Bringing Edge AI to low-power nRF and STM microcontrollers

University of Turku
Master of Science in Technology Thesis
Department of Computing
Robotics and Autonomous Systems
June 2025
Lauri Jussinmäki

Supervisors:
Prof. Juha Plosila
Dr. Hashem Haghbayan

UNIVERSITY OF TURKU
Department of Computing

Lauri Jussinmäki : Bringing Edge AI to low-power nRF and STM microcontrollers

Master of Science in Technology Thesis, 51 p.
Robotics and Autonomous Systems
June 2025

This thesis investigates the development and deployment of embedded artificial intelligence applications on low-power microcontrollers using the Edge Impulse platform. The work focuses on implementing a real-time keyword spotting system on two platforms: the Nordic Semiconductor nRF5340, and the STMicroelectronics STM32L476. These devices represent commonly used microcontrollers in battery-powered embedded systems without dedicated AI acceleration hardware.

The study demonstrates a complete workflow from dataset creation to model training, optimization, and integration with real-time firmware using both Zephyr and FreeRTOS. The Edge Impulse platform was evaluated for its usability, performance estimation tools, and deployment options. Performance was measured in terms of inference latency, memory usage, and real-time feasibility.

Results show that meaningful machine learning tasks can be executed on general-purpose low-power MCUs, with the nRF5340 offering better performance due to its more modern architecture and higher clock speed. The STM32L476, while functional, showed limited headroom for real-time inference with the tested model. Edge Impulse's performance estimations were found to be slightly optimistic for STM32 and slightly conservative for nRF5340, highlighting the importance of early on-device testing.

The findings provide practical guidance for selecting microcontrollers for Edge AI applications and underscore the importance of early-stage performance testing. The thesis also outlines future directions for exploring newer MCU families and comparing general-purpose microcontrollers with systems that include dedicated AI accelerators.

Keywords: Edge AI, IoT, Edge Impulse, nRF5340, STM32L4

Contents

1 Introduction

The Internet of Things (IoT) refers to connected embedded systems that collect, process, and exchange data, often autonomously. Traditionally, these systems rely on cloud services for data processing, which introduces latency, network load, and privacy concerns.

Edge AI addresses these challenges by enabling artificial intelligence inference directly on devices. By processing data locally on microcontrollers, Edge AI reduces latency and bandwidth usage while improving reliability and privacy.

This thesis focuses on implementing Edge AI on low-power microcontrollers, demonstrating how modern toolchains allow intelligent behavior at the device level without dedicated AI hardware.

1.1 Thesis motivation and research questions

Thesis is written for a company called Dream Devices. This thesis is assigned to research Edge AI toolchains and implementation methods. Specifically Nordic Semiconductors and ST-Microelectronics microcontrollers, as those are the most popular microcontrollers for device development in Dream Devices.

Original Dream Devices requirements for the thesis:

1. Provide information on Nordic ML Studio
2. Implement some AI application in the platform

3. Contain a demonstration on nRF MCU

The tool requested for research was originally Nordic ML Studio. Nordic ML Studio provides AI tools for their processors, and is powered by Edge Impulse. Edge Impulse is a largely used platform for embedded & IoT artificial intelligence products. During the design phase of the thesis, the subject evolved to include the Edge Impulse as the main tool, as it is the superset of Nordic ML Studio. In other words, Nordic ML studio provides Edge Impulse tools only for Nordic products, whereas including the original Edge Impulse Studio provides support for more platforms. This opened the way to include STM processor and multiple compile options of the Edge Impulse in to the research.

After reconsidering the requirements, thesis has 3 research questions:

1. How artificial intelligence models can be developed in Edge Impulse?
2. How to implement and infer on nRF5340 and STM32L4 microcontrollers?
3. Which microcontroller performs better, and how to choose MCU for known Edge AI application?

Implementing models for nRF5340 and STM32L4 is expected to provide information on the deployment options of the Edge Impulse platform. Implementing and providing a working demonstration was one the Dream Devices original requirements. During the thesis design phase, the implemented application was decided to be audio recognition. To be more precise, keyword spotting will be used. The selected application has the balance of some complexity but little enough for low-power devices. Other options included accelerometer pattern/ gesture recognition, or some sensor data anomaly detection, but these were discarded as a more complex application is expected to provide better performance metrics.

Performance results from the implemented audio recognition is expected to provide a reference point for selecting neural networks and MCUs for AI applications in

future customer projects. With the performance metrics of these two ARM-based MCUs, the complexity of suitable networks can be estimated. From this information suitability for faster or slower similar MCUs can be extrapolated. With inference latency and inference interval information, the performance and power consumption can be estimated to match customer specifications.

1.2 Thesis structure

The structure of the thesis is as follows:

- Chapter 2 provides introduction to IoT and Edge AI. Edge processing and its benefits are discussed, to provide motivation and significance for developing devices with these technologies. Also development operations related to Edge AI devices are compared to traditional IoT development operations.

- Chapter 3 presents Edge Impulse platform, the main focus in this thesis. Edge Impulse combines multiple tools and steps for developing intelligent applications to low-power devices. The supported devices also include vendors other than Nordic & STM. These are discussed shortly, because some embedded projects may require more performance than Nordic & STM MCUs can provide. Chapter also discusses implementing and optimizing steps.

- Chapter 4 describes the impulse and application development. For this thesis, a keyword spotting application is designed, optimized and implemented using the Edge Impulse platform. Designing the impulse also contains data gathering for training the model. After impulse design, it is deployed to nRF5340 running Zephyr and to STM32L4 running FreeRTOS. Respective sections describe these microcontrollers and justify their selection. C++ library and CubeMX-package deployment options are studied.

- Chapter 5 presents the performance metrics of the selected microcontrollers. Performance metrics are analyzed and differences explained.

- Chapter 6 concludes the thesis. Conclusions describe the suitability and usage of Edge Impulse, and the tested deployment options. It describes the performance metrics, and discusses how to use the work of this thesis to select suitable microcontrollers for edge AI applications.

2 Intelligent connected devices

2.1 Internet of Things

Internet of Things (IoT) refers to devices that function independently over some network. IoT applications collect, process and exchange data communicatively. IoT independence means that applications include machine-to-machine communications (M2M) and don't require user actions for normal operations. IoT devices often include different types of sensors, actuators, embedded systems or other smart applications. The used communication network may be LoRaWAN, Cellular, WiFi or any other wireless or wired solution. Compared to traditional embedded systems, IoT devices are always connected.

IoT architecture is traditionally described using a three layer model, consisting of the device layer, network layer and application layer[1]. The device layer provides the sensing, actuator or other application function in the physical IoT device location. This layer serves as a foundation for the IoT application. The work in this thesis creates solutions for device layer. The implementations in chapters 4.2 and 4.3 creates device layer prototypes.

The network layer describes the connectivity of the IoT application. Without connectivity IoT devices are useless. Even thought the IoT name includes internet, applications may work over private networks as well, like a Sigfox or some wired network. In addition, the network layer has the responsibility to provide connectivity

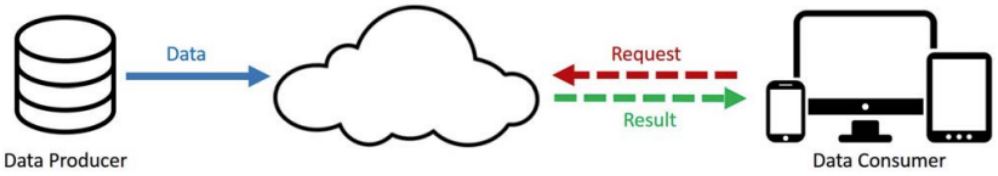

Figure 2.1: Cloud computing paradigm [2]
Copyright ©2016, IEEE

inside the application layer, if needed. This refers to the needs of accessing the application level data or results. The Figure 2.1 shows the connections needed in traditional cloud based IoT architecture. Device layer acts as the data producer, connection layer provides the necessary connections, and application layer analyses displays the data to the data consumer.

The application layer receives and analyses the IoT data in the cloud. Application layer provides user interfaces and applications that allow monitoring, automation or actuation, depending on the function of the system. Traditional sensing applications layer consists of a cloud server that analyses the data, and a client user interface to display the results and possibly control the devices.

2.2 Edge processing and edge AI

Traditional IoT only processes data in the cloud. In the traditional model device layer acts only as a data producer. If any actions are needed in the device layer, two way communications are required. This leads to increased latency, higher bandwidth consumption, and dependencies on active communication for system operation. These problems introduce the need to process data faster, without any need for communication. Edge processing refers to computations performed near the source of the data, typically on the sensing device or on a centralized node or gateway. Edge AI refers to artificial intelligence applications inferred in the device layer.

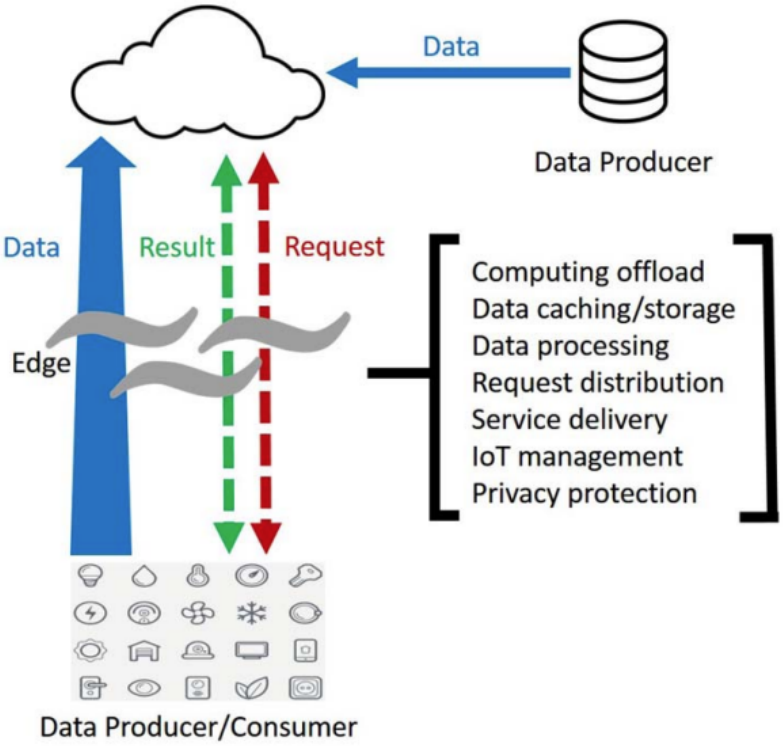

Figure 2.2: Edge computing paradigm [2]
Copyright ©2016, IEEE

Integrating edge computing into IoT expands the simplified three layer architecture. As shown in Figure 2.2, edge can be thought of as an additional layer between device and network layer[2]. The figure illustrates a traditional IoT node and an intelligent edge device, each connected to the cloud. The intelligent device is labeled as both a data producer and consumer. However, this does not imply the replacement of traditional consumers at the application level, as it is meant to describe added functionality.

The edge layer consists of various processing and analysis software operating on the device layer before the network layer is engaged. As most of the data is processed locally, the network layer transmits only selective or preprocessed data. The application/cloud layer continues long-term data storage and user interface services. It may also perform further analysis over long-term data, or comparing

analysis between multiple devices and aggregated analysis of the whole system.

Artificial intelligence has revolutionized the implementation of applications and functionalities. Traditional predefined algorithms with hard-coded instructions have been enhanced by AI, enabling intelligent features that can learn from data, manage uncertainty, and adapt to new inputs. Popular AI applications—such as pattern recognition, anomaly detection, audio recognition, and intelligent vision—further extend the capabilities IoT devices.

2.2.1 Benefits of intelligent edge

In addition to enabling new AI-based applications, traditional IoT applications can also benefit from intelligent edge features. However, since edge processing and AI-capable devices require more computational resources than conventional IoT systems, the benefits must be weighed against the increased demands. Development costs also rise as system complexity increases. Therefore, the advantages of adopting a more complex and power-intensive system should be carefully considered. The potential benefits are outlined below.

Lower latency

Edge processing and inference provides local real-time analysis, enabling devices to make adjustments or take actions in their own. This mitigates the delay in the traditional process of acquiring data, cloud communication, cloud analysis, response communication, and taking action. Local AI inference provides possibilities to have intelligent features in real-time applications with hard latency requirements.

Reduced bandwidth usage

Processing data locally can significantly reduce the amount of communications needed. Instead of continuous communication, only anomalities or aggregated pre-

processed data packages need to be sent. This prevents network congestion and increases the number of possible devices in limited connectivity networks.

Improved security and privacy

Local data processing within the device eliminates the need to transmit all information over a network, reducing data exposure. As sensitive data remains on the device, the benefits and therefore risks of cyberattacks reduces. Additionally powerful intelligent edge devices can implement enhanced security solutions to further protect the communications.

Reliability

Intelligent edge ensures that the system can continue to operate even when the network layer connections fail. Since edge devices don't rely on continuous connection, they remain operational. For device developers the operation during a disruption should be explicitly defined, depending on the length of the disruption and the nature of the application. Disruptions may be caused by cyberattacks or equipment malfunctions, so some applications may need to differentiate their future actions respectively. For example, communication with the cloud server may be deliberately cut off if the intelligent device detects message tampering by a malicious party.

Energy efficiency

As implied by Nordic Semiconductor, the world does not have enough power to keep increasing the cloud servers capacity at today's rate[3]. Depending on the network, on-device processing is also less energy consuming for the device, thanks to the smaller communication needs.

On general purpose MCUs without AI accelerators, intelligent edge features translates to extra CPU cycles. For energy optimizations, this raises the compari-

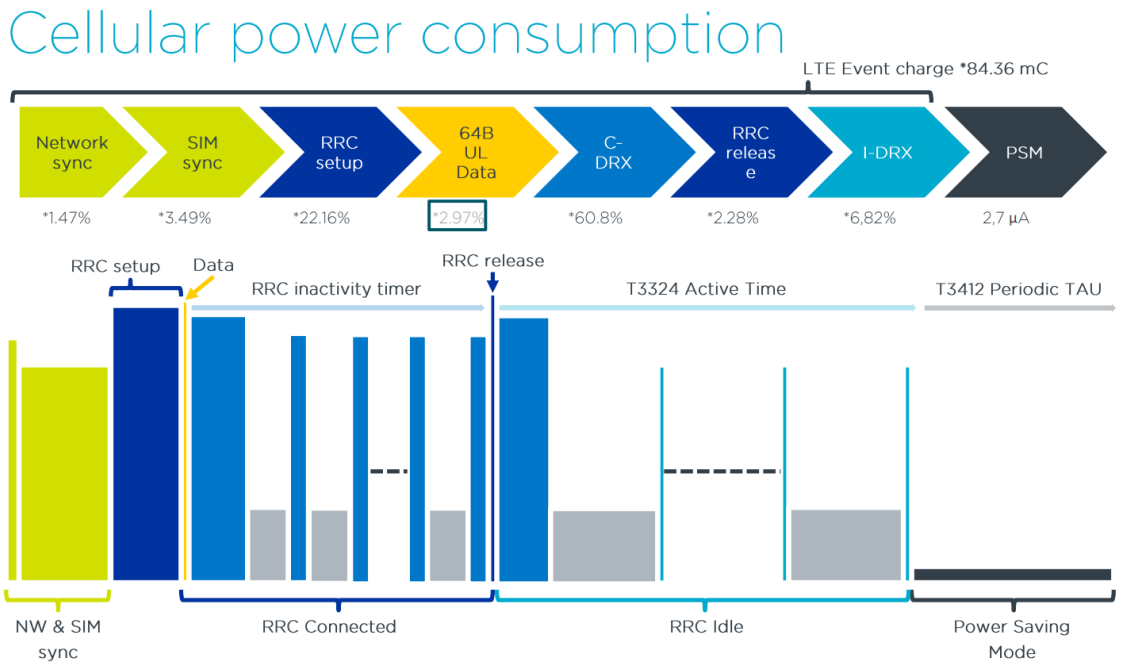

Figure 2.3: Cellular power consumption, ©*Nordic Semiconductor*[3]

son between local processing and networking consumption. nRF9160, an ARM-M33 64Mhz MCU, consumes about 2.2mA when executing from flash. The energy cost of sending 64 bytes of data over LTE is about 84mC.[3]

$$t_{CPU}\,[s] = \frac{Energy_{LTE}\,[mC] \times 1\,[s]}{I_{CPU[mA]}}, t_{CPU}\,[s] = \frac{84mC \times 1s}{2,2mA} \simeq 38s$$

The equation above divides the energy needed for an communication event with the energy needed for normal execution. With the equation we can estimate that the CPU could run for 38 seconds with the energy cost of a single 64 bytes communication event.

As shown in the figure 2.3, cellular transmissions consume power in many stages, starting all the way from synchronizing to the network. In the case of sending 64B of data, the actual transmission only took about 3% of the energy. According to Nordic Semiconductor, sending 1024B would consume 96mC, and 2048B would consume 108mC.[3] This highlights the massive energy savings available by aggregating data

at the device layer and enhancing edge intelligence and independence to enlarge communications intervals.

For most applications, the processing/inferring time is between 10 - 500 milliseconds. For hard real-time applications the acceptable scale is from microseconds to some milliseconds. Compared to the equivalent of 38 seconds of energy used per communication, the power required for processing locally is significantly lower. I have observed LTE modems drawing up to 300mA of peak current during operation, with such high currents posing additional strain on battery systems. Wireless communication is always energy consuming and prolonged operations generate heat within the system.

2.3 Development and machine learning operations

Development Operations, or DevOps, describes a collaborative approach that integrates software development with IT operations[4]. This methodology emphasizes automation and the use of continuous integration and continuous delivery (CI/CD) pipelines to enhance the efficiency and reliability of software deployment.

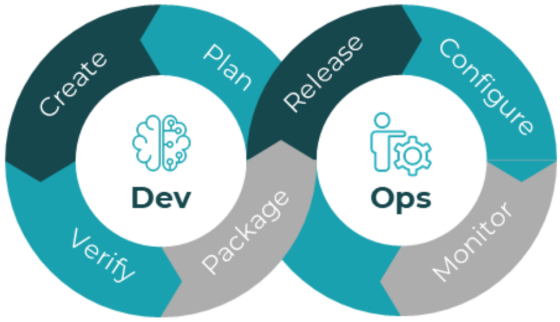

Figure 2.4: Development & Operations cycle [4]

As illustrated in Figure 2.4, the development team plans the project or its next version, creates the software, and packages the version for an official release. The operations team then manages this release by deploying updates to users, monitoring

system performance, and reporting issues. Development efforts are continuously delivered to end users, with constant feedback flowing through the operations team[5].

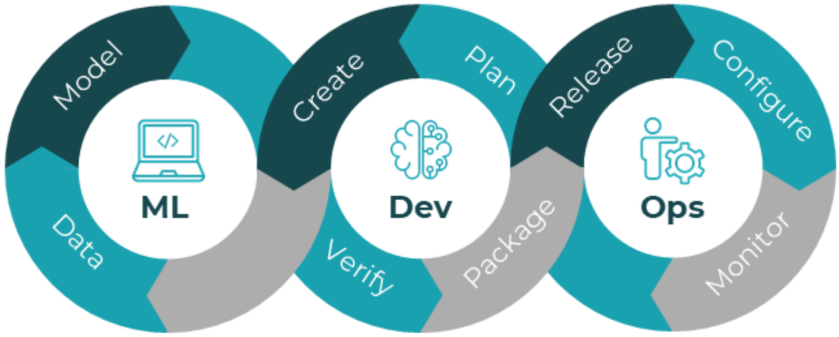

Figure 2.5: ML & DevOps cycle [4]

MLOps, short for Machine Learning Operations, extends these principles by adding the cycle of ML in the development, as shown in Figure 2.5. Machine learning operations cover tasks involved in the creation, training, and maintenance of machine learning models. As shown in the figure, once the revision of the model is completed, it is handed to the development team for integration into the software. An ML team may consist of professionals specializing in data science, artificial intelligence, or software engineering. The team works closely with the main development team, or in small projects it may operate as a part of it.

The work of ML team consists of several phases. In the context of this thesis, the first phase is to research machine learning models and select a suitable model to begin with. The ML team gathers the necessary training and testing data, often using available databases and supplementing them with data collected in-house. The team trains and tests the model, releasing it to the development team for integration into the main software. Data collection continues alongside model monitoring and maintenance.

Model development also continues in the maintenance phase. Maintaining high-performing machine learning models often requires periodic retraining to address

challenges such as overfitting, data shifts, and changes in data distribution over time [6]. These may be caused by sensor degradation or variations in environmental conditions. If the application environment data diverges from the training dataset, retraining becomes necessary to ensure the model remains aligned with current data patterns. To address these issues, the model needs to be retrained. Retraining practices ensure that models evolve, learn, and adapt over time, maintaining their efficacy in changing environments.

3 Edge Impulse platform

Edge Impulse is a machine learning development platform for Edge applications, developed and maintained by Edgeimpulse Inc. It provides AI model developing, training and deploying tools for resource-constrained MCU based devices. Edge Impulse is suitable for both AI-powered IoT applications and traditional standalone embedded AI products. By providing the tools for different steps in the same environment, the platform simplifies and streamlines the steps in AI development.

The platform is designed for both beginners and professionals. This is achieved by offering both automated tools and code-level access to every aspect in the development process. Enabling developers to build functional AI systems without requiring extensive machine learning expertise, but being able to interfere in each step by modifying the program code, is a bit similar to traditional code generating development environments like the STMCubeMX. These generators can establish the foundation for the project, which is then further developed, with any automatic code generation refinement needs addressed manually. The following sections provide a deeper exploration of Edge Impulse's MLOps tools, supported hardware, model implementation, training and optimizing techniques.

3.1 Edge Impulse development tools

To complete all MLOps tasks within the platform, Edge Impulse provides several tools, categorized according to the development phases. The arrangement of the

sections also reflects the order in which these tools are used in the Edge Impulse workflow.

3.1.1 Data collection & processing tools

Effective model training requires a large, labeled dataset. Edge Impulse provides data acquisition tools to collect, centralize, and validate datasets. Data can be collected using smartphones or pre-built firmware for supported platforms. The platform also provides free datasets of various types. Depending on the data type and sensor quality, the choice of the data acquisition platform should be carefully considered. For example, gathering vibration or gesture data from a high-quality MEMS accelerometer is far superior to using a smartphone, as its enclosed accelerometer and the phone's mass can hinder movement.

Edge Impulse's data uploader supports over 10 file types for raw data, audio, and images. After data collection, datasets can be validated using the Data Explorer tool. Labels can be created through the Labeling Queue or AI Labeling. Additionally, Edge Impulse allows metadata to be included in the dataset for storing user information that the model will not use, such as dates or descriptions.

Datasets can be further advanced using Edge Impulses synthetic data tools[7]. Based on the provided data, the platform can synthesize additional training data for images and sounds. The platform can create synthetic images using GPT-4 (DALL-E), generate human speech with Whisper, and sounds with Eleven Labs sound effects models.

After acquiring data, the dataset can be analysed by the Data Explorer tool, to detect outliers in the trainig data.

3.1.2 Impulse design tools

An impulse is a concept in Edge Impulse that represents the complete AI pipeline that transforms data from sensors to some result through various phases. Each impulse includes at least three essential blocks, with the possibility of multiple components within each block.

Input Block provides the data and preprocessing based on the type of the sensor data. Traditional sensor data, such as accelerometer readings and temperature measurements, represent time-series data. These signals require a time-series signal processing input block. Edge Impulse offers input blocks for various data types beyond simple time-series signals, like audio and video. As the complexity and volume of input data increase, more advanced preprocessing is required, especially in low-power systems. Audio and image input blocks introduce more complex preprocessing techniques. For example, image data is often transformed to grayscale or scaled down on resolution, to reduce computational demands in later stages. Similarly, audio signals are transformed into spectrograms, which convert the raw time-domain signal into a frequency-domain representation.

Processing Block transforms the preprocessed data into a structured format suitable for machine learning models through feature extraction. In cases of simple datatypes, a sophisticated input block can process the data enough to omit this block. This stage includes operations such as spectral analysis of spectograms, feature detection in images, or other domain-specific transformations that extract meaningful patterns or characteristics from the data. Processing block(s) allows chaining multiple processes. Users can modify the parameters of these processing blocks to fit their application-specific data. In addition to built-in processing blocks, Edge Impulse also supports custom processing blocks, enabling further flexibility in feature extraction and data transformation.

Learning block applies the machine learning model to the processed data.

Within the Learning Block, users can configure various model parameters, including neural network architecture, activation functions, and optimization algorithms. Adjusting these parameters allows for fine-tuning the model to achieve better accuracy and generalization. Edge Impulse supports deploying an eternal pretrained model with the "Bring your own model" tool, in TensorFlow SavedModel (`savedmodel .zip`), ONNX model (`.onnx`) or LiteRT (Tensorflow Lite, `.tflite`) model formats.

3.1.3 EON Tuner

The Edge Optimized Neural (EON) Tuner is a tool for both selecting and optimizing machine learning models[8]. Selection and optimizing work can be somewhat repetitive, as it requires consecutive testing of different settings. EON automates the evaluation of different neural network architectures based on performance metrics and hardware constraints. By applying automated hyperparameter tuning, the EON Tuner explores trade-offs between model accuracy, inference latency, and memory footprint. As it automates model selection and optimization, it functions as an AutoML tool, and reflects the increase on automated ML development techniques, i.e. using AI to develop AI.

3.1.4 Deployment tools

For model deployment, there are currently 6 deployment options[9]:

1. **Customizable library.** Library options currently include 11 different options, including generic C++, Arduino, WebAssembly and TensorRT libraries. Deploying as a generic library provides the developer full access to the source code. This option provides the most customizable deployment option, but is time-consuming in testing setups.

2. **Prebuilt firmware.** Edge Impulse provides a ready-made flashable binary

for fully supported development kits. The prebuilt firmware allows the impulse to be executed directly on the supported platform. The impulse can be run using the Edge Impulse local client on a PC connected to the development board, with the Impulse Runner displaying the results of the model running on the board. This option enables fast performance testing but is limited to supported development kits with provided firmware and compatible data types in terms of supported sensors.

3. **Linux client.** The Linux client enables deployment of Edge Impulse models on Linux-based systems, such as Raspberry Pi and NVIDIA Jetson[10]. By using this client, developers can execute models locally, facilitating real-time inference. This option supports both testing and deployment directly on Linux platforms, making it a suitable choice for edge devices running Linux operating systems.

4. **Docker container.** Edge Impulse allows deployment in Docker containers, offering a portable solution for running models in isolated environments. This creates a HTTP accessible inference server. This method is ideal for testing and deploying models across various systems without needing to manage dependencies directly on the host machine. Docker containers ensure that models can be consistently executed across different platforms and are particularly useful for scaling inference across multiple devices. While Linux targets can use hardware accelerators, other optimizations, such as local GPU or NPU detection, are unavailable. This option is most suitable for deployment on gateways or cloud targets with Linux, where the isolation is necessary.

5. **Browser.** With this option the impulse can be run directly on computer or smartphone. As the Edge Impulse Library options include WebAssembly, it is also possible to deploy the model on the browser. This creates a site under

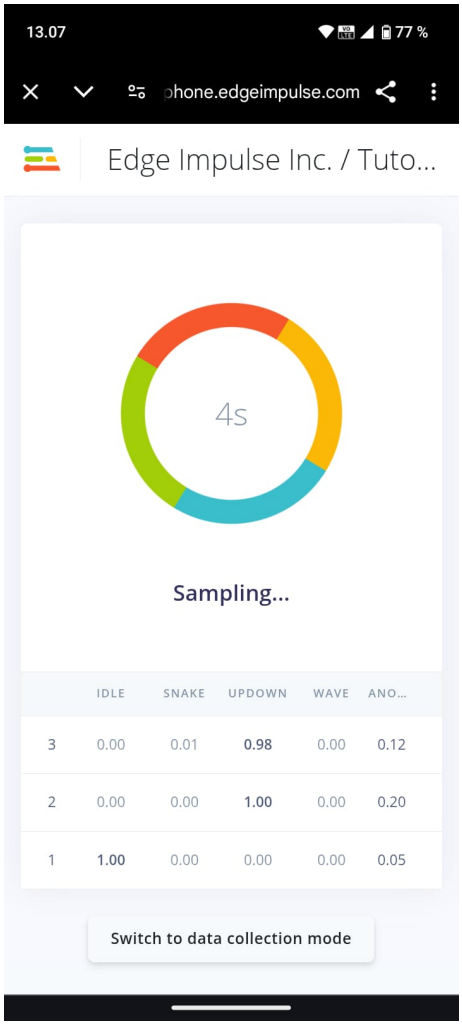

Figure 3.1: Edge Phone inference

`https://smartphone.edgeimpulse.com`, where the model can be tested. The same site can be used for data-acquisition with phone. Figure 3.1 is from a gesture recognition tutorial model running on an Android phone. This option is suitable for testing.

6. **Custom deployment block.** Edge Impulse's custom deployment blocks allow developers to create tailored deployment solutions for specific hardware or application requirements. It is suitable for scenarios that demand specific processing capabilities or when built-in options do not meet all needs.

3.1.5 Retraining

Edge Impulse provides retraining tools that retain the original input and processing blocks, while also allowing modifications to the model architecture during training if needed. Newly collected data can be added to the database through the data acquisition tools. After retraining, the updated model can be deployed, and its performance monitored. Additionally, when over-the-air (OTA) updates are available, retrained models can be seamlessly deployed, ensuring continuous adaptation to evolving conditions.

3.2 Supported hardware platforms

Edge Impulse offers compatibility with many types of hardware platforms, enabling developers to deploy machine learning models across various types of devices.[11] The largest support is for traditional MCUs, with the coverage of 15 vendors including companies like Arduino, Nordic Semiconductor, Raspberry Pi and ST-Microelectronics. As discussed in the deployment options, traditional C++ libraries and Linux options provide support for custom platforms.

Besides MCUs, many custom boards with MCU+AI-Accelerators are supported. These platforms can run larger and deeper AI models, and inhabit larger memories. Larger performance comes with the trade off of larger energy consumption and component cost. On the scale from consumption to power, there are multiple options to select from. For example, STM32N6 platform contains ST Neural-ART Accelerator, a custom neural processing unit (NPU) running at 1GHz, besides its 800MHz Cortex-M55 cores. In Arduino Nila voice, Nordic Semiconductors nRF52832, an 64MHz Cortex-M4 MCU, is placed alongside Syntiants NDP120. The ST accelerator has the capability of 600 GOPS, as the Syntiant has 6.4 GOPS (giga-operations per second). The two mentioned platforms each provide larger capabilities than

traditional MCUs, and provide an example how largely the performance inside this product category can differ. A general rule in low-power devices is to choose the smallest possible suitable platform, to minimize energy consumption and component cost.

General purpose CPU's, like Raspberry Pi 4 (Cortex-A) or any PC x86_64 CPU, can be used with the Linux deployment option. Traditional CPU's offer more computational resources than MCU's. However, using a x86 CPU is usually not suitable for edge devices, as other options outperform them. Compared to MCU's with AI-accelerators, x86 CPU's lose in inferring performance, are costly and consume more energy. PC's are suitable for cloud or gateway applications, where energy consumption or other optimizations makes little difference. Small CPU platforms can be a justified choice when Linux support is essential for the implementation and model inference does not require AI-accelerators.

General purpose CPU's capabilities can be enhanced with AI-accelerators. Edge Impulse provides support for aftermarket accelerators that connect to an existing Linux-based PC platforms via PCI-E or M.2 interfaces, such as the BrainChip AKD1000[12]. There are also multiple single-board systems, that host an CPU and an AI-accelerator. Typically these contain Cortex A-processors with limited operating system support, restricted to specific Linux distributions. These platforms aim to offer performance to more demanding tasks than the MCU's with accelerators can handle, while their CPU's consume more power. The line between MCU+AI-accelerators and CPU+AI-accelerators is a bit blurry. Generally, the latter category includes Cortex-A series cores, whereas MCUs typically feature Cortex-M series cores. In the hardware supported by Edge Impulse, no single-board system with an x86 core and an AI-accelerator was available.

GPU-based platforms provide another alternative for machine learning inference, offering high parallel processing capabilities for deep learning tasks. Edge Impulse

supports several GPU-based solutions, consisting of NVIDIA's Jetson and other variants.[13] Jetson modules integrate Cortex A-series cores with CUDA GPU's to accelerate AI workloads. Compared to MCU+AI accelerators, GPU-based solutions provide greater computational power but at the cost of higher energy consumption and system complexity. These platforms are widely used in applications such as robotics, computer vision, and industrial automation, where real-time inference and high processing power are required. Despite their importance in another fields, their size and power draw make them not suitable for low-power Edge IoT devices.

4 Implementation

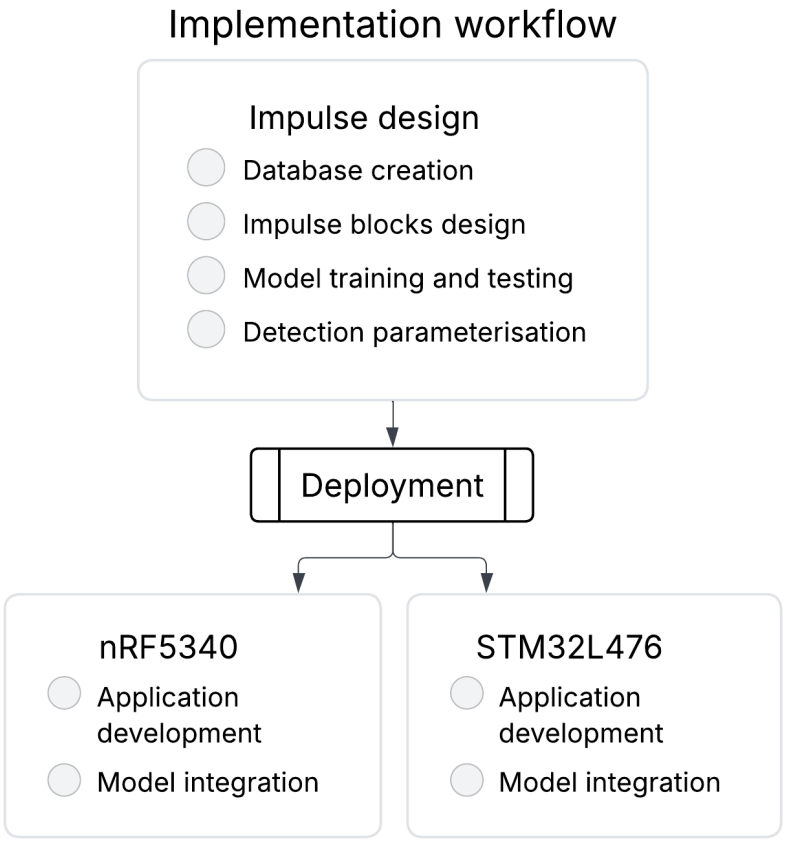

Figure 4.1: Implementation workflow

This chapter describes the whole workflow needed to create and run a keyword recognition model. The model is trained to distinguish "Dream Devices" keyword, and then deployed to both nRF Zephyr environment and STM FreeRTOS environment. To enable later performance comparisons, the same model will be used for both devices. As shown in Figure 4.1, the work consists of many phases, starting

with the Impulse design.

4.1 Impulse design

The impulse was designed to prioritize classification accuracy and robustness over model size or latency. Although the final model was later deployed to both nRF5340 and STM32L476, resource constraints—especially on the STM32—were not a major consideration during initial development. Both of these devices have sufficiently large memories, and processing power for this application. Instead, the focus was on selecting signal processing and model architecture components that best supported reliable keyword recognition across varying voices, tones, and background conditions.

The design process involved dataset creation, selecting MFCC-based audio preprocessing, followed by training a convolutional neural network classifier. These choices were guided by Edge Impulse's built-in tools for feature visualization and performance tuning, described in detail in the following subsections.

4.1.1 Dataset creation

As discussed in chapter 3.1, machine learning development begins with acquiring a large, labeled dataset. Initially, data collection was intended to be carried out using the nRF5340 with Edge Impulse provided firmware. With X-NUCLEO-IKS02A1 sensor board, which is later utilized in inference purposes, this would have accomplished training and inferencing data coming from the same sensor. Although the nRF5340-DK is supposedly fully supported by Edge Impulse, with an Edge firmware available on GitHub[14], the firmware currently fails to build.

Therefore keyword data is collected using a smartphone. This is suboptimal, as discussed in chapter 3.1.1, it would be ideal to use the same sensor that will be employed in the final inference device. However, it is anticipated that the microphones

in both devices are sufficiently accurate for this purpose.

Dataset was collected by having multiple colleagues say the keyword in different tones and speeds. Besides keyword, the dataset also requires background noises, like doors, TV playing, different words spoken. For this purpose, free dataset from Edge Impulse was imported, providing 577 noises, 572 unknown words, and 17 "hello world" keywords.[15] After having several colleagues reading the "Dream Devices" keyword in phone booth, there are 50 "Dream Devices" samples. The "helloworld" keyword samples were excluded.

After gathering data, Whisper tool inside the Edge Studio was used to generate 30 more samples, to test synthetic tools presented in chapter 3.1.1. Using Whisper in Edge Impulse requires an OpenAI API-key. For this key, a paid plan or another payment quota is required. After generating first 30 samples, the OpenAI balance was reduced by one cent. It shows that this type of speech generation is inexpensive. However, this small test proves the simplicity of adding synthetic data in Edge Impulse to enlarge training datasets and acquire differing voices. The generated samples differ in speed, intonation and accent. As it proved to be simple, cheap and fast, 120 more samples were generated to reach a total of 200. After this, the balance was again reduced by one cent. By default Edge Impulse divided new samples to 80% training and 20% testing.

After acquiring data, it is inspected in Data Explorer. Figure 4.2 on the next page shows the data explorer view of the Edge Impulse. Blue dots are the keyword samples, green samples are other words from Edge Impulse provided dataset, and orange samples are environmental noises like train or car passing. One blue sample is highlighted, locating right from the hovering blue label text, showing sample DreamDevices.5oc5mqsp alongside noise samples.

As can be seen from the figure, the dataset is not completely segregated, as the mentioned and some other blue samples are close to noise. The time series data of

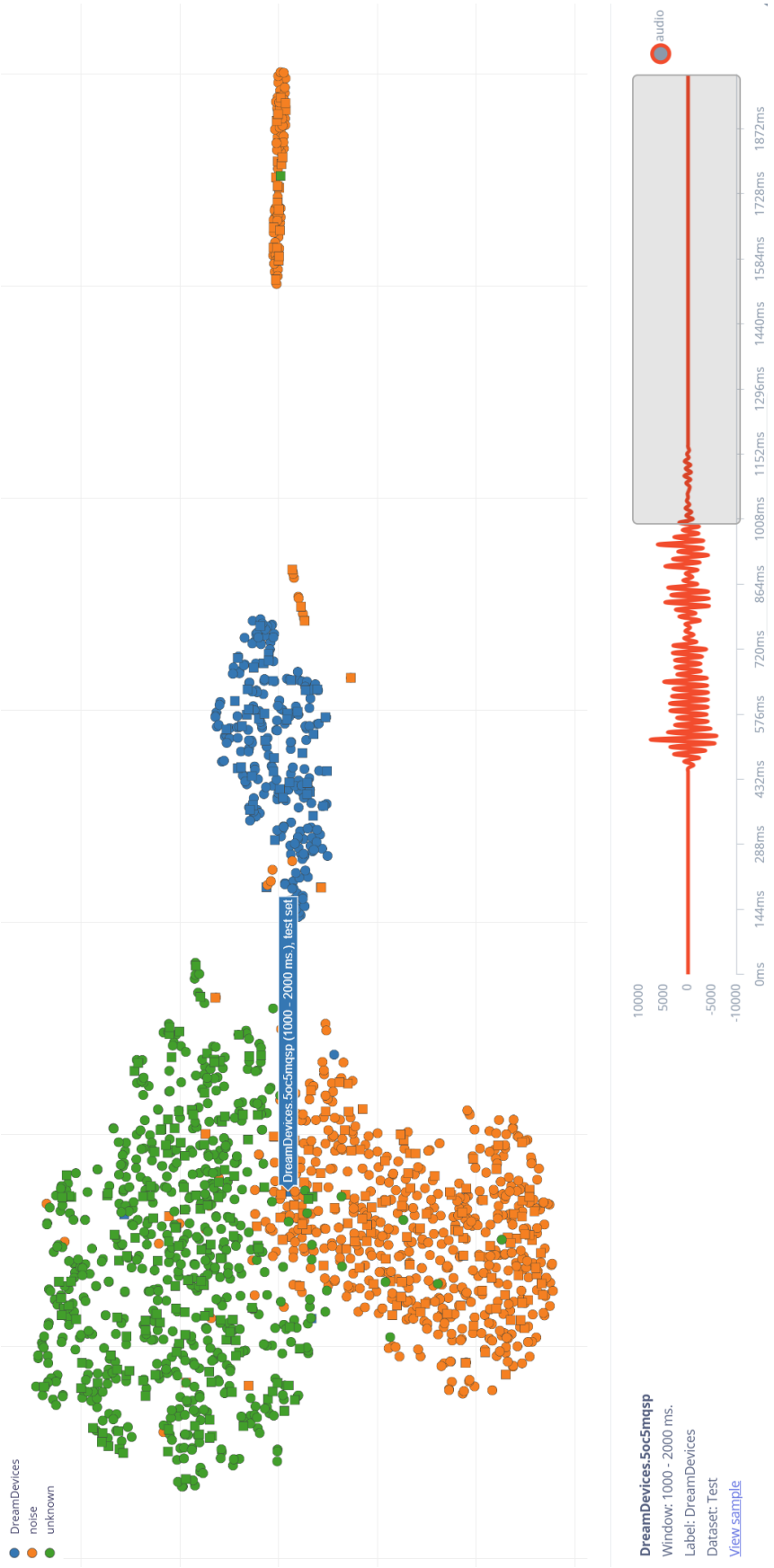

Figure 4.2: Data explorer outlier example

the highlighted sample is shown in the bottom of the figure, with its highlighted right side being the part where the feature was extracted from. Clearly, this outlier came from keyword recording being too long, including unnecessary noise that was extracted. Outliers came mostly from noise adjacent to the keyword if the sample was too long, or if the keyword did not fit the window. Samples were trimmed, and some bad quality ones deleted. Noise samples being close to blue samples should not matter much, as the machine learning model is expected to distinquish these.

4.1.2 Impulse blocks design

Input and processing blocks

As discussed in chapter 3.1.2, the Impulse begins with input and preprocessing block(s). For audio data, this typically involves transforming raw time-series audio signals into a frequency representation suitable for machine learning models. A common approach is to compute a spectrogram, which shows how the frequency content of the signal evolves over time. To better align with human hearing sensitivity, the Mel scale is often applied, resulting in Mel-scaled spectrograms [16].

For the work of this thesis, Mel-Frequency Cepstral Coefficients (MFCC) preprocessing is selected due to its efficiency on resource-constrained devices. MFCC provides a compact representation by applying a Mel filterbank, taking the logarithm of the energies, and performing a Discrete Cosine Transform (DCT). Mel filterbank emphasizes lower frequency components of speech (vowels, voiced sounds) while de-emphasizing finer spectral details [17]. The DCT decorrelates the features and reduces dimensionality, which reduces complexity when deploying models on low-power hardware.

Edge Impulse supports several alternative feature extraction methods based on similar principles. Mel Filterbank Energies (MFE) omit the DCT step and retain the log-Mel filterbank energies directly. This preserves more of the structure of the

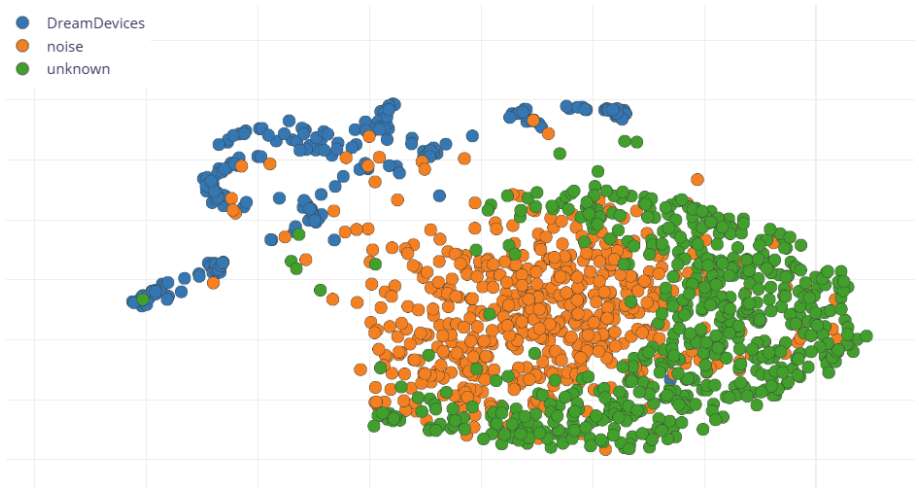

Figure 4.3: Edge Impulse Feature explorer

frequency content and is particularly effective when using deep learning models such as Convolutional Neural Networks (CNNs) [18]. Although an Impulse built around MFE demands more computational resources, it may offer improved accuracy where such resources are available.

Alternatively, raw or log-transformed spectrograms can be used directly. This retains the most information from the signal but typically requires larger models and higher computational capacity. Such representations are especially useful for recognizing non-speech audio, including environmental sounds or animal vocalizations [19].

After the MFCC preprocessing block is configured, the extracted spectral features can be explored using the Edge Impulse Feature Explorer, shown in Figure 4.3. Before training a neural network, it is important to verify that the features are reasonably well separated. While certain samples may display similar noise characteristics — and thus appear in close proximity within the "Dream Devices" sample space in the figure — this is not expected to pose a significant problem. The classifier neural network is expected to learn to distinguish between these cases. Outlier samples can be listened to in this view and they mostly contain noise similar to additive white Gaussian noise.

Learning block

Once the feature extraction results have been verified, the next step is to train a neural network classifier to distinguish the desired classes. The classifier operates on the features produced by the MFCC preprocessing block and learns to map them to the corresponding class labels.

The default classifier neural network architecture provided by Edge Impulse is typically well-suited for audio classification tasks. This neural network architecture consists of fully connected layers, followed by non-linear activation functions and a final output layer corresponding to the number of target classes. Model parameters such as the learning rate, number of training epochs, and network size can be adjusted to optimize classification performance, depending on the complexity of the task and the characteristics of the dataset.

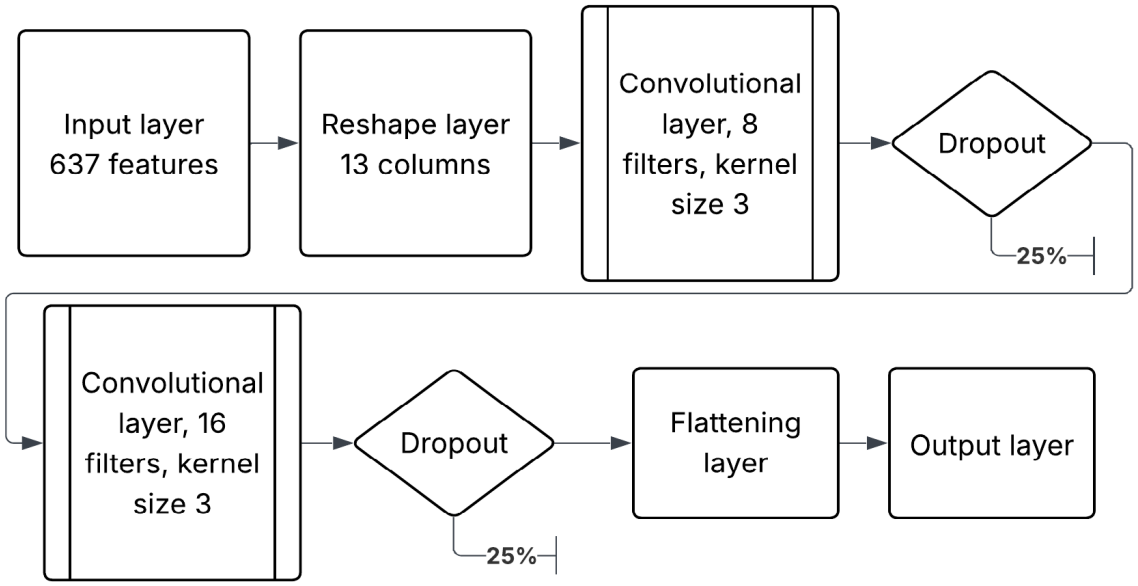

Figure 4.4: Neural network architecture

As seen in the Figure 4.4 classifier model has an input layer for 637 features, which are reshaped into a two-dimensional representation with 13 columns to facilitate

convolutional processing. The reshaped input is passed through a one-dimensional convolutional block consisting of 8 filters and a kernel size of 3. This is followed by a pooling operation and a dropout layer with a rate of 0.25, which serves to reduce overfitting by randomly deactivating a subset of neurons during training.

The second convolutional block increases the model's capacity by applying 16 filters, again with a kernel size of 3, followed by another pooling operation and dropout layer with the same dropout rate. These stacked convolutional blocks enable the model to progressively extract higher-level temporal features relevant to keyword classification.

After the convolutional processing, the output is flattened into a one-dimensional vector, which is fed into the final dense output layer comprising three units corresponding to the target keyword classes. This architecture is optimized for efficient and effective classification of short-duration audio features in a keyword recognition context.

4.1.3 Model training and testing

The recommended settings from Edge Impulse set epochs, the number of training cycles, to 100, with batch size of 32. During the training phase, the network parameters are iteratively updated to minimize a loss function that measures the difference between the predicted class probabilities and the true class labels. For multi-class classification tasks, the commonly used loss function is categorical cross-entropy. The optimization process aims to generalize beyond the training data to handle variations in the input features, including samples affected by background noise or signal distortions.

Testing the model

Model performance is then evaluated using a separate validation set, providing insight into the network's ability to classify previously unseen data. As stated in section 3.1.1, by default the database is divided to 80% training & 20% testing. Standard evaluation metrics include overall classification accuracy, class-wise precision and recall, and analysis of the confusion matrix to identify potential sources of misclassification.

Confusion matrix

	DREAMDEVICES	NOISE	UNKNOWN	UNCERTAIN
DREAMDEVICES	**98%**	0%	0%	2%
NOISE	0%	**93.8%**	2.8%	3.5%
UNKNOWN	0%	5.6%	**90.3%**	4.2%
F1 SCORE	0.99	0.94	0.94	

Figure 4.5: Confusion matrix

The model was first tested in unoptimized float32 mode. As shown in the confusion matrix in Figure 4.5, the model performed outstandingly, classifying 98% of the keyword test samples correctly. Uncertainty and disclassification between noise and unknown words is irrelevant. The F1 score is a metric that balances how many predicted positives were correct (precision) and how many actual positives were identified (recall) to measure a model's accuracy in classifying positive instances. With a score of 0.99, the model demonstrates excellent overall performance. Results mean that one "Dream Devices" sample was incorrectly classified. The disclassified sample was a generated 1 second sample, that had the keyword said with fast tempo inside the first 0,6 seconds. The sample was then trimmed to proper length. After this, the test was conducted again using both float32 and optimized uint8 models.

Both models performed similarly, failing the same now cropped sample. It seems the problem was not the 0,4s of noise, but fast pace that makes it hard to distinguish. This autogenerated sample is indeed faster than any collected naturally spoken sample.

4.1.4 Detection parameters

Edge Impulse provides the Performance Calibration tool, which assists in optimizing the interpretation of model outputs. The tool uses the dataset to synthesize 10 minutes of continuous test audio by layering samples and combining them with artificial noise. After running the test, the calibration tool recommends a set of parameters to be applied in a post-processing block following the impulse. This block utilizes three key parameters: window duration, detection threshold, and suppression period. The window duration defines a time span over which multiple inferences are aggregated to calculate an average certainty. The detection threshold, a value between 0 and 1, specifies the confidence level required to classify a detection as positive. The suppression period prevents additional positive detections for a defined time following an initial detection, thereby reducing repeated detections of the same keyword.

The added noise significantly challenged the model's performance. During the 10-minute evaluation period, 37% of positive instances were not detected, and there were no false detections. The Performance Calibration tool recommended configuring a post-processing block with the window size of 18ms, effectively relying on a single positive inference, and lowering the detection threshold to 0.43. Recommended suppression period was 967ms. With these settings, the false rejection rate was lowered to 29.6%, and false detection rate increased to 0.5%. These suggestions improve the performance, as 0.5% false positive rate is an acceptable trade off.

Final detection parameters are an application level design choice, iterated during development on the final hardware, and possibly updated in the maintenance cycle.

The Performance Calibration tool gives insight on the direction of the adjustments, and helps to prune inefficient models before deployment.

4.2 nRF5340

The Nordic Semiconductor nRF5340 is a dual-core microcontroller, featuring separate application and network processors.[20] As its design suggests, the nRF5340 is optimized for connectivity. With an application processor clock speed of 128 MHz, it is also the fastest Cortex-M33 based MCU from Nordic Semiconductor.

Both cores are Cortex M33 ARMv8-M architecture RISC cores, with the network core operating at a lower clock speed of 64 MHz. The higher clock speed of the application core comes at a slight efficiency trade-off, with the application core achieving 66 CoreMark points per mA, while the network core delivers 101 CoreMark points per mA.[21] CoreMark is a benchmark designed to measure the performance of microcontrollers by evaluating core CPU operations such as integer processing, control, and memory access. For better efficiency, the application cores clockspeed can be slowed when the highest performance is not required. The application core has 1MB of flash and 512KB of RAM, whereas the network core has 256KB of flash and 64KB of RAM.

The nRF5340 was selected for this thesis over other nRF MCUs due to its recent use in several customer projects and high processing power. These projects chose the MCU primarily for its proficiency in Bluetooth Low Energy (BLE) connectivity, thanks to the separate network core. The work of this thesis will test its application cores computing power against neural network model in keyword recognition, providing real world Edge AI performance metrics.

As the nRF5340-DK lacks any sensors, an STM sensor board, namely X-NUCLEO-IKS02A1 shield will be used. The shield has pressure, humidity and temperature sensors, an IMU with a magnetometer, and most importantly, a digital omni-directional microphone with high sensitivity and a wide frequency range. Microphone is located on the right side of the sensor board in Figure 4.6, it is the chip with the golden/yellow hole.

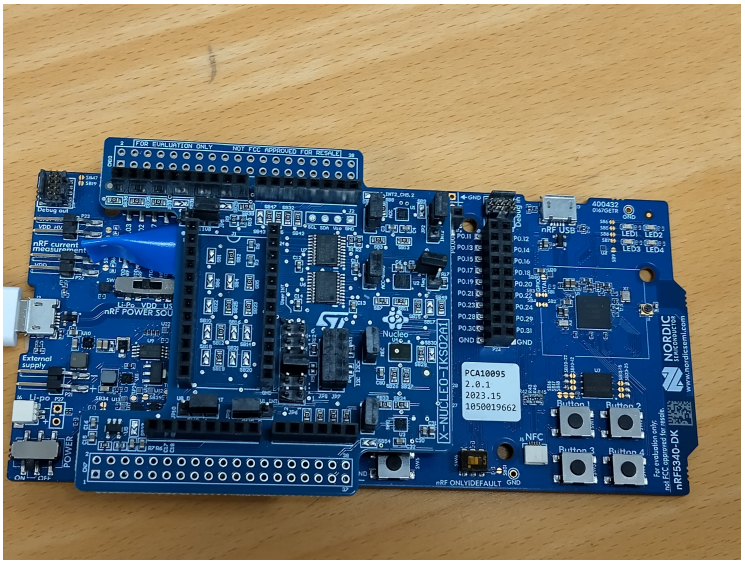

Figure 4.6: nRF5340-DK with NUCLEO-IKS02A1

4.2.1 Non-continuous prototype

The first version of the software was developed to achieve a working implementation of the microphone driver and Edge Impulse-based inferencing. The aim was to validate the end-to-end functionality of real-time audio capture and machine learning inference on a low-power embedded system. This version serves as a foundation for future improvements and performance optimization. It also provides a sample on running the inference in non-continuous mode, meaning that the inference is run once on acquired data.

Software architecture

The software consists of three key components, all implemented in `main.c`, that is shown in Appendix A: audio capture, buffering, and machine learning inference calls. Audio is recorded using a digital PDM (Pulse-Density Modulation) microphone configured to sample at 16 kHz. Every 250 milliseconds, a chunk of audio data is stored in a circular buffer. Four consecutive chunks are later combined into a 1-second audio window, which is passed to the machine learning model. A new

inference window is prepared every 250 milliseconds, allowing overlapping input for better detection responsiveness.

To enable parallel processing, the software uses two separate Zephyr threads: one for handling microphone input and another for running inference. Synchronization is managed using a semaphore to signal when sufficient data has been collected. The audio thread initializes the PDM interface and writes incoming audio samples into buffers. The inference thread is then triggered to extract features using MFCC (Mel-frequency cepstral coefficients) and run classification using Edge Impulse's `run_classifier()` function. This process is repeated every 250 milliseconds.

The classifier outputs probability scores for each label, which are printed to the serial console. In practice, detection confidence typically ranged between 30% and 50%. This result reflects both the limitations of the initial buffering method and the blocking nature of inference execution, which may occasionally lead to dropped samples between 250 millisecond chunks.

4.2.2 Continuous inference

The first software prototype performs inference using a block-based method. In this approach, MFCC feature extraction and classification are run on a newly assembled 1-second audio window every 250 milliseconds. While this allows for overlapping inference windows, it also means that each 1-second window is processed independently, and all features must be recalculated from scratch for each new cycle. This method is relatively simple to implement but introduces unnecessary computational overhead and latency, especially on low-power embedded systems.

Edge Impulse's continuous inference framework addresses these limitations by using a more efficient internal buffering system and overlapping feature windows. Instead of recalculating MFCC features from the entire audio buffer each time, the continuous inference engine maintains a rolling window of audio spectograms and

only computes features for the newly added portion. The result is a smoother and more responsive detection process, with reduced processor usage and lower risk of missing events that occur near the boundary between inference windows.

In continuous inference mode, classification happens at smaller intervals, while still using a full 1-second window of features. As the processing is faster, the intervals may be shorter. This increases the effective sampling resolution of detections and provides more accurate and timely responses to transient audio events such as keywords. Furthermore, the framework includes internal logic to handle buffer management and overlap scheduling, reducing the need for manual implementation of sliding window logic.

Continuous version adopts this continuous inferencing method to improve efficiency, reduce sample loss, and achieve more stable keyword detection performance. This transition also simplifies the codebase by offloading buffer handling and timing logic to the Edge Impulse SDK. The first software version prototype serves as a working example on non-continuous inference, useful for other applications.

Implementing continuous inference

Continuous version is shown in Appendix B. The software reuses the same microphone driver and PDM configuration from the earlier prototype. Incoming audio is captured into a double buffer instead of ring buffer, using DMA-backed PDM sampling. The audio thread continuously fills either buffer with 16-bit PCM samples at 16 kHz, while the other is used by the classifier. Buffers are switched every interval. Instead of manually assembling 1-second windows, the system now pushes each new chunk into the Edge Impulse SDK's internal buffers. This makes the continuous version code size smaller and simpler.

When enough new audio has accumulated, the SDK triggers feature extraction and inference using `run_classifier_continuous()`. This non-blocking interface

allows the application to maintain real-time performance, as classification happens in small steps, reducing peak CPU load. The inference thread checks buffer readiness and dispatches inference in regular intervals without halting the audio pipeline.

The updated implementation simplifies buffer management logic and avoids sample loss caused by inference delays. Overlap handling, sliding window timing, and classification output filtering are now managed by the SDK. In practice, this enables more stable detection rates and significantly reduces missed keywords compared to the original version. With the continuous version, classifier outputs are 1 or 0, regarding the detection parameters set in the Performance Calibration tool in chapter 4.1.4. These parameters control the post-processing behavior of classification output, including thresholds, averaging, and suppression time. Proper configuration helps reduce spurious detections and ensures that short, valid events are not missed. Unlike the earlier prototype, which directly printed raw probabilities, the continuous version applies calibrated decision logic based on these thresholds.

Continuous version software uses the same original model trained to detect the keyword "dreamdevices" and continues to output results over the serial console. As an added feature, the software blinks the LED's of the development kit. This feature also demonstrates how to add threaded functionality triggered by the classifier.

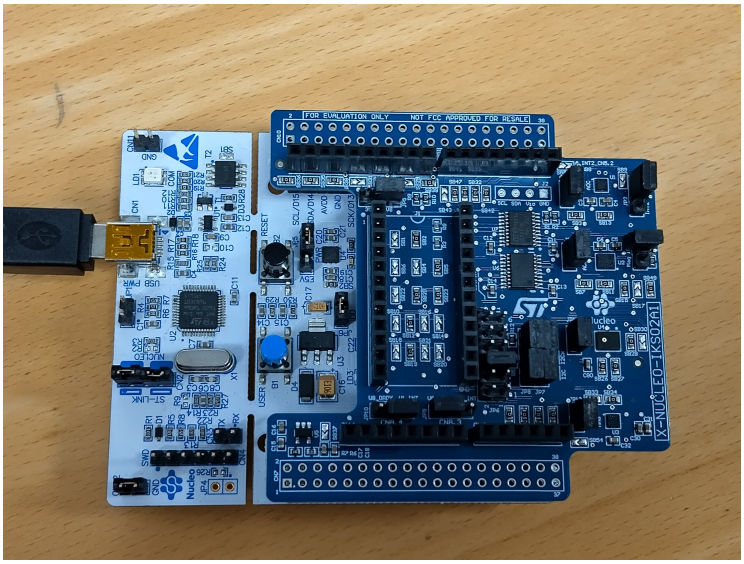

Figure 4.7: NUCLEO-L476RG with NUCLEO-IKS02A1

4.3 STM32L4

The STM32L4 series processor, specifically the L476, was chosen for the STM implementation. The L series of microcontrollers by STMicroelectronics is designed for low-power applications requiring high energy efficiency. For even lower power requirements, there is the newer U series, while the F and H series offer higher processing power. The L series serves as a solid reference point, although in many cases it may be replaced by the newer U series.

Figure 4.7 shows the NUCLEO-L476RG development board with the NUCLEO-IKS02A1 shield. Based on the ARM Cortex-M4F core with DSP extensions and a floating-point unit (FPU), the STM32L4 line offers a balance between performance and power consumption, making it suitable for low-power embedded AI workloads in energy-constrained environments.

While the newer STM32U series introduces ARM Cortex-M33 cores—comparable to the application core of the nRF5340—the STM32L4 series was selected to enable a more meaningful performance comparison between the Cortex-M4 and Cortex-M33 architectures.

This thesis uses the STM32L476RG variant, which features a 32-bit ARM Cortex-M4 core operating at 80 MHz. It includes 1 MB of flash memory and 128 KB of SRAM, which are sufficient for storing the machine learning model and the intermediate buffers required for signal processing and inference, although having less RAM than nRF5340. The MCU also integrates a rich set of peripherals, including multiple timers, ADCs, I^2C, SPI, USART, and notably, a flexible Serial Audio Interface (SAI) peripheral, which is employed in this work to acquire digital audio from a PDM microphone (IKS02A1) using DMA.

In addition to the hardware capabilities, the STM32 ecosystem provides extensive development support. STM32CubeMX is a graphical configuration tool that allows developers to generate initialization code for peripherals, middleware, and RTOS components through a hardware configuration—or more precisely, a pin-mapping—interface. The code generator tool is integrated with STM32CubeIDE and supports generation across various toolchains. Edge Impulse offers a dedicated STM32Cube deployment option, which packages the trained machine learning model, signal processing code, and classifier into a CubeMX-compatible software module, Cube package. This package can be directly imported into STM32CubeMX, accelerating the integration of Edge AI functionality into STM32-based projects. CubeMX packages can later be refreshed to update the model version. However, this creates the requirement that application developers write all custom code strictly within the designated user code sections, embedded in the CubeMX-generated codebase.

The following section will describe the development of a real-time keyword detection application targeting the STM32L476, using Edge Impulse for feature extraction and inference. The implementation reuses core concepts from the earlier nRF5340-based continuous audio classification prototype, but is adapted to the STM32 platform and the FreeRTOS operating system.

4.3.1 STM32L476 application development

The STM32L476-based implementation is structured around the STM32CubeMX-generated codebase and integrates the Edge Impulse SDK for continuous audio classification. The system utilizes FreeRTOS for task scheduling and SAI (Serial Audio Interface) with DMA support for efficient real-time audio acquisition from a digital PDM microphone. Dictated by the CubeMX generated codebase, core application logic is distributed across three key source files: `main.c`, `freertos.c`, and `sai.c`. `freertos.c`, and `sai.c` are listed in the Appendix C.

`main.c` contains only the system entry point and initialization stubs generated by STM32CubeMX. It serves as a minimal launcher, initializing the Hardware Abstraction Layer (HAL), clock configuration, and RTOS kernel before transferring control to FreeRTOS.

`sai.c` implements the microphone driver using the STM32 HAL SAI API. A statically allocated internal buffer is divided into two halves to enable double-buffered audio streaming via DMA. The two audio buffers are updated in the respective DMA transfer callbacks. These callbacks copy the DMA-managed memory into the application-facing buffers and flag new data availability.

`freertos.c` defines the application-level task responsible for executing the inference loop. A single FreeRTOS thread is created, which initializes the Edge Impulse classifier and starts audio recording. The thread periodically checks for new audio buffers signaled as ready, performing classifications using `run_classifier_continuous()`. The inference results are printed to a UART terminal, including keyword detection notifications and timing breakdowns.

This architecture provides continuous keyword spotting using a double buffer management scheme, minimal synchronization overhead, and a single classification thread. Audio preprocessing and inference are executed only when a complete buffer slice is available, allowing the system to operate within time constraints.

4.3.2 Performance limitations

During development, it became clear that the STM32L476 could not run the chosen keyword spotting model efficiently. Because of this, only the continuous inference version was implemented. The device struggled to keep up with real-time performance even in continuous mode with 250ms slices, so the non-continuous version was not developed. Compared to the nRF5340, the STM32L476 is at a disadvantage due to its lower clock speed and older core architecture, which lacks more advanced DSP and acceleration features. STM's M4F cores use the older ARMv7E-M architecture, whereas nRF5340's M33 use newer ARMv8-M architecture.

If real-time performance or comparison with the nRF5340 had not been required, a smaller classification model and simpler MFCC setting could have been used to better fit the processing power of the STM32L476.

While the STM32L4 series is well suited for low-power embedded systems, it may not be powerful enough for running larger machine learning models. STMicroelectronics also offers more advanced devices, such as the STM32N6 series, which include faster cores (like Cortex-M55) and optional NPUs (neural processing units). These would be more suitable for edge AI tasks but use more power, making them less ideal for battery-powered applications. A closer match to the nRF5340 in terms of performance and power consumption would be the STM32U series, having the same Cortex-M33 cores, but in lower clock speeds.

5 Performance

5.1 Performance of nRF5340

This section shows the resource usage and runtime performance of the non-continuous and continuous inference prototypes running on the nRF5340 development kit. Both implementations use the same keyword spotting model trained with Edge Impulse, but differ in how input buffering, MFCC feature extraction, and inferencing are handled.

Table 5.1: Performance and memory usage comparison

Metric	**Non-continuous**	**Continuous**
DSP time (MFCC)	93–94 ms	36–37 ms
Classification time	2–3 ms	2–3 ms
Window length	1 s	1 s
Inference interval	250 ms	100 ms
Flash usage	125536 B (11.97%)	132536 B (12.64%)
RAM usage	81488 B (17.76%)	35024 B (7.63%)

As seen in the Table 5.1 the most significant runtime improvement is in the DSP stage, where the continuously inferencing version reduces processing time from approximately 94ms to 37ms. This reduction is due to Edge Impulse's rolling MFCC buffer, which avoids recomputing spectrogram features for overlapping audio segments. The improvement is not 10-fold despite the 10-fold decrease in input length,

due to overhead associated with buffer management, overlapping window alignment, and internal scheduling.

The classification time remains identical in both cases, around 2–3 ms, since the underlying neural network model and its structure are the same.

In terms of memory usage, the continuous version consumes slightly more flash (an increase of 0.67%) due to the inclusion of additional SDK logic for rolling buffers and scheduling. However, RAM usage is significantly reduced, dropping from 81KB to 35KB. This is attributed to the removal of manual ring buffer logic and the use of more efficient internal memory handling in the Edge Impulse SDK.

Overall, the continuous inference implementation offers lower DSP latency, reduced RAM footprint, and more frequent classification intervals, resulting in a more responsive and resource-efficient system. In addition, the continuous model benefits from Edge Impulse's Performance Calibration tool, which provides recommended post-processing parameters—such as detection threshold, window duration, and suppression period—based on synthesized continuous audio tests. These parameters help fine-tune the interpretation of model outputs and improve detection reliability.

5.2 Performance of STM32L4

This section presents the resource usage and runtime performance of the non-continuous and continuous inference implementations on the STM32L476. The same keyword spotting model was used as in the nRF5340-based system, but this time exported as a Cube package and running under FreeRTOS instead of Zephyr.

Table 5.2: Performance and memory usage comparison

Metric	**Non-continuous**	**Continuous**
DSP time (MFCC)	400–402 ms	198–203 ms
Classification time	39–40 ms	39–40 ms
Window length	1 s	1 s
Inference interval	—	250 ms
Flash usage	—	215410 B (21.04%)
RAM usage	—	31560 B (24.65%)

As shown in Table 5.2, the continuous version requires nearly the entire 250 ms slice duration to perform inference, with DSP consuming around 198–203 ms and classification taking an additional 39–40 ms. This leaves very little margin, making real-time continuous inference barely feasible on the STM32L476. On the other hand, the model is optimally as demanding as it can be, when using 250ms windows. This "barely feasible" level of optimization may prove hard to achieve, if the window size is dictated beforehand.

The non-continuous version was briefly tested using pre-recorded audio. Processing a full 1-second audio block took approximately 402 ms for DSP and 39 ms for classification. Since this test was conducted only to confirm performance limitations, RAM and flash usage figures were not separately measured, as the same software was used.

Compared to the nRF5340, STM32L4 performs significantly worse when running the same model. This discrepancy can be attributed to differences in clock speed and core architecture: the STM32L476 uses a single-core ARM Cortex-M4F running 80 MHz, while the nRF5340 benefits from a faster Cortex-M33 application core running 128MHz.

The slightly higher flash usage is due to differences in RTOS implementation and

system libraries rather than the inference code or model itself, as model remained unchanged and application code size is insignificant. The figures show the memory necessary for the operating system and the artificial intelligence model, highlighting how much is left for the application development. As all microcontrollers have several versions with differing RAM and flash sizes, the memory estimations are necessary for the optimal selection.

6 Conclusions

This thesis explored the Edge Impulse platform, with implementations on low-power microcontrollers nRF5340 and STM32L476. The motivation behind the work was to evaluate how artificial intelligence models can be designed, deployed, and executed efficiently on resource-constrained low-power hardware without using any AI accelerators. The study focused on keyword spotting as the tested application, using the Edge Impulse platform for model development, optimization, and deployment.

The successful implementation of a real-time keyword recognition system on both target microcontrollers demonstrates the feasibility of deploying AI workloads on traditional MCUs. Furthermore, performance measurements and memory usage statistics were analyzed to help understand how microcontroller characteristics influence Edge AI performance. For low-power platforms, often battery-powered, local inference offers energy savings by avoiding constant communication with the cloud. Other benefits of intelligent edge computing—such as improved privacy, reduced latency, and enhanced reliability—are discussed in Section 2.2.1.

6.1 Developing AI models on Edge Impulse

One of the key outcomes of this thesis was to show that developing AI models for low-power microcontrollers can be done effectively using modern toolchains such as Edge Impulse. The platform provides an accessible web-based interface, suitable for developers with limited background in machine learning. At the same time,

it allows full control over each development step, including preprocessing, model architecture, and deployment. The platform combines ease of use for non-experts with deep access to code-level customization for experienced developers.

Model deployment was streamlined thanks to built-in integration with Zephyr for nRF5340 and STM32CubeMX for STM32. While CubeMX requires developers to write code only within designated "user code" sections, this constraint does not limit functionality. However, it does introduce inconvenience in later application development. Developers may discard CubeMX and use the standard C++ library and toolchains for full flexibility, or use CubeMX for initial codebase generation and model testing, and discard it later.

Overall, the development process was productive and approachable. Developers working on embedded AI applications can benefit from these tools to build and refine models quickly.

6.2 Selecting Suitable MCUs for Edge AI Applications

This work also provided a side-by-side comparison of two low-power MCU platforms, demonstrating how differences in architecture and clock speed affect their suitability for AI workloads. The nRF5340, with its dual-core Cortex-M33 running at 128 MHz, showed strong performance in both continuous and non-continuous inference modes. Its faster DSP times and shorter inference latency made it well-suited for real-time applications.

The STM32L476, while also capable of running the same AI model, reached the limits of its processing power. Its older Cortex-M4F architecture and 80 MHz clock speed resulted in longer DSP and classification times, leaving very little timing margin for real-time inference. However, with careful tuning and the use of Edge

Impulse's continuous inference tools, real-time operation was still achieved using 250 ms slices.

Edge Impulse provides estimation tools that predict inference latency, RAM, and flash usage during model development. While these estimates are useful for quick planning, this thesis found that they are not fully accurate for all platforms. For the STM32L4, the estimations were somewhat optimistic, while for the nRF5340, they were slightly conservative. Fortunately, Edge Impulse enables rapid deployment of models to real hardware. This makes it possible to measure actual performance early, enabling quick iteration with model and DSP parameters. It is recommended to run real performance tests as early as possible, before integrating the model into the full application. This allows selecting or optimizing the model based on actual measurements and memory usage, rather than predictions alone. Using a simple RTOS implementation without additional application logic, the RAM and flash memory usage of both the model and the operating system can be measured early in the process to evaluate how much capacity remains available for application development.

These results show that selecting a microcontroller for Edge AI requires careful consideration of:

- Latency and timing requirements
- Power consumption limits
- Available memory resources
- Compatibility and selection of toolchains and RTOS

Importantly, the results also demonstrate that MCUs without AI accelerators can still meet the needs of many real-world inference tasks, provided the model is optimized appropriately and the timing constraints are reasonable. Even larger

models may be used with upcoming low-power microcontrollers without AI accelerators, like STM32U5A9, that features Cortex-M33 cores running at 160 MHz. The U-series is expected to replace the older L-series. For even higher performance, developers may consider MCUs outside the traditional low-power category, such as the STM32H series, which scales up to 480 MHz. However, using such high-performance MCUs raises new questions about overall energy efficiency compared to low-power MCUs paired with dedicated AI accelerators, which introduces future investigations.

6.3 Design Scope and Future Work

A clear design choice in this thesis was to limit the scope to general-purpose MCUs without dedicated AI acceleration hardware. This approach reflects realistic constraints in many commercial projects, especially where battery life, cost, and simplicity are priorities.

Nevertheless, evaluating performance on more advanced microcontrollers—such as those with Cortex-M55 cores or built-in NPUs—would be a valuable extension. Such platforms may support deeper models or higher inference frequencies and can provide additional options for developers facing stricter performance requirements or for expanding the range of viable AI applications.

Future research could also explore:

- Broader comparisons including the STM32U series or similar M33-based MCUs
- Microcontrollers that don't belong to the low-power category, like STM32H series
- Comparisons of high-performance microcontrollers vs. low-power microcontrollers + AI accelerators

These directions could further enhance understanding of trade-offs between performance, accuracy, and power efficiency across a range of embedded platforms.

6.4 Final Remarks

This thesis shows that artificial intelligence is not limited to high-performance hardware or cloud computing. With the right tools and careful engineering, meaningful machine learning models can be deployed directly onto low-power microcontrollers using only their general-purpose compute resources. The results highlight how modern development platforms such as Edge Impulse can enable embedded developers to adopt AI techniques without deep AI expertise.

By comparing nRF and STM platforms, and demonstrating complete working implementations, this thesis offers practical insights and performance data that can support future design decisions. As embedded AI continues to expand into new application domains, the ability to match model and hardware capabilities efficiently will be increasingly important—and tools and workflows validated in this thesis will play a central role in achieving that.

References

[1] M. R. Abdmeziem and D. Tandjaoui, "Architecting the internet of things: State of the art", in *Advanced Internet of Things*, P. Sabo, S. Sahraoui, and A. Bouhoula, Eds., Springer, 2015, pp. 1–25. DOI: 10.1007/978-3-319-23126-6_1.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges", *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. DOI: 10.1109/JIOT.2016.2579198.

[3] Nordic Semiconductor, "Introduction to Edge AI and Nordic ML Studio", *Nordic Tech Tour, EMEA*, 2024.

[4] *What is edge MLOps? | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/concepts/edge-ai-fundamentals/what-is-edge-mlops, [Accessed 03-03-2025].

[5] *What is CI/CD? — redhat.com*, https://www.redhat.com/en/topics/devops/what-is-ci-cd, [Accessed 03-03-2025].

[6] *Retrain model | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-impulse-studio/retrain-model, [Accessed 18-03-2025].

[7] *Synthetic data | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition/synthetic-data, [Accessed 03-03-2025].

[8] *EON Tuner | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-impulse-studio/eon-tuner, [Accessed 17-03-2025].

[9] *Deployment | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-impulse-studio/deployment, [Accessed 17-03-2025].

[10] *Edge Impulse for Linux | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/tools/edge-impulse-for-linux, [Accessed 17-03-2025].

[11] *Overview | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-ai-hardware/edge-ai-hardware, [Accessed 19-03-2025].

[12] BrainChip Holdings Ltd., *BrainChip Brings Neuromorphic Capabilities to M.2 Form Factor - BrainChip — brainchip.com*, https://brainchip.com/brainchip-brings-neuromorphic-capabilities-to-m-2-form-factor/, [Accessed 19-03-2025].

[13] *GPU | Edge Impulse Documentation — docs.edgeimpulse.com*, https://docs.edgeimpulse.com/docs/edge-ai-hardware/gpu, [Accessed 19-03-2025].

[14] *GitHub - edgeimpulse/firmware-nordic-nrf52840dk-nrf5340dk: Official Edge Impulse firmware for nRF52840 DK and nRF5340 DK — github.com*, https://github.com/edgeimpulse/firmware-nordic-nrf52840dk-nrf5340dk, [Accessed 07-04-2025].

[15] *Audio classification - keyword spotting | Edge Impulse*, Apache 2.0, 2024. [Online]. Available: https://studio.edgeimpulse.com/public/499022/latest.

[16] D. O'Shaughnessy, *Speech Communication: Human and Machine*. Addison-Wesley, 1987.

[17] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 357–366, 1980.

[18] S. Hershey, S. Chaudhuri, D. P. Ellis, *et al.*, "Cnn architectures for large-scale audio classification", in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–135.

[19] *Responding to your voice - audio classification | Edge Impulse*, Accessed: 2025-04-10, 2024. [Online]. Available: https://docs.edgeimpulse.com/docs/tutorials/end-to-end-tutorials/audio/responding-to-your-voice.

[20] *NRF5340 - Nordic Semiconductor — nordicsemi.com*, https://www.nordicsemi.com/Products/nRF5340, [Accessed 19-03-2025].

[21] *Nordicsemi.com*, https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF5340-SoC-PB.pdf, [Accessed 22-05-2025].

Appendix A: nRF5340

Non-continuous software

```
#include <zephyr/kernel.h>
#include <zephyr/irq.h>
#include <nrfx_pdm.h>
#include <nrfx_clock.h>
#include <string.h>
#include "edge-impulse-sdk/classifier/ei_run_classifier.h"
#include "edge-impulse-sdk/dsp/numpy.hpp"

// --- GPIO pin configuration ---
#define PDM_CLK_PIN 37
#define PDM_DIN_PIN 38

// --- Audio sampling parameters ---
#define SAMPLE_RATE_HZ 16000
#define SAMPLE_DURATION_MS 250
#define SAMPLE_COUNT ((SAMPLE_RATE_HZ * SAMPLE_DURATION_MS) / 1000)
#define PDM_BUFFER_SIZE SAMPLE_COUNT

#define CHUNK_SIZE 512
#define MAX_BUFFERS 4
#define INFERENCE_BUFFER_SIZE (PDM_BUFFER_SIZE * MAX_BUFFERS)

// --- Zephyr threads configurations ---
#define PDM_STACK_SIZE 2048
#define PDM_THREAD_PRIORITY 1
#define INFERENCE_STACK_SIZE 2048
#define INFERENCE_THREAD_PRIORITY 2

K_THREAD_STACK_DEFINE(pdm_stack_area, PDM_STACK_SIZE);
static struct k_thread pdm_thread_data;
K_THREAD_STACK_DEFINE(inference_stack_area, INFERENCE_STACK_SIZE);
static struct k_thread inference_thread_data;
K_SEM_DEFINE(inference_trigger, 0, 1);

// --- Static PDM instance ---
static const nrfx_pdm_t my_pdm = NRFX_PDM_INSTANCE(0);

// --- Impulse runner ---
void start_inference();

// --- Buffers ---
static int16_t pdm_buffers[MAX_BUFFERS][PDM_BUFFER_SIZE];
static int16_t pdm_chunk_buffer[CHUNK_SIZE];
static int16_t inference_buffer[INFERENCE_BUFFER_SIZE];
static volatile size_t samples_collected = 0;
static volatile bool recording_done = false;
static volatile int current_buffer_index = 0;
static volatile int inference_buffer_index = -1;

// --- PDM event handler ---
static void pdm_event_handler(nrfx_pdm_evt_t const *p_evt)
{
    if (p_evt->buffer_released && !recording_done)
```

```
    {
        size_t remaining = PDM_BUFFER_SIZE - samples_collected;
        size_t copy_count = (CHUNK_SIZE > remaining) ? remaining : CHUNK_SIZE;

        memcpy(&pdm_buffers[current_buffer_index][samples_collected],
            pdm_chunk_buffer, copy_count * sizeof(int16_t));
        samples_collected += copy_count;

        if (samples_collected >= PDM_BUFFER_SIZE)
        {
            nrfx_pdm_stop(&my_pdm);
            recording_done = true;
            printk("PDM: Recording complete (%d samples) in buffer index %d\n",
                (int)samples_collected, current_buffer_index);
        }
    }

    if (p_evt->buffer_requested && !recording_done)
    {
        nrfx_err_t err = nrfx_pdm_buffer_set(&my_pdm, pdm_chunk_buffer,
            CHUNK_SIZE);
        if (err != NRFX_SUCCESS)
        {
            printk("PDM: Buffer set failed: %d\n", err);
        }
    }

    if (p_evt->error != 0)
    {
        printk("PDM: Error occurred: %d\n", p_evt->error);
    }
}

#define PDM_IRQ 38

static void pdm_irq_wrapper(const void *arg)
{
    ARG_UNUSED(arg);
    nrfx_pdm_0_irq_handler();
}

// --- PDM initialization ---
static uint32_t pdm_clock_calculate(uint64_t sampleRate)
{
    const uint64_t PDM_RATIO = 80ULL;
    const uint64_t CLK_32MHZ = 32000000ULL;
    uint64_t clk_control = 4096ULL * (((sampleRate * PDM_RATIO) * 1048576ULL) /
        (CLK_32MHZ + ((sampleRate * PDM_RATIO) / 2ULL)));

    return (uint32_t)clk_control;
}

static void pdm_init(void)
{
    nrfx_pdm_config_t config_pdm = NRFX_PDM_DEFAULT_CONFIG(PDM_CLK_PIN,
        PDM_DIN_PIN);
    config_pdm.clock_freq = (nrf_pdm_freq_t)pdm_clock_calculate(SAMPLE_RATE_HZ);
    config_pdm.ratio = NRF_PDM_RATIO_80X;
    config_pdm.edge = NRF_PDM_EDGE_LEFTRISING;
    config_pdm.gain_l = NRF_PDM_GAIN_MAXIMUM;
    config_pdm.gain_r = NRF_PDM_GAIN_MAXIMUM;

    nrfx_err_t err = nrfx_pdm_init(&my_pdm, &config_pdm, pdm_event_handler);
    if (err != NRFX_SUCCESS)
    {
        printk("PDM: Initialization failed: %d\n", err);
        return;
    }
    printk("PDM: Initialized\n");

    irq_disable(PDM_IRQ);
    irq_connect_dynamic(PDM_IRQ, 6, pdm_irq_wrapper, NULL, 0);
    irq_enable(PDM_IRQ);
}

// --- PDM recording thread ---
void pdm_thread(void *p1, void *p2, void *p3)
```

```
126 {
127     pdm_init();
128 
129     while (1)
130     {
131         printk("PDM: Starting recording into buffer index %d...\n",
                current_buffer_index);
132 
133         samples_collected = 0;
134         recording_done = false;
135 
136         nrfx_err_t err = nrfx_pdm_start(&my_pdm);
137         if (err != NRFX_SUCCESS)
138         {
139             printk("PDM: Start failed: %d\n", err);
140             k_sleep(K_SECONDS(1));
141             continue;
142         }
143 
144         while (!recording_done)
145         {
146             k_sleep(K_MSEC(10));
147         }
148 
149         inference_buffer_index = current_buffer_index;
150         k_sem_give(&inference_trigger);
151 
152         current_buffer_index = (current_buffer_index + 1) % MAX_BUFFERS;
153     }
154 }
155 
156 // --- Combine 4x250ms into 1s buffer ---
157 void audio_init_data()
158 {
159     for (int i = 0; i < MAX_BUFFERS; i++)
160     {
161         int buff_index = (inference_buffer_index + i) % MAX_BUFFERS;
162 
163         memcpy(&inference_buffer[i * PDM_BUFFER_SIZE],
164                pdm_buffers[buff_index],
165                PDM_BUFFER_SIZE * sizeof(int16_t));
166     }
167 }
168 
169 int audio_get_data(size_t offset, size_t length, float *out_ptr)
170 {
171     memcpy(out_ptr, inference_buffer + offset, length * sizeof(float));
172     return 0;
173 }
174 
175 void inference_thread(void *p1, void *p2, void *p3)
176 {
177     while(1)
178     {
179         k_sem_take(&inference_trigger, K_FOREVER);
180 
181         ei_impulse_result_t result = {0};
182         signal_t signal;
183         signal.total_length = 16000;
184         audio_init_data();
185         signal.get_data = &audio_get_data;
186 
187         EI_IMPULSE_ERROR res = run_classifier(&signal, &result, false);
188         printk("run_classifier returned: %d\n", res);
189 
190         printk("Timings\n    DSP: %d ms., Classification: %d ms., Anomaly: %d
                ms.): \n",
191             result.timing.dsp, result.timing.classification,
                    result.timing.anomaly);
192 
193         for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++)
194         {
195             printk("    %s: %.5f\n", result.classification[ix].label,
196                 result.classification[ix].value);
197         }
198 
199         if (result.classification[0].value > 0.33)
```

```
        {
            printk("\n****************** Keyword spotted ******************");
        }
    }
}

// --- Main entry ---
int main(void)
{
    setvbuf(stdout, NULL, _IONBF, 0);
    nrfx_clock_divider_set(NRF_CLOCK_DOMAIN_HFCLK, NRF_CLOCK_HFCLK_DIV_1);
    printk("System: Starting nrfx PDM sampling (4x250ms = 1s @ 16kHz)...\n");

    k_thread_create(&pdm_thread_data, pdm_stack_area,
                    K_THREAD_STACK_SIZEOF(pdm_stack_area),
                    pdm_thread,
                    NULL, NULL, NULL,
                    PDM_THREAD_PRIORITY, 0, K_NO_WAIT);

    k_thread_create(&inference_thread_data, inference_stack_area,
                    K_THREAD_STACK_SIZEOF(inference_stack_area),
                    inference_thread,
                    NULL, NULL, NULL,
                    INFERENCE_THREAD_PRIORITY, 0, K_SECONDS(1));

    return 0;
}
```

Appendix B: nRF5340 Continuous software

```
1
2  #include <zephyr/kernel.h>
3  #include <zephyr/irq.h>
4  #include <nrfx_pdm.h>
5  #include <nrfx_clock.h>
6  #include <string.h>
7  #include <stdio.h>
8  #include <zephyr/drivers/gpio.h>
9
10 #include "edge-impulse-sdk/classifier/ei_run_classifier.h"
11 #include "edge-impulse-sdk/dsp/numpy.hpp"
12
13 #define PDM_CLK_PIN 37
14 #define PDM_DIN_PIN 38
15
16 #define SAMPLE_RATE_HZ 16000
17 #define SAMPLE_DURATION_MS 100
18 #define SLICE_SIZE ((SAMPLE_RATE_HZ * SAMPLE_DURATION_MS) / 1000)
19 #define CHUNK_SIZE 512  // Multiple of 16 for DMA
20
21 static const nrfx_pdm_t my_pdm = NRFX_PDM_INSTANCE(0);
22
23 /**
24  * Buffering ensures that while one buffer is used for inference, the
25  * other fills with new data. int16_t buffer[2][SLICE_SIZE] holds PDM data.
26  * int buf_select and int buf_ready are toggled in SW, as buf_count is used to
27  * detect current buffer state for switching and overflows.
28  */
29 static int16_t buffer[2][SLICE_SIZE];
30 static volatile int buf_select = 0;
31 static volatile int buf_ready = 0;
32 static volatile uint32_t buf_count = 0;
33
34 static int16_t pdm_chunk[CHUNK_SIZE];
35 static bool debug = false;
36
37 /**
38  * LED configurations and thread to indicate keyword detection without terminal
39  * Works as an example on how to add threaded functionality in the SW with
40  * simple binary semaphore.
41  */
42 #define LED1_NODE DT_ALIAS(led1)
43 #define LED2_NODE DT_ALIAS(led2)
44 #define LED3_NODE DT_ALIAS(led3)
45 #define LED4_NODE DT_ALIAS(led4)
46 #define NUM_LEDS (sizeof(leds) / sizeof(leds[0]))
47 #define LED_STACK_SIZE 1024
48 #define LED_PRIORITY 5
49 static const struct gpio_dt_spec leds[] = {
50     GPIO_DT_SPEC_GET(LED1_NODE, gpios),
51     GPIO_DT_SPEC_GET(LED2_NODE, gpios),
52     GPIO_DT_SPEC_GET(LED3_NODE, gpios),
53     GPIO_DT_SPEC_GET(LED4_NODE, gpios),
```

```
};
K_THREAD_STACK_DEFINE(led_stack, LED_STACK_SIZE);
struct k_thread led_thread_data;
void toggle_leds_worm_thread(void *p1, void *p2, void *p3);
K_SEM_DEFINE(led_trigger, 0, 1);

// Audio signal callback for Edge Impulse
int get_audio_signal(size_t offset, size_t length, float *out_ptr)
{
    numpy::int16_to_float(&buffer[buf_select ^ 1][offset], out_ptr, length);
    return 0;
}

// PDM audio IRQ handler wrapper
#define PDM_IRQ 38
static void pdm_irq_wrapper(const void *arg)
{
    ARG_UNUSED(arg);
    nrfx_pdm_0_irq_handler();
}

// PDM audio event handler
static void pdm_event_handler(nrfx_pdm_evt_t const *p_evt)
{
    if (p_evt->buffer_requested && buf_ready == 0)
    {
        nrfx_pdm_buffer_set(&my_pdm, pdm_chunk, CHUNK_SIZE);
    }

    if (p_evt->buffer_released && buf_ready == 0)
    {
        size_t remaining = SLICE_SIZE - buf_count;
        size_t copy = CHUNK_SIZE > remaining ? remaining : CHUNK_SIZE;

        memcpy(&buffer[buf_select][buf_count], pdm_chunk, copy * sizeof(int16_t));
        buf_count += copy;

        if (buf_count >= SLICE_SIZE)
        {
            buf_ready = 1;

            // Handle overflow: copy remaining chunk to the other buffer
            size_t overflow = CHUNK_SIZE - copy;
            if (overflow > 0)
            {
                memcpy(&buffer[buf_select ^ 1][0], &pdm_chunk[copy], overflow *
                    sizeof(int16_t));
                buf_count = overflow;
            }
            else
            {
                buf_count = 0;
            }

            buf_select ^= 1;
        }
    }
}

// PDM audio initialization
static void pdm_init(void)
{
    nrfx_pdm_config_t config_pdm = NRFX_PDM_DEFAULT_CONFIG(PDM_CLK_PIN,
        PDM_DIN_PIN);
    config_pdm.clock_freq = NRF_PDM_FREQ_1032K;
    config_pdm.ratio = NRF_PDM_RATIO_80X;
    config_pdm.edge = NRF_PDM_EDGE_LEFTRISING;
    config_pdm.gain_l = NRF_PDM_GAIN_MAXIMUM;
    config_pdm.gain_r = NRF_PDM_GAIN_MAXIMUM;

    nrfx_err_t err = nrfx_pdm_init(&my_pdm, &config_pdm, pdm_event_handler);
    if (err != NRFX_SUCCESS)
    {
        printk("PDM: Initialization failed: %d\n", err);
        return;
    }
    printk("PDM: Initialized\n");
```

```
129
130     irq_disable(PDM_IRQ);
131     irq_connect_dynamic(PDM_IRQ, 6, pdm_irq_wrapper, NULL, 0);
132     irq_enable(PDM_IRQ);
133 }
134
135
136 // Main loop
137 int main(void)
138 {
139     setvbuf(stdout, NULL, _IONBF, 0);
140
141     k_thread_create(&led_thread_data, led_stack, LED_STACK_SIZE,
142                     toggle_leds_worm_thread, NULL, NULL, NULL,
143                     LED_PRIORITY, 0, K_NO_WAIT);
144
145     // 128 MHz CPU clock
146     nrfx_clock_divider_set(NRF_CLOCK_DOMAIN_HFCLK, NRF_CLOCK_HFCLK_DIV_1);
147
148     pdm_init();
149     nrfx_pdm_start(&my_pdm);
150
151     static signal_t signal;
152     signal.total_length = SLICE_SIZE;
153     signal.get_data = &get_audio_signal;
154     run_classifier_init();
155
156     printk("MAIN: run_classifier_continuous inferencing starting.\n");
157
158     while (1)
159     {
160         if (buf_ready == 1)
161         {
162             buf_ready = 0;
163
164             ei_impulse_result_t result = {0};
165             EI_IMPULSE_ERROR r = run_classifier_continuous(&signal, &result,
                    debug);
166
167             if (r != EI_IMPULSE_OK)
168             {
169                 printk("ERROR: run_classifier_continuous returned error (%d)\n",
                        r);
170             }
171
172             // Performance calibration is configured for this project,
173             // meaning the outputs of continuous classification are all 1 or 0.
174             if ((float)result.classification[0].value)
175             {
176                 k_sem_give(&led_trigger);
177                 printk("***************** Keyword recognized *****************\n");
178             }
179         }
180         else
181         {
182             k_msleep(1);
183         }
184     }
185     return 0;
186 }
187
188 void toggle_leds_worm_thread(void *p1, void *p2, void *p3)
189 {
190     printk("LED thread starting");
191     int err = 0;
192     err += gpio_pin_configure_dt(&leds[0], GPIO_OUTPUT_INACTIVE);
193     err += gpio_pin_configure_dt(&leds[1], GPIO_OUTPUT_INACTIVE);
194     err += gpio_pin_configure_dt(&leds[2], GPIO_OUTPUT_INACTIVE);
195     err += gpio_pin_configure_dt(&leds[3], GPIO_OUTPUT_INACTIVE);
196     if (err)
197     {
198         printk("LED GPIO configuration invalid.");
199         return;
200     }
201
202     const int worm_interval_ms = 100;
203
```

```
204     while (1)
205     {
206         k_sem_take(&led_trigger, K_FOREVER);
207 
208         for (size_t i = 0; i < NUM_LEDS; i++)
209         {
210             gpio_pin_set_dt(&leds[i], 1);
211             k_msleep(worm_interval_ms);
212         }
213 
214         k_msleep(worm_interval_ms);
215 
216         for (size_t i = 0; i < NUM_LEDS; i++)
217         {
218             gpio_pin_set_dt(&leds[i], 0);
219             k_msleep(worm_interval_ms);
220         }
221     }
222 }
```

Appendix C: STM32L476

Continuous software

Listing 1: freertos.c

```
1
2  #include "FreeRTOS.h"
3  #include "task.h"
4  #include "main.h"
5  #include "cmsis_os.h"
6
7  /* USER CODE BEGIN Includes */
8  #include "sai.h"
9  #include "buffers.h"
10 #include "edge-impulse-sdk/classifier/ei_run_classifier.h"
11 #include "edge-impulse-sdk/dsp/numpy_types.h"
12 /* USER CODE END Includes */
13
14 /* Definitions for defaultTask */
15 osThreadId_t defaultTaskHandle;
16 const osThreadAttr_t defaultTask_attributes = {
17   .name = "defaultTask",
18   .stack_size = 1512 * 4,
19   .priority = (osPriority_t) osPriorityNormal,
20 };
21
22 /* Private function prototypes -----------------------------------------------*/
23 /* USER CODE BEGIN FunctionPrototypes */
24 int get_audio_signal(size_t offset, size_t length, float *out_ptr);
25
26 extern "C" {
27 void MX_FREERTOS_Init(void);
28 }
29
30 /* USER CODE END FunctionPrototypes */
31
32 void StartDefaultTask(void *argument);
33
34 void MX_FREERTOS_Init(void);
35
36 void MX_FREERTOS_Init(void) {
37   defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);
38 }
39
40 void StartDefaultTask(void *argument)
41 {
42   /* USER CODE BEGIN StartDefaultTask */
43     ei_impulse_result_t result = {0};
44     static signal_t signal;
45     signal.total_length = SLICE_SIZE;
46     signal.get_data = &get_audio_signal;
47
48     run_classifier_init();
49
50     if (pdm_start_recording() != PDM_OK) {
```

```
        Error_Handler();
    }

    uart_log("\n\n\nSystem initialized\n");
    /* Infinite loop */
    while (1)
    {
        if (buf_ready)
        {
            buf_ready = 0;

            EI_IMPULSE_ERROR err = run_classifier_continuous(&signal, &result,
                false);

            if (err != EI_IMPULSE_OK)
            {
                uart_log("ERROR: run_classifier_continuous returned error\n");
            }

            // Performance calibration is configured for this project,
            // meaning the outputs of continuous classification are all 1 or 0.
            if ((float)result.classification[0].value)
            {
                uart_log("\n***************** Keyword recognized
                    ******************\n");
                char uart_buf[64];
                snprintf(uart_buf, sizeof(uart_buf), "Timings: classification:
                    %dms, dsp: %dms\r\n",
                          result.timing.classification, result.timing.dsp);
                uart_log(uart_buf);
            }
        }
        osDelay(1);
    }
  /* USER CODE END StartDefaultTask */
}

/* USER CODE BEGIN Application */
int get_audio_signal(size_t offset, size_t length, float *out_ptr)
{
    int local_buf = buf_select;  // Prevent race condition
    numpy::int16_to_float(&audio_buffer[local_buf][offset], out_ptr, length);
    return 0;
}

// For optional edge impulse model debugging
void ei_printf(const char *format, ...) {
    char buffer[128];
    va_list args;
    va_start(args, format);
    vsnprintf(buffer, 128, format, args);
    va_end(args);
    uart_log(buffer);
}
/* USER CODE END Application */
```

Listing 2: sai.c

```

#include "sai.h"

/* USER CODE BEGIN 0 */
#include "buffers.h"
#include <string.h>

uint16_t SAI_InternalBuffer[SLICE_SIZE * 2];

int16_t *audio_buffer[2] = {
    (int16_t*)&SAI_InternalBuffer[0],
    (int16_t*)&SAI_InternalBuffer[SLICE_SIZE]
};

volatile int buf_select = 0;
volatile int buf_ready = 0;

int pdm_start_recording(void)
```

```
19 {
20   return HAL_SAI_Receive_DMA(&hsai_BlockB1, (uint8_t*)SAI_InternalBuffer,
         SLICE_SIZE * 2) == HAL_OK ? 0 : -1;
21 }
22
23 void HAL_SAI_RxHalfCpltCallback(SAI_HandleTypeDef *hsai)
24 {
25     memcpy(audio_buffer[0], &SAI_InternalBuffer[0], SLICE_SIZE * sizeof(int16_t));
26     buf_select = 0;
27     buf_ready = 1;
28 }
29
30 void HAL_SAI_RxCpltCallback(SAI_HandleTypeDef *hsai)
31 {
32     memcpy(audio_buffer[1], &SAI_InternalBuffer[SLICE_SIZE], SLICE_SIZE *
           sizeof(int16_t));
33     buf_select = 1;
34     buf_ready = 1;
35 }
36 /* USER CODE END 0 */
37
38 SAI_HandleTypeDef hsai_BlockB1;
39 DMA_HandleTypeDef hdma_sai1_b;
40
41 void MX_SAI1_Init(void)
42 {
43
44   hsai_BlockB1.Instance = SAI1_Block_B;
45   hsai_BlockB1.Init.Protocol = SAI_FREE_PROTOCOL;
46   hsai_BlockB1.Init.AudioMode = SAI_MODEMASTER_RX;
47   hsai_BlockB1.Init.DataSize = SAI_DATASIZE_16;
48   hsai_BlockB1.Init.FirstBit = SAI_FIRSTBIT_MSB;
49   hsai_BlockB1.Init.ClockStrobing = SAI_CLOCKSTROBING_RISINGEDGE;
50   hsai_BlockB1.Init.Synchro = SAI_ASYNCHRONOUS;
51   hsai_BlockB1.Init.OutputDrive = SAI_OUTPUTDRIVE_DISABLE;
52   hsai_BlockB1.Init.NoDivider = SAI_MASTERDIVIDER_ENABLE;
53   hsai_BlockB1.Init.FIFOThreshold = SAI_FIFOTHRESHOLD_EMPTY;
54   hsai_BlockB1.Init.AudioFrequency = SAI_AUDIO_FREQUENCY_16K;
55   hsai_BlockB1.Init.SynchroExt = SAI_SYNCEXT_DISABLE;
56   hsai_BlockB1.Init.MonoStereoMode = SAI_STEREOMODE;
57   hsai_BlockB1.Init.CompandingMode = SAI_NOCOMPANDING;
58   hsai_BlockB1.FrameInit.FrameLength = 256;
59   hsai_BlockB1.FrameInit.ActiveFrameLength = 1;
60   hsai_BlockB1.FrameInit.FSDefinition = SAI_FS_STARTFRAME;
61   hsai_BlockB1.FrameInit.FSPolarity = SAI_FS_ACTIVE_LOW;
62   hsai_BlockB1.FrameInit.FSOffset = SAI_FS_FIRSTBIT;
63   hsai_BlockB1.SlotInit.FirstBitOffset = 0;
64   hsai_BlockB1.SlotInit.SlotSize = SAI_SLOTSIZE_DATASIZE;
65   hsai_BlockB1.SlotInit.SlotNumber = 1;
66   hsai_BlockB1.SlotInit.SlotActive = 0x00000001;
67   if (HAL_SAI_Init(&hsai_BlockB1) != HAL_OK)
68   {
69     Error_Handler();
70   }
71 }
72 static uint32_t SAI1_client =0;
73
74 void HAL_SAI_MspInit(SAI_HandleTypeDef* saiHandle)
75 {
76
77   GPIO_InitTypeDef GPIO_InitStruct;
78   RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};
79
80     if(saiHandle->Instance==SAI1_Block_B)
81     {
82
83     PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_SAI1;
84     PeriphClkInit.Sai1ClockSelection = RCC_SAI1CLKSOURCE_PLLSAI1;
85     PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_MSI;
86     PeriphClkInit.PLLSAI1.PLLSAI1M = 1;
87     PeriphClkInit.PLLSAI1.PLLSAI1N = 18;
88     PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV17;
89     PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
90     PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
91     PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_SAI1CLK;
92     if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
93     {
```

```
94        Error_Handler();
95      }
96
97        if (SAI1_client == 0)
98        {
99          __HAL_RCC_SAI1_CLK_ENABLE();
100       }
101     SAI1_client ++;
102
103     GPIO_InitStruct.Pin = GPIO_PIN_4;
104     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
105     GPIO_InitStruct.Pull = GPIO_NOPULL;
106     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
107     GPIO_InitStruct.Alternate = GPIO_AF13_SAI1;
108     HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
109
110     GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_5;
111     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
112     GPIO_InitStruct.Pull = GPIO_NOPULL;
113     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_MEDIUM;
114     GPIO_InitStruct.Alternate = GPIO_AF13_SAI1;
115     HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
116
117
118     hdma_sai1_b.Instance = DMA2_Channel2;
119     hdma_sai1_b.Init.Request = DMA_REQUEST_1;
120     hdma_sai1_b.Init.Direction = DMA_PERIPH_TO_MEMORY;
121     hdma_sai1_b.Init.PeriphInc = DMA_PINC_DISABLE;
122     hdma_sai1_b.Init.MemInc = DMA_MINC_ENABLE;
123     hdma_sai1_b.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
124     hdma_sai1_b.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
125     hdma_sai1_b.Init.Mode = DMA_CIRCULAR;
126     hdma_sai1_b.Init.Priority = DMA_PRIORITY_HIGH;
127     if (HAL_DMA_Init(&hdma_sai1_b) != HAL_OK)
128     {
129       Error_Handler();
130     }
131
132     __HAL_LINKDMA(saiHandle,hdmarx,hdma_sai1_b);
133     __HAL_LINKDMA(saiHandle,hdmatx,hdma_sai1_b);
134     }
135 }
136
137 void HAL_SAI_MspDeInit(SAI_HandleTypeDef* saiHandle)
138 {
139
140     if(saiHandle->Instance==SAI1_Block_B)
141     {
142     SAI1_client --;
143       if (SAI1_client == 0)
144       {
145       __HAL_RCC_SAI1_CLK_DISABLE();
146       }
147
148     HAL_GPIO_DeInit(GPIOA, GPIO_PIN_4);
149
150     HAL_GPIO_DeInit(GPIOB, GPIO_PIN_3|GPIO_PIN_5);
151
152     HAL_DMA_DeInit(saiHandle->hdmarx);
153     HAL_DMA_DeInit(saiHandle->hdmatx);
154     }
155 }
```