification of temporal connections. However, the basic RNNs may encounter problems in preserving long-term dependencies due to the vanishing or exploding gradients [10]. To overcome these difficulties, more advanced designs of RNN, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), have been developed. This models incorporate gating mechanisms to control the flow of information through the network. These gates aid in the resolution of the vanishing gradient problem and enable models to capture long-term dependencies in sequential data.

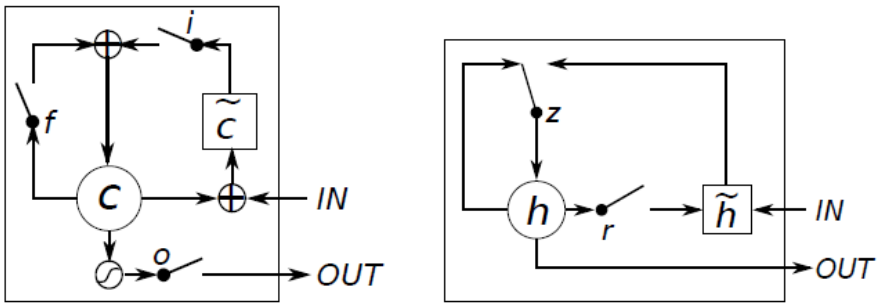
Hochreiter and Schmidhuber (1997) [11] introduced the Long Short-Term Memory architecture, which is designed specifically to retain information over extended temporal intervals. Traditional techniques, that rely on recurrent backpropagation to store information, often suffer from prolonged learning periods due to reduced error backflow. The LSTM model ingeniously overcomes this challenge. As outlined in their groundbreaking study, LSTMs demonstrate a higher level of proficiency in comparison to other recurrent neural network architectures. Particularly, they effectively handle noise, embrace dispersed representations, and manage continuous values in cases with considerable time lags. Furthermore, LSTMs eliminate the necessity for pre-defined finite state numbers, illustrating their flexibility in accommodating an unlimited number of states. Within the framework of LSTMs, there are several mechanisms that play a significant role:

- Input Gate (i): determines how much new data should be stored in the memory cell. It employs a sigmoid activation function to generate values between 0

and 1, which represent the amount of information to be allowed through.

- Forget Gate (f): determines whether or not to discard a portion of the current memory cell content. It also employs sigmoid activation.
- Output Gate (o): determines how much of the current state of the memory cell should be output to the hidden state. Again, a sigmoid activation function is used.
- Memory Cell (c): despite not being a gate, is an essential component of LSTMs. It acts as the LSTM's 'long-term memory', maintaining information across long sequences. In this cell, the input, forget, and output gates collaborate to regulate and update the information stored.
- New Memory Cell Content ($\tilde{c}$): it is a temporary value calculated from the current input and the prior memory cell state. This new content may be added to the current memory cell (c) to update its state, depending on the decisions made by the input and forget gates. It uses a hyperbolic tangent function (tanh).

The activation function of the LSTM unit is defined as $h_t^j = o_t^j \tanh(c_t^j)$.
In Fig. 2.2 (a) is reported the graphical illustrations for the LSTM model.



(a) Long Short-Term Memory (b) Gated Recurrent Unit

Figure 2.2: Illustration of (a) LSTM and (b) GRU [12].

The Gated Recurrent Unit, as explained by Chung et al. (2014) [13] in their seminal paper 'Empirical Evaluation of Gated Recurrent Neural Networks on Sequence

Modeling' uses gating mechanisms to manage information flow inside the network as LSTM. These gates allow the GRU to capture long-term dependencies in sequential data more effectively than ordinary RNNs. The study by Chung and colleagues provides a rigorous empirical evaluation of the GRU's usefulness in various sequence modeling tasks, emphasizing its efficiency and potential advantages over other RNN architectures. The fundamental mechanism inside the GRU model are:

- Update Gate (z): this is similar to a mix of the LSTM's input and forget gates. It determines how much of the prior hidden state is to be kept and how much new information is to be introduced. It employs a sigmoid activation function.

- Reset Gate (r): this gate specifies how much of the previous concealed state should be forgotten. It is used to calculate the new candidate activation.

- Activation (h): it represents the GRU's memory and serves as the unit's output for that time step. The activation is the result of a combination of the prior concealed state and the candidate activation, which is controlled by the update gate (z).

- Candidate Activation ($\tilde{h}$): this is a possible new memory or state for the GRU. It is computed using the current input and the prior hidden state, however the previous hidden state is modulated before being utilized in this computation by the reset gate (r). The candidate activation suggests what the GRU's new memory should be, but whether it is adopted is determined by the update gate. It uses a hyperbolic tangent function.

In Fig. 2.2(b) is reported the graphical illustrations for the GRU model.

2.2 Deployment on Microcontroller - TinyML

TinyML is a paradigm that enables machine learning on battery-powered embedded edge devices with minimal processing power and memory and a few milliwatts of power consumption. It entails the application of architectures, frameworks,

techniques, tools, and approaches that include machine learning to perform on-device analytics for a variety of sensing modalities. It offers a wide range of practical applications, including speech recognition, picture identification, autonomous cars, anomaly detection, and others [14]. TinyML is made up of three major components: software, hardware, and algorithms.

1. Software: frameworks, libraries, and tools that allow developers to construct and deploy machine learning models on low-resource devices. TensorFlow [15], TensorFlow Lite [16], and the Edge Impulse [17] cloud service were used in this thesis.

2. Hardware: the physical equipment that operate machine learning models. Microcontrollers, system-on-chip (SoC), and Internet of Things (IoT) devices are examples of battery-powered embedded edge devices with limited computing capability and memory. A Raspberry Pi RP2040 [18] was chosen as the platform for this thesis.

3. Algorithms: This category comprises machine learning algorithms that are used to train and execute machine learning models on limited-resource devices. These algorithms are lightweight and efficient in order to run on devices with limited processing power and memory. Model compression and quantization are techniques used to minimize the size of machine learning models while preserving their accuracy.

Chapter 3

Data Acquisition and Preprocessing

3.1 Input Data

Due to the development stage of the *Evergrin* project, direct access to the acceleration pedal signals fed to the controller was not available. As a result, the inputs to the neural network models were generated using mathematical transformations applied to a time series of the throttle valve position of an internal combustion engine vehicle. The raw time series data was carefully collected by scanning the throttle position sensor (TPS) through the on-board diagnostic (OBD-II) system. OBD-II is a sophisticated vehicle diagnostic technology that allows users to retrieve fault logs and recorded data from various vehicle systems. This retrieval is aided by an OBD-II reader, also known as a diagnostic scanner or scan tool.

The throttle position sensor is a critical component of the vehicle sensors that regulates the air intake into the engine. As an integral part of the fuel management system, the TPS is critical in providing the ideal mixture of fuel and air, both of which are required for the engine to function. The adjustment of the throttle opening is the operating element of the TPS.

The pressure that the driver applies to the accelerator pedal has a direct effect on the position of the throttle valve. When the driver fully depresses the pedal

for maximum acceleration, the throttle valve opens completely. Conversely, if the pedal is not touched and is fully released, the throttle valve remains closed. This operating paradigm highlights the correlation between the position of the throttle in a combustion engine vehicle and the signal voltage an electric vehicle sends to the engine control unit (ECU) when the accelerator is depressed.

3.1.1 Generation of the Acceleration Signal from Throttle Valve Position Data

The first stage in translating the raw accelerator position information into a representation of the acceleration signal was pre-processing. The raw data, which included time stamps corresponding to the accelerator positions, was modified and the time stamps were changed to a simpler unit of milliseconds. These timelines were modified to start from zero to ensure a consistent starting point for subsequent analysis. Given the presence of rapid, transient oscillations or spikes in the raw data, sometimes caused by sensor errors or external disturbances, it was critical to resolve these inconsistencies in the accelerator position data. To this end, a smoothing technique based on a moving average filter with a ten-sample window was used. This filtering method performed two functions: first, it efficiently smoothed out these high-frequency transient disturbances, resulting in less jagged and noisy data. Secondly, by reducing these variations, the filter emphasised the true behaviour or primary tendency of the throttle position over time, resulting in a clearer and more accurate representation of the data.

After smoothing, the data was differentiated to determine the rate of change of throttle valve position over time. The rate at which the throttle valve was activated or released was effectively measured by the first derivative. A second differentiation was performed to obtain the acceleration of the throttle valve position to further investigate the complicated nature of the throttle behaviour. This generated data which showed how the rate of change of throttle valve position changed over time. The second derivative was integrated using the trapezoidal rule. Due to its cumulative nature, this numerical integration produced a signal that provided a representation of how the throttle valve position has changed over time, given

the acceleration of the throttle valve position. This integrated signal, rich in information about the throttle response over time, served as the basis for further modifications.
The process culminated in the embedded signal being subjected to a series of modifications and scaling. It was first modified to have a new baseline aligned to a preset zero point. The zero point varies with each raw data set and is set when the raw data indicates that the throttle is fully closed. Following this modification, the signal was scaled to ensure that the required voltage ranges were maintained, specifically $[0,5]V$ and $[0,10]V$. This is because the accelerator pedal in *Evergrin* is configured to communicate the same information on two similar signals with different voltage limits. This feature was later used in the construction of the NN architectures.

Random Gaussian noise has been introduced into both rescaled signals to better match real-world conditions and provide a sense of authenticity. This feature not only approximated possible measurement errors, but also ensured that the synthesized signals corresponded to the characteristics of real, imperfect data. Given its numerous advantages, Gaussian noise is an excellent choice for research, with the objective of potentially studying different types of noise in the future to assess their faithfulness to real-world settings. Gaussian noise is prevalent in many natural systems, making it an appropriate representation for the plethora of little random disturbances present in real-world data. The Central Limit Theorem asserts that a sum of several independent random variables frequently converges to a Gaussian distribution, confirming the Gaussian distribution as a common noise model for a wide range of real-world scenarios. Furthermore, adding Gaussian noise improves the durability of models trained on such data. When exposed to this noise, these models improve their ability to recognize patterns, increasing their resistance to real-world variations.

The original data refers to four different car journeys totalling 107.7 minutes for the whole dataset. The four couple signals obtained from the original data sets are shown in Figs. 3.1, 3.2, 3.3, and 3.4.

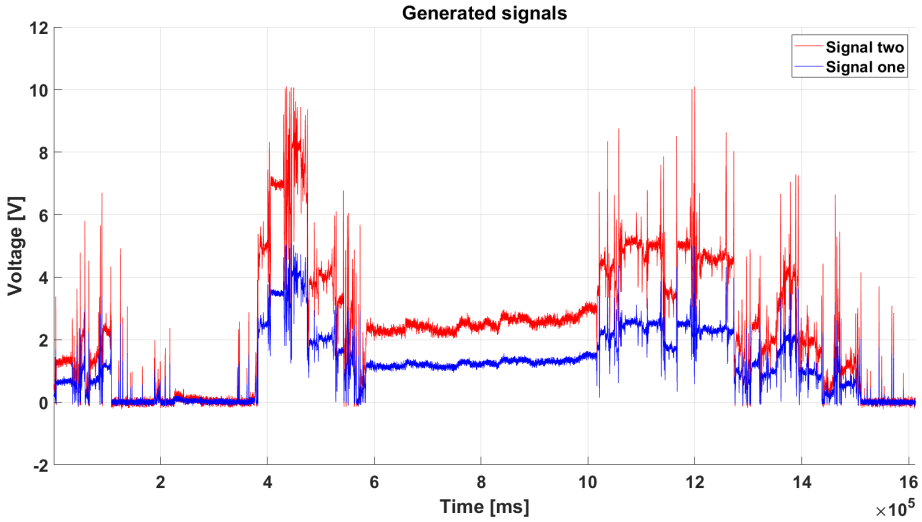


Figure 3.1: Acceleration signal generated from throttle valve position – first dataset.

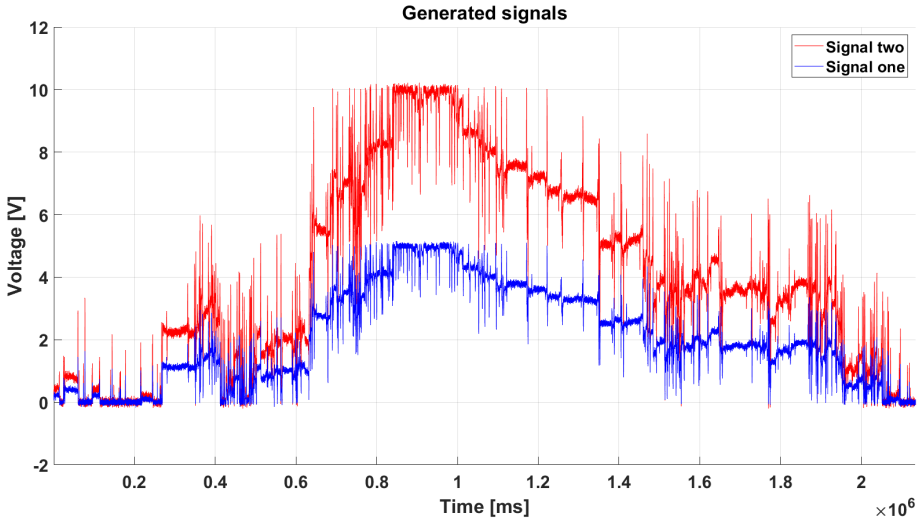


Figure 3.2: Acceleration signal generated from throttle valve position – second dataset.

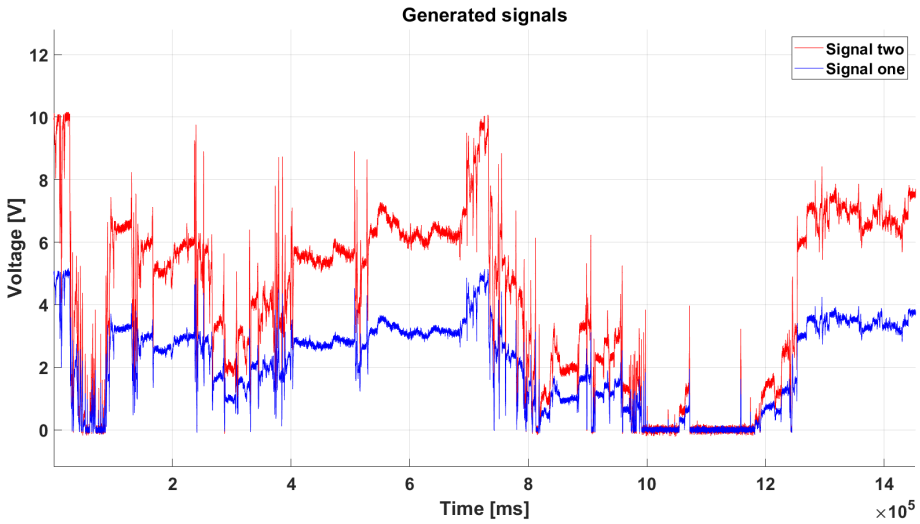


Figure 3.3: Acceleration signal generated from throttle valve position – third dataset.

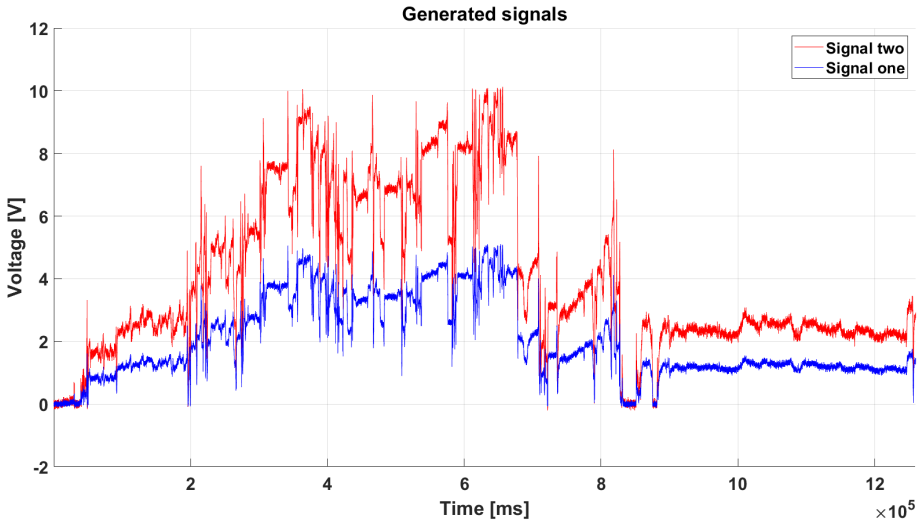


Figure 3.4: Acceleration signal generated from throttle valve position – fourth dataset.

3.2 Error Analysis

The integrity and precision of input signals create the foundation for trustworthy and accurate results in the field of neural network-driven system control. The signals generated thus far in the thesis workflow are meant to be labeled as positive for neural network classification. These signals describe the essential condition of a vehicle that has no evidence of functional faults. However, in order to provide comprehensive and resilient training, it is critical to expand beyond these ideal conditions and investigate errors and discrepancies that may occur in real-world contexts. The error analysis performed focused on the incorporation of the conventional functional safety method in the thesis work. The automotive sector follows tight standards – such as ISO 26262 [3] – to assure the safe operation of electronic and electrical vehicle systems even in the presence of errors.

Fifteen discrete error types have been established to structurally summarize the approach used. This results in a combinatorial set of 225 potential error possibilities in the dual-signal framework. The first signal went through a segmentation process, producing fifteen different partitions. After that, each of these partitions was subjected to one of the fifteen error perturbations listed below. Similarly, the second signal was split into the same fifteen primary segments, with further subdivisions made inside each of these main segments to accommodate all fifteen error perturbations consecutively. As a result, each main segment of the second signal displays the whole error spectrum, with each error happening in its own temporal segment.

3.2.1 Error Generation and Functional Safety

Sensor redundancy is the foundation of functional safety: traditional vehicles utilize two sensors to monitor important components such as the accelerator pedal signals. Divergence in these sensors' values indicates probable errors. Consequently, when the duplicate signals of the *Evergrin* prototype indicate disparity, following the logic of the duplicate sensor anomalies, an error is identified. The collection includes the

following error types:

- **Signal shifted with positive mean**: This error causes the entire signal segment to be increased by its mean value. It shows circumstances in which an offset could be introduced to the signal, causing its magnitude to increase.
- **Signal shifted with positive mean and random error**: Adds Gaussian noise after the mean shift based on the prior error. It simulates the system's combined impacts of offset and unexpected noise.
- **Signal shifted with negative mean**: Unlike positive shift, this error subtracts the signal segment's mean value, resulting in a downward shift. It represents potential calibration or grounding issues that could cause signal offset.
- **Signal shifted with negative mean and random error**: Gaussian noise is used to simulate the combined negative offsets and system noise as before.
- **Signal at voltage extremes**: Signals can saturate in certain conditions. To explore such saturation effects, this error sets the signal to its maximum, either $5V$ or $10V$.
- **Signal cancellation**: This error simulates scenarios in which signal transmission is fully lost or halted, resulting in data gaps, by setting the entire segment to zero.
- **Signal scaling**: Signals can be attenuated for a variety of reasons, including resistance or interference. Each signal is rescaled by eight distinct errors with factors ranging from 0.01 to 0.09 (excluding 0.05 which, given the signal processing performed to create the database, corresponds to the scaling factor of the positive label). Each scaling attenuates the signal segment, imitating varying degrees of signal degradation.
- **Random signal cancellation**: The transmission is randomly cancelled in order to examine irregular losses. This replicates occasional signal transmission losses or drops.

Starting with the established error types, it is important to understand the broader implications between the errors generated and the corresponding role of functional safety functions. It is well established that the usage of redundant signals to detect a single parameter is critical for fault detection. The signals sent by the vehicle's accelerator pedal are typically within predefined voltage ranges. Sudden and unexpected variations in signal values can be replicated by introducing Gaussian noise, emulating fast transitions. The incorporation of this error allows the NN to be trained to closely monitor both the signal spectrum and its rate of change.
Plausibility evaluation, which includes comparing associated signals to discover contradictions, is another feature of the security standard. The *Evergrin* prototype's dual signals provide an inherent framework for such analyses. Plausibility discrepancies are inconsistencies that can be reflected as inconsistent scaling coefficients or shifted means within the voltage domains.
It is important to notice that in vehicle systems malfunctions can pose substantial risks. This is why emergency mechanisms, or fallback mechanisms, are used, which are essential to ensure that the system maintains its safety parameters, even in the presence of adversity. In the absence of an additional system, if an acceleration sensor in a vehicle malfunctions, the vehicle could experience unintended acceleration or a complete lack of propulsion. Therefore, anomalies such as signal scaling may be perceived as a simulation of diminished effectiveness and attenuated signals or excessively accentuated signals.
Furthermore, the ECU is responsible for the consistency of communication with the sensors. Anomalies in which signal segments are sporadically cancelled can be interpreted as sporadic communication gaps or interruptions.
To illustrate the error mechanisms introduced, Figure 3.5 shows an example of the representative signals modified from their original form. The first signal has been rescaled and its amplitude altered by a factor of 0.07. The second signal shows a more complex transformation. It has been subjected to the full set of fifteen different types of error explained above. These errors include mean shifts in both positive and negative directions, the introduction of Gaussian noise, amplitude restrictions limited to 10V or cancelled at 0V, scaling by various factors and random cancellation of certain data points. This example is intended to illustrate

the robustness and comprehensiveness of the database created, which includes all 225 possible error combinations.

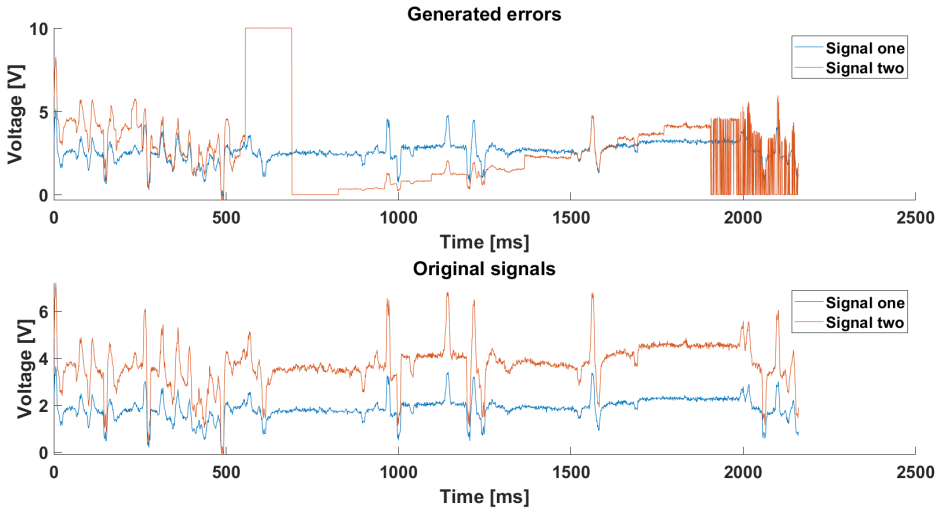


Figure 3.5: Example generated errors from original signals.

Chapter 4

Neural Network Architectures for Signal Classification

The generated data signals and their corresponding errors form the foundation for the construction of a robust database suitable for further research. A distinctive feature of the *Evergrin* project is its dual-signal nature. This useful feature prompted an investigation into the performance of the neural networks under different data input methods. To facilitate this, all models were implemented using Python, utilizing the TensorFlow [15] library and the Keras API [19]. In particular, there was an interest in understanding the network's behaviour when the data is presented as a single signal versus when it's divided into two separate channels. In order to systematically evaluate these scenarios, two different databases were formulated. The first merges the two signals sequentially, effectively doubling the temporal length of the input data compared to the original. In contrast, the second database introduces the two signals simultaneously, preserving their individual identities and presenting them in parallel. This dichotomy applies to almost all neural networks architectures studied. It provides a comprehensive framework for understanding the nuances of NNs performance in different data representation paradigms.

4.1 Convolutional Neural Networks

The initial architecture selected for investigation was the Convolutional Neural Network (CNN). The initial focus was on its one-dimensional layers (Conv1D) used for time series data, and the model was tested using both the one-channel and two-channel databases. In addition, the use of two-dimensional convolutional layers was explored (Conv2D). This was achieved by transforming the 1D input time series into 2D inputs using the Gramian Angular Field method.

4.1.1 One-dimensional CNN

The neural network model under investigation is primarily intended for the classification of accelerator pedal inputs. The architecture starts with an input layer designed specifically for the needs of the data structure. Data samples with dimensions of $batch\_size \times 30 \times 1$ are supported for the single-channel model, where each sample represents a sequence of 30 time-steps, each separated by an average of 6 ms, with a single feature. As a result of the concurrent processing of the two signals, the input dimension for the two-channel model is $batch\_size \times 15 \times 2$. Beyond this difference in input dimensions, the models' structures are the same.

The three main processing layers are designed as Conv1D layers. Convolutional layers are essential for finding local patterns in time series data. Each of these layers contains a set of hyperparameters, such as the number of filters, kernel size and the activation function, which will be decided during the future hyperparameter tuning phase. In addition, each convolutional layer is followed by a batch normalization layer to ensure stable and fast convergence during training. Batch normalization modifies neural network activations to have a mean close to 0 and a standard deviation close to 1. This prevents internal covariate shift by stabilizing the input distributions to each layer during training. Covariate shift refers to changes in the distribution of input data between the training and testing phases, whereas internal covariate shift relates to changes inside the training process itself. Training becomes quicker, more consistent, and less reliant on weight initialization by reducing these

fluctuations.

The data is delivered to a Global Average Pooling layer after being transformed through the convolutional layers. Global Average Pooling (GAP) computes the average output of each feature map in the preceding layer, as opposed to the traditional Max Pooling or Mean Pooling layers. This helps to reduce the spatial dimensions of the feature maps while maintaining their depth, ensuring that the important temporal characteristics are preserved while minimizing computation overhead. The GAP layer decreases the number of parameters significantly, making the model less prone to overfitting and more suited for deployment on resource-constrained devices such as microcontrollers.

Depending on the optimal outcomes of the hyperparameter optimization, an optional Dropout layer can be added after the GAP layer. Dropout is a regularization approach in which a portion of the neurons in the layer are 'dropped' or deactivated at random during training. This prevents a single neuron from becoming highly specialized, resulting in a more generic and resilient model.

A Dense layer with two neurons and a softmax activation function performs the final transformation. The use of two output neurons in conjunction with the categorical crossentropy loss function is intentional. While a binary classification task might normally be accomplished with a single neuron and binary crossentropy, this particular structure facilitates the implementation on microcontrollers via platforms such as Edge Impulse. The softmax activation guarantees that the output values are probabilities that sum up to one, indicating the two classes' confidence evaluations.

To avoid exploding gradients, the model employs the Adam optimizer, a popular adaptive learning rate optimization algorithm, with a clip value of 1.0, which means that every component of the gradient vector is clipped to lie between -1.0 and 1.0. So, if a particular value in the gradient vector exceeds this range, it is set to the maximum or minimum value (i.e. 1.0 or -1.0). The categorical crossentropy loss function is used to assess the difference between predicted and

true probabilities, and accuracy is the major metric used to assess the model's performance on validation data.

4.1.2 Two-dimensional CNN

The research then extended to convolutional neural networks using two-dimensional layers (Conv2D). The transition to a two-dimensional representation was achieved using the Gramian Angular Field (GAF). In essence, the GAF captures the temporal correlation between different time points in the series and visualizes them in a 2D space.

Once the data is transformed into its 2D representation, it is shaped to support the one channel structure. The core architecture involves a series of Conv2D layers, designed to extract patterns and features from these 2D representations. Each convolutional layer, followed by a Batch Normalization layer, has hyperparameters set for optimization during the tuning phase.
The network subsequently utilizes a Global Average Pooling 2D layer, which condenses the spatial dimensions of each feature map, preserving the depth. Depending on the results of hyperparameter optimization, a Dropout layer with a rate of 0.2 might be integrated to enhance model regularization.
The architecture culminates in a Dense layer with two neurons, employing a softmax activation. The model's optimization strategy hinges on the Adam optimizer, and its performance is evaluated using the categorical crossentropy loss function, with accuracy as the primary validation metric.

In addition to the architecture described above, an alternative model that combines data augmentation techniques immediately after the input layer has been developed. These augmentation techniques can incorporate variances in the training data to improve the model's ability to generalize to new and previously unseen data points. The model can perform random flips both horizontally and vertically, random rotations up to a factor of 0.1, and/or a random zooming up to a factor of 0.1, using the appropriate Keras layers. The hyperparameters chosen during the optimization

phase determine the use of these data augmentation strategies.
By including these levels, the model can benefit from a more diversified collection of training examples, thereby increasing its robustness and ability to generalize across different inputs. When working with restricted datasets, data augmentation is especially useful since it artificially expands the training data pool, potentially leading to higher model performance. The purpose was to examine the feasibility of extracting new information from previously changed data using GAF.

Thus, the general structure of the models is similar to that already used for one-dimensional CNNs. The two-channel version was not pursued because, as will be better explained in the Chapter 7, models with 2D input data do not perform well with the type of time series used in this research. In addition, loading 2D images into memory is not practical given the limited computational resources of microcontrollers.

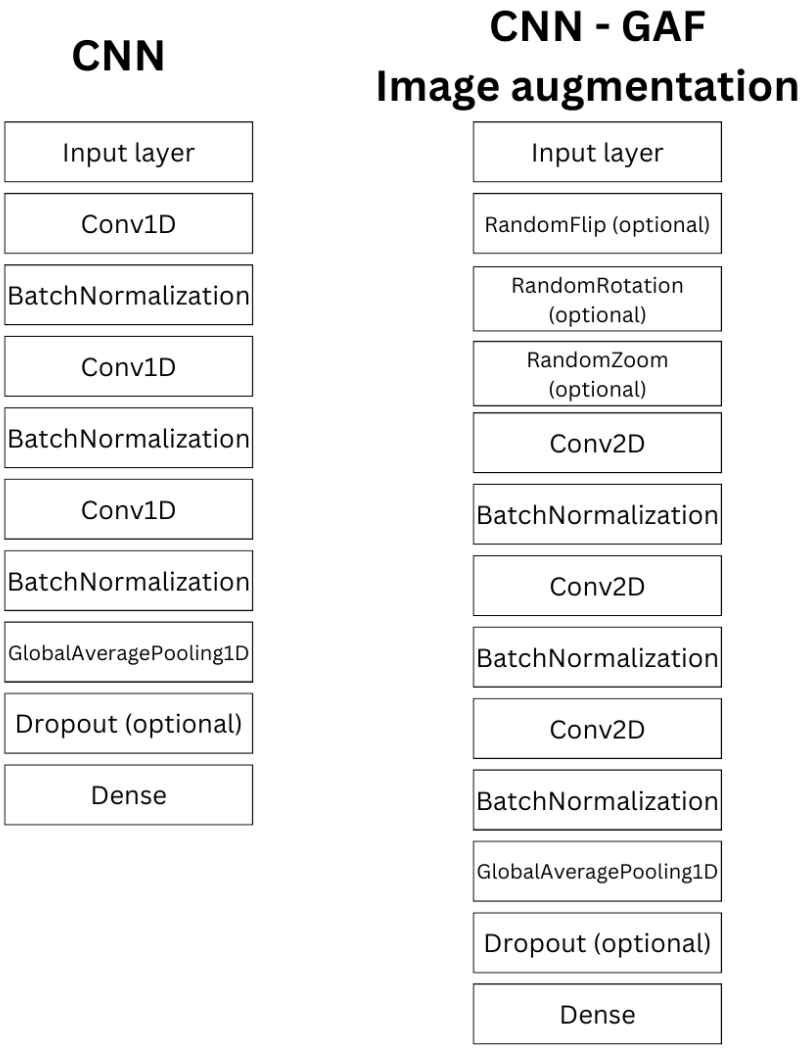


Figure 4.1: Architectures CNN models.

Figure 4.1 illustrates the general architectural structures of CNN models. The left side outlines the architecture employed by CNNs s with one or two input channels. In contrast, the right side presents the custom architecture for CNNs s using GAF images as input. Specifically, the structure depicted incorporates layers of data augmentation. If the initial three data augmentation layers are omitted, the architecture without data augmentation is obtained.

Gramian Angular Field

The Gramian Angular Field (GAF) [20] is gaining popularity as a sophisticated method for converting one-dimensional time series data into two-dimensional image-like structures. Such a transformation is particularly important in the context of deep learning. Given that CNNs are designed especially to analyze two-dimensional data, such as images, the GAF provides a way to harness the power of CNNs for time series data.
The GAF, in its essence, represents the complicated temporal correlations inherent in time series data. This is accomplished by coding the mutual correlations between different time points as angles. This novel encoding method is based on the representation of time series data in polar coordinates. While the magnitude of each data point – its distance from the origin – remains unchanged in this transformation, its phase, or angular position relative to the origin, reflects the essence of its value.

Building on this transformed series, the Gramian matrix is constructed. The specific elements of this matrix are determined by the Gramian Angular Summation Field (GASF). For the implementation, the `pyts` package — a Python library dedicated to time series classification — is utilized [21]. Specifically, the class `pyts.image.GramianAngularField` is employed. The matrix components encapsulate the cumulative angles between pairs of points in the series.

From a mathematical point of view, the `fit_transform(x)` method performs three important steps:

1. **Normalization** Suppose to have a time series $X = \{x_1, x_2, \ldots, x_i, \ldots, x_N\}$, containing N observations. Firstly, X is normalized so that all values of X can be in the range of $[-1, 1]$ which can be expressed as follows:
$$\tilde{x}_i = \frac{(x_i - \max(X)) + (x_i - \min(X))}{\max(X) - \min(X)} \tag{4.1}$$
2. **Calculate the polar coordinates** The following step define the inverse cosine angle, ϕ, from the normalised amplitude values and the radius, r, from

the time label i/N, as shown in Equation 4.2. This returns angular values in the range $[0, \pi]$.

$$\begin{cases} \phi_i = \arccos(\tilde{x}_i), & -1 \leq \tilde{x}_i \leq 1, \tilde{x}_i \in \tilde{X} \\ r_i = \frac{i}{N}, & i \in N \end{cases} \tag{4.2}$$

A new insight into understanding time series is provided by this polar coordinate system based representation. As time passes, the value of the sequence changes from the original amplitude variation to the angular variation in the polar coordinate system.

3. **GASF** By calculating the sum of the trigonometric function between the sampling points, the temporal correlation between them is identified from an angular point of view. Therefore, the GASF is defined as follows:

$$\mathbf{GASF} = \begin{bmatrix} \cos(\phi_1 + \phi_1) & \cdots & \cos(\phi_1 + \phi_n) \\ \cos(\phi_2 + \phi_1) & \cdots & \cos(\phi_2 + \phi_n) \\ \cdots & \ddots & \cdots \\ \cos(\phi_n + \phi_1) & \cdots & \cos(\phi_n + \phi_n) \end{bmatrix} \tag{4.3}$$

One of the standout attributes of GAF is its unwavering commitment to preserving the temporal dynamics of the series. The matrix representation, by design, ensures the retention of patterns, overarching trends, and other inherent temporal relationships. This makes the transformed data highly suitable for the discerning filters of CNNs.

Although GAF has been praised for its ability to preserve temporal relationships and for its synergistic compatibility with CNN architectures, its inherent limitations must be acknowledged. In particular, when dealing with noisy datasets or subtle patterns, the GAF may struggle to accurately capture data nuances. This limitation became apparent in the context of this study.

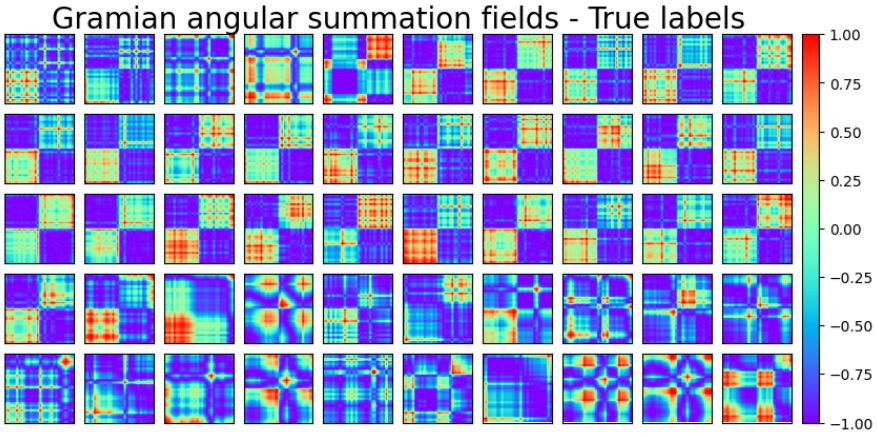


Figure 4.2: GASF representations of 50 positively labeled time series.

The performance of NN models using the matrices produced by GAF modifications as input was found to be poor compared to models using the raw time series as input. This performance difference was first observed in the GAF outputs shown in Figs. 4.2 and 4.3. These figures show the GAF matrices for two distinctive series: a collection of 50 time series with positive labels and another collection of 50 time series with negative labels.

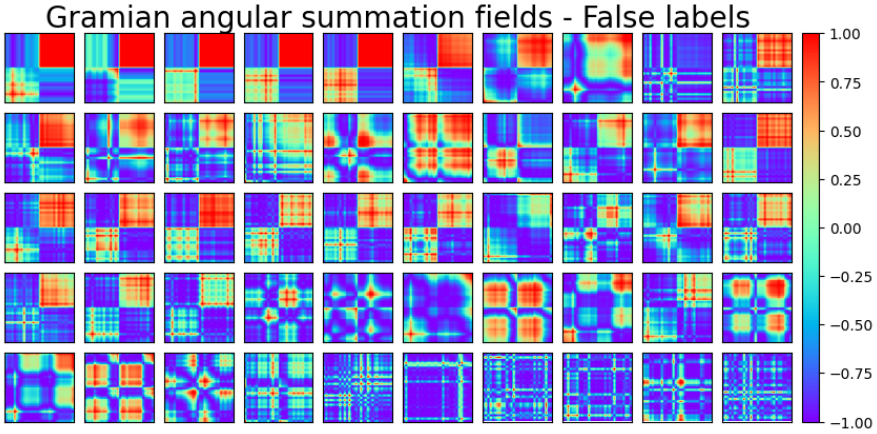


Figure 4.3: GASF representations of 50 negatively labeled time series.

A preliminary analysis of these figures indicates some resemblance in the visual representations of the time series in both labelled sets. This observation suggests

that the GAF transformation may not have been optimal for the individual characteristics of the dataset in question, potentially resulting in feature space overlap and, as a result, diminished discriminatory strength.

4.2 Recurrent Neural Networks

The second macro-category of neural network models studied belongs to the recurrent neural networks: both the Long Short-Term Memory (LSTM) and the Gated Recurrent Unit (GRU) models were developed to fit the two databases created, with only one channel of input data or with two channels. In the course of this research, two separate versions of both models were carefully designed. The aim was not only to understand the intrinsic behaviour of the LSTM and GRU layers in the context of the datasets, but also to assess how variation in network depth affected the performance of the model and subsequent implementation on a microcontroller.

4.2.1 Long-Short Term Memory

Long short-term memory networks excel at processing sequences. Their ability to deal with temporal dependencies makes them particularly well suited to data based on sequences and time series.

The first version of the LSTM model has two LSTM layers. The input layer is intended to receive data that matches the shape of the training dataset. The first LSTM layer is then applied. It contains a number of LSTM cells or units, which are essentially the memory cells responsible for maintaining temporal dependencies across sequences. During the optimization phase, the exact number of these units is defined. Each unit can be thought of as a memory element with built-in gating mechanisms that control the flow of information, determining what to store, discard, or update. The layer employs a hyperbolic tangent (tanh) activation function in conjunction with sigmoid recurrent activation. Sequences from this layer are then passed on to the second LSTM layer, which represents the hidden state of the

LSTM at each time step for each input. When stacking multiple LSTM layers, such a configuration needs to be performed because the subsequent LSTM layer requires a sequence input to handle temporal dependencies across time steps.

The configuration of the next layer is identical to the first, with one exception: it does not return sequences, making it suitable for connection to subsequent non-recurrent layers. Both LSTM layers are followed by batch normalization layers to increase learning and stabilize activations. As previously stated, an optional dropout layer can be incorporated to reduce overfitting. The model ends with an output layer suitable for binary classification tasks, which contains two neurons controlled by a softmax activation function.

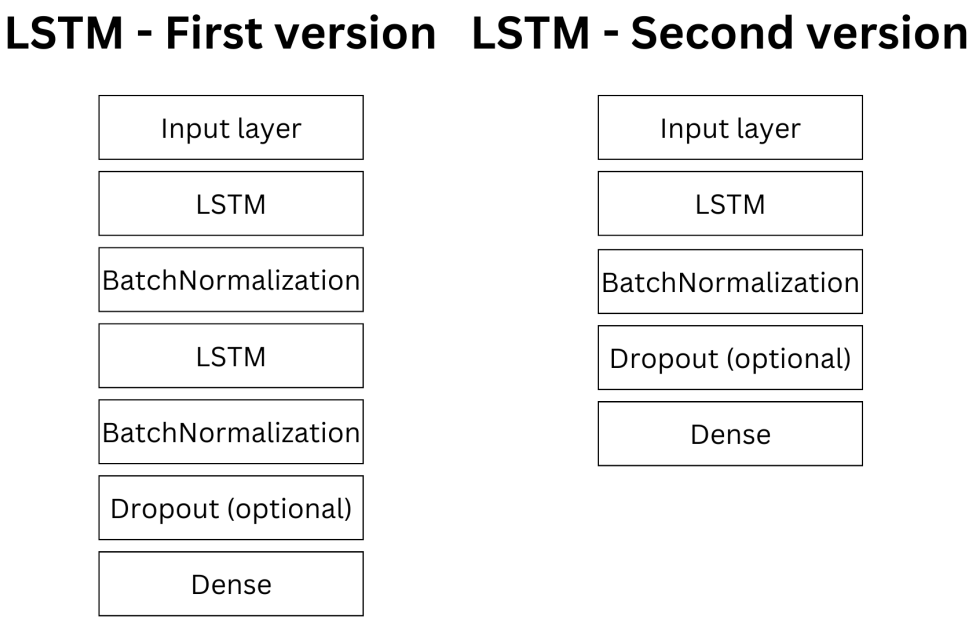


Figure 4.4: Architectures LSTM models.

In contrast to the prior model the second version is more straightforward, with a single LSTM layer. This difference can be easily identified in Fig. 4.4. The input layer remains consistent, processing the training dataset's data. The single LSTM layer, which has the same setup as the preceding model, is immediately followed by

a batch normalization layer. Again, a dropout layer with the same goal of reducing overfitting may be incorporated. The architecture is completed with a dense layer with two output neurons and a softmax activation function. Both models train to minimize the loss of categorical cross-entropy using the Adam optimizer with a clipping value of 1.0.

4.2.2 Gated Recurrent Unit

Gated Recurrent Units (GRUs) are another version of recurrent neural network designed to handle sequences. While they are theoretically similar to LSTMs in their attempt to capture long-term relationships in data, they have a distinct internal structure, which is principally defined by their update and reset gates.
For the purposes of this study, GRU models were built using a structure similar to the LSTM models previously addressed. The two GRU models paralleled the two LSTM models: one has a dual-layer design that aims to leverage the potential of depth in capturing complex temporal patterns, while the other has a single-layer arrangement that leans toward simplicity. The core recurrent layers in both setups are GRUs, which stand in for the LSTMs. In Fig. 4.5 are reported the two architectures for the GRU versions of the models.
The subtle changes in the internal gating mechanisms of GRUs and LSTMs offer possible performance variations. GRUs reduce the gating system to two gates, which may improve computational efficiency.

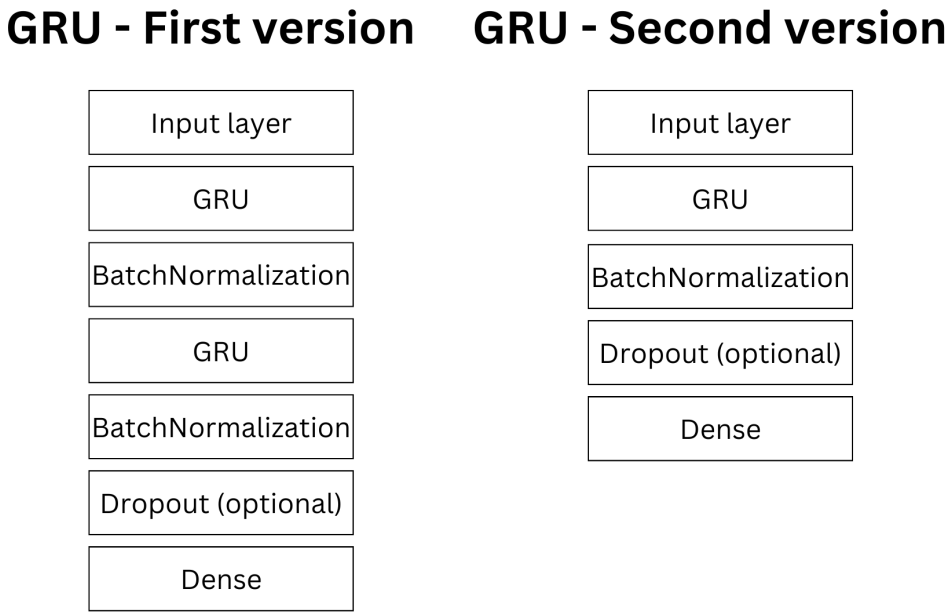


Figure 4.5: Architectures GRU models.

Chapter 5

Models Optimization

Optimizing neural network models is a critical step in assuring their effectiveness, particularly when dealing with complicated systems like electric vehicle acceleration pedal signals. This chapter digs into the complexities of model optimization, emphasizing the need of hyperparameter adjustment and data normalization. Using tools like the KerasTuner API allows for systematic exploration of the hyperparameter space with the goal of improving model performance. Furthermore, the choice between normalized and non-normalized data has been investigated because it can have a significant impact on the behavior and correctness of a model. The next sections will explain the approaches used for these optimizations as well as the reasoning behind each decision.

5.1 Hyperparameter Tuning

Optimizing hyperparameters is a key step in the development of robust and efficient neural network models, as the choice of hyperparameters can significantly influence model performance. For hyperparameter optimization, the Keras Tuner library has been implemented, specifically the `keras_tuner.BayesianOptimization` method. This method is rooted in the principles of Bayesian optimization, with an underlying Gaussian process (GP) model. Upper confidence bound (UCB) is the acquisition function employed.

Bayesian Optimization

Bayesian optimization, as elucidated by Garnett (2023) [22], is a probabilistic model-based optimization approach that is best suited for global optimization problems when the objective function behaves as a 'black box'. This approach is especially useful when evaluating the function required to run expensive simulations or when the behavior of machine learning models is highly dependent on their parameters, such as in convolutional neural networks.
Bayesian optimization works by retaining a probabilistic belief about the objective function and creating an acquisition function to determine the next evaluation point. In order to learn about the function, before any data are observed, the method frequently uses a Gaussian process prior, which is a pre-existing distribution for an uncertain quantity. Then, given a set of observations, it conditions the distribution accordingly. The main challenge then becomes the selection of the next observation point. In Bayesian optimization, this is accomplished by creating an acquisition function that is proportional to the expected desirability of assessing the objective function at a specific point and is often an inexpensive function to evaluate. While the acquisition function is faster to evaluate, this does not mean that it can be evaluated on a dense grid over the entire parameter space, especially if the dimensionality of the space is high. This distinction is critical because Bayesian optimization gives a rational technique to select the most informative points to assess next without having to intensively sample the whole space.

The activation function used by the Keras Tuner's Bayesian optimization method is the upper confidence bound function, also known as GP-UCB in the context of Gaussian processes [23]. It can be interpreted in the framework of Bayesian decision theory as evaluating an expected loss associated with evaluating the function at a specific point. The objective is to select the point that minimizes this expected loss. The UCB acquisition function is usually described in terms of function maximization, rather than function minimization; however in the context of minimization, it takes the form:

$$a_{\text{ucb}}(x;\beta) = \mu(x) - \beta\sigma(x) \tag{5.1}$$

where $\beta > 0$ is a tradeoff parameter and $\sigma(x)$ is the marginal standard deviation of the function. The GP-UCB acquisition function contains explicit exploitation ($\mu(x)$ - evaluating at points with low mean) and exploration ($\sigma(x)$ - evaluating at points with high uncertainty). In the context of optimization, exploitation entails selecting points that are predicted to produce the greatest results based on prior information. It is all about capitalizing on what is already known to gain instant advantages. Exploration, on the other hand, represents the model's uncertainty about the function. It is about exploring unexplored function areas, particularly in regions where the model is uncertain. The goal is to discover new information, even if it does not immediately bring advantages. By doing so, the model may be able to identify previously unknown better optima. Strong theoretical results are known for UCB, namely that under certain conditions, the iterative application of this acquisition function will converge to the true global minimum of the function.

Detailed Hyperparameter Selection

The models were constructed based on distinctive hyper parameter values adapted to their architecture and the nature of the data they were supposed to analyze after the hyperparameter optimization procedure. As seen in Table 5.1:

- The CNN models are made up of convolutional layers, each with its own set of hyperparameters. The number of filters ranged from 32 to 128, while kernel sizes ranged from 3 to 5. For these layers, two activation functions, ReLU and tanh, were evaluated. In addition, an optional dropout layer was implemented, and the batch size for training was varied from 10 to 32.

- The LSTM and GRU models have been developed with units ranging from 128 to 256. An optional dropout layer, similar to the CNN model, was included. The batch size for these models was evaluated across a wider range, up to 64.

- Conv2D layers were used in the CNN with image augmentation model. The hyperparameters of these layers were identical to those of the other CNN models but this model also investigated imagine augmentation techniques such as random flipping, rotation, and zooming.

Model Type	Hyperparameters Tested	Ranges/Values
CNN	Conv layers: `filters`	32-128, step 32
	Conv layers: `kernel_size`	3-5, step 1
	Conv layers: `activation`	ReLU, tanh
	Dropout layer	True, False
	Batch size	10, 16, 24, 32
LSTM/GRU	LSTM/GRU layers: `units`	128-256, step 64
	Dropout layer	True, False
	Batch size	10, 16, 24, 32, 64
CNN with Image Augmentation	Conv2D layers: `filters`	32-128, step 32
	Conv2D layers: `kernel_size`	3-5, step 1
	Conv2D layers: `activation`	ReLU, tanh
	Dropout layer	True, False
	RandomFlip layer	True, False
	RandomRotation layer	True, False
	RandomZoom layer	True, False
	Batch size	10, 16, 24, 32

Table 5.1: Hyperparameters tested for each model architecture.

The lower amount of hyperparameters in the LSTM and GRU models compared to the CNN models is a noticeable difference. This design decision was made to take full advantage of Keras' cuDNN capabilities. CUDA Deep Neural Network library, or cuDNN, is a GPU-accelerated library of primitives for deep neural networks developed by NVIDIA [24]. Keras will automatically transition to this fast cuDNN implementation if it detects a suitable GPU and meets specified layer criteria. The advantages are obvious in terms of computational speed and efficiency. However, in order to be able to use this implementation of cuDNN, the GRU and LSTM classes of Keras must fulfil certain requirements [25] [26], such as:

- `activation` == tanh
- `recurrent_activation` == sigmoid
- `recurrent_dropout` == 0
- `unroll` is False

- `use_bias` is True
- `reset_after` is True (for GRU layers)
- Inputs, if use masking, are strictly right-padded.
- Eager execution is enabled in the outermost context.

The decision to optimize the models for cuDNN was dictated by practical reasons. Model training times were prohibitive without the use of cuDNN, making repeated testing and modifications impractical. By adhering to the cuDNN standards, training time was significantly reduced. This acceleration was made even more achievable by exploiting the GPU resources made available by Google Colaboratory, also known as Colab [27]. Colab is a cloud-based platform that provides free GPU access for machine learning and data analysis workloads. It is essentially a Jupyter notebook environment that runs entirely in the cloud and requires no setup. The NVIDIA Tesla T4 GPU with 16GB VRAM [28][29] is the default GPU given by Colab, and it is powerful enough for a wide range of machine learning tasks. Even with the computational needs of the dataset, the combination of cuDNN optimization and Colab's GPU resources guaranteed that the RNNs models were trained efficiently.

5.2 Normalized Input Signals

Normalization is a fundamental preprocessing technique that is especially useful in neural network training and time series analysis. Normalization supports faster convergence in model training and perhaps increases performance by standardizing the input data to a consistent range. This is especially important in signal classification, where many different values may be encountered. The goal of normalization is to align the intervals and distributions of the data such that no feature or scale has a disproportionate impact on the model. Although there are indisputable benefits to normalization in many circumstances, its indiscriminate use can be damaging. Normalization is especially important with time series data where the emphasis is

on pattern and sequence progression rather than raw numbers, therefore it may not be appropriate in this context.
Distinct series or channels in multivariate time series are able to capture distinct phenomena, stressing the requirement for channel-specific normalization. This method considers each series as a separate entity, identifying and conserving its unique qualities. The values of each channel are mathematically modified based on its individual mean (μ) and standard deviation (σ), guaranteeing uniform scaling and keeping channel distinctiveness. The formula is as follows:

$$x' = \frac{x - \mu_{\text{channel}}}{\sigma_{\text{channel}}} \tag{5.2}$$

where x' is the normalized value and x is the original value
However, since the signals in the two channels of the database are similar but at different voltage ranges, normalization could result in a decrease in the knowledge that can be extracted from the data by the neural network.
Even the single-channel approach, in which the entire data set is normalized to a common mean and standard deviation, can cause problems. For example, normalizing a signal that goes from a range of [0,5]V in its first half to a range of [0,10]V in its last half can obscure the intrinsic magnitude shift between segments. Such normalization could inadvertently diminish critical events or features, potentially misrepresenting crucial shifts in the signal. In particular, the high [0,10]V segment, if interrupted by outliers, may distort the normalization metric, making the [0,5]V segment appear compressed. This could lead to under representation of patterns in the lower range, potentially affecting subsequent analysis or modeling results.
Since it cannot be determined a priori whether normalization will have a positive or negative effect on model performance, the models were dragged and tested with both normalized and un-normalized data.

5.3 Optimized Architectures

This section presents the architectures that were selected as optimal after the hyperparameter tuning process. These architectures provide the optimum hyperparameter combination for each model type, ensuring maximum performance on the

dataset.

Table 5.2 displays the models of convolutional neural networks based on different inputs. Three convolutional layers were utilized in the CNN models, each with its own filter size, kernel size, and activation function. The tanh activation function was employed by all layers in the single-channel input CNN model, with no dropout layer and a batch size of 16. The two-channel CNN model, on the other hand, kept the tanh activation for its layers but added a dropout layer and reduced the batch size to 10.

The presence of a dropout layer in the two-channel CNN model, as opposed to the absence of one in the single-channel model, indicates the increased complexity and potential overfitting issues posed by multi-channel data. This inclusion is probably the result of a regularization method to prevent overfitting, particularly when dealing with richer input data.

Model	`filters`	`kernel_size`	`activation`	**Dropout layer**	**Batch size**
CNN one channel	32 64 64	5 5 3	tanh tanh tanh	False	16
CNN two channels	32 128 96	5 3 4	tanh tanh tanh	True	10
CNN GAF images	96 96 64	4 4 5	ReLU tanh tanh	False	24
CNN GAF images augmentation	32 128 64	3 3 5	tanh tanh ReLU	False	10

Table 5.2: Optimal hyperparameters for the CNN models.

The CNN models constructed for the GAF images use a combination of tanh and ReLU activations, with no dropout layer and batch sizes of 24 and 10, respectively. Furthermore, the GAF image-based CNN model's data augmentation technique is extremely intriguing, which is displayed in Table 5.3. The usage of just the

'RandomRotation' layer, ignoring the 'RandomFlip' and 'RandomZoom' layers, highlights the relevance of rotational variations for the generated model when working with GAF images. When translated into GAF images, this decision can highlight the distinguishing aspects of time series data, stressing that rotating patterns can lead to optimal model training.

Augmentation Layer	**RandomFlip Layer**	**RandomRotation Layer**	**RandomZoom Layer**
CNN GAF images augmentation	False	True	False

Table 5.3: Optimal hyperparameters for the data augmentation layers.

When looking more closely at the best hyperparameters selected, a noteworthy pattern emerges. The consistent employment of the tanh activation function across the convolutional layers in the CNN models demonstrates its efficiency in capturing the complexities of the dataset, regardless of the shape of the input data. This is especially intriguing given the extensive use of ReLU activation in many modern CNN architectures, yet tanh appears to be the preferred option in this scenario. This result is also highly advantageous in the context of following the Keras instructions for LSTM and GRU layers to take advantage of the cuDNN implementation, which involves setting the layer activation function as tanh a priori.

Turning to recurrent neural networks (RNNs), the hyperparameter for LSTM and GRU models with a single channel input, showed in Table 5.4, primarily used 128 units across layers, and the dropout layer was not used, indicating a balance between computational efficiency and model expressiveness.

Model	units	Dropout layer	Batch size
LSTM two layers	128 128	False	32
LSTM one layer	128	False	16
GRU two layers	128 128	True	24
GRU one layer	96	False	24

Table 5.4: Optimal hyperparameters for the RNN models – one channel.

The choice of hyperparameters in two-channel models (Table 5.5), on the other hand, can occasionally reach 192 units, indicating the necessity to represent more intricate inter channel relationships and dynamics. Indeed, the inclusion of dropout layers in nearly all two-channel RNN designs emphasizes the delicate trade off between model complexity and overfitting risk.

Model	units	Dropout layer	Batch size
LSTM two layers	128 192	False	64
LSTM one layer	128	True	24
GRU two layers	192 128	True	24
GRU one layer	128	True	10

Table 5.5: Optimal hyperparameters for the RNN models – two channels.

The hyperparameters of the models trained using normalized data are presented below. When these hyperparameters are compared to those of the non-normalized models, various distinctions and parallels become apparent. Table 5.6 shows a significant shift toward the adoption of the ReLU activation function for CNN models, indicating its greater efficacy in handling normalized data. Furthermore, batch sizes have generally decreased, indicating a more granular technique for the normalized dataset.

Model	`filters`	`kernel_size`	`activation`	Dropout layer	Batch size
CNN one channel	64 128 32	5 4 4	ReLU tanh ReLU	True	10
CNN two channels	128 96 64	3 3 3	tanh tanh ReLU	False	10
CNN GAF images	32 64 64	5 5 3	tanh tanh tanh	False	16
CNN GAF images augmentation	96 64 32	3 3 4	tanh ReLU ReLU	False	10

Table 5.6: Optimal hyperparameters for the CNN models with normalized input data.

The data augmentation layers, as shown in Table 5.7, demonstrate a preference shift with the addition of the 'RandomFlip' and 'RandomZoom' layers, while the 'RandomRotation' layer is omitted. This is in contrast to the non-normalized data, which favored the 'RandomRotation' layer. This disparity might be attributable to the fact that normalization can change the distribution and the characteristics of the data. Normalization of GAF images may emphasize or de-emphasize specific patterns within the time series.

Augmentation Layer	RandomFlip Layer	RandomRotation Layer	RandomZoom Layer
CNN GAF images augmentation	True	False	True

Table 5.7: Optimal hyperparameters for the data augmentation layers in the model with normalized input data.

The original patterns in the non-normalized data time series could have been more rotationally variant, making 'RandomRotation' more effective. However, these rotational patterns may become less distinct or less relevant for model training after

normalization. Furthermore, given that data augmentation is utilized to increase the strength of models by supplying them with different versions of the training data, the non-normalized model may have needed rotational variations, but the normalized model may benefit more from flips and zooms. Normalized data may be smoother or have fewer extreme values, which might cause models to overfit to specific patterns more easily. Using 'RandomFlip' and 'RandomZoom' may add additional diversity to the training process, allowing the model to generalize more effectively. Other hyperparameters may also influence the choice of data augmentation approaches. For example, if the normalized model employs smaller filters or different activation functions, it may react differently to different augmentations.

Model	**units**	**Dropout layer**	**Batch size**
LSTM two layers	128 128	True	24
LSTM one layer	256	False	10
GRU two layers	256 256	False	16
GRU one layer	128	False	10

Table 5.8: Optimal hyperparameters for the RNN models with normalized input data – one channel.

There are noticeable discrepancies between the hyperparameters of the recurrent neural networks in Tables 5.8 and 5.9, and their non-normalized counterparts in Tables 5.4 and 5.5. To begin with, models trained on normalized data favor a larger number of units. This implies that normalized data may benefit from a more complex structure in order to better capture its temporal variations. In contrast, normalized datasets frequently choose smaller batch sizes, suggesting a desire for more frequent model updates, perhaps to better manage the altered data distribution. The existence or lack of dropout layers, on the other hand, follows a pattern similar to that of non normalized models.

Model	`units`	Dropout layer	Batch size
LSTM two layers	128 128	True	16
LSTM one layer	128	False	10
GRU two layers	256 128	True	16
GRU one layer	128	False	10

Table 5.9: Optimal hyperparameters for the RNN models with normalized input data – two channels.

These distinctions highlight the significant impact of data normalization on hyperparameter selection. Changing the normalization of the data may alter its underlying distribution, which can affect the learning dynamics of the model. As a result, different hyperparameter setups are required to achieve optimal performance, even if the model primary structure is the same.

Visual representations of neural network designs frequently reveal more about the model's complexity and structure than textual descriptions alone. As a result, what follows are a few noteworthy examples of implemented NN models:

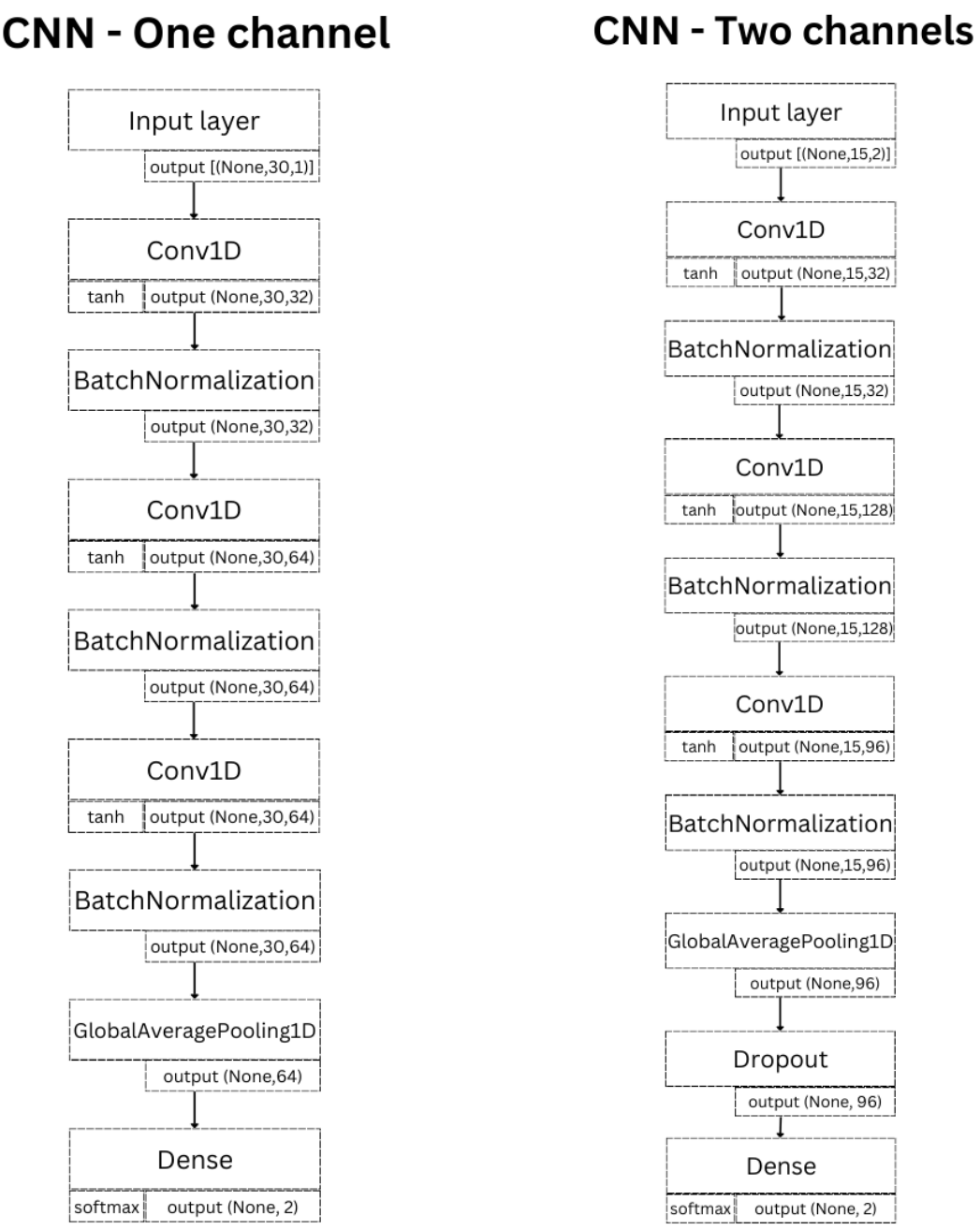
- Figure 5.1 depicts the architectures of CNN models with one-channel and two-channel input data. The structural similarities between the two are obvious at first glance, highlighting CNNs' innate adaptability to different input dimensions. Notably, the two-channel model includes a dropout layer, which is most likely an adaptation to deal with the extra complexity of dual-channel input. When dealing with multidimensional input, the presence of dropout shows that there is an emphasis on avoiding overfitting.

- In Figure 5.2, the earliest versions of the LSTM and GRU models are displayed, which are distinguished by their dual-layered approach. Their complexity reflects an attempt to capture more complicated patterns and relationships in the data.

- Figure 5.3 shows the later version of the model, which features a streamlined

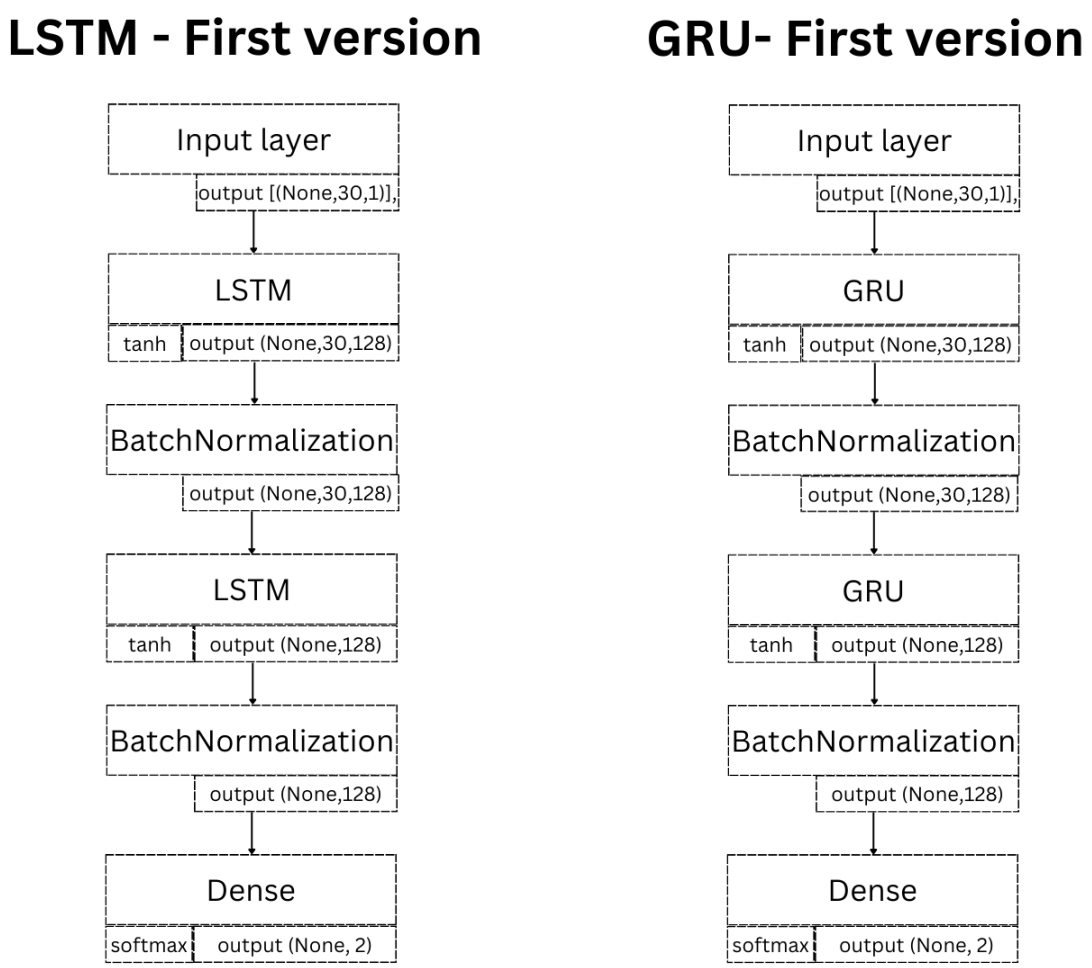
approach that is implemented with a single recurrent layer, indicating an intentional trade-off between model complexity and computational efficiency.

The addition of these visual aids improves the understanding of the neural network architectures that are being implemented and provides a comparison view of the differences between various models.



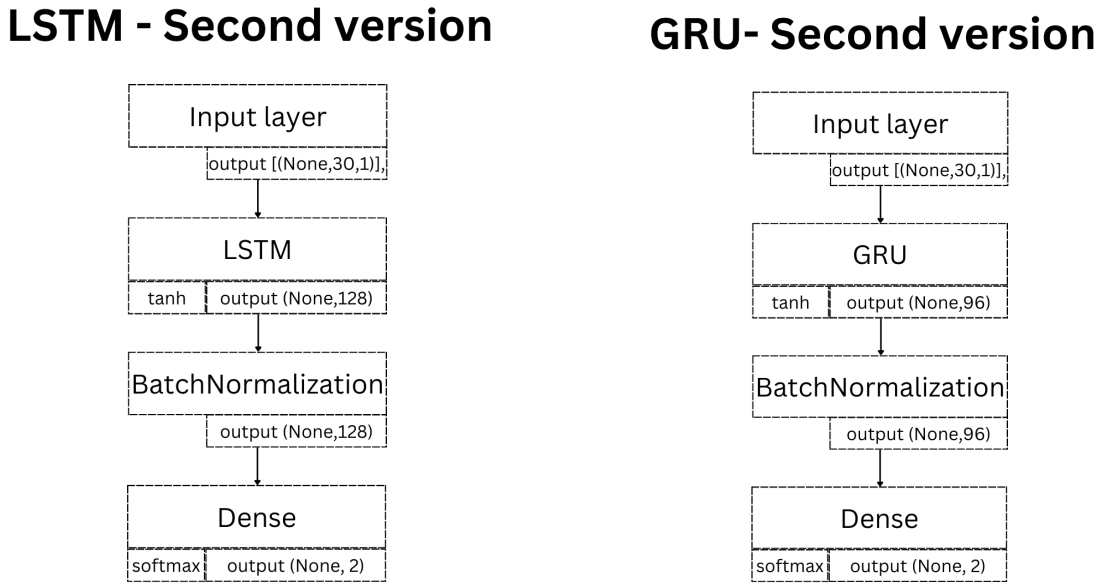
(a) One channel model. **(b)** Two channels model.

Figure 5.1: Final architectures CNN models.



(a) LSTM model. **(b)** GRU model.

Figure 5.2: Final architectures RNN models - first version.



(a) LSTM model. **(b)** GRU model.

Figure 5.3: Final architectures RNN models - second version.

Chapter 6

TFLite Conversion and Deployment on Microcontroller

6.1 TensorFlow Lite

TensoFlow Lite (TFLite) [16] is a specific extension for on-device applications that emerged from the TensorFlow framework. Its primary purpose is to compact and optimize TensorFlow models in order to ensure they are adequate for edge applications. Given the specific challenges provided by edge devices, this is not only a question of convenience, but a requirement. TFLite addresses these issues by minimizing latency because there is no need to request and send data to a server, ensuring data privacy through on-device processing, operating even in the absence of Internet connectivity, minimizing model and binary size, and taking power consumption into account, which is critical for mobile and embedded devices. It has been designed to integrate with various platforms, including Android, iOS, microcontrollers, and a variety of programming languages: the Python API (`tf.lite`) [30] was used for this thesis.

6.1.1 TensorFlow Lite Conversion

Moving from TensorFlow to TFLite is a rigorous procedure that necessitates an in-depth understanding of the requirements and conversion stages.
The process of conversion can be visualized as a series of steps:

1. **Prerequisites**: Ensure the model's compatibility with the TFLite conversion.
2. **Model Conversion**: Transform the TensorFlow model into the TFLite format by utilizing the proper tools and APIs.
3. **Optimization**: Apply approaches to minimize the model's size while maintaining its accuracy.
4. **Quantization**: Compress the model even more by reducing the accuracy of its weights and, potentially, activations.

Prerequisites

It is crucial to confirm that the model to be converted was developed and trained using TensorFlow's core libraries and tools before beginning the conversion. To achieve a smooth conversion, the model's architecture, particularly the layers and operations employed, must be consistent with the operations offered by TFLite. Layers like Conv1D, Conv2D, LSTM, and GRU, for example, are widely used and supported. Prior to conversion, a performance baseline for the TensorFlow model was established. This will be relevant later when comparing the performance of the TFLite model, the results of which are provided in Chapter 7.
Before conversion, the model must be evaluated in terms of data volume and overall complexity to determine whether it is suitable for edge implementations. It is natural to believe that more complicated models, such as RNNs, will cause more issues during conversion than basic, linear models.

Model Conversion

Model conversion happens once all prerequisites have been thoroughly checked. Figure 6.1 clearly depicts the overall workflow that is required for the conversion process.

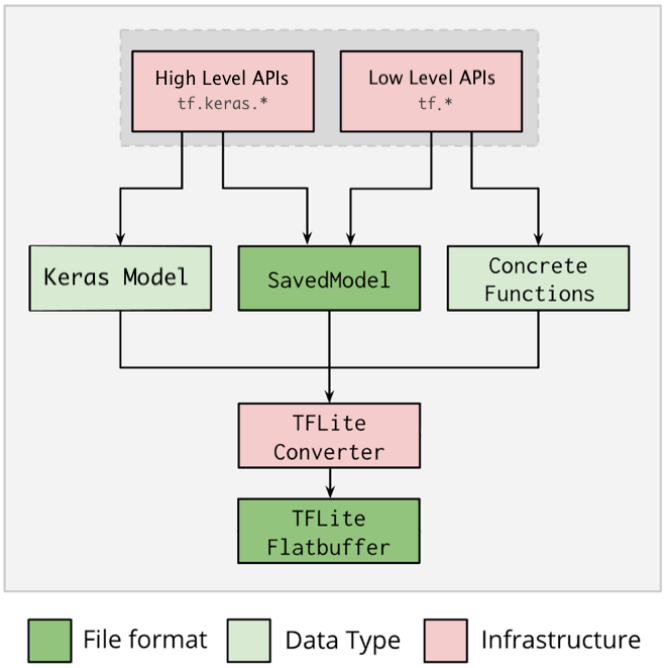


Figure 6.1: Conversion from TensorFlow to TFLite workflow.

Although there are many APIs available for the purpose of performing this task, the Python API emerges as the ideal choice. The recommendation stems from its numerous capabilities: not only does it seamlessly integrate the conversion process into the larger development framework, but it also enables the application of optimizations, the incorporation of critical metadata, and a host of other features that make the conversion process more intuitive. For this work, the TensorFlow model was initially stored in the `SavedModel` format, an approach strongly recommended by the official TensorFlow guidelines. By using the high-level API `tf.keras.*`, which results in a Keras model, this format is generated. However, as shown in the Fig. 6.1, the low-level API `tf.*` may be utilized as well. As a result, a TensorFlow

Lite model in the optimized FlatBuffer format, identified by the `.tflite` file extension, was obtained using the `tf.lite.TFLiteConverter.from_saved_model()` function [31].

Optimization

In machine learning models, optimization refers to a collection of techniques indicated to enhance model performance, particularly when deploying it on edge devices with limited computational resources. When a model is optimized, several benefits can be observed that can aid in model deployment. Optimization techniques, for example, can drastically reduce the size of the model. A reduced model takes up less storage space on devices and requires less bandwidth and time to download. Smaller models consume less RAM when running, resulting in less RAM usage during inference, freeing up memory for other portions of the application, which can translate to improved performance as well as stability. Reduced latency is another benefit of model optimization: through optimization, latency, defined as the time it takes a model to execute a single inference, may be significantly lowered. This not only increases the responsiveness of real-time applications, but it can also reduce power consumption.
It is critical, however, to understand that improvements may involve trade-offs, potentially affecting model accuracy. While some models may see a slight reduction in accuracy, others may see more noticeable alterations. In exceptional cases, optimization may even improve a model's accuracy.

TensorFlow Lite includes, but is not limited to, the following optimization techniques:

- Pruning is the process of removing parameters from a model that have a minimal impact on its predictions. While pruned models retain their original size and runtime latency, they can be compressed more effectively, making pruning useful for minimizing model download size.
- Clustering is the process of categorizing the weights of each level in a model into predetermined clusters based on their similarity. Weights within the

same cluster are then represented by a single value, the centroid of that cluster. Clustering minimizes model complexity in a data-driven approach and determines values based on the actual distribution of weights in the model. As a result, clustered models can be compressed more efficiently, providing distribution benefits similar to pruning.

- Quantization is a method that decreases the precision of the numbers that represent the parameters of a model. As a result, the model is both smaller in size and quicker in computation [32].

The initial step was to examine the behavior of two simple optimization algorithms, pruning and clustering, on convolutional models with one and two channels, respectively. Both techniques were performed using the TensorFlow Model Optimization Toolkit [33].
A sparsity level of 0.5 was used for pruning, which implies the elimination of about fifty percent of the weights. Thanks to the function `ConstantSparsity`, this sparsity level was kept constant throughout the pruning operation. For clustering, which represents the model parameters using the centroids of these clusters, the number of clusters was set to four for this study, with the centroids initialized linearly.
After optimization, the two versions of the models were converted to TFLite format and their accuracy was evaluated. Notably, because pruning outperforms clustering, the latter was eliminated from model optimization in this thesis.

Quantization

In the context of optimizing machine learning models, quantization emerges as a critically important approach. Essentially, quantization relies on reducing the numerical precision required to describe a model's parameters. These parameters are often expressed using 32-bit floating-point numbers. Using lower precision numbers, such as 8-bit integers, not only compacts the model but also increases its computational speed. However, as with any optimization strategy, the advantages must be balanced against the potential loss of accuracy.
The influence of quantization on accuracy varies depending on the application

and model, so a comprehensive study of the quantized model is required before implementation.
Quantization is not only desirable, but frequently required for microcontroller implementations, especially in this study where there are additional inference constraints. Given microcontrollers' limited memory and processing resources, the benefits of quantization in lowering model size and enhancing inference speed become essential. Also, the Raspberry Pi RP2040 is an ARM Cortex-M0+ dual-core microprocessor. The ARM Cortex-M0+ [34] core is integer-only, which means it lacks a hardware floating-point unit (FPU). As a result, all floating-point operations would have to be emulated through software routines or libraries, which would be time-consuming.

It is worth mentioning that the attempt to integrate machine learning models on microcontrollers is a relatively new topic. As a result, quantization support for various Keras layers, including recurrent, 1D convolution and batch normalization layers, is still in the early stages of development [35]. Because of this major constraint, even the simplest models cannot be properly quantized. Consequently, while the results presented in this thesis are promising, they may not represent the pinnacle of what is possible to achieve. However, it is important to understand the expanding potential of this field. The community and industry are working diligently to fill these gaps, and as the field evolves, additional support and enhanced tools are likely to emerge, paving the way for more optimum microcontroller implementations in the future.

TensorFlow Lite provides numerous quantization approaches suitable for a variety of requirements:

- **Post-training quantization:** This is a form of quantization applied after the model has been trained. This implies that the model is quantized without having to be re-trained. One of the key advantages of this strategy is its simplicity and broad application. Significant reductions in model size and latency can be accomplished by quantizing the weights and, optionally, the activations. Weight quantization, integer quantization, and 16-float quantization are three

types of post-training quantization, and each offers a trade-off between model size, speed and accuracy.

- **Quantization-aware training (QAT):** QAT, as opposed to post-training quantization, guarantees that the model is aware of the quantization process while training. This suggests that the model has been trained to predict the noise and errors that quantization introduces. As a result, when the model is quantized, it retains its accuracy more effectively than models quantized after training. QAT was explicitly implemented in the context of this thesis to take full advantage of its potential in obtaining optimal performance on microcontrollers, because QAT is especially helpful for deployment on hardware that primarily supports integer operations. QAT guarantees that the model remains robust and performs well in such environments by training it with quantized data. Additionally, using QAT, not only is the model size decreased but the model is also designed to manage the decreased precision, guaranteeing that there is minimal degradation in performance.

The pruned models were subjected to the additional optimization procedure called Quantization Aware Training. The use of QAT varies according to the type of model. Two different scenarios for convolutional models were investigated: one with a representative dataset and one without. A representative dataset is a smallsubset of the training data, usually 0.01%. It is crucial because it captures the overall properties and distribution of the dataset, ensuring that the quantization process appropriately represents the data's dynamics. This dataset facilitates quantization by giving a snapshot of the data distribution, allowing for more efficient and precise quantization.
However, due to the inherent complexity of RNN models and their ability to retain information from previous inputs, the representative dataset could not be used. RNNs process data sequences in which the outcome of each step is determined not only by the current input but also by past inputs. The total memory requirement could be significant when combined with the memory-intensive QAT procedure. The use of a representative dataset for RNNs contributes to the problem by adding additional memory usage. This could cause memory exhaustion in environments

with limited RAM, such as the Google Colab, resulting in unexpected restarts of execution.

6.1.2 TensorFlow Lite for Microcontroller

The TensorFlow ecosystem is well-known for its strong deployment tools, which allow developers to deploy models almost everywhere, from cloud environments to mobile devices. This deployment capability has been extended to embedded devices, which require efficient, portable code, with the introduction of TensorFlow Lite for Microcontrollers [36]. TensorFlow Lite for Microcontrollers (TFLM) is a new version of TensorFlow Lite that is specifically built to handle the unique challenges that microcontrollers provide. Microcontrollers have restricted computational, memory, and storage capabilities. Given these limits, it may appear unrealistic to run powerful machine learning models on these devices. TFLM is meant to be extremely lightweight, allowing basic machine learning operations to be performed even when memory is restricted to a few kilobytes.

Several steps must be taken to turn a TensorFlow model into a version that can be executed on a microcontroller. The model must first be trained using the TensorFlow library. Following training, the model is optimized and then converted to TensorFlow Lite format using the TensorFlow Lite converter. Through Tensorflow Lite Interpreter [37], the accuracy of the lite model is calculated, so that it can then be compared with the baseline model. The model is subsequently transformed once more, this time into a C byte array, to make it easier to store in the read-only memory of the microcontroller. This operation is made possible by TensorFlow Lite for microcontrollers. This process culminates in inference on the device, where the microcontroller uses a C++ library to execute the model. The produced C++ library enables TFLM inference to be performed on nearly any device, including microcontrollers, as long as the hardware supports C++. The Edge Impulse [17] platform was utilized in order to facilitate the TFLM model conversion process.

6.2 Deployment on Microcontroller

6.2.1 Edge Impulse

Edge Impulse is a platform that connects advanced machine learning models to the limitations of edge devices, primarily microcontrollers. The platform is meant to function in combination with TensorFlow Lite for Microcontrollers (TFLM), providing an end-to-end solution for the machine learning lifecycle, from the collection of data and model training through deployment on edge devices. One of Edge Impulse's distinguishing features is its Python SDK, which allows developers to profile and deploy machine learning models generated in almost any machine learning framework to numerous hardware targets. TensorFlow Lite and TFLM, as well as vendor-specific toolchains, are among the targets. The SDK wraps the model in a variety of pre- and post-processing functions, as well as device-specific optimizations, making deployment simple.

Edge Impulse is a cloud-based machine learning operations (MLOps) [38] platform designed for developers who want to deploy models on embedded systems. It speeds up model deployment with TensorFlow Lite for Microcontrollers by helping users through the process of training a model in TensorFlow with Keras and then translating that model into a C++ library that integrates with TFLM. It provides an easy-to-use interface or Python SDK to integrate into the code.

Edge Impulse's C++ library is versatile, allowing inference with TensorFlow Lite for Microcontrollers on practically any device that supports C++. The library can handle input and output processing, such as windowing and resampling time series or audio inputs, in addition to executing the model. It can also support post-processing operations such as adding a moving average filter to a stream of classifier outputs. The library is also designed to apply the optimum optimizations for the targeted processor automatically.

A key problem in edge deep learning is ensuring that the model fits within the memory limitations of the target hardware and fulfills the needed inference performance. Edge Impulse's Python SDK assists in profiling the model for various

target hardware architectures, ranging from microcontrollers to neural network accelerators. This profile offers information on RAM, ROM, and inference execution time for the chosen hardware, which is useful in the design flow of a model architecture for edge machine learning.

Once the model has been profiled and optimized, Edge Impulse makes it easy to deploy. The model is loaded into an Edge Impulse project, where it is transformed into a C++ library using TensorFlow Lite for microcontrollers. Once the model is deployed to RP2040, it can be seen that the model is working either by a LED that lights up if the signal has an error, or by connecting the microcontroller to the computer and checking the serial output [39] .

Chapter 7

Discussion and Comparison of Results

The primary purpose of this chapter is to provide an overview of the empirical findings that emerged from this research. The results that are provided here are critical in addressing the study's original objective of researching the efficiency of neural networks in identifying errors in electric vehicle acceleration pedal signals and their subsequent deployment on microcontrollers.
To offer context, it is essential to revisit the research questions that motivated this investigation:

- How effective are neural networks in detecting anomalies in accelerator pedal signals?
- What are the performance disparities, if any, between TensorFlow Lite models and their more extensive neural network counterparts?
- Can a balance be achieved between the latency of neural network models on microcontrollers and their performance accuracy?

The following sections will discuss the data, analysis and results in response to these questions in a structured way.

7.1 Neural Network Model Performance

As previously stated (Chapter 4), the first models were built with TensorFlow. This strategy was adopted deliberately to provide a solid baseline, allowing for a more informative comparison with the TFLite models. While the chosen architectures are known for their effectiveness in time series classification tasks, it was critical to assess their performance in light of the specific challenges given by this study's database and objectives.
This section will delve into the empirical results of these models, highlighting their advantages and potential limitations in the context of electric vehicle acceleration signal classification. This will include metrics such as accuracy, precision and F1 score, with the goal of providing a comprehensive picture of the performance of these models in addressing the research goals set at the beginning of this thesis.

Performance metrics are critical tools for assessing and comparing the efficacy of machine learning models. They are frequently constructed from basic components that encapsulate the core results of a classification operation. True positives, true negatives, false positives and false negatives are these components. They are the heart of the confusion matrix, a table that describes the performance of a classification model on the test set.

- True Positives (TP): These are instances where the model properly predicted a positive outcome. It corresponded to the signals that were correctly classified as error free in the context of electric vehicle acceleration signal classification.
- True Negatives (TN): These are instances of negative outcomes that were accurately predicted to be negative. It denotes faulty signals that were appropriately detected as errors.
- False Positives (FP): These are instances that were negative but were wrongly forecast as positive by the model. This indicate that signals with errors were misclassified as error-free.
- False Negatives (FN): These are instances of positives that were wrongly

predicted as negative. It represents true signals that the model labeled as errors.

The confusion matrix provides a thorough view of model performance. It offers information on the precise nature of the errors committed by the model. Understanding these fundamentals is critical since it allows for a more refined evaluation of the model. This guarantees that the model not only achieves high accuracy but also corresponds with the specific requirements and complexities of the task in question. In the context of this study, for example, a model that mostly produces false negatives is preferable than one that produces false positives. The reason for this is that with the latter, the vehicle may incorrectly believe there are no issues when, in fact, there are, thereby risking driver safety. A system that occasionally warns an error, even when none occurs, on the other hand, is a safer preventive approach.
The following metrics were considered for this study:

- **Loss (or Error):** this quantifies how much the model's predictions differ from the actual value. Lower loss levels imply greater performance, whereas larger values indicate possible model inefficiencies. In this context, the categorical cross-entropy is the loss function employed when training the models, where the goal is to minimize it. The specific formula depends on the type of loss function used, in this case:

$$L(y, p) = -\sum_{i=1}^{C} y_i \log(p_i) \tag{7.1}$$

  where y_i is the true label for class i, p_i is the predicted probability of the instance belonging to class i and C is the number of classes.

- **Accuracy:** the accuracy value quantifies the percentage of correct model predictions. It is determined by dividing the number of instances correctly predicted by the total number of instances tested. Although accuracy is a simple and intuitive metric, it does not provide information about the errors made by the model, so it cannot be used alone to evaluate model performance.

It is calculated as:

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{7.2}$$

- **Precision:** this metric assesses the accuracy of positive forecasts. It is the proportion of correctly predicted positive observations to total expected positives. The low false positive rate is associated with high precision. It is defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7.3}$$

- **Recall:** also referred to as sensitivity or true positive rate, recall is the proportion of the actual positives that were correctly identified. It is critical when the cost of false negatives is significant. It is provided by:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{7.4}$$

- **F1 score:** The F1 score is a model performance metric that combines precision and recall into a single value, providing a more complete picture of a model's performance. The F1 score ensures that both false positives and false negatives are considered by taking the harmonic mean of these two metrics. This makes it especially beneficial because it does not allow a high value of one metric to compensate for a low value of the other. The F1 score is calculated as follows:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.5}$$

7.1.1 Analysis of Accuracy and Loss

As an initial analysis, the metrics of accuracy and loss are frequently the easiest and most basic to be evaluated. The accuracy bar plot (Fig. 7.1) for various models provide a clear visual representation of how effectively each model identifies the vehicle's acceleration signals. Higher precision means that the model classifies the test database better than those with lower precision. Meanwhile, Fig. 7.2 show the losses, meaning how well the models' predictions match the actual labels; lower loss values indicate greater model performance.

This section's analysis focuses on models trained on unnormalized data. The Appendix contains results comparing the accuracy and loss of models trained on normalized data.

The model 'Two channels LSTM v1' stands out from the visual data shown in Figs. 7.1 and 7.2, achieving the highest accuracy (0.96) among the rest of the models. This demonstrates its more effective ability to determine the validity of acceleration signals within the test set. In addition to its high level of accuracy, the model's low loss attests to its refined performance. While a high accuracy implies that the model frequently correctly classifies the signals, a low loss reflects the model's confidence in its predictions. Even when the model misclassifies, the confidence level of its incorrect prediction remains quite close to the correct outcome, ensuring that errors are not extremely off-target. Due to its high accuracy and low loss, it is the optimal but also the most complex model in this study.

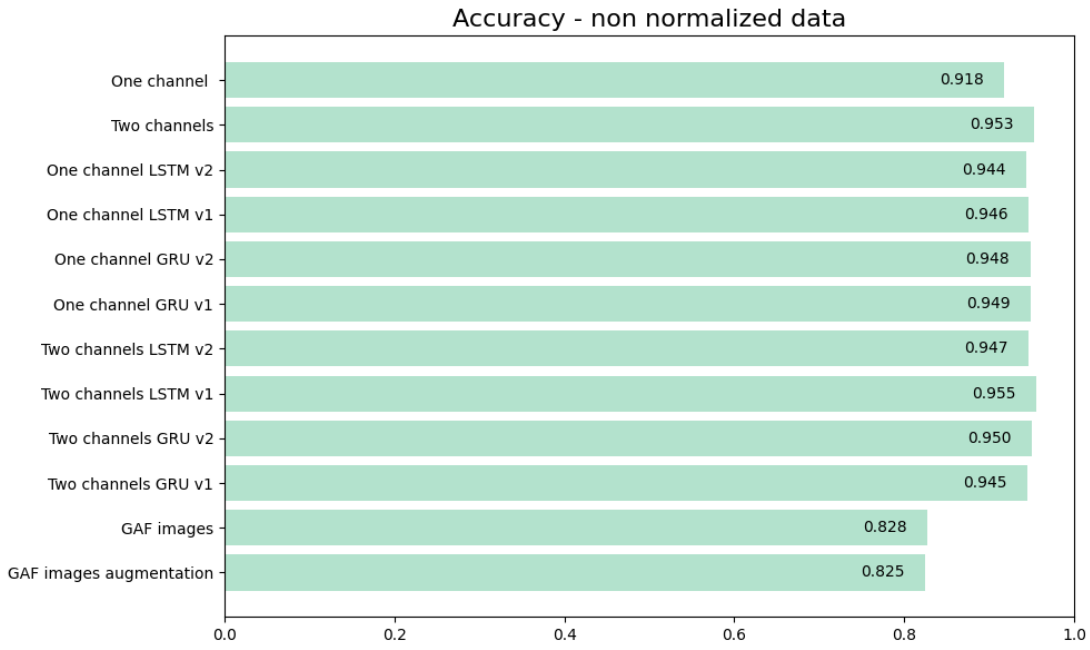


Figure 7.1: Comparison of models accuracy - non normalized data.

When it comes to model versions, there are subtle differences in performance between v1 and v2 for both LSTM and GRU designs. The close performance metrics

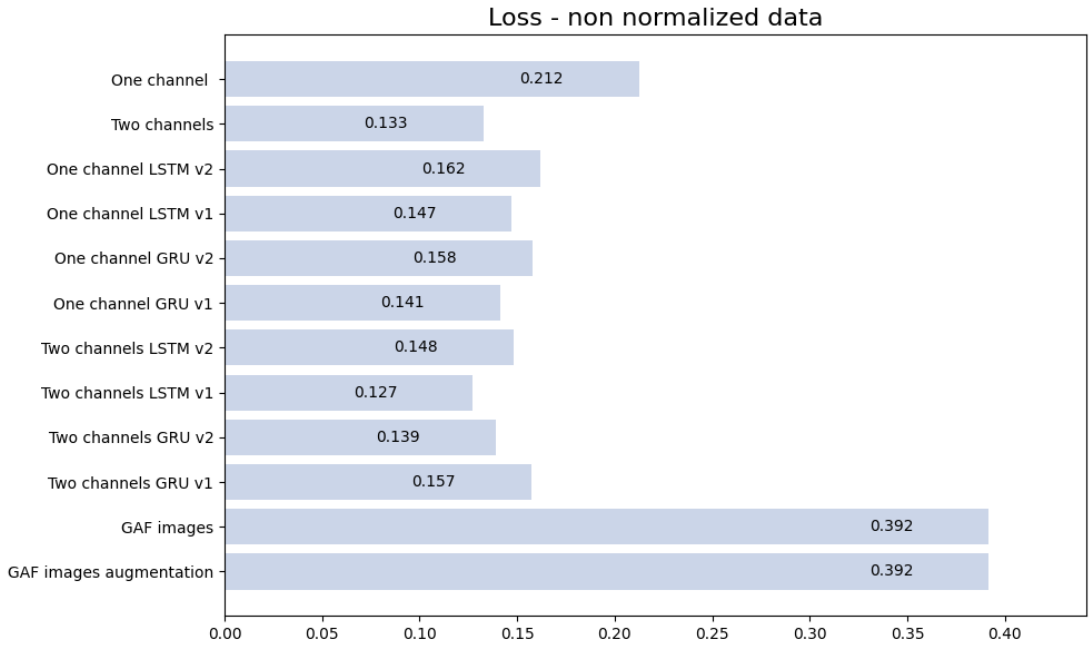


Figure 7.2: Comparison of models loss - non normalized data.

between v1 and v2, with v1 representing the more intricate models equipped with two LSTM/GRU layers and v2 representing a streamlined version, highlight an important realization: the added complexity of v1 may be superfluous, especially when computational efficiency is paramount, as in the case of subsequent deployments on a microcontroller. While the differences in accuracy are slight, the difference in loss, while still minor, is more noticeable. This underscores the fact that architectural differences between versions do have an impact on the overall effectiveness of the model, but it is rather small.

The comparison of the 'GAF images' and 'GAF image augmentation' models makes it clear that their performance metrics are similar, but that they remain behind of other models in terms of accuracy and loss. Their pronounced loss values indicate bigger prediction errors, meaning that they are less reliable classifiers for the task in question.

When comparing one-channel (1C) and two-channel (2C) configurations, a noticeable trend can be seen: two-channel models, albeit slightly, outperform their one-channel counterparts. This confirms the incremental advantages of the second

channel in the context of signal classification. Moreover, the performance of the models, with the exception of those using GAF images as input data, is still very comparable, with a minimum accuracy of 0.92 of the simplest model, the one with only one input channel. In addition, it is important to note that the two-channel model achieves an accuracy and loss of 0.953 and 0.133, respectively, which are very similar to the optimal values of the 'Two channels LSTM v1' model (accuracy 0.955 and loss 0.127), which, however, has a much complex structure.

When the models were trained using normalized data (Figs. A.1 and A.2), their performance patterns shifted slightly. Normalization, which is often used to standardize the range of independent variables or data features, appeared to have a subtle influence on the models. For the vast majority of them, there was either a marginal reduction in accuracy or stayed basically unchanged. This is intriguing because normalization is frequently used to improve the training dynamics of models, particularly neural networks, by ensuring all input features have a similar scale.

More notably, when trained on normalized data, the loss values for these models increased in general. A high loss suggests that, despite potentially correct classifications, the model's predictions may not be as close or confident to the true labels as desired.

These observations give rise to a few hypotheses. It is possible that the underlying scale and distribution of the original, non normalized data include essential nuances or patterns critical to the project at hand. Therefore, models trained with normalized data were not analyzed further since the performance of these models is inferior to that achievable with the original data.

7.1.2 Analysis of Precision, Recall and F1 Score

Focusing on the other key parameters, precision, recall, and F1 score, is critical for thorough examination. These measures give light on the models' intricate performance features, particularly when it comes to classification complexities. Precision and recall are frequently in conflict. A model with a high precision shows

that it is very confident in its forecast when it predicts a positive class. A high precision in the context of detecting anomalies in accelerator pedal signals means that when the model predicts a signal as error-free, it is mostly error-free. A high recall, on the other hand, shows that the model properly identifies a large proportion of the actual positive class. A model with a high recall would capture the majority of truly error-free signals but may misclassify some faulty signals as error-free. The F1 Score combines the precision and recall strengths into a single score. A high F1 Score implies that the model's precision and recall are both high, which is excellent because it suggests the model is both reliable when predicting a positive class and captures a large proportion of the actual positive class.

Heatmaps, as shown in Figs. 7.3, 7.4 and 7.5, give a comprehensive visual depiction of these metrics for one-channel and two-channels configurations of the LSTM and GRU models and for the convolutional models. Several conclusions can be drawn from these heatmaps: the precision of the 'Two channels GRU v1' model is 0.936, suggesting its resilience in classifying error-free signals. The 'Two-channel LSTM v1' model, on the other hand, stands out with an excellent recall of 0.989, indicating that it accurately captures almost all real signals but may potentially produce some false positives. In general, 2C RNN models outperform 1C RNN models in terms of precision and recall. The 'Two-channel GRU v1' model, on the other hand, is an outlier, with a recall of 0.957, the lowest among the RNN models.

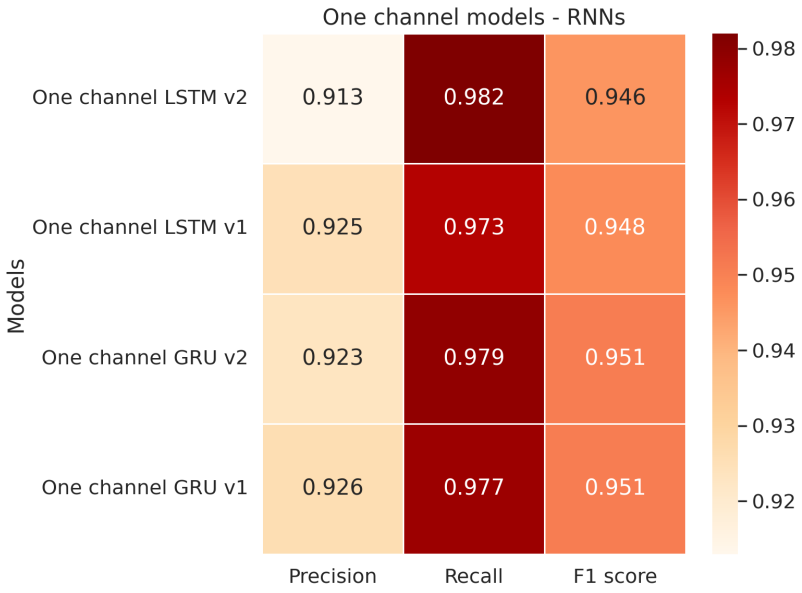


Figure 7.3: Heatmap of precision, recall and F1 score for one channel RNNs models.

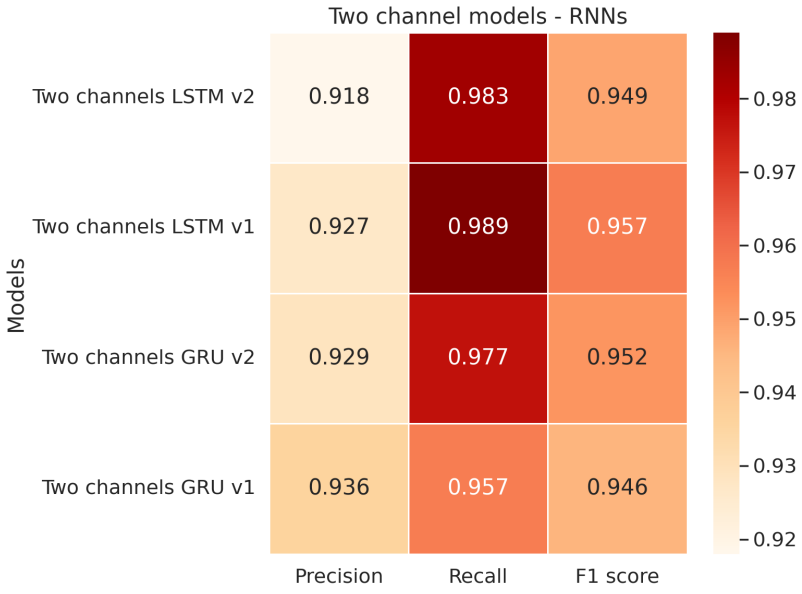


Figure 7.4: Heatmap of precision, recall and F1 score for two channel RNNs models.

Precision values for the 'GAF images' and 'GAF images augmentation' models are 0.79 and 0.792, respectively (Fig. 7.5). This lower precision suggests a tendency to confuse erroneous signals as error-free signals, which is a major concern for the real-world applications. Therefore, these models were removed from next steps of the thesis due to their poor performance, as evidenced also by their accuracy scores.

Precision is critical in the context of electric vehicle acceleration signals. Mistaking an erroneous signal for error-free can have major safety consequences. According to this perspective, the convolutional model with two channels is the most balanced option. It not only competes with the 'Two-channel LSTM v1' model in accuracy and loss, but also outperforms it in precision, with a score of 0.934 versus 0.927, while being less complex.

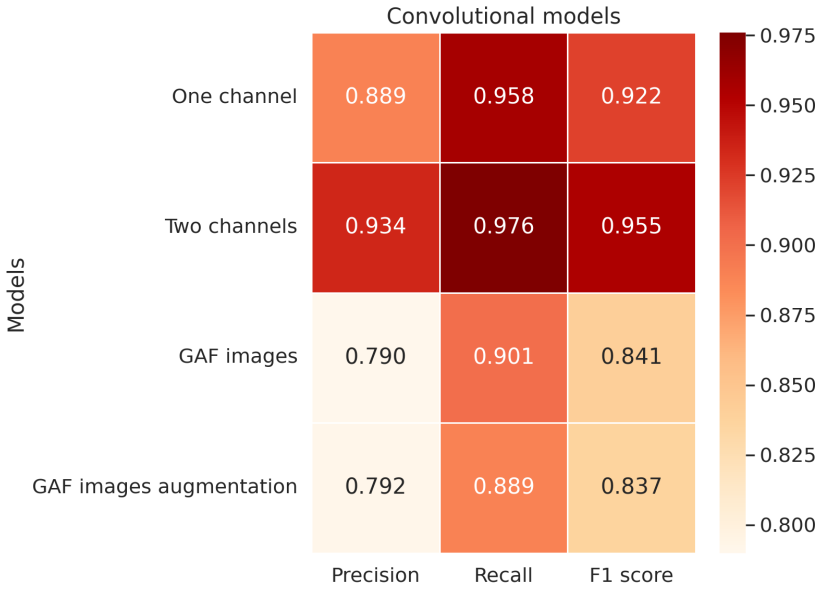


Figure 7.5: Heatmap of precision, recall and F1 score for convolutional models.

7.2 TensorFlow Models vs TFLite Models

Following the evaluation of the performance of generated TensorFlow models, the next step was to investigate the behavior of pruning and clustering on convolutional models to assess which of the two techniques achieves better results when converting the models to their TFLite version. The objective is to create models that are accurate but much smaller in size than the original models. Figures 7.6 and 7.7 demonstrate the findings for models with one and two channels, respectively. The data visualizations compare the accuracy of the test set for each model to its size in Kilobytes. The light blue bars reflect model accuracy plotted against the left y-axis, while the light coral bars show model size plotted against the right y-axis. The baseline models, '1C_baseline' and '2C_baseline', serve as reference

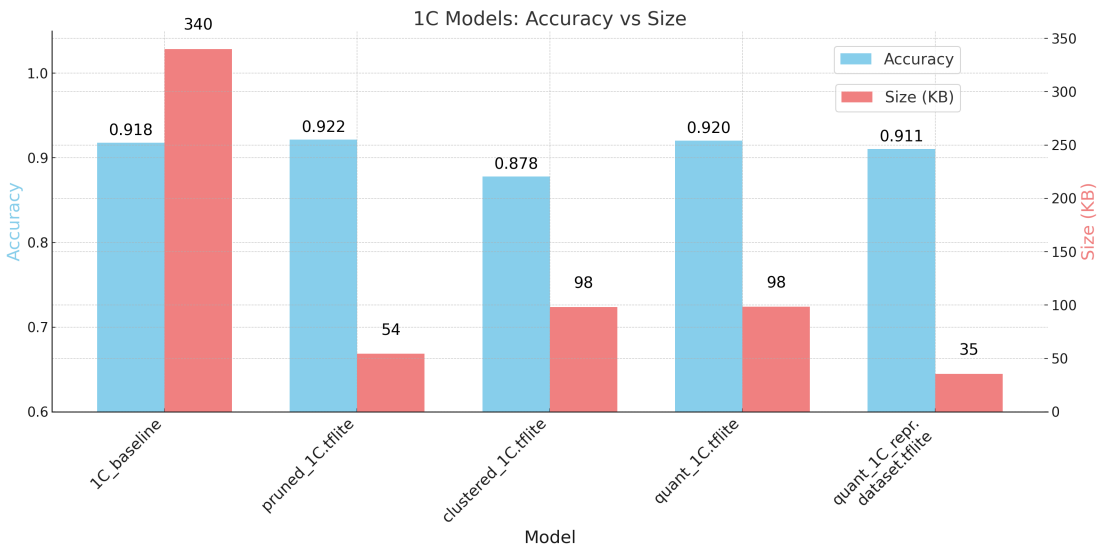


Figure 7.6: Accuracy versus size of the optimized 1C CNNs.

points, emphasizing the large reductions in model size obtained by optimization while preserving or even enhancing accuracy. The baseline models are the ones presented on the left of the Figures: as can be seen, the sizes of both the 1C and 2C CNN models, 340 KB and 803 KB respectively, are significantly larger than their optimized counterparts. Thus, while adding two channels for input data improves performance, it also results in a noticeable increase in model size.

The 'pruned_1C.tflite' model has a worthy accuracy of 0.922, which is slightly higher than the accuracy of the baseline model. Surprisingly, this outcome was obtained while retaining an extremely small size of only 54 KB. Regarding the two-channel (Fig. 7.7) models, the pruned model, labeled as 'pruned_2C.tflite', also performed better than the original model by registering an accuracy of 0.956, all contained in a size of 131 KB. It is worth noting that clustered models perform poorly in terms of accuracy as well as their inability to reduce size as well as pruned models. Because both clustering and pruning are optimization techniques that set the ground for future quantization, only pruning was used. As a result, quantization is done on the pruned models.

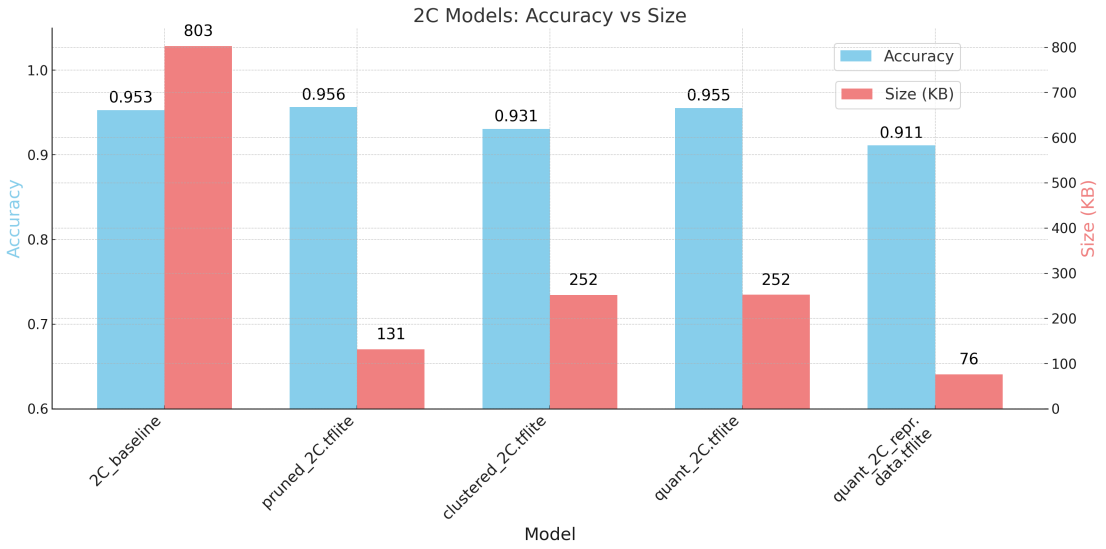


Figure 7.7: Accuracy versus size of the optimized 2C CNNs.

The quantized variation, 'quant_1C.tflite', matched the unquantized pruned counterpart's accuracy for the 1C model, hovering around 0.920 while being roughly twice as large in size. Interestingly, the quantized model with a representative dataset, 'quant_1C_repr_dataset.tflite', sacrificed some accuracy (approximately 0.911) in exchange for a drastically decreased size of 35 KB. The same behavior is followed by the 2C model: when quantized using a representative dataset it achieves an accuracy of around 0.911, but with a significant decrease in size compared to its quantized counterpart model without the representative dataset, from 252 KB to

76 KB.

Given the inherent compactness of quantized models, even in the absence of a representative dataset, the choice was to prioritize accuracy over model size. Moreover, as anticipated in the previous chapter, the attempt to quantize the RNN models using the representative dataset proved problematic. This reinforced the decision to discontinue its use, resulting also in a more consistent comparison between all versions of the model.

In accordance with the analysis of convolutional models, a similar analytical technique was used for RNN models. The purpose is to compare the performance of pruned and quantized RNN models to their respective baseline equivalents. Figures 7.8, 7.9, 7.10 and 7.11 show these comparisons, which give further insight into the accuracy and dimension of the models. The visualizations depict the LSTM and GRU models in the two versions proposed and compared to the baselines, split by input data from one or two channels.

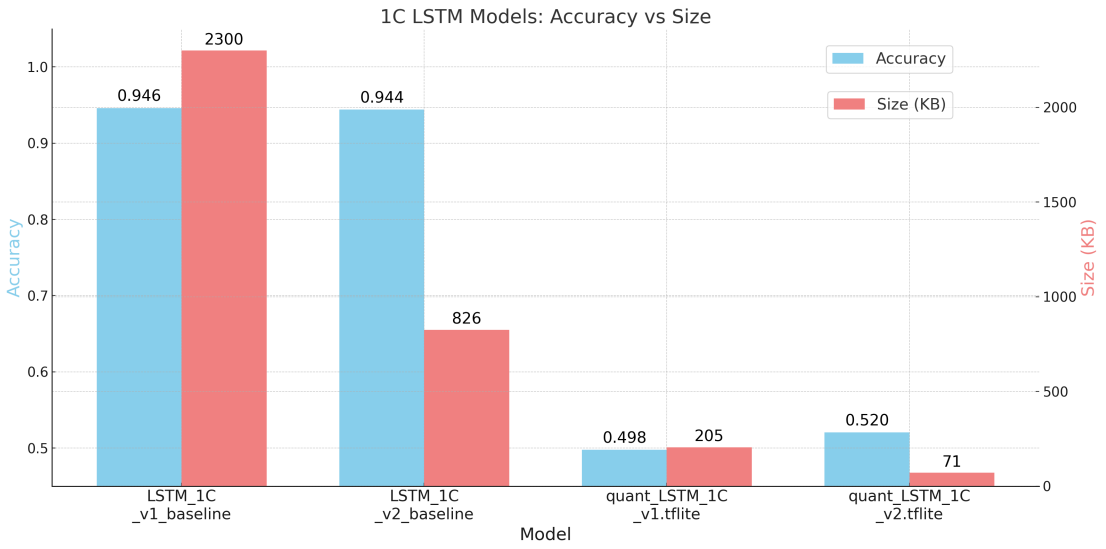


Figure 7.8: Accuracy versus size of the LSTM 1C models.

To begin, it is critical to emphasize how much the decision of having two LSTM or GRU layers vs one layer, version 1 and version 2, effects the size. Version 1

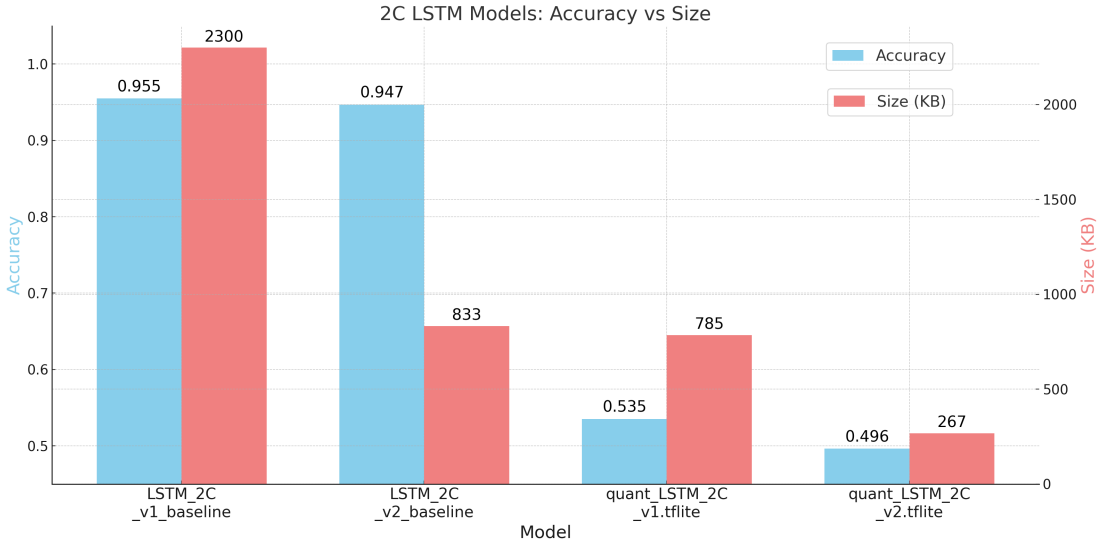


Figure 7.9: Accuracy versus size of the LSTM 2C models.

is approximately three times bigger than the simpler version for LSTM models, while version 1 is more than four times larger than version 2 for GRU models. As previously stated, the complexity provided by version 1 does not provide a sufficient benefit in accuracy to justify its usage; so, the one-layer versions will be the focus of the following investigation. For example, the 'LSTM_1C_v2_baseline' model, which had an accuracy of 0.944 and a size of 826 KB before quantization, has an accuracy of roughly 0.497 after quantization ('quant_LSTM_1C_v2.tflite'). The accuracy of its two-channel equivalent, 'LSTM_2C_v2_baseline', has likewise decreased from 0.947 to 0.496 in the 'quant_LSTM_2C_v2.tflite' model.

TensorFlow has not fully optimized and incorporated the quantization for RNN layers, as stated in the previous Chapter. This constraint becomes clear when looking at the accuracy drop showed in Figs. 7.8 and 7.9. The accuracy of the LSTM models almost halves as compared to their baseline versions, resulting in a considerable performance reduction. This considerable drop shows the difficulties and limitations associated with quantizing RNNs using existing approaches.

The GRU models, on the other hand, provide a more optimistic picture. The 'quant_GRU_1C_v2.tflite' model, Fig. 7.10, for example, with an accuracy of 0.949 and a size of 122 KB, marginally outperforms the baseline version while half its size.

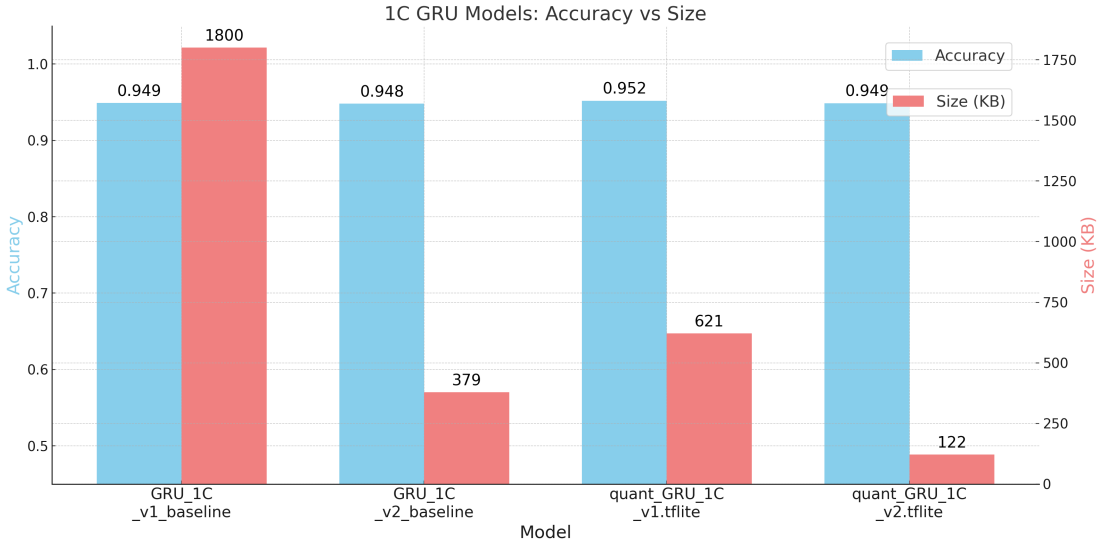


Figure 7.10: Accuracy versus size of the GRU 1C models.

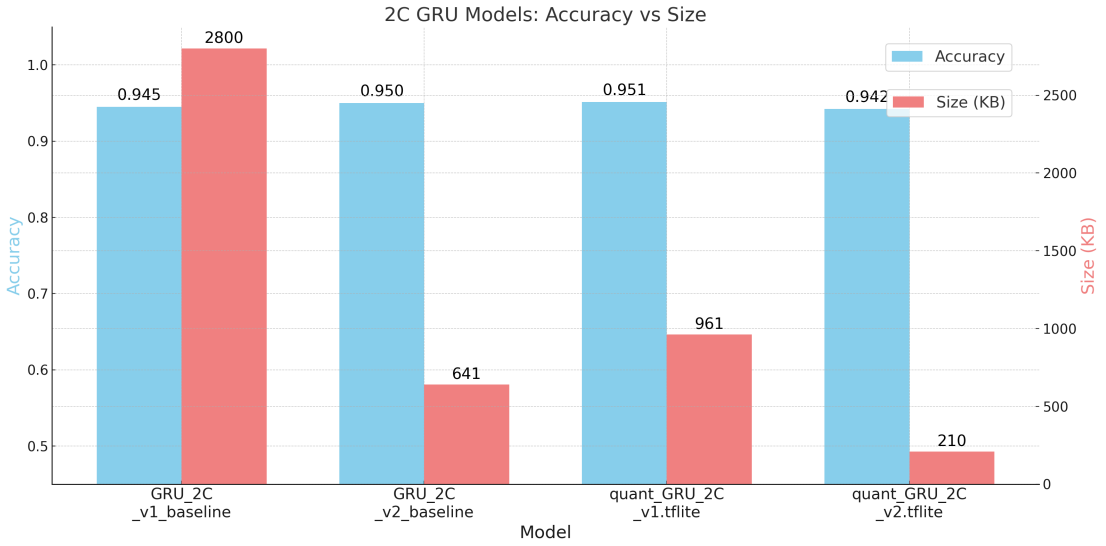


Figure 7.11: Accuracy versus size of the GRU 2C models.

Unlike LSTM models, the accuracy of GRU models remains largely consistent after quantization. Given the current status of quantization methods, this robustness implies that GRU models may be better appropriate for usage in constrained contexts such as microcontrollers. Because of their consistent performance and modest size, they are a potential choice for such real-world applications.

7.3 Results for Model Deployment on RP2040

It is difficult to implement machine learning models on microcontrollers such as the RP2040. Memory, computing power, and real-time requirements frequently necessitate careful optimization and model selection. Edge Impulse includes a `profile()` function that is a response to these problems. This method accomplishes two tasks. First, it determines if a particular model is compatible with the target hardware, in this case the RP2040. It also allows users to assess their model's inference time, offering insight into its real-time performance. To offer an example, the result of convolutional model profiling with two channels of input data is provided in the Appendix.
The result includes a large amount of information. It begins by providing details on the RP2040-specific performance metrics, such as the TensorFlow Lite model's file size, compatibility with the microcontroller, and memory requirements for the TensorFlow Lite and EON compilers. EON is a compiler developed by Edge Impulse to optimize NN for deployment on embedded devices. Furthermore, the time necessary for a single inference on the RP2040 is provided (Listing B.1). After that, performance estimates are supplied for a variety of devices, ranging from low-end microcontrollers to high-end GPUs or neural network accelerators. These estimates comprise device type descriptions, inference times, memory requirements, and compatibility statuses. These detailed profile data help developers in determining the feasibility and efficiency of executing their models on a variety of hardware platforms (Listing B.2).

Following the results of the performance of the machine learning models created, the profile of the one- and two-channel versions of the following models: CNN, LSTM v2, and GRU v2 was deepened. The outputs given by Edge Impulse's `profile()` function prompted this further in-depth investigation. Table 7.1 summarizes RP2040's support for these models as well as their corresponding inference times. Interestingly, the `profile()` method can determine inference time also for unsupported models. This is due to the fact that the function simulates the model's execution to assess its computational complexity, providing insight into

TFLite model	Supported on RP2040	Inference Time (ms)
1C	True	1693
2C	True	1889
GRU_1C_v2	False	6704
GRU_2C_v2	False	4120
LSTM_1C_v2	False	43821
LSTM_2C_v2	False	30240

Table 7.1: Comparison of model support and inference time on Raspberry Pi RP2040.

how long an inference could take, even if the real implementation on the target device is impeded by memory or architectural restrictions. This simulation can give significant insights into a model's computing requirements. It should, however, be regarded as a recommendation rather than an absolute. In this research in particular, all models have estimated inference times that are much longer than the intended 100 ms threshold. The value of this threshold was decide in the *Evergrin* project as a adequate value for a functionality evaluation of the system.
The research project considers neural networks to be an auxiliary control mechanism. While the control unit monitors the hardware's performance, the neural network models monitor any possible malfunctions or abnormalities in the driver's behavior. The 100 ms threshold was not picked at random: it represents the lowest limit of human reaction time, making it an important benchmark for real-time responsiveness. Given this significant disparity, it was decided that testing the inference time in the real-world setting was unnecessary because the models had already surpassed permissible bounds in the simulated environment.

Although CNN models are the only ones that can be used with the RP2040, their inference times are still not ideal. This raises concerns regarding the real-time applicability of even the most basic models on such constrained hardware, particularly in applications where response time is crucial.
LSTM v2 models, which are noted for their capacity to memorize patterns across lengthy sequences, require more memory by definition. The error 'computed arena size is >6MB' obtained from the `profile()` function emphasizes not just the

memory-intensive nature of LSTMs, but also the challenges of implementing advanced neural network architectures on microcontrollers. The recurring nature of LSTMs, along with the requirement to retain each unit's internal states, produces a memory demand that the RP2040 cannot satisfy. This constraint implies that, in addition to not working well with present quantization approaches, LSTMs may be better suited to more powerful devices or platforms with greater memory.
On the other hand, GRU v2 models, a different class of recurrent neural network, provide another level of difficulty. The error 'The model has multiple subgraphs, only one is supported' implies that the model is structurally complex. Multiple subgraphs can be produced by branching architectures or models designed to process multiple types of input at the same time. Although such designs might be powerful and adaptable, they can be difficult to implement on platforms like the RP2040 with today's technology. Handling models that involve multiple computational paths requires complications beyond the capability of the microcontroller implementation architecture, which is meant for simplicity and efficiency.

Given these findings, it is evident that microcontrollers like the RP2040 do not provide adequate opportunities for edge computing and embedded machine learning due to tangible restrictions. Anyway, taking into account all of the previous evaluations, the TFLite version of the convolutional model with two input channels (2C CNN) emerges as the best option. This model not only has excellent accuracy and precision, but it is also natively supported by the RP2040.
Figures 7.12 and 7.13 show the performance of the 2C CNN model in the two different versions: the TensorFlow implementation and its simplified TFLite equivalent, which has been quantized and pruned. A comparison of these settings shows that the TFLite version has a slightly higher false positive rate than its TensorFlow equivalent. This increase is not optimum for the purposes mentioned in this thesis, because errors in signals should always be identified, but the difference between the two models is minimal. The TFLite version, on the other hand, has a lower false negative rate, indicating that it is more successful at properly classifying instances that are positive. This observed improvement is further confirmed by the TFLite version's higher percentage of true positives.

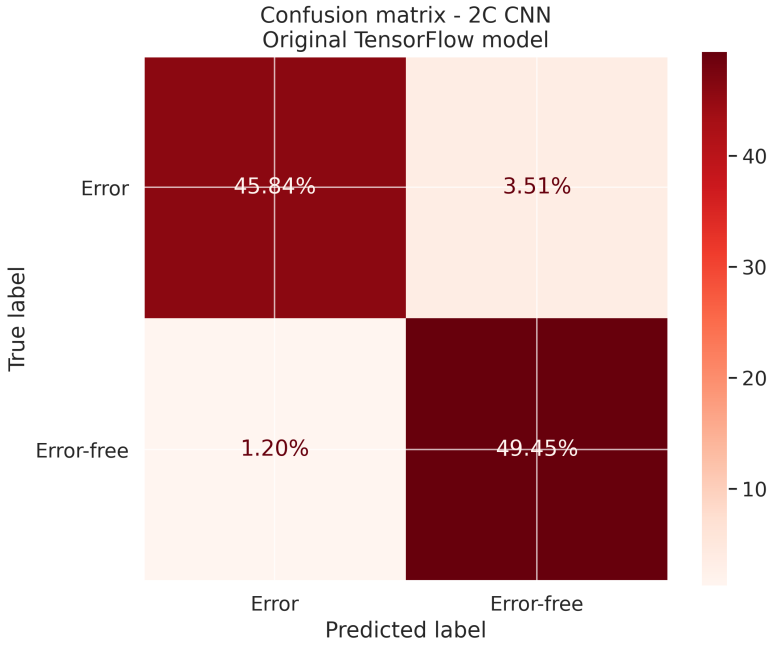


Figure 7.12: Confusion matrix for the 2C CNN TensorFlow model.

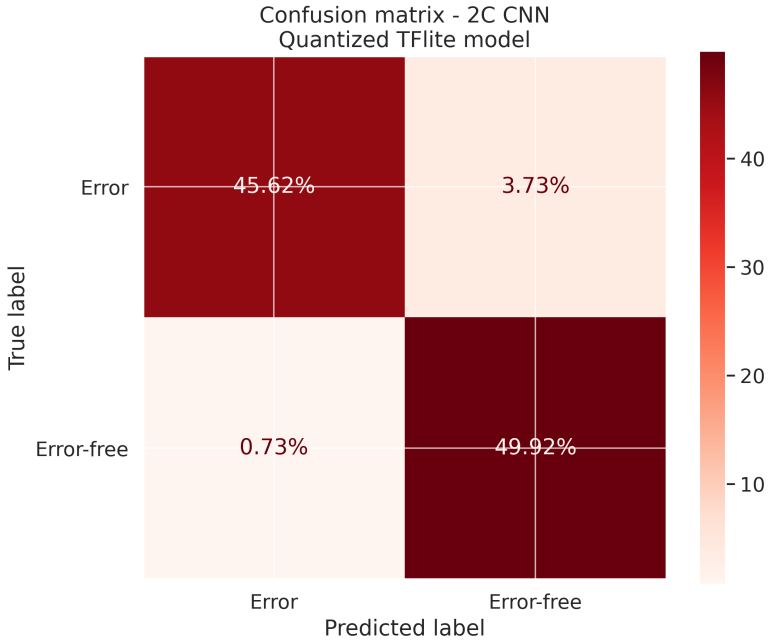


Figure 7.13: Confusion matrix for the 2C CNN TensorFlow Lite model.

Despite the great performance of the TFLite version of the 2C CNN model, more powerful computer systems are required to obtain an inference time according to the established threshold. The results supplied in the Appendix B demonstrate that the profiling function offers evidence of this. For example, high-end Microcontroller Units (MCUs), such as the Cortex-M7 [40] or comparable devices operating at 240 MHz, have dramatically reduced inference time by 86 ms. The Cortex-M7 is a high-performance ARM Cortex series core intended for usage in a variety of devices ranging from microcontrollers to complete embedded systems. It is a popular choice for demanding embedded applications due to its characteristics such as double-precision floating point, high code density, and ease of use. These high-end MCUs or DSPs (Digital Signal Processors) are purpose-built to perform complicated mathematical operations and algorithms, making them perfect for running advanced machine learning models.

Chapter 8

Conclusions and Future Work

The incorporation of artificial intelligence into electric vehicles indicates a major transformation in the world of automotive manufacturing. This thesis delves into the complex world of accelerator pedal signal classification using neural networks, a research project that has significant potential to improve vehicle safety and control. The role of artificial intelligence in ensuring functional safety and optimizing control mechanisms will become increasingly important as the AI field expands. In particular, given the growing need to convert internal combustion vehicles to EVs, as demonstrated by the *Evergrin* project, the market for EVs is also growing, leading to a consequent merging of the two sectors.

To answer key research questions, this study conducted an in-depth analysis of neural network architectures and their efficiency in anomaly detection. The significant potential of these networks in identifying errors in accelerator pedal signals was evident. The research uses a funnel approach to find the best model for deployment on a microcontroller. Several TensorFlow models were tested by initially evaluating metrics such as accuracy and loss, and then performing a more comprehensive study using the metrics of precision, recall, and F1 score. The results showed that some models can discriminate very well between error-free signals and erroneous

signals, providing the basis for the incorporation of AI-driven safety systems in future EVs. CNN models with time series input, LSTM and GRU are the best performing models, with one of the LSTM models achieving the highest accuracy of 96%.

The study also highlighted the performance dynamics of TensorFlow Lite models compared to their more complex neural network equivalence. The transition of TensorFLow models to TFLite, which had been enhanced by pruning and quantization techniques, revealed several performance changes. Although the TFLite models appear to be suited for microcontroller implementation due to their compact structure, a noticeable difference in their performance was seen when compared to the original models, particularly for recurrent networks. This is due to the existing scarcity of suitable quantization approaches for complicated models like LSTM and GRU. A selection of these TFLite models were evaluated for compatibility with microcontroller integration using the Python SDK offered by Edge Impulse. The RP2040 only supports CNN models, but even though these are the simplest models, the inference duration was significantly longer than the 100 ms limit set by this research.

The inference time associated with neural network models deployed on microcontrollers was highlighted as a major challenge. The demanding need for real-time responses in the automotive industry is motivated by both security imperatives and the prospect of being able to take advantage of future technology breakthroughs for functional checks and against potential malicious attacks.

This research sets neural networks as an auxiliary control mechanism within EVs. While the primary control system is intended to take care of hardware failures, the neural network serves as a fallback system, meticulously monitoring any user-induced malfunctions or security risks. Consider the following scenario: a driver, due to unanticipated circumstances such as as an illness, keeps the accelerator pushed for an extended length of time. In such circumstances, a sophisticated neural network supplied with data from many sensors – including steering wheel inputs, brake pedal usage, and other ADAS (Advanced Driver Assistance Systems)

components – can identify this as unusual driving behavior that deviates from conventional driving patterns. The system can determine if the extended accelerator pressure is a genuine anomaly, such as driver illness, or the consequence of other circumstances, such deliberate acceleration over a lengthy stretch of road, by assessing the holistic data, and can then take corrective action.
Moreover, as vehicles become increasingly interconnected and part of complex vehicular networks, the importance of neural networks may expand beyond their traditional applications. They will be able to be used to discover complex anomalies in the huge amounts of data transmitted through these networks. Neural networks will be able to quickly detect deviations or anomalies that could be indicative of a cyber attack by learning common patterns of data transmission and system behavior. This proactive detection capability is critical to ensuring the cybersecurity of modern vehicles, especially as they are increasingly connected to smart city infrastructure and other IoT devices. In essence, neural networks will be able to act as vigilant sentinels, strengthening cybersecurity defenses against possible cyber attacks in these complex systems.
Given these factors, there is a clear and pressing need for models that can analyze data rapidly without affecting accuracy. For microcontroller implementation, the 2C CNN model has emerged as the most promising and practicable so far. To obtain the necessary latency results, however, a more powerful platform than the RP2040, such as the Cortex-M7, may be better suited to the task.

Looking ahead to the future of neural network applications in the automobile field, various opportunities for research and improvement arise. In the context of this thesis, it is necessary to expand the dataset used for training and testing neural networks. The existing approach, which involves retrieving data from a combustion engine vehicle's throttle valve position data, was critical for this research, as there was no possibility to directly obtain data from the *Evergrin* prototype, which was not yet operational. In the future, with *Evergrin*'s working prototype, it will be practicable to obtain real data with which to train the networks, potentially improving the resilience and reliability of the models.
Furthermore, the quick rate of technical advancements in the field of pattern

quantization should not be underestimated. Complex models, like LSTM and GRU, are currently challenging to incorporate into microcontrollers. However, as quantization techniques evolve and become more sophisticated, it is expected that these models will be able to be deployed on microcontrollers without issue in the near future. Given the higher performance parameters associated with LSTM models in preliminary testing, there is a strong argument in favor of investing research effort in order to adapt these models for microcontroller implementation. The potential benefits could alter in the future how neural networks contribute to vehicle safety and functionality.

Appendix A

Analysis of Accuracy and Loss for Normalized Input Data

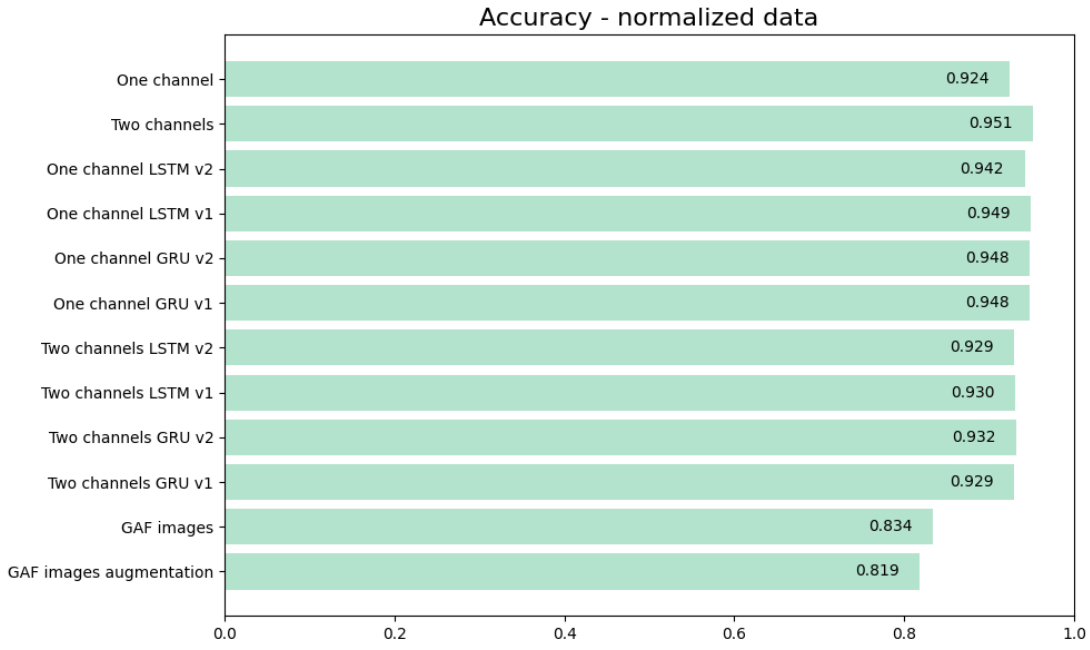


Figure A.1: Comparison of models accuracy - normalized data.

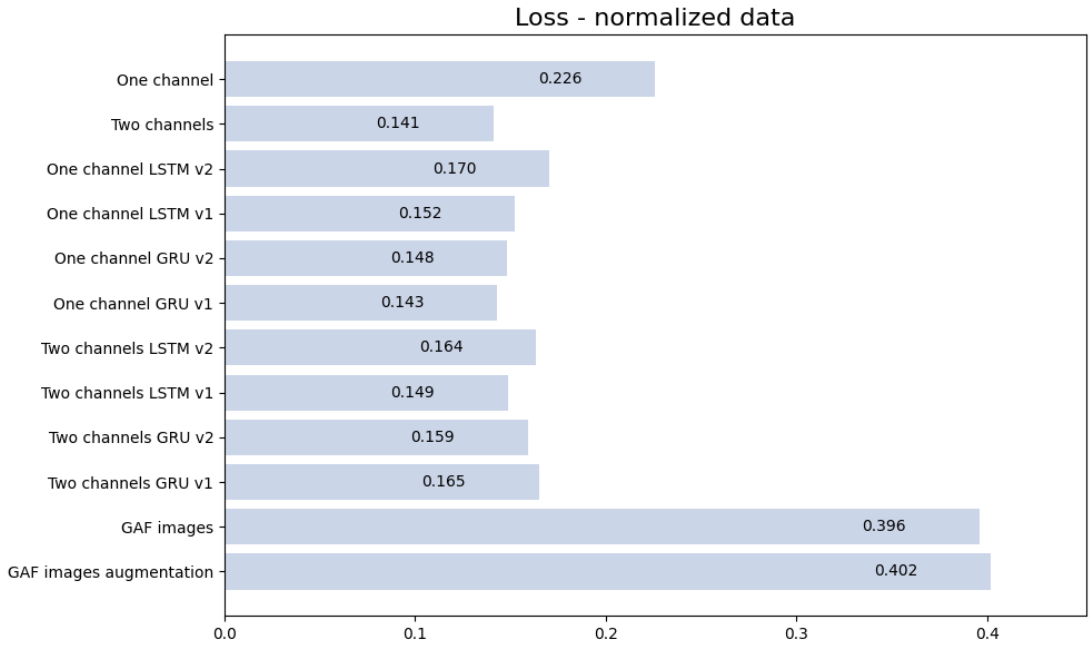


Figure A.2: Comparison of models loss - normalized data.

Appendix B

Profile Output 2C CNN model

Listing B.1: Performance on Raspberry Pi RP2040


```
{
    "device": "raspberry-pi-rp2040",
    "tfliteFileSizeBytes": 258336,
    "isSupportedOnMcu": true,
    "memory": {
        "tflite": {
            "ram": 27275,
            "rom": 323824,
            "arenaSize": 26771
        },
        "eon": {
            "ram": 20664,
            "rom": 295792
        }
    },
    "timePerInferenceMs": 1889
}
```

Listing B.2: Performance on other device types


```
{
    "variant": "float32",
    "lowEndMcu": {
        "description": "Estimate for a Cortex-M0+ or similar
   , running at 40MHz",
        "timePerInferenceMs": 6281,
        "memory": {
            "tflite": {
                "ram": 29675,
                "rom": 307920
            },
            "eon": {
                "ram": 22648,
                "rom": 287136
            }
        },
        "supported": true
    },
    "highEndMcu": {
        "description": "Estimate for a Cortex-M7 or other
   high-end MCU/DSP, running at 240MHz",
        "timePerInferenceMs": 86,
        "memory": {
            "tflite": {
                "ram": 27275,
                "rom": 323824
            },
            "eon": {
                "ram": 20664,
                "rom": 295792
            }
        },
        "supported": true
```

```
    },
    "highEndMcuPlusAccelerator": {
        "description": "Most accelerators only accelerate quantized models.",
        "timePerInferenceMs": 86,
        "memory": {
            "tflite": {
                "ram": 27275,
                "rom": 323824
            },
            "eon": {
                "ram": 20664,
                "rom": 295792
            }
        },
        "supported": true
    },
    "mpu": {
        "description": "Estimate for a Cortex-A72, x86 or other mid-range microprocessor running at 1.5GHz",
        "timePerInferenceMs": 2,
        "rom": 258336.0,
        "supported": true
    },
    "gpuOrMpuAccelerator": {
        "description": "Estimate for a GPU or high-end neural network accelerator",
        "timePerInferenceMs": 1,
        "rom": 258336.0,
        "supported": true
    }
}
```

Ringraziamenti

Desidero esprimere la mia profonda gratitudine al relatore di questa tesi di laurea, il Professor Luca Vassio, per avermi indirizzato e guidato durante la stesura ed elaborazione di questo lavoro. La sua disponibilità e i consigli preziosi sono stati fondamentali per il completamento di questo percorso.
Un sentito ringraziamento va a Brain Technologies per avermi dato la possibilità di immergermi in un ambiente aziendale autentico. Anche se il contesto era leggermente differente dal mio campo di studi, avete dimostrato supporto e pazienza, guidandomi passo dopo passo nella realizzazione di questa tesi. Lavorare con un team così eccezionale mi ha infuso fiducia e speranza per il mio futuro professionale. In particolare, desidero ringraziare Luca Bussi per la sua costante disponibilità, per avermi motivata e stimolata durante questo periodo.
Infine, non posso non esprimere la mia eterna gratitudine alla mia famiglia e ai miei amici. Grazie per il vostro incondizionato supporto, per la pazienza dimostrata e per l'amore che mi avete sempre riservato.

Bibliography

[1] IEA. *Global EV Outlook 2023*. License: CC BY 4.0. 2023. URL: https://www.iea.org/reports/global-ev-outlook-2023 (cit. on pp. 1, 2).

[2] *DECRETO 1 dicembre 2015, n. 219*. Gazzetta Ufficiale. Available from: https://www.gazzettaufficiale.it/eli/id/2016/01/11/15G00232/sg. 2017 (cit. on p. 2).

[3] International Organization for Standardization. *ISO 26262: Road vehicles – Functional safety*. 2018. URL: https://www.iso.org/standard/68383.html (cit. on pp. 3, 23).

[4] Ayushi Chahal and Preeti Gulia. «Machine Learning and Deep Learning». In: *International Journal of Innovative Technology and Exploring Engineering* 8 (Oct. 2019), pp. 4910–4914. DOI: 10.35940/ijitee.L3550.1081219 (cit. on p. 4).

[5] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. First Edition, December 2019. ISBN: 978-1-492-05204-3. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2020. URL: http://oreilly.com/catalog/errata.csp?isbn=9781492052043 (cit. on p. 5).

[6] Zhiguang Wang, Weizhong Yan, and Tim Oates. «Time series classification from scratch with deep neural networks: A strong baseline». In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 1578–1585. DOI: 10.1109/IJCNN.2017.7966039 (cit. on p. 11).

[7] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. «Convolutional neural networks for time series classification». In: *Journal of Systems Engineering and Electronics* 28.1 (Feb. 2017), pp. 162–169 (cit. on p. 11).

[8] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2019. arXiv: 1803.08375 [cs.NE] (cit. on p. 12).

[9] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. 2022. arXiv: 2109.14545 [cs.LG] (cit. on p. 12).

[10] Y. Bengio, P. Simard, and P. Frasconi. «Learning long-term dependencies with gradient descent is difficult». In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166 (cit. on p. 14).

[11] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 14).

[12] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE] (cit. on p. 15).

[13] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL] (cit. on p. 15).

[14] Partha Pratim Ray. «A review on TinyML: State-of-the-art and prospects». In: *Journal of King Saud University - Computer and Information Sciences* 34.4 (2022), pp. 1595–1623. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.11.019. URL: https://www.sciencedirect.com/science/article/pii/S1319157821003335 (cit. on p. 17).

[15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 17, 27).

[16] TensorFlow. *TensorFlow Lite: Deploy Machine Learning Models on Mobile and Edge Devices*. TensorFlow Lite is a mobile library for deploying models on mobile, microcontrollers and other edge devices. 2023. URL: https://www.tensorflow.org/lite (cit. on pp. 17, 56).

[17] Edge Impulse. *Build with the World's Top Hardware, Sensors, and Cloud Platforms*. Benefit from built-in integrations with leading partner ecosystem including MCUs to MPUs and GPUs, sensors, cloud services, data science tools, and digital twin platforms. 2023. URL: https://www.edgeimpulse.com/ (cit. on pp. 17, 63).

[18] Raspberry Pi Foundation. *RP2040: High Performance. Low Cost. Small Package*. 2023. URL: https://www.raspberrypi.com/products/rp2040/ (cit. on p. 17).

[19] François Chollet. *Keras*. https://github.com/fchollet/keras. 2015 (cit. on p. 27).

[20] Hongji Xu, Juan Li, Hui Yuan, Qiang Liu, Shidi Fan, Tiankuo Li, and Xiaojie Sun. «Human Activity Recognition Based on Gramian Angular Field and Deep Convolutional Neural Network». In: *IEEE Access* 8 (2020), pp. 199393–199405. DOI: 10.1109/ACCESS.2020.3032699 (cit. on p. 33).

[21] Johann Faouzi and Hicham Janati. «pyts: A Python Package for Time Series Classification». In: *Journal of Machine Learning Research* 21.46 (2020), pp. 1–6. URL: http://jmlr.org/papers/v21/19-763.html (cit. on p. 33).

[22] Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023. DOI: 10.1017/9781108348973 (cit. on p. 41).

[23] Roman Garnett. *Bayesian Optimization*. Washington University in St. Louis, course : CSE 515T - Bayesian Methods in Machine Learning. 2015. URL: https://www.cse.wustl.edu/~garnett/cse515t/spring_2015/files/lecture_notes/12.pdf (cit. on p. 41).

[24] NVIDIA. *The NVIDIA CUDA®Deep Neural Network library (cuDNN)*. GPU-accelerated library of primitives for deep neural networks. URL: https://developer.nvidia.com/cudnn (cit. on p. 43).

[25] Keras. *GRU Layer*. 2023. URL: https://keras.io/api/layers/recurrent_layers/gru/ (cit. on p. 43).

[26] Keras. *LSTM Layer*. 2023. URL: https://keras.io/api/layers/recurrent_layers/lstm/ (cit. on p. 43).

[27] Google. *Google Colab*. A hosted Jupyter Notebook service that requires no installation and offers free access to computing resources such as GPUs and TPUs. Machine learning, data science, and education are particularly well matched. 2023. URL: https://colab.google/ (cit. on p. 44).

[28] TensorFlow Blog. *Colab's Pay As You Go offers more access to powerful NVIDIA compute for machine learning*. Standard GPUs are typically NVIDIA T4 Tensor Core GPUs, while premium GPUs are typically NVIDIA V100 or A100 Tensor Core GPUs. 2022. URL: https://blog.tensorflow.org/2022/09/colabs-pay-as-you-go-offers-more-access-to-powerful-nvidia-compute-for-machine-learning.html (cit. on p. 44).

[29] NVIDIA. *NVIDIA Tesla T4 GPU*. Accelerates a wide range of cloud tasks, including high-performance computing, deep learning training and inference, machine learning, data analytics, and graphics processing. NVIDIA Turing-based architecture. 2023. URL: https://www.nvidia.com/en-us/data-center/tesla-t4/ (cit. on p. 44).

[30] TensorFlow. *TensorFlow Lite Python API*. 2023. URL: https://www.tensorflow.org/lite/api_docs/python/tf/lite (cit. on p. 56).

[31] TensorFlow. *TensorFlow Lite Conver Models*. 2023. URL: https://www.tensorflow.org/lite/models/convert/convert_models (cit. on p. 59).

[32] TensorFlow. *TensorFlow Lite Conver Models*. 2023. URL: https://www.tensorflow.org/lite/performance/model_optimization (cit. on p. 60).

[33] TensorFlow. *TensorFlow Model Optimization Toolkit*. https://github.com/tensorflow/model-optimization. GitHub repository. 2023 (cit. on p. 60).

[34] ARM. *Cortex-M0+*. ARM, 2023. URL: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m0-plus (cit. on p. 61).

[35] TensorFlow. *TensorFlow Lite Conver Models*. 2023. URL: https://github.com/tensorflow/model-optimization/blob/master/tensorflow_model_optimization/python/core/quantization/keras/default_8bit/default_8bit_quantize_registry.py#L213 (cit. on p. 61).

[36] TensorFlow. *TensorFlow Lite for Microcontrollers*. 2023. URL: https://www.tensorflow.org/lite/microcontrollers (cit. on p. 63).

[37] TensorFlow. *TensorFlow Lite Interpreter*. 2023. URL: https://www.tensorflow.org/lite/api_docs/python/tf/lite/Interpreter (cit. on p. 63).

[38] Daniel Situnayake. *How TensorFlow helps Edge Impulse make ML accessible to embedded engineers*. 2021. URL: https://blog.tensorflow.org/2021/06/how-tensorflow-helps-edge-impulse-make-ml-accessible.html (cit. on p. 64).

[39] Shawn Hymel et al. *Edge Impulse: An MLOps Platform for Tiny Machine Learning*. 2023. arXiv: 2212.03332 [cs.DC] (cit. on p. 65).

[40] ARM. *Cortex-M7*. ARM, 2023. URL: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m7 (cit. on p. 85).