UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

BACHELOR'S THESIS

TinyML: From Basic to Advanced Applications

Author:
Marc MONFORT GRAU

Director:
Dr. Felix FREITAG

Co-director:
Roger Pueyo Centelles

A thesis submitted in fulfillment of the requirements
for the degree of Informatics Engineering specialising in Computing

June 21, 2021

Abstract

TinyML aims to implement machine learning (ML) applications on small, and low-powered devices like microcontrollers. Typically, edge devices need to be connected to data centers in order to run ML applications. However, this approach is not possible in many scenarios, such as lack of connectivity. This project investigates the tools and techniques used in TinyML, the constraints of using low-powered devices, and the feasibility of implementing advanced machine learning applications on microcontrollers.

To test the feasibility of implementing ML applications on microcontrollers, three TinyML programs were developed. The first, a basic keyword spotting application able to recognize a set of words. The second, a program for training a neural network model on a microcontroller following an online learning approach. And the third, a federated learning program able to train a single global model with the aggregation of local models trained on multiple microcontrollers. The results show optimal performance in all three applications once deployed on microcontrollers. The development of basic TinyML applications is straightforward when the machine learning pipeline is understood. However, the development of advanced applications turned out to be very complex, as it requires a deep understanding of both machine learning and embedded systems.

These results prove the feasibility of successfully implementing advanced ML applications on microcontrollers, and thus, unveil a bright future for TinyML.

Keywords: TinyML, machine learning, microcontroller, keyword spotting, federated learning

Resumen

TinyML tiene como objetivo la implementación de aplicaciones de aprendizaje automático en dispositivos de poco tamaño y baja potencia, como los microcontroladores. Normalmente los dispositivos periféricos necesitan estar conectados a centro de datos para poder ejecutar aplicaciones de aprendizaje automático. Sin embargo, este método no es posible en muchos escenarios, como en la falta de conectividad. Este estudio investiga las herramientas y técnicas utilizadas en TinyML, las limitaciones en la utilización de dispositivos de baja potencia y la viabilidad de implementar aplicaciones avanzadas de aprendizaje automático en microcontroladores.

Se desarrollaron tres programas para comprobar la viabilidad de implementar aplicaciones de aprendizaje automático en microcontroladores. El primero, una aplicación capaz de reconocer un conjunto de palabras claves. El segundo, un programa capaz de entrenar un modelo de red neuronal en el microcontrolador siguiendo un enfoque de aprendizaje en línea. Y el tercero, un programa de aprendizaje federado capaz de entrenar un único modelo global con la agregación de modelos locales entrenados en múltiples microcontroladores. Los resultados muestran un óptimo rendimiento de las tres aplicaciones una vez desplegadas en los microcontroladores. El desarrollo de aplicaciones básicas de TinyML resulta sencillo una vez entendidos el proceso de aprendizaje automático. Sin embargo, el desarrollo de aplicaciones avanzadas es muy complejo, ya que requiere un profundo conocimiento tanto del aprendizaje automático como de los sistemas embebidos.

Estos resultados demuestran la viabilidad de implementar con éxito aplicaciones avanzadas de aprendizaje automático en microcontroladores, y por lo tanto, desvelan un futuro brillante para TinyML.

Palabras Clave: TinyML, aprendizaje automático, microcontrolador, aprendizaje federado

Contents

List of Figures

List of Abbreviations

ADC	**A**nalog to **D**igital **C**onverter
AI	**A**rtificial **I**ntelligence
CPU	**C**entral **P**rocessing **U**nit
FL	**F**ederated **L**earning
FPU	**F**loating **P**oint **U**nit
GC	**G**eneric **C**osts
GPU	**G**raphics **P**rocessing **U**nit
IC	**I**ntegrated **C**ircuit
IDE	**I**ntegrated **D**evelopment **E**nvironment
IID	**I**ndependent and **I**dentically **D**istributed
I/O	**I**nput / **O**utput
IoT	**I**nternet **o**f **T**hings
MCU	**M**icro **C**ontroller **U**nit
MFCC	**M**el **F**requency **C**epstral **C**oefficient
ML	**M**achine **L**earning
PCB	**P**rinted **C**ircuit **B**oard
PCT	**P**ersonnel **C**osts per **T**ask
RAM	**R**andom **A**ccess **M**emory
RAM	**R**ead **O**nly **M**emory
SOTA	**S**tate **o**f **t**he **A**rt
TPU	**T**ensor **P**rocessing **U**nit
TF	**T**ensor**F**low

1 Introduction and Context

1.1 Introduction

Machine learning applications often rely on cloud services offered by external companies. Devices (e.g., smartphones) running these applications have to transmit the data captured by their sensors (e.g., cameras, microphones) to data centers. This data is then processed by GPUs or TPUs [1], which can offer high computing power. The result of the machine learning algorithm is sent back to the device to continue the application workflow. Although this approach allows running high computing power applications on low-powered devices, it also has some disadvantages and requirements that cannot be met in all scenarios. For instance, having to send data from two separate locations can lead to excessive latency or, worse, can compromise data privacy (data leakage). In addition, the devices must be connected to the internet for the application to work. With these disadvantages it is no wonder that of the 5 petabytes of data produced each day by IoT devices, less than 1 % of this data is analyzed or used at all [1].

TinyML is a new field that aims to implement machine learning applications on microcontrollers capable of performing data analytics at extremely low power. Therefore, TinyML applications can run continuously for a long period of time only using battery power or energy harvesting. The devices running the TinyML application do not need to be connected to the Internet. There is no need to worry about the privacy, as the data is analysed on the device itself. Microcontrollers are the only option when the power supply is restricted, size is a constraint or budget is limited. However, the use of microcontrollers for machine learning applications will bring with it many challenges to overcome.

[1] A TPU (Tensor Processing Unit) is an AI accelerator integrated circuit developed by Google for training neural networks.

1.1.1 Context

This Bachelor's thesis is submitted in fulfillment of the requirements of the degree in Informatics Engineering specialising in Computing, coursed in the Barcelona School of Informatics of the Polytechnic University of Catalonia. The thesis is authored by Marc Monfort Grau and supervised by Felix Freitag and Roger Pueyo Centelles.

1.1.2 Concepts

TinyML lies at the intersection between machine learning and microcontrollers. Therefore, the reader should be familiar with the following concepts in order to understand the thesis correctly.

Microcontroller

A microcontroller is an integrated circuit (IC) device designed to govern a specific operation in an embedded system. It is important not to confuse a microcontroller with a microprocessor, as the latter is used in general-purpose computers. Microcontrollers typically include a processor (CPU), memory and input/output peripherals (I/O). Microcontrollers are ubiquitous and can be found in a wide range of devices (e.g., washing machines, cars, stove, etc.)[2].

Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) that, with the help of statistics and a lot of data, generates a model that is capable of identifying interesting patterns. The model can then be used to identify patterns in unseen data (in a process called inference) and make decisions based on this. There are many machine learning techniques (e.g., linear regression, SVM, decision trees, etc.), but for TinyML we will have a special interest in artificial neural networks.

There are two approaches to machine learning algorithms: supervised and unsupervised learning. In supervised learning, the data is labeled, so the machine knows what patterns to look for. In unsupervised learning, the data is not labelled and therefore the machine is responsible for identifying the patterns. Supervised learning has become more popular and tends to produce better results. However, in many

problems the data cannot be labeled, and the only way is to use unsupervised learning algorithms.

Artificial Neural Networks

Artificial neural networks are algorithms that are intended to mimic the functioning of the brain. Figure 1.1 shows a representation of a neural network. A neural network is composed of several layers, and each layer contains several nodes representing neurons. The connections between neurons are represented by edges. Each neuron applies a mathematical function (e.g., linear regression) to the input values, and return an output value that will be propagated to the connected neurons. But before sending the output value, the neuron applies a function called the *activation function* (e.g., the sigmoid). The activation function is often used to make the system non-linear. With a non-linear system, the neural network can identify much more complex patterns in data.

Gradient descent is an optimization algorithm used to train neural networks. First, the algorithm attempts to identify the correct pattern from a labeled dataset (the training dataset). Then, the error is backpropagated from the output to the input layers. During backpropagation, the parameters of each neuron (the model's parameters) are modified to reduce the error. After many training iterations, the neural network can identify patterns from the training dataset (and hopefully, from unseen data) with very low error.

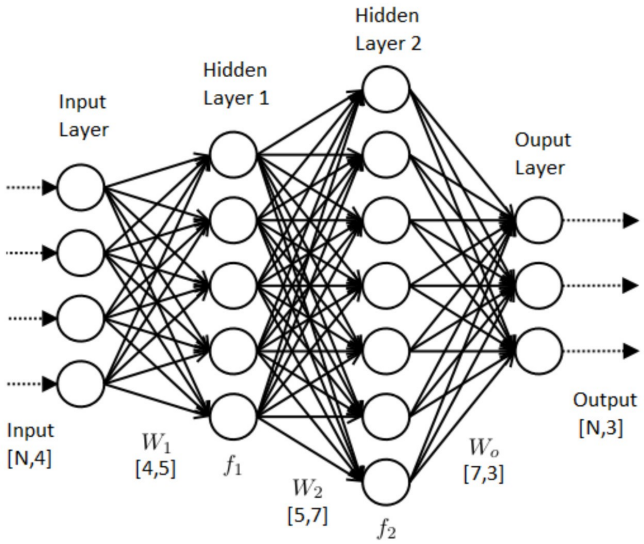

FIGURE 1.1: Diagram of an artificial neural network (source: Data Science Central).

1.1.3 Problem Definition

As mentioned above, TinyML tries to implement machine learning applications on microcontrollers. Therefore, the challenge is how to adapt machine learning algorithms to be used by low-powered devices. We also aim to discover the various scenarios where TinyML applications could be deployed. Below we describe some constraints of microcontrollers that could hinder their use of TinyML applications. And then we mention the trend of machine learning and what problems may arise.

Microcontroller hardware

Every computing system consists of three fundamental building blocks: computation (CPU), memory and storage. Table 1.1 shows the differences between microcontrollers and general-purpose computers. There are several orders of magnitude between the two platforms. We have the challenge of adapting the machine learning algorithms to these extreme features.

TABLE 1.1: PC vs. Microcontroller (source: EDX)

Component	PC	Microcontroller
Compute	$1GHz - 4GHz$	$1MHz - 400MHz$
Memory	$512MB - 64GB$	$2KB - 512KB$
Storage	$64GB - 4TB$	$32KB - 2MB$
Power	$30W - 100W$	$150\mu W - 23.5mW$

Microcontroller software

In general-purpose computers there are three levels of abstraction: the high-level application, the libraries that provide support for those applications and the operating system that provides support for the libraries and the applications. This architecture allows a lot of flexibility. However, microcontrollers are not general-purpose systems. Microcontrollers are typically designed to perform one task, and therefore, they usually do not have any operating system. Many common libraries may not work on their standard version and may have to be adapted. This supposes a loss in portability. We cannot be sure that the same code will work in different microcontroller devices, since they may not have the same components. Therefore, the second challenge we face is to enable TinyML applications across different microcontrollers.

Machine learning trends

The following two figures show the evolution of machine learning. Figure 1.2 shows the increase in size of machine learning models over the last few years. State-of-the-art (SOTA) models have many times more parameters than old models. This trend makes it more difficult to fit the latest models into memory restricted devices. As well as the size, the computing power needed to train the models is also increasing. Therefore, from the machine learning perspective, we have the challenge to shrink the size of the model and, at the same time, improve the training performance.

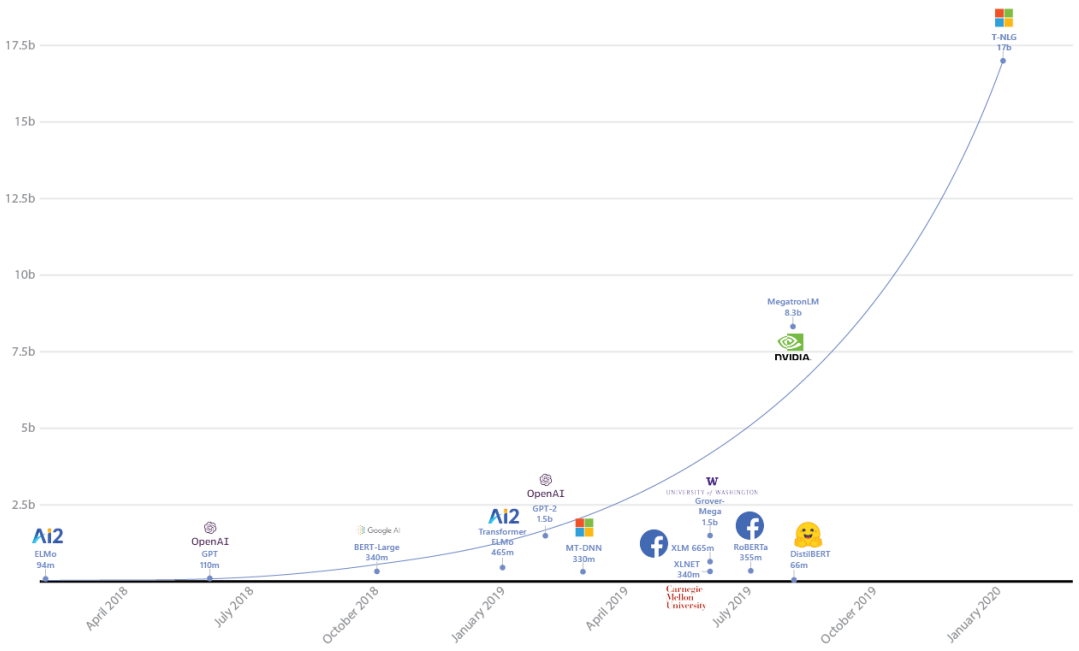

FIGURE 1.2: Model growth over the years (Source: DistilBERT).

1.1.4 Stakeholders

- **Research team**: With the development of this thesis, the research team will become an expert in the field of TinyML, which it is expected to be decisive for the future of machine learning.
- **Companies**: The bright future of TinyML will impact very positively to companies that invest and research about the field. An early investment will strengthen a dominant position. This thesis will show some of the most successful TinyML applications in the market.
- **Cientific community**: We hope that this thesis will inspire new lines of research about TinyML, or the development of innovative applications.

1.2 Justification

As mentioned above, machine learning applications deployed in edge devices (e.g., smartphones, embedded systems) usually require a constant connection to data centers. This approach is not suitable for many scenarios in which the edge devices do not have connectivity or enough energy supply [2]. Therefore, it is justified the need

[2]The transmission of data usually requires a much higher amount of power than computation.

for new strategies to allow the deployment of machine learning applications in these restricted scenarios.

A possible strategy could be the deployment of a general-purpose computer closer to the edge devices. The computer could act as a small data center. This approach would considerably reduce the distance, and therefore the communication cost. However, the edge devices would still have to send all the raw data from the sensors to the computer. Therefore, the network would, most probably, become the bottleneck of this approach. To solve it, a smarter strategy would be to only send the interesting data captured in the sensors. Yet, the edge devices should be able to classify between interesting and non-interesting data. With TinyML, we completely forget about transmitting the data, and we only focus on how to process the data in the same edge device.

TinyML is a recent field of machine learning. Therefore, the documentation is very limited. However, many machine learning techniques have been deeply studied, and there is a lot of documentation that can be used in our project. Yet, we need to adapt these concepts to the constraints of microcontrollers.

1.3 Scope

This section will define the global objectives and sub-objectives, the functional requirements, and the potential obstacles and risks that carry out the development of the thesis.

1.3.1 Objectives and Sub-objectives

The thesis has two objectives. The **first objective** is to develop a basic TinyML application. This objective has been broken down into smaller sub-objectives:

- Identify the most common TinyML techniques.
- Identify the tools used for developing TinyML applications.
- Recognize the hardware and software constraints.
- Understand the steps for developing a basic TinyML application.

- Develop and deploy a custom TinyML application.

The **second objective** is to identify advanced research directions of TinyML, and try to develop some advanced applications. The scope of this objective will be highly influenced by the obstacles and the lack of documentation, since we may go beyond the state of the art. Again, we have broken down the objectives into more detailed sub-objectives to be completed sequentially.

- Develop a TinyML application to identify the feasibility of training a machine learning model on a microcontroller.
- Develop a federated learning application to identify the feasibility of training a single model between multiple microcontroller devices.
- Upgrade the previous federated learning application to use wireless communication (LoRa or WiFi) between the edge devices in a distributed orchestration.

1.3.2 Potential Obstacles and Risks

Every project involves a series of potential obstacles and risks that must be taken into account. In this thesis, we have considered that the greatest obstacle is the time available. The field of TinyML comprises two very different disciplines such as machine learning and microcontrollers. It is therefore required to be, not only software expertise, but also embedded-hardware expertise. The lack of time can compromise the quality and the scope of the work, and we may fail to complete all the sub-objectives defined above.

A second major obstacle that we may face, is the non-compatibility between the software (libraries and frameworks) used for machine learning and the hardware (microcontrollers boards and sensors) available for this project. We must carefully analyze all the components that are going to be used.

Lastly, we should be very careful when developing the TinyML application. Even tough modern Integrated Development Environments (IDE) have very sophisticated debugging functionalities, when developing for microcontrollers it is completely different. Microcontroller IDEs very rarely are able to debug the code due to the nature

of deploying the application into a separate hardware (the microcontroller board). Therefore, any minor mistake in the code can suppose a major delay in the schedule.

1.4 Methodology and Rigor

This section defines the working methodology, the monitoring tools and the validation methods. It is very important for any project to correctly define these aspects in order to optimize the work and avoid the obstacles.

1.4.1 Agile Methodology

The methodology is based on the Agile Manifesto. We will work in short iterations (around two weeks). Each iteration has six phases to be completed sequentially: planning, design, development, verification, review and deployment. At the end of each iteration, the result will be evaluated among the project team. Then, we will start the planning phase for the next iteration.

The objective of this methodology is to obtain results as soon as possible and to continuously improve the work on every part of the TinyML pipeline. Moreover, we want to avoid spending too much time developing one of the components (e.g., the data pre-processing), only to realise that we lack time for developing another component (e.g., the model training).

1.4.2 Monitoring Tools and Validation

To monitor and revise the progress of the report, we will use cloud services that allow remote collaboration. Specifically, we will use Google Drive for sharing files and Overleaf for writing the final version of the report. To maintain the application code we will use the Git version control, and to store the application, we will use a public Github repository. The tools have been chosen because they are very popular, reliable and easy to use.

For organizing the project is has been agreed to hold meeting every two weeks between the author, the director and the co-director of the thesis. These meetings will be held using the Jitsi platform.

2 Project Planning

2.1 Duration

The thesis starts on February 26, 2021, and ends on June 30, 2021, with a formal presentation. This period is equivalent to 125 days (including holidays). Considering four hours of daily dedication to the project (on average), this becomes into 500 hours. This is an approximation and can be affected by external factors.

2.2 Task Definition

The project tasks have been grouped into sections from *A* to *F*. Each section has several tasks, and each task may be split into subtasks. The requirements are inherited from section to task and from task to subtask. The section's estimated time, is the sum of the time of its task, and the task's estimated time is the sum of the time of its subtask. Table 2.1 shows a summary of the tasks with its dependencies and the required resources.

A - Project Management

This section includes the tasks of defining and organising the project. Estimated time of 76 hours.

A1 - Context and scope: Definition of the project scope in the context of its study. It includes the justification of the project and the methodology to be followed. Estimated time of 14 hours.

A2 - Project Planning: Time planning of the project. It includes the definition of each task, the resources needed and the risk management. Estimated time of 14 hours. It requires the A1 task to be completed.

A3 - Budget and Sustainability: Analysis of the economic dimensions and the sustainability of the project. Estimated time of 14 hours. It requires the A2 task to be completed.

A4 - Final Project Definition: Revision of A1, A2 and A3 tasks, and integration of these tasks into a single document for the final memory. Estimated time of 14 hours. It requires the A1, A2 and A3 tasks to be completed.

A5 - Meetings: Project team meetings to organize and discuss the progress. Estimated time of 20 hours. It requires access to Jitsi Meet platform.

B - Basic Applications

This section includes the first objective tasks. Estimated time of 146 hours.

B1 - Research (state of the art): Research on the TinyML state of the art. Estimated time of 50 hours.

B2 - Analysis of Techniques: Analysis of typical TinyML techniques. Estimated time of 20 hours. It requires the B1 task to be finished.

B2.1 - Keyword spotting: Analysis of keyword spotting technique. Estimated time of 6 hours.

B2.2 - Visual Wake Word: Analysis of visual wake word technique. Estimated time of 6 hours.

B2.3 - Anomaly Detection: Analysis of anomaly detection technique. Estimated time of 8 hours.

B3 - Learn Basic Tools: Study of the basic software tools used for developing TinyML applications. Estimated time of 12 hours. It requires the B1 task to be finished.

B3.1 - Python and Jupyter Notebooks: Study of Python and Jupyter Notebooks (taking into account the prior knowledge). Estimated time of 2 hours. It requires the Python 3 programming language and Jupyter Notebook environment.

B3.2 - Google Colab: Study of Google Colab (taking into account the prior knowledge). Estimated time of 2 hours. It requires access to the Google Colab environment.

B3.3 - TensorFlow and Keras: Study of TensorFlow and Keras framework. Estimated time of 8 hours. It requires access to TensorFlow library.

B4 - Analysis of Microcontrollers: Analysis of microcontrollers and the components needed to deploy TinyML applications. Estimated time of 8 hours. It requires the B1 task to be finished.

B4.1 - Microcontrollers: Analysis of microcontrollers characteristics, components, restrictions and advantages. Estimated time of 4 hours. It requires access to a microcontroller.

B4.2 - Development Environments: Analysis of development environments, i.e. its file structure, how to interact with microcontrollers and how to deploy applications. Estimated time of 4 hours. It requires the B4.1 task to be completed and access to a microcontroller and a development environment (IDE).

B5 - Analysis of TinyML Workflow: Analysis of TinyML workflow in order to deploy a TinyML application. Estimated time of 16 hours. It requires the B2, B3, and B4 task to be finished.

B5.1 - Machine Learning Workflow: Analysis of machine learning workflow in order to generate a model. Estimated time of 6 hours.

B5.2 - Model Conversion: Analysis of model conversion techniques in order to reduce its size and be ready to be deployed in a microcontroller. Estimated time of 5 hours.

B5.3 - Model Deployment: Analysis of the model deployment stage with the use of development environments. Estimated time of 5 hours.

B6 - Application Development: Development of a TinyML application in order to validate the workflow analyzed in B5 task. Estimated time of 40 hours.

It requires the B5 task to be finished and access to a development environment and a microcontroller with sensors.

B6.1 - Definition and Design: Definition and design of the application to be developed. Estimated time of 6 hours.

B6.2 - Collect and Process Data: Collection and processing of data in order to have a dataset ready for training the model. Estimated time of 10 hours. It requires the task B6.1 to be finished.

B6.3 - Design and Train Model: Designing and training the neural network model. Estimated time of 10 hours. It requires the B6.2 task to be finished.

B6.4 - Model Evaluation and Optimization: Evaluation of the trained model, and possible optimizations for size reduction and performance. Estimated time of 8 hours. It requires the B6.3 task to be finished.

B6.5 - Deployment on Microcontroller: Deployment of the model to the microcontroller, and validation of the application. Estimated time of 6 hours. It requires the B6.4 task to be finished.

C - Advanced Applications

This section includes the second objective tasks. Estimated time of 160 hours. It requires the B tasks to be finished successfully.

C1 - Research (Beyond State of the Art): Identify advanced advanced research directions of TinyML. Estimated time of 40 hours.

C2 - Training on device: Development of an application to train a model inside a microcontroller. Estimated time of 40 hours. It requires the C1 task to be finished.

C3 - Federated Learning: Development of a federated learning application with microcontrollers. Estimated time of 40 hours. It requires the C2 task to be finished successfully.

C4 - Decentralized Federated Learning: Development of a decentralized federated learning application. Estimated time of 40 hours. It requires the C3 task to be finished successfully.

D - Project Documentation

This section includes the project documentation tasks. Estimated time of 60 hours.

D1 - Memory Draft: Drafting of each chapter to be included in the memory. Estimated time of 30 hours.

D2 - Memory Revision: Revision of the drafted chapters from section D1. Estimated time of 32 hours. It requires the task D1 to be finished.

D3 - Final Memory: Writing the final memory to be submitted to the jury. Estimated time of 28 hours. It requires the task D2 to be finished and access to Overleaf.

E - Project Defense

This section includes the task to be completed before the project defense. Estimated time of 20 hours. It requires the D tasks to be finished.

E1 - Prepare Slides: Preparation of the slides to be used in the project defense. Estimated time of 12 hours.

E2 - Rehearse Presentation: Rehearsing the project defense presentation. Estimated time of 8 hours. It requires the E1 task to be finished.

F - Post-Mortem Analysis

This section includes a retrospective analysis of the project. This is intended to improve for later projects, and should be done after the jury evaluation. Estimated time of 2 hours (outside the deadline). It requires the presence of the author, the director and the co-director of this project.

2.3 Resources

Human resources

PM - Project Manager: Personnel in charge of organizing the project to meet the content and deadlines. This includes the author, the director and the co-director of the thesis.

RE - Researcher: Personnel in charge of the research tasks. This includes the author of the thesis.

PR - Programmer: Personnel in charge of the development of the TinyML applications. This includes the author of the thesis.

ED - Editor: Personnel in charge of the project documentation. This includes the author of the thesis.

Material resources

Hardware

PC - Personal Computer: General-purpose computer for researching, developing and documenting the project.

uC - Microcontroller: A microcontroller device used for deploying TinyML applications. It will be used an Arudino Nano BLE Sense and a ESP32 with different configurations.

ES - Electronic Sensors: Sensors to be used by the microcontroller for gathering data (e.g., microphone, camera, accelerometer, etc.).

Software

JM - Jitsi Meet: Video-communication platform used for meetings.

GD - Google Drive: File storage and synchronization service used to share files.

OV - Overleaf: Collaborative cloud-based LaTeX editor for documenting the project.

GC - Google Collaboratory: Jupyter Notebook environment executed in the cloud. Will be used for machine learning development.

TF - TensorFlow: Open-source software library for machine learning.

IDE - Microcontrollers IDE: Integrated development environment for microcontrollers.

TABLE 2.1: Project tasks and resources (own creation)

Id.	Task	Time(h)	Dependencies	Resources
A	**Project Management**	76		PM, PC
A1	**Context and Scope**	14		
A2	**Project Planning**	14	A1	
A3	**Budget and Sustainability**	14	A2	
A4	**Final Project Definition**	14	A3	
A5	**Meetings**	20		PE, PR
B	**Basic Applications**	146		PC
B1	**Research (State of the art)**	50		RE
B2	**Analysis of Techniques**	20	B1	RE
B2.1	Keyword Spotting	6		
B2.2	Visual Wake Word	6		
B2.3	Anomaly Detection	8		
B3	**Learn Basic Tools**	12	B1	RE
B3.1	Python and Jupyther Notebooks	2		
B3.2	Google Colab	2	B3.1	GC
B3.3	TensorFlow and Keras.	8	B3.2	GC, TF
B4	**Analysis of Microcontrollers**	8	B1	RE
B4.1	Microcontrollers	4		uC
B4.2	Development Environments	4	B4.1	uC, IDE
B5	**Analysis of TinyML Workflow**	16	B2, B3, B4	RE
B5.1	Machine Learning Workflow	6		
B5.2	Model Conversion	5	B5.1	
B5.3	Model Deployment	5	B5.2	
B6	**Application Development**	40	B5	PR
B6.1	Definition and Design	6		
B6.2	Collect and Process Data	10	B6.1	GC
B6.3	Design and Train Model	10	B6.2	GC, TF
B6.4	Model Evaluation and Optimization	8	B6.3	GC, TF
B6.5	Deployment on Microcontroller	6	B6.4	uC, IDE, ES
C	**Advanced Applications**	160	B	PC, uC, IDE
C1	**Research (Beyond state of the art)**	40		RE
C2	**Training on Device**	40	C1	PR
C3	**Federated Learning**	40	C2	PR
C4	**Decentralized Federated Learning**	40	C3	PR
D	**Project Documentation**	90		ED, PC
D1	**Memory Draft**	30		
D2	**Memory Revision**	32	D1	
D3	**Final Document**	28	D2	OV
E	**Project Defense**	20	D	ED, PC
E1	**Prepare Slides**	12		OV
E2	**Rehearse Presentation**	8	E1	
F	**Post-Mortem Analysis**	2	E	PM,RE,PR,ED
Total		494		

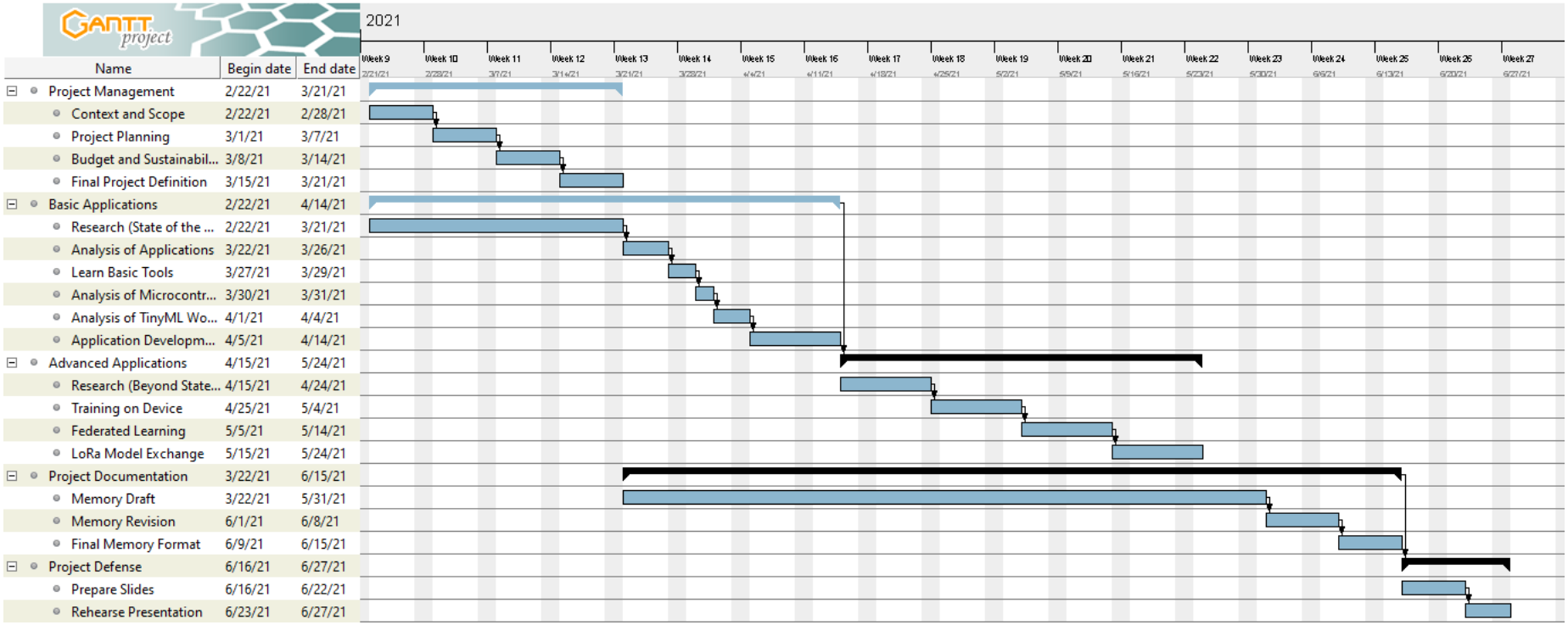

FIGURE 2.1: Gantt chart.

2.4 Risk Management: Alternative Plans

As mentioned in the Section 1.3.2, the development of the thesis entails several obstacles and risks. Below we have defined some possible solutions and alternative plans that will be followed in the case of having to face any of these obstacles.

Deadline (high risk)

The deadline is the most important risks we face. To avoid underestimating the time of each task, we have added some extra time for unforeseen events. However, if this extra time is not enough, we have the possibility, as a last resort, to extend the deadline until October, 2021 (4 extra months).

Scope (high risk)

The scope of the project is likely to be affected by the results of section C. Since the task are strongly dependent, a negative result in one of the tasks would produce the inability to perform any of the subsequent tasks. This risk is high since the second objective is beyond the *state of the art* and may be unfeasible. This issue is already foreseen by the team, and it has been agreed that section C will have the scope that can be reached within the deadline.

Debug (medium risk)

The complexity of developing applications on microcontrollers does not facilitate debugging tasks. Any error can mean a significant delay. For this reason, we have made a loose estimation in the more technical tasks. Also, we will use the IDE with the maximum amount of debugging functionalities for microcontrollers.

3 Budget

This section analyzes the economic costs involved in carrying out this thesis. It includes the personnel costs per activity, the generic costs, the contingency and the incidental costs.

3.1 Personnel Costs per Task (PCT)

First, we will analyze the cost per hour of the human resources required by the project. We distinguish between the roles already mentioned in Section 2.3. Table 3.1 shows the gross hourly salary of each role, the social security cost (calculated as 0.3 of the gross salary), and the final salary as the sum of these two. Table 3.2 shows the cost of each task based on the amount of hours and the salary of the personnel in charge.

TABLE 3.1: Personnel salary (own creation)

Role	Gross salary(€)/h	Social security	Salary(€)/h
Project Manager	30 [5]	9	39
Researcher	20 [6]	6	26
Programmer	15 [7]	4.5	19.5
Editor	12	3.6	15.6

TABLE 3.2: Personnel cost per task (own creation)

Id.	Task	Time(h)	Role	Cost(€)
A	**Project Management**	76		4524
A1	**Context and Scope**	14	PM	546
A2	**Project Planning**	14	PM	546
A3	**Budget and Sustainability**	14	PM	546
A4	**Final Project Definition**	14	PM	546
A5	**Meetings**	20	3xPM	2340
B	**Basic Applications**	146		3536
B1	**Research (State of the art)**	50	RE	1300
B2	**Analysis of Techniques**	20		520
B2.1	Wake Word Detection	6	RE	156
B2.2	Visual Wake Word Detection	6	RE	156
B2.3	Anomaly Detection	8	RE	208
B3	**Learn Basic Tools**	12		312
B3.1	Python and Jupyther Notebooks	2	RE	52
B3.2	Google Colaboratory	2	RE	52
B3.3	TensorFlow and Keras.	8	RE	208
B4	**Analysis of Microcontrollers**	8		208
B4.1	Microcontrollers	4	RE	104
B4.2	Development Environments	4	RE	104
B5	**Analysis of TinyML Workflow**	16		416
B5.1	Deep Learning Workflow	6	RE	156
B5.2	Model Conversion	5	RE	130
B5.3	Model Deployment	5	RE	130
B6	**Application Development**	40		780
B6.1	Definition and Design	6	PR	117
B6.2	Collect and Process Data	10	PR	195
B6.3	Design and Train Model	10	PR	195
B6.4	Model Evaluation and Optimization	8	PR	156
B6.5	Deployment on Microcontroller	6	PR	117
C	**Advanced Applications**	160		3380
C1	**Research (Beyond state of the art)**	40	RE	1040
C2	**Training on Device**	40	PR	780
C3	**Federated Learning**	40	PR	780
C4	**LoRa Model Exchange**	40	PR	780
D	**Project Documentation**	90		1404
D1	**Memory Draft**	30	ED	468
D2	**Memory Revision**	32	ED	499.2
D3	**Final Document**	28	ED	436.8
E	**Project Defense**	20		316
E1	**Prepare Slides**	12	ED	187.2
E2	**Rehearse Presentation**	8	ED	128.8
F	**Post-Mortem Analysis**	2	3xPM	234
Total		494		13394

3.2 Generic Costs (GC)

Amortization

Table 3.3 shows the amortization of the hardware devices used in the project. To calculate the amortized cost, we have used the following formula:

$$cost\_per\_hour = resource\_price * \frac{1}{life\_expectancy} * \frac{1}{working\_days} * \frac{1}{working\_hours} \tag{3.1}$$

$$amortization = cost\_hour * used\_hours \tag{3.2}$$

The life expectancy of hardware devices is set to 10 years [8]. The working days in Catalonia in 2021 are 252, and each day has 8 working hours. The used hours of each device are obtained from the sum of the hours of each task requiring the device. The software used is free and does not suppose any cost.

TABLE 3.3: Hardware amortization (own creation)

Device	Price (€)	Used hours	Amortization cost (€)
Personal Computer	800	492	19.52
Periphereals	200	492	4.88
Arduino Nano 33 BLE Sense	27	174	0.23
2 x ESP32 LoRa	35	174	0.3
Camera sensor OV7675	4.88	174	0.04
Total			24.97

Indirect costs

The list below shows the indirect resources of the project and the parameters used to calculate the cost.

Work space: The work will be done remotely. Monthly rent of 400€. Total of 494 project hours.

$$Work\ space\ cost = 400 * \frac{1}{30\ days} * \frac{1}{24\ hours} * 494$$

Electricity: Average price about 0.1199 €/kWh. Average personal computer consume about 200 Wh. Total of 494 project hours.

$$Electricity\ cost = 0.1199 \times 10^{-3} * 200 * 494$$

Internet: Monthly cost of 50€. Average of 8 working hours. Total of 494 project hours.

$$Internet\ cost = 50 * \frac{1}{30\ days} * \frac{1}{8\ hours} * 494$$

Table 3.4 summarizes the indirect costs of the project.

TABLE 3.4: Indirect costs (own creation)

Resource	Cost (€)
Work space	274.44
Electricity	11.84
Internet	102.91
Total	389.19

3.3 Contingency

This project may have unforeseen events that result in cost overruns. The contingency cost is a percentage of the budget value that is added to it in order to cover the cost of unforeseen events that finally appear. We have decided to set a 15% of the sum of the PCT and GC as the contingency cost. This gives us 2071.224 €.

3.4 Incidental Costs

The obstacles that we may encounter while working on the project have already been described. To mitigate the cost of these obstacles we will add to the budget the percentage of the probability of encountering the obstacle over the estimated cost it would entail. Table 3.5 shows the incidental cost added to the budget for each obstacle.

TABLE 3.5: Incidental costs (own creation)

Incident	Estimated cost (€)	Risk (%)	Cost (€)
Deadline of the project (20 hours)	455 (footnoe)	40	182
Debugation time (10 hours)	195 (footnote)	50	97.5
Total			279.5

3.5 Final Budget

Table 3.6 shows the final budget as the sum of the personnel costs per task, the generic costs, the contingency costs and the incidental costs.

TABLE 3.6: Final budget (own creation)

Activity	Cost (€)
PCT	13394
Amortization	24.97
Indirect costs	389.19
Contingency	2071.22
Incidental costs	279.5
Total	16158.88

3.6 Management Control

It is likely that during the course of the project the costs will vary from those estimated in the previous sections. It is important to monitor these deviations and act accordingly.

The **personnel costs** deviation is calculated as follows. A delay in any task from the Gantt chart will increase the personnel cost. In the same way, if any task is finished before the estimated time, the cost will be reduced, and we could cover the cost for the delayed tasks.

$$Personnel\_deviation = (Estimated\_cost\_per\_hour - Real\_cost\_per\_hour) * Total\_hours$$

The **generic costs** deviation is calculated as follows. It applies the same task delay situation as the personnel costs deviation. If we have a delay in any task that requires a resource, the cost of using the resource will raise.

$$Generic\_costs\_deviation = (Estimated\_used\_hours - Real\_used\_hours) * Price\_per\_hour$$

The **contingency and incidental costs** deviation are calculated as follows. If we finally face more unforeseen events than the ones expected in the budget, the cost will increase, and therefore, the deviation. In the opposite case, where we face less unforeseen events than the ones expected, we could use the surplus to cover the loss in other sections.

$$Contingency\_deviation = (Estimated\_incidental\_hours - Real\_incidental\_hours) * Cost\_per\_hour$$

The **total deviation** is the sum of the partial deviations mentioned above.

$$Total\_deviation = Personnel\_deviation + Generic\_costs\_deviation + Contingency\_deviation$$

4 Tools and Techniques

4.1 TinyML Techniques

In this section we will describe the three most promising TinyML techniques. These techniques have been successfully applied in many applications that are commercialized in different products. These products can be found in domestic, office or industrial scenarios.

4.1.1 Keyword Spotting

Keyword spotting (KWS) is a speech recognition technique that deals with the identification of specific words from a short voice recording. Usually, the applications that are based on this technique are constantly capturing sounds from the environment in order to identify predefined keywords with a machine learning model. When the model recognizes a keyword, the application triggers a signal to wake up a more powerful system which will be responsible to perform a more complex task (e.g., recognizing a full sentence). In simpler applications, the keyword only triggers an actuator (e.g., turn on an LED, opening a door, etc.).

There are several commercial products that use a keyword spotting approach. For example, many smartphones come with Google's "Ok Google" or Apple's "Hey Siri". Both applications allow to send a query after saying a specific keyword, without touching the device. Another application is Amazons's "Alexa". A home virtual assistant with the same functionality. Also, it is common to find keyword spotting products for car voice assistant. These assistants allow the driver to interact with the multimedia system without distraction. It is expected that keyword spotting applications will be very important for the future of human-machine interfaces and IoT devices.

4.1.2 Visual Wake Word

Visual wake word technique is the extension of keyword spotting for images. The application pipeline is very similar: A camera is constantly capturing images from the environment in order to identify (with a machine learning model) whether a specific object or person appears in the image. Again, if the model recognizes the object or the person, it triggers a response (e.g., activating an alarm, turning on the lights), or a more complex system (e.g., face recognition).

There are several applications on the market using the visual wake word technique. For example, Bird Buddy have designed a smart bird feeder with a built-in camera that is able to recognize the presence of birds, and then send a notification to the user. Other popular applications are camera sensors that are able to detect the presence of persons in a room. The application could turn off the lights if no person is detected, or it could be used for security control. Again, the growth in IoT will only increase the use of visual wake word applications for automation and human-machine interaction.

4.1.3 Anomaly Detection

Anomaly detection is a technique used for identifying unexpected events. Unlike the previous techniques, anomaly detection follows an unsupervaised learning approach, where the model have to discover patterns on unlabelled data. The goal of this application is to discover anomalies which occur very rarely in the data. Although anomaly detection seems unrelated to TinyML, it is actually one of the most promising techniques for TinyML applications.

Anomaly detection applications are mainly deployed in the industry. They can be used for detecting malfunctions on factory machines. Tiny devices are attached to the machines and are constantly analyzing their sounds and vibrations in order to train a machine learning model (autoencoder). After the model is trained, if the device detects an anomaly on the sound, it can send a signal to warn about a possible malfunction. Then, the operator can repair the machine and prevent a shutdown of the entire factory production. Anomaly detection applications can also be used in other scenarios, like detecting vehicle engine failures or pipeline water leakage.

4.2 Tools and Frameworks

In this section we will analyze the tools and libraries used to develop a typical TinyML application.

Python

Python is and interpreted, high-level and general-purpose programming language, that has gained a lot of popularity for the development and research in machine learning. In addition to its simplicity, the biggest advantages that Python offers over other programming languages is the number of libraries and frameworks focused on machine learning (e.g., TensorFlow, Keras, PyTorch, etc.). With these libraries we can considerably reduce the development time. In this project we will use Python and TensorFlow for the development of basic TinyML applications.

Jupyther Notebooks

Jupyter Notebook is and interactive web application that combines software code, computational output, explanatory text and multimedia resources, all in a single document. This approach facilitates the collaboration between different members on a project with a common development environment. Python with Jupyter Notebooks have gained a lot of popularity for machine learning researchers.

Google Colaboratory

Google Colaboratory (or Google Colab) is an online web application based on Jupyter Notebook that allows high computing power using cloud computing. It runs using a Chrome web browser without any previous configuration. Moreover, it makes it even easier to share, and it allows to edit simultaneously a Jupyter Notebook file.

TensorFlow

TensorFlow is an open source machine learning framework developed by Google. It has a Python front-end with a highly optimized C++ code at the core. It is both used in the industry and academia, having a large developer community. It is mainly

designed to facilitate the building of machine learning models. It has many functionalities for data pre-processing, data ingestion, model evaluation, visualization and serving.

TensorFlow is designed to be highly portable being able to run on a variety of devices and platforms. However, the standard version of Tensorflow has a size of 400 MB, but our microcontrollers only have around 1MB. Therefore, it is not possible to deploy the standard Tensorflow framework into these tiny devices. Luckily, there is a smaller version of the framework called TensorFlow Lite, which is specifically designed for more restrictive devices.

TensorFlow Lite. In TinyML we work on very constraint devices with limited computing power, memory and storage. However, the basic TinyML applications usually do not have to perform the whole machine learning pipeline, neither need all the advanced techniques used by researchers. Therefore, many functionalities of the TensorFlow are not required. Only a small subset will be used. This is exactly what TensorFlow Lite is about.

TensorFlow Lite (TFLite) is and optimized subset of TensorFlow designed to run models on mobile, embedded and IoT devices. It enables on-device inference with low latency and small binary size (about 1MB) [9]. It is achieved by removing the superfluous functionalities that are unnecessary for mobile deployment. The workflow consists in first training a model using the standard TensorFlow framework, and then converting this model to a much smaller format suitable for edge devices, using optimizations like pruning or quantization. Then, the compiled model can be deployed and used by TFLite for inference. However, the 1MB size of TFLite still will not fit in our microcontroller flash memory. An even smaller version of TensorFlow has been created for this extreme scenario.

TensorFlow Lite Micro. TensorFlow Lite Micro is the state-of-the-art inference framework from Google. It is designed to run machine learning on microcontrollers and other devices with only a few kilobytes of memory. It takes the compression of the standard framework to the extreme, removing all but the essential functionalities. The core runtime just fits in 16 KB. It does not require any operating system support

(can run on bare metal), any standard C or C++ library, nor dynamic memory allocation. On the downside, it is harder to troubleshoot any issue since the framework does not support any plotting or debugging tool. TensorFlow Lite Micro runs on a wide variety of embedded microcontrollers. The exact list of supported devices can be found on the official page.

Other Embedded ML Frameworks

We decided to use TensorFlow Lite Micro framework for our TinyML applications because it is free, open-source and very well documented. However, it is important to notice that there are several alternatives:

- Apache TVM: Open source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators.
- uTensor: Free and open source embedded machine learning infrastructure designed for rapid-prototyping and deployment.
- GLOW: Machine learning compiler engine and execution engine for hardware accelerators. It is designed to be used as a back-end for high-level machine learning frameworks.

5 Embedded Systems

5.1 Microcontroller Boards

A microcontroller board refers to a microcontroller built onto a printed circuit board (PCB), which provides all of the circuitry necessary for a useful control task (e.g., I/O circuits, clock generator, RAM, etc.) [10]. We can distinguish between two types of microcontroller board; the commercial boards designed to perform one particular task, and the developer boards designed for a general purpose. For this project we will use a developer board, since it allows more flexibility for testing and developing different kinds of applications, and we can avoid having to design our own PCB.

We have available an Arduino Nano 33 BLE Sense, and an Espressif ESP32 board. Both boards have enough computing power and memory capacity for basic TinyML applications, and both are supported by the TensorFlow Lite Micro framework. Table 5.1 compares the hardware characteristics of each board.

TABLE 5.1: Board specification

Board	MCU	Clock Speed	Flash Memory	SRAM	Cost
Arduino Nano 33 BLE Sense	32-bit nRF5284	64 MHz	1 MB	384 kB	€27.0
ESpressif ESP32	32-bit ESP32-PICO-D4	240 MHz	4 MB	520 kB	€13.0

The above table shows that the Espressif board is cheaper, has more clock speed, more memory and more storage than the Arduino board. However, the Arduino PCB offers many integrated sensors that can be very useful for TinyML applications.

5.2 Sensors

Sensors are basic components for any TinyML application. In TinyML we try to bring computing power closer to the edge devices where the data is collected. Sensors are required to obtain the data that will later be used by the machine learning pipeline. Therefore, the performance and feasibility of any TinyML application strongly depend on the kind of sensor and the quality of the data captured.

The Arduino Nano 33 BLE Sensen board already comes with several sensors integrated on the board. It has a nine axis inertial sensor, a humidity and temperature sensor, a barometric sensor, a microphone, and a light sensor. We also have available an external image sensor that could be plugged into the board. This high variety of sensors allow us to develop a wide range of TinyML applications. However, it is hard to find existing datasets for some of the listed sensors, and therefore, we will focus on the most common (microphone).

5.3 Development Environments

An Integrated Development Environment (IDE) is an application with a set of features that simplifies the development of software for a general or specific purpose. TinyML is about developing applications in microcontrollers. Therefore, we need a specific IDE for these devices. There are several IDEs for embedded software devices. Some of the most popular IDEs are Arduino IDE, Keli uVision, PlatformIO and MPLAB X. For this project we will use both the Arduino IDE and PlatformIO, since they are free, very complete and easy to use.

Arduino IDE

The Arduino IDE is a light-weight application that minimizes functionalities in return for simplicity. The Arduino's mission is to create easy-to-use hardware and software. This implies a tradeoff that limits the features of the development environment to the essentials. However, the Arduino IDE has all the basic features necessary to deploy a program into a microcontroller and to monitor the running

application. Besides the standard version, there is a cloud-based (Create Web Editor) and a professional version (Arduino Pro IDE). For this project we will use the standard Arduino Desktop IDE only for prototyping and fast deploying.

PlatformIO

PlatformIO is a multi-framework professional development environment for embedded applications built on top of Microsoft's Visual Studio Code. PlatformIO supports more than 1000 microcontroller boards, including the ones we have for this project (Arduino Nano 33 and ESP32). Moreover, Visual Studio Code has many extensions that allow features like code completion, reference tracking, debugging, etc. It is very comfortable to develop embedded applications and share them using PlatformIO, since it has its own library manager. Therefore, it will be our preferred development environment for this project.

6 Basic Keyword Spotting Application

In this chapter, we will develop a basic keyword spotting application. The application will be able to recognize a custom set of keywords using the Arduino Nano 33 BLE Sense board. The application will be developed following two different approaches. First, using the TensorFlow framework, both for training and calling a neural network model. Second, using the Edge Impulse web application. Finally, we will evaluate the benefits of using one approach over the other.

6.1 First approach: TensorFlow

The first approach will be based on the Micro Speech example from the TensorFlow Lite Micro framework. This example provides a pre-trained keyword spotting model that can recognize two keywords, "yes" and "no", from audio samples. The application is listening to the environment with a embedded microphone. If any of the keywords is recognized, the application turns on a LED. In the example, a green LED means that the application has recognized the "yes" keyword, and a red LED means that the application has recognized the "no" keyword.

In the following sections we will go through all the steps necessary for developing a TinyML keyword spotting application. Some of these steps are common in other techniques (e.g., visual wake word, anomaly detection). We want the application to be able to recognize the words *vermell*, *verd* and *blau* (red, green and blue in Catalan language). However, for creating our custom application, we will not develop all the parts (that are explained below), but rather modify the necessary parts in order to adapt the example to our set of keywords.

6.1.1 Data Collection

The first step is to collect all the data that will be used to train the machine learning model. The collection stage is often the most challenging and time-consuming, unless an existing dataset is available. The dataset must be large and specific, as the feasibility of our application would be highly determined by the quality and size of the dataset. A bad dataset would result in a poor performance of the machine learning model, which is the core component of the application.

Although there are some known dataset for keyword spotting, we want the application to recognize our custom set of keywords. Therefore, we will not reuse any dataset, but create our own. In order to do so, we will record several samples of each of the three keywords (*vermell*, *verd* and *blau*), using a voice recorder application. The process of creating the dataset is cumbersome, since we have to save every recorded word in a single file. This can suppose a lot of time of audio editing. Our final dataset has around 100 samples (of one second) for each keyword. Although typical datasets often have thousands (or even millions) of samples, we have neither the time nor the resources to obtain that many. Anyway, a keyword spotting model is usually very small compared to typical machine learning models. Therefore, with a small dataset we can still obtain good performance.

For commercial applications, it is very important to keep in mind the requirements for a good dataset. Before starting the process of collecting the keyword spotting data, we should determine who might be the end users (e.g., language, accent, slang, etc.), and in which environment will be used the application (e.g., noisy place, crowded street, quite room, etc.). For our demo application, we will create a simple dataset with only the author's voice in a quiet room. Therefore, it is not expected that the application will have the same accuracy with other users or in other scenarios.

6.1.2 Data Processing

After obtaining enough samples for the dataset, we have to process this data in order to extract more relevant features. This process is known as feature engineering.

Another possibility is to simply leave the input data unperturbed. However, preprocessing the data usually gives us better performance and reduces the size of the model.

The type of data obtained for our keyword spotting dataset is in the form of digital audio signals. To obtain this digital representation, the microphone has to capture the vibrations from the sound waves, and then convert these vibrations to electrical signals in the form of voltage distortions. Finally, the electrical signals are converted into digital signals using an analog-to-digital converter (ADC). The number of data points captured each second depends on the sample rate used by the microphone. For example, a typical 16 kHz sample rate captures 16000 data points each second, which may be too much data for only one sample. Figure 6.1 shows the digital representation of the *vermell*, *verd* and *blau* words.

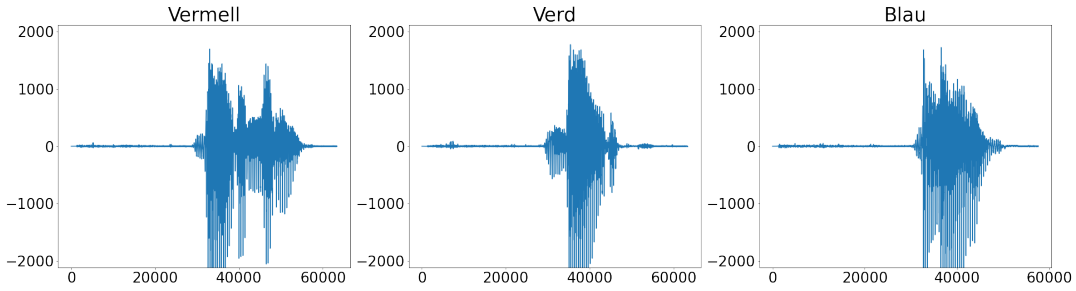

FIGURE 6.1: Digital representation of *vermell*, *verd* and *blau* keywords

To obtain the feature vector from an audio signal, we have to perform several steps. The first step is to align the critical part of the signal that contains the spoken word. As it is seen in Figure 6.1, the start and the end of the audio has no sound, and therefore it should be removed. The simplest way is to extract the loudest second from the audio sample [11].

The second step is to extract the unique features of each sound in the word. The audio signals can be decomposed into primitive signals of fundamental frequencies. Using the Fourier transform we can decompose the audio and obtain the frequency domain representation of each word. Figure 6.2 shows the frequencies that are present in the three keywords (*vermell*, *verd* and *blau*). By extracting out the critical frequencies, we are obtaining the fingerprint of each word. However, to generate the frequency spectrogram of a word, we have to apply the Fourier transform along

all the word sound using short sliding windows (from 20 to 30 milliseconds). This way, we will obtain the time domain spectrogram of the frequencies of each word, as seen in Figure 6.3.

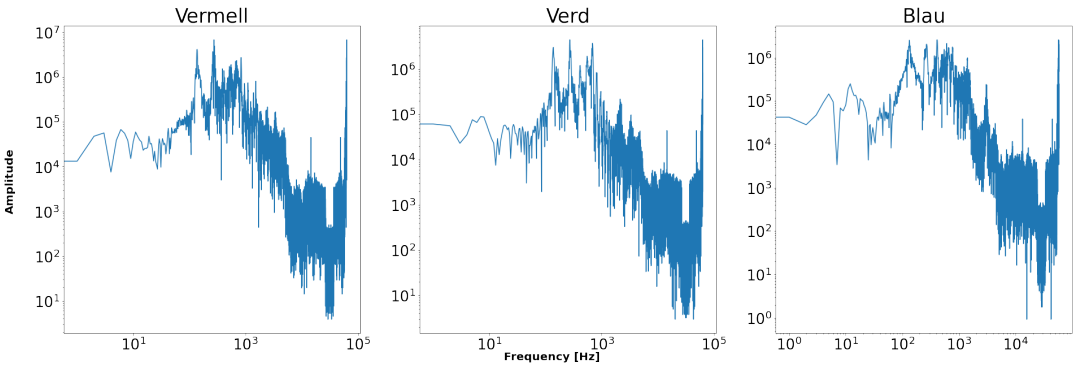

FIGURE 6.2: Fourier transform of *vermell*, *verd* and *blau* keywords

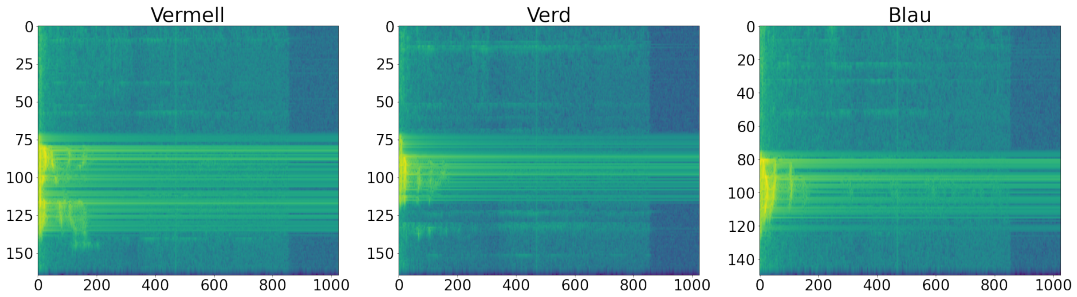

FIGURE 6.3: Spectogram of *vermell*, *verd* and *blau* keywords

The last step of the data processing stage is to obtain the Mel Frequency Cepstral Coefficient (MFCC) from the spectrogram (Figure 6.3). This technique is based on the phenomenon that the human ear can hear the low frequencies easier than the high frequencies according to the Mel Scale. Therefore, we are able to extract more salient features from the higher frequency signals using the Mel Filter Bank (Figure 6.4). The result of the MFCC is shown in Figure 6.5. The difference between the MFCC and the previous spectrogram is that with the MFCC we take into account the human audio perception.

After applying all the processing steps, we converted the audio time signal into a frequency signal in the form of an image (Figure 6.5). This image is the feature vector that will be used to feed the neural network model. We could further process

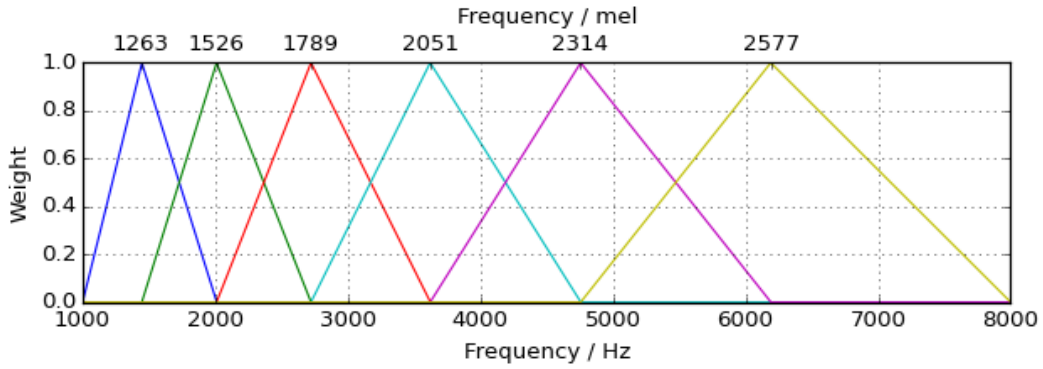

FIGURE 6.4: Mel Filter Bank (PyFilterbank)

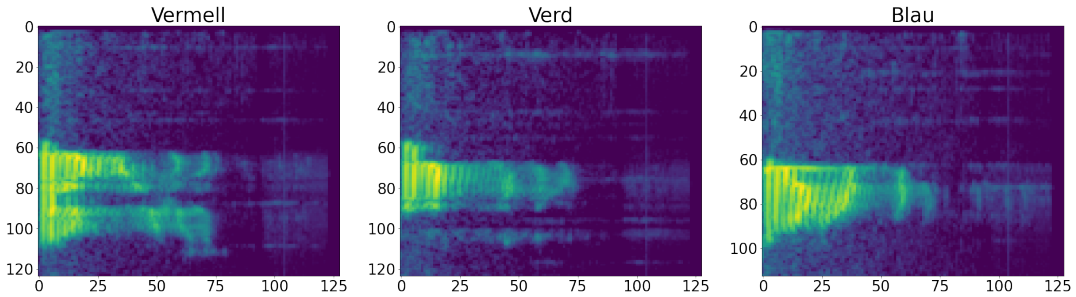

FIGURE 6.5: MFCC of *vermell*, *verd* and *blau* keywords

the audio signal (e.g., signal cleaning, amplitude normalization, etc.), however, with this feature vector we can already get good performance on the application.

6.1.3 Model Design

Once our dataset is processed, we can start designing the machine learning model. Usually, models are selected based only on their performance, however, other metrics may also be important, such as simplicity. In this application, we will use a neural network model. This network needs to be very small in order to be deployed into our microcontroller.

Convolutional neural networks are very popular for images, and since the type of data of the feature vector is in the form of images (Figure 6.5), the convolutional neural networks would be a perfect fit. The model used in the TensorFlow example, and the one we are going to use in our application, is called **TinyConv**. This convolutional model is composed of a 2D convolutional layer, followed by a single dense

layer with a softmax activation function. The model architecture can be seen in Figure 6.6. The model has a size of 16652 parameters (calculated below). Therefore, the model could perfectly fit into the Arduino Nano 33, which has 1MB of flash memory and 256 kB of RAM. Although the model is very small, it is possible to get good performance because TinyML applications (like keyword spotting) are domain-specific. Therefore, we only need a simple model that performs very good on a very specific task.

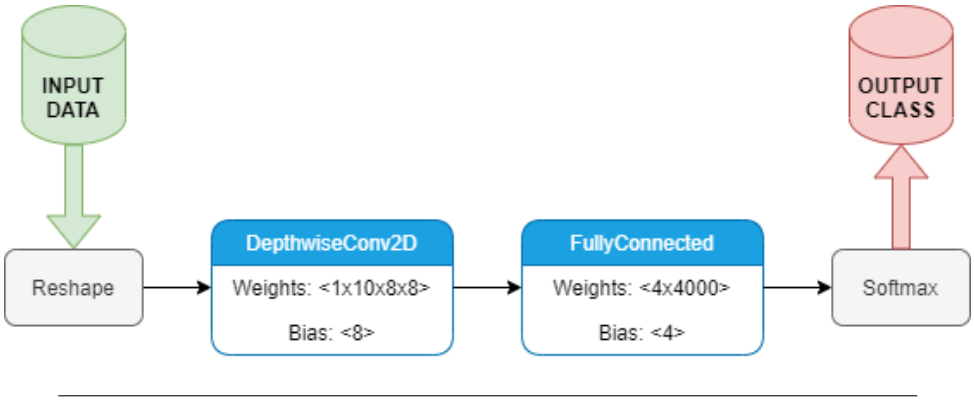

FIGURE 6.6: TinyConv diagram (own creation)

$$DepthwiseConv2D : 10 * 8 * 8 + 8 = 648 \; parameters \tag{6.1}$$

$$FullyConnected : 4 * 4000 + 4 = 16004 \; parameters \tag{6.2}$$

$$TinyConv : 648 + 16004 = 16652 \; parameters \tag{6.3}$$

6.1.4 Model Training

With the model already defined, the next step is to train it using our dataset. The training process consists in tuning the model's parameters (weights and biases) in order to learn associations that could find patterns on a particular dataset. The Python script we used to train our custom keyword spotting model is based on the Google Colab template from HarvardX's Tiny Machine Learning course. At the same time, this script is using the TensorFlow train.py script to perform the model

training. With this script, we are basically training the TinyConv model using the gradient descent algorithm for 12000 training steps with a learning rate of 0.001, and 3000 final steps with a learning rate of 0.0001. This way, in the last iterations, the model is slowly approximating to the local minimum, and not missing it. The used batch size is 100, which represents the number of data samples to train with at once. The time it takes to train the model using Google's Colab GPUs is around two hours. The accuracy obtained is 0.957. This result could be optimized by fine-tunning the training hyperparameters.

6.1.5 Model Conversion

The last step before deploying the model in the microcontroller is to convert it to an appropriate format. As we already mentioned, the TinyConv model has 16652 parameters. If these parameters are represented using floats (4 bytes), the model will occupy 66.6 kB. But if we represent the model using unsigned integers (1 byte) the size will be reduced to 16.6 kB. To obtain such a small size, we have to quantize the model. This technique allows our model to run using only 8-bit, which is available in most embedded systems.

Neural networks are able to find patterns in data, despite the noise (e.g., a blur picture, lighting changes). When the precision of the model is reduced from four to one bytes, we are just adding a bit more noise that needs to be filtered out. However, the model can still produce good results. With quantization we can reduce the model size and improve the inference latency, with a small loss in accuracy. Moreover, the consuming power is also reduced, which is very important for edge devices that run on batteries.

There are two forms of quantization: **post-training quantization** and **quantization aware training**. Post-training quantization is easier to use. It works by reducing the range of the parameters to an interval between 0 and 255. For example, if the minimum value of a model is -2, and the maximum value is 7, after the post-training quantization (8-bit), a value of 0 would represent a -2, a value of 255 would represent a 7 and a value of 128 would represent a 2.51.

In quantization aware training the model is quantized during the training phase in order to reduce the errors introduced by the quantization. The weights are adjusted considering the quantization noise. However, we will only perform post-training quantization, since it adds less complexity to the training phase.

Pruning is another technique to reduce the size of the model. This technique consists in removing the weights with the lowest values. These weights do not contribute much to the final model performance. The result after pruning is a model with the same architecture, but with fewer parameters (sparser) [12]. Figure 6.7 shows a representation of a model after being pruned. This technique is done automatically by TensorFlow.

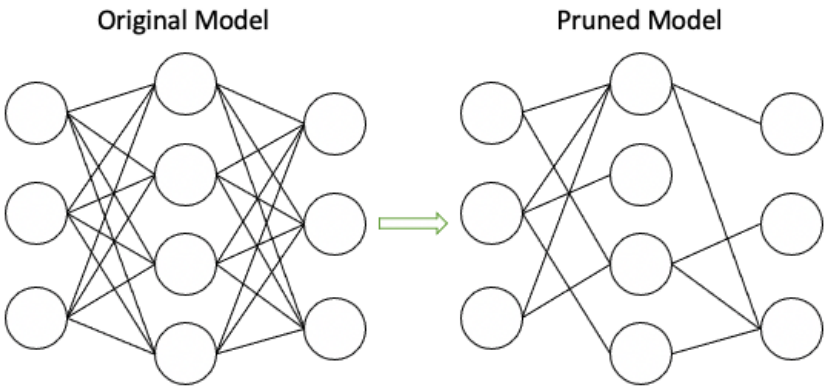

FIGURE 6.7: Original model vs. pruned model (towards data science)

After pruning and quantizing the model, we have to convert it into a FlatBuffer format (the format used by TensorFlow Lite, see Figure 6.8). FlatBuffer is a cross platform serialization library compatible with many programming languages (C++, C#, Python, JavaScript, etc.). Some of the advantages of using FlatBuffer for tiny devices are: access to serialized data without parsing/unpacking, memory efficiency and speed, and tiny code footprint [13]. The FlatBuffer model will be serialized to a byte array in order to be deployed in the microcontroller.

6.1.6 Deployment and Inference

Finally, we can deploy the trained and serialized model into the microcontroller to run the keyword spotting application. This basic application was based on the already mentioned TensorFlow example micro speech. Therefore, we will only modify

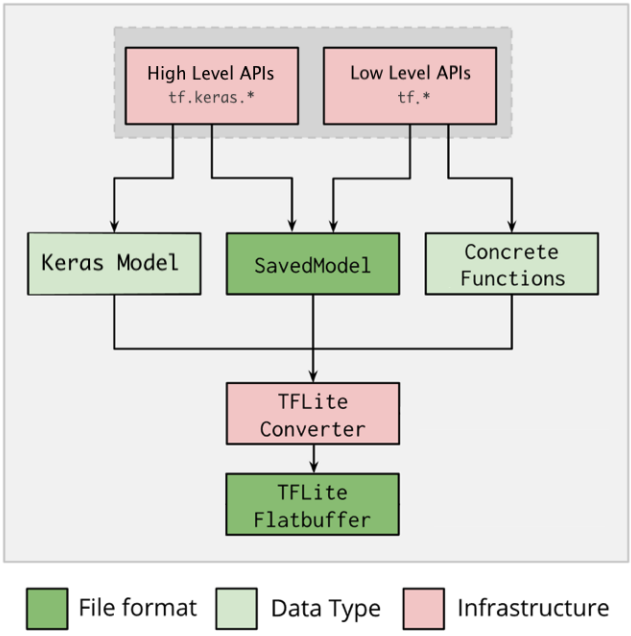

FIGURE 6.8: Diagram of TFLite Converter (source: TensorFlow)

the necessary files to deploy the custom model [1].

The deployed keyword spotting application is able to recognize the words *vermell*, *verd* and *blau*. When the device recognizes any of the keywords, it turns on the RGB LED with the corresponding color. This simple application could be easily upgraded to perform more complex tasks using the already pre-trained model. Although we just implemented a simple keyword spotting application based on previous examples, the research on the TinyML pipeline will help us to develop more complex applications in the next chapters.

6.2 Second approach: Edge Impulse

Keyword spotting have been proved as a very useful technique. Therefore, it is common to find tools that allow a very fast implementation without expertise in machine learning nor microcontrollers. For example, the Edge Impulse platform is a very intuitive tool for developing all kinds of basic TinyML applications, including keyword spotting. In this second approach we will use the Edge Impulse platform to implement a keyword spotting application that is able to recognize the same three

[1] The modified project can be found in the Github page https://github.com/MarcMonfort/TinyML-KeywordSpotting.

words (*vermell*, *verd* and *blau*). The development process will be much faster than the previous approach, since the Edge Impulse platform is very fast and simple to use.

Again, the first step is to create a dataset to train the keyword spotting model. Edge Impulse provides a firmware that allows to connect a microcontroller board to the platform. Therefore, it is possible to create our dataset using the same board's sensors that will be used by the application when deployed. For creating our keyword spotting dataset, we have connected the Arduino Nano 33 BLE Sense board to the platform, and we have used the embedded microphone to record samples of the keywords, which are sent directly to the Edge Impulse platform. The maximum length of the recorded audio is about 10 seconds. However, it is possible to split the audio into one second sample containing a single word. Figure 6.9 shows a long audio sample that is split into six segments using the Edge Impulse platform. The segments can be randomly shuffled to make the application more robust to un-aligned audios. After obtaining enough samples for each keyword, we have to rebalance our dataset between the training and testing set.

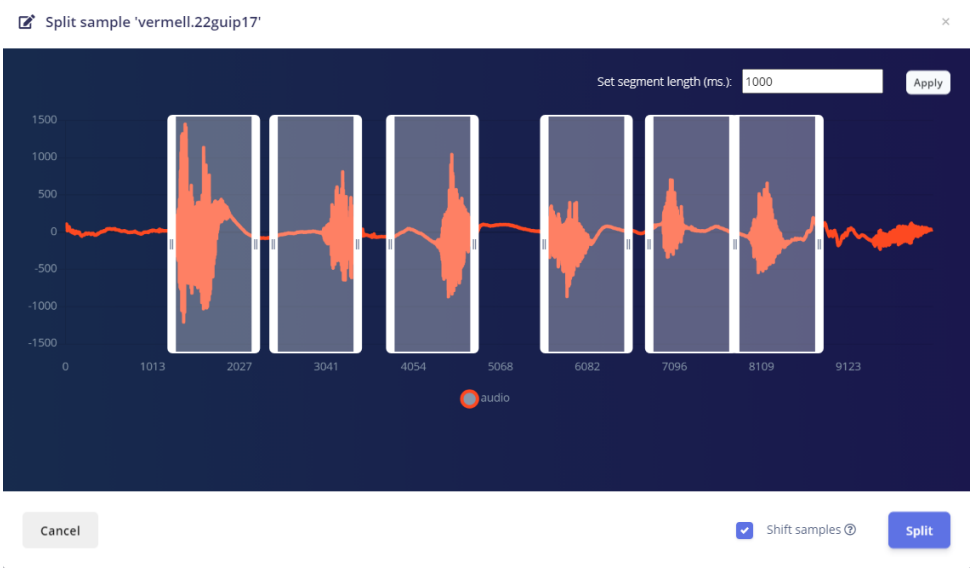

FIGURE 6.9: Edge Impulse interface to split audio recording into several shifted sections.

The next step is to configure the machine learning pipeline. The raw data have to be processed before being sent to the neural network. We will use the Mel Frequency Cepstral Coefficient (with 13 coefficients) to generate the feature vector of each audio sample, like we did in the previous approach. To configure the neural network

model, we can use the Keras library. However, the platform allows to tweak the parameters with a visual interface. We will use the recommended neural network, which can be seen in Figure 6.10.

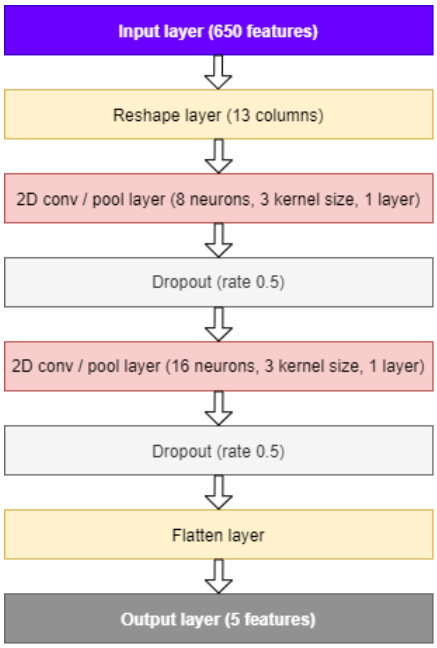

FIGURE 6.10: Diagram of the keyword spotting model used in Edge Impulse.

Our implementation with Edge Impulse is almost over. The last step is to train the model. The model is trained with a learning rate of 0.005 for 100 training cycles. We have obtained an accuracy of 0.948 and a loss of 0.17. A very useful feature of the Edge Impulse platform is the *Feature explorer*, which allows to visualize the classification of the data in a 3D plot, and see the cluster created by each keyword.

After training the model, we can generate the firmware that contains the application code and all the necessary libraries for the microcontroller board (Arduino Nano 33). In this final step, the neural network can be quantized in order to optimize the latency and reduce the RAM and the ROM usage, with a very low drop in accuracy [2].

To deploy the application we have to flash the downloaded firmware into the microcontroller board. By default, the application driver is programmed to constantly

[2]The project can be found on the page `https://studio.edgeimpulse.com/public/25158/latest`

record one second audio samples, then process the audio (MFCC) to obtain the feature vector, and finally call the trained model to classify the audio among all the keywords. The result is sent over the serial output, and can be seen using a serial monitor. This driver can be easily adapted in order to perform more sophisticated keyword spotting applications.

6.3 Results

In both approaches, we have obtained a very good model accuracy (above 0.94) in the testing set. However, the performance of the first approach application, when deployed in the microcontroller, has dropped considerably. On average, the application correctly recognizes a keyword once in four times (around 25 % accuracy). This low performance is due to the difference in the quality of the training dataset and the audio recorded by the embedded microphone. The training dataset has been created using a laptop microphone, which is able to capture with a much higher quality than the Arduino's microphone.

In the second approach, we have created the training dataset using the same low-quality microphone from the Arduino board. The performance obtained by the deployed application is higher than in the first approach, with the application being able to correctly recognize a keyword two out of three times (around 66 % accuracy). The drop in accuracy is produced by keywords that are recorded with a strange alignment. The application is constantly recording the environment, so it is not always possible to capture each keyword from the start. This could be solved by creating a bigger dataset that contains audios with a wide variety of alignments.

Using the Edge Impulse platform, we have obtained a better performance. Moreover, the development process was much more easier and faster than in the first approach. However, it has been very useful to develop the first approach application, since we were able to understand more deeply the TinyML pipeline and the challenges of the keyword spotting technique. The results have also shown the importance of having a good training dataset. In the next chapters we will develop more advanced applications based on the techniques learned here.

7 On-Device Model Training

7.1 Training Phase

Training a neural network model is all about finding the best parameters (weights and biases) in order to minimize the loss function. The loss function depends on the ability of the model to correctly classify (classification problem) or predict a value (regression approach) from data. In a supervised learning approach, we need a dataset for training the model. It is very important to have a good dataset, since the model will be trained to recognize patterns in it. For training a neural network model we use the gradient descent algorithm. Gradient descent is an iterative optimization algorithm that allows to find the local minimum of the model's loss function. In each iteration of the algorithm, the model tries to classify a batch of data. From the error obtained, the algorithm tries to adjust the parameters in order to reduce it. This last step is called backpropagation. Usually, the training algorithm has to iterate many times over the dataset.

Typical TinyML applications usually perform the training phase outside the microcontroller board. In the previous keyword spotting application, we were using a pre-trained model. The model was trained in Google Colab with the standard TensorFlow framework. Then, the model was pruned and quantized to reduce its size. Finally, the model was compiled to a specific format to be flashed into the microcontroller board. However, it was not possible to improve the model after being deployed. The application only used the model to classify new data. In this approach, the user cannot improve the model with its own local data (his/her own voice) in order to improve the performance of the application.

Training a model is a computationally intensive task that some devices cannot perform. Some of the biggest artificial neural networks can take up to months to be

trained and many watts of power. This huge amount of computing power is not feasible in tiny devices like microcontrollers. However, our intention is not to train a general purpose model (e.g., GPT-3), but to train a very specific model that performs well in a single task. The keyword spotting model we used in the previous application had a size of 16 kB and only 16652 parameters. Running a gradient descent iteration on this model with the Arduino Nano 33 would take less than a second. Therefore, it is computationally possible to train a specific model using a tiny device.

Transfer learning is a technique that can significantly reduce the time and the computing power required to train a model. This technique takes advantage of a previously trained model to train a new one. For example, we could train a model that recognizes Greyhounds in images, making use of a pre-existing model that recognizes dogs. Instead of training a new model from the beginning, with transfer learning we reuse some of the parameters (weights and biases) already trained on a previous model. Usually, we only take the parameters from the first layers, since their job is to recognize the most basic patterns of the data. The new model only has to train the layers that are closer to the output. These layers are responsible to recognize the most specific patterns of the data. Therefore, with transfer learning we only have to train a small subset of layers of the neural network. However, transfer learning has some limitations. If the purpose of the previous trained model is not related to the purpose of the new model, the result will be very poor. The problems need to be similar enough, which is quite hard to discern [14].

An important requirement for training a neural network model is to have a high floating point precision. In each iteration of the gradient descent, we take a small step in the opposite direction from the loss function gradient. The value of this step can be very small, therefore, requiring a high precision floating point type to represent it. This does not suppose a problem in normal scenarios, where a general-purpose computer is used to train a model. Any computer should be able to support enough floating point precision for the gradient descent algorithm. However, microcontrollers are very limited devices. Only microcontrollers with a Floating Point Unit (like the Aruino Nano 33) can accurately perform the gradient descent.

7.2 On-Device Training Application

This section shows the development of a TinyML application[1] that demonstrates the feasibility of training a model in a microcontroller. We will develop a program that is able to train a keyword spotting model. Moreover, the application will be able to recognize three different keywords, which will be decided by the user when starting to train the model. When the application is restarted, the model has to be trained once again, using the same or a new set of keywords. The user could also test the application accuracy at recognizing the keywords to see the training progress.

7.2.1 Device Setup

To interact with the application, we need to set up the board. The application will be deployed on an Arduino Nano 33 BLE Sensen board, which already comes with several components. The board has an integrated microphone, that will be used to record the keywords. It also has an integrated white and RGB LED. The white LED will be used to visualize the application stage (e.g., IDLE, busy), and the RGB LED will be used to show the output class of the keyword spotting model. To train the model, we need to plug three buttons to the board. Each button will allow to train one of the keywords. A fourth button will be added for testing the model, but without training it. Figure 7.1 shows a way to connect the buttons to the Arduino board.

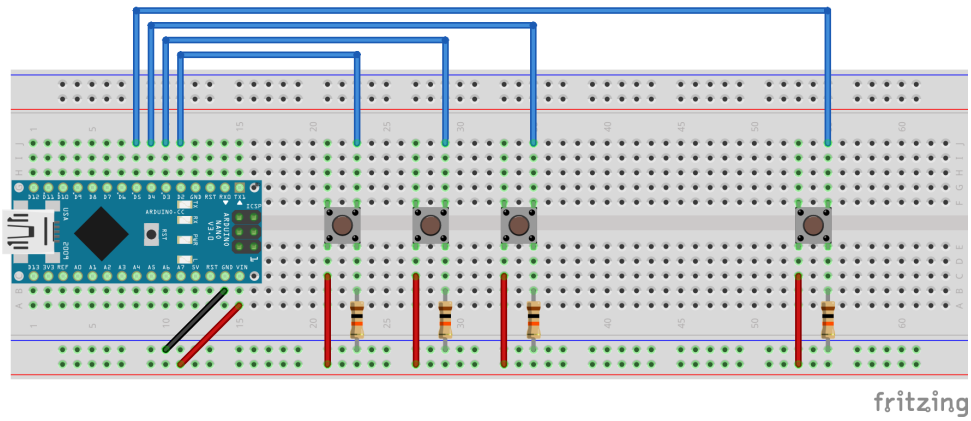

FIGURE 7.1: Microcontroller board setup with four external buttons for on-device training (own creation).

[1]The program can be found on the Gihub page https://github.com/MarcMonfort/TinyML-OnDeviceTraining

7.2.2 Workflow

The application starts when the program is flashed to the Arduino board, or the board is restarted with the program already flashed. The board should have a similar setup as the one shown in Figure 7.1. Every time the application starts, a new model is created. The weights and biases of the model are initialized to random numbers. Once the model is initialized, the user can start to train it using the three training buttons. Each button allows to train one of the three keywords. When a training button is pressed, the RGB LED will turn on with a color identifying the button (red, green or blue). When the button is released, the Arduino built-in microphone will start recording a one second audio. The keyword should be said within this second. The recorded audio will be processed to obtain the feature vector. Finally, the model will be trained with the feature vector and the expected keyword (known from the button pressed). The result of each training iteration is sent through the serial port. The fourth button has the same workflow as the three training buttons, but it will not train the model. Instead, it will turn on the RGB LED with the color associated with the keyword that the model had recognized. Figure 7.2 shows a diagram of the application workflow.

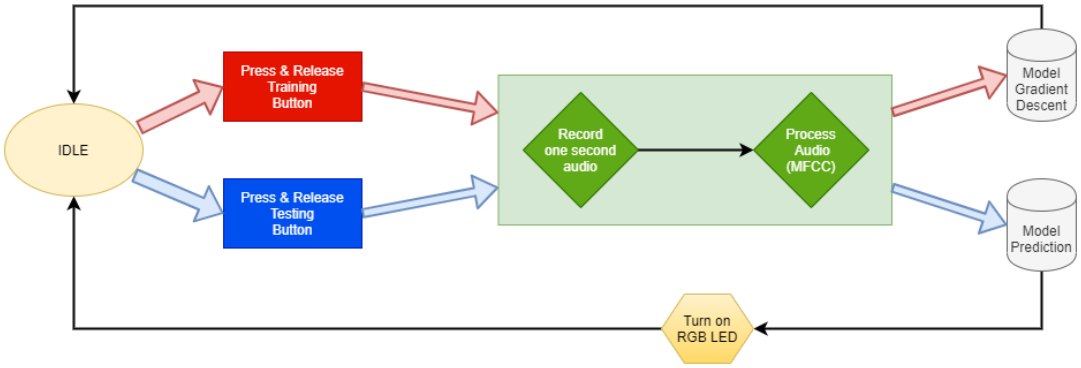

FIGURE 7.2: Workflow diagram of on-device training application (own creation).

7.2.3 Feature Extraction

Before training the model, we need to obtain the feature vector. As mentioned above, just after releasing any of the buttons, the microphone will start to record for one second. The microphone will record with a sample rate of 16 kHz, and the result will

be stored in an array of 16000 values. Each value is represented by a 16-bit signed integer. Therefore, the recorded audio will have a size of 32 kB. From the recorded audio we have to obtain the features that will be used to train the model. A popular way to extract features from human voice is using the Mel Frequency Cepstral Coefficients (MFCC) (see Data Processing). Using the MFCC (with 13 coefficients), we obtain a spectrogram of 13 rows and 50 columns, as shown in Figure 7.3. This spectrogram will be the feature vector used to train the model.

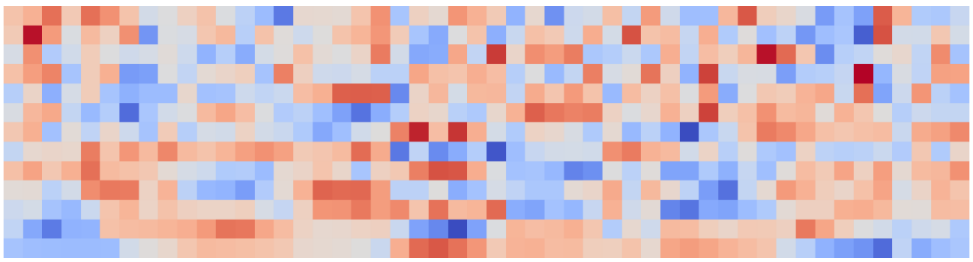

FIGURE 7.3: MFCC from *Montserrat* keyword (source: Edge Impulse).

7.2.4 Artificial Neural Network

Our application aims at training a model on a microcontroller. In the previous applications, we used the TensorFlow Lite Micro framework to load a pre-trained model and use it to classify new data. However, this framework does not support the training of the model, and therefore, we will have to implement an artificial neural network interface and the training algorithms. To do so, we will use the neural network template implemented by Ralph Heymsfeld on the Arduino UNO board [15], and modify it for a keyword spotting application.

The neural network model has to be small and fast to work on the Arduino board. The keyword spotting problem is very simple, if compared to general speech recognition problems. Therefore, our model does not need to have millions of parameters for a good performance. The model that will be used in our application is a feed-forward neural network with an input layer of 650 nodes, a single hidden layer of 16 nodes, and an output layer of 3 nodes (each node representing a keyword). The activation function used is a sigmoid. Although the Relu function is more popular for deep learning models, for smaller models Sigmoid can be a better choice. This architecture gives a sum of 10467 parameters (10448 weight and 19 biases). We will

use a 4 byte float to represent the parameters (the maximum precision allowed by the Arduino framework). Therefore, the model will have a total size of 41868 bytes. These bytes are not stored in the slow Arduino flash memory (1MB) because they are constantly modified in the training phase. They need to be stored in the RAM (preferably in the heap). This is not an issue with the Arduino Nano 33, since it has 256 kB of SRAM. Figure 7.4 shows a diagram of the neural network.

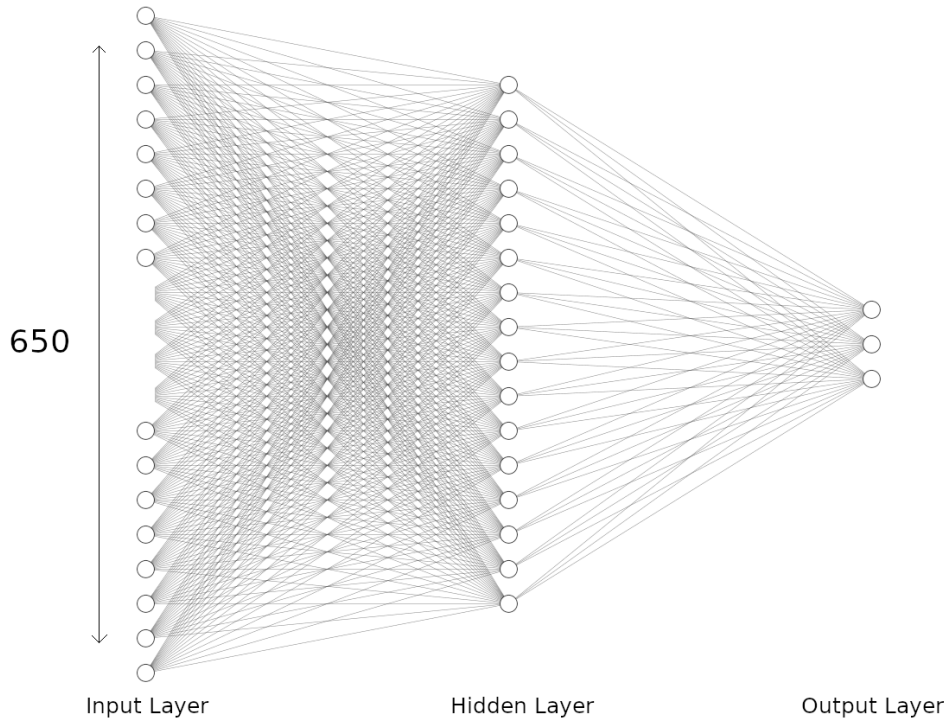

FIGURE 7.4: Neural network diagram for on-device training application (own creation).

The model is trained following an online learning approach [16]. As already mentioned, the model is created every time the application is restarted. The weights and values are initialized to random floats in the range between -0.5 and 0.5. When a training button is pressed (and released), the board will record a keyword (said by the user) and generate the feature vector (MFCC). Then, the feature vector will be sent to the model to perform a forward propagation in order to obtain the three output values. With these output values and the expected values (known from the button pressed), we can calculate the mean squared error. The final step is to calculate the delta [2] of each neuron in order to perform a gradient descent iteration. To

[2]The delta value reflects the magnitude of the error of each node.

sum up, we are following an online learning approach, where the model is trained using the most recent recorded audio.

To optimize the model, we can fine tune training hyperparameters. The most important hyperparameter is the learning rate, which controls how much the model is updated in response of the estimated error of each training epoch. Choosing the correct learning rate is quite challenging. A too small value may result in a long training phase, and a too high value may result in an unstable training process [17]. The default learning rate is 0.3. This value is quite high if compared with the values used for training more complex models. However, since the application is fully trained by the user, it is necessary to not extend the training phase for too long.

The second hyperparameter is the momentum. Similar to the learning rate, the momentum tries to maintain a consistent direction on the gradient descent algorithm. It works by combining the previous heading vector and the new computed gradient vector. The default value used in our setup is 0.9, which adds 90 % of the previous direction to the new direction. We note that the use of the momentum consumes additional RAM, since it is necessary to store the previous gradients.

7.2.5 Results

In order to evaluate the performance of on-device training, the model is trained with two different sets of keywords. The first set of keywords contains two spoken words, namely *Montserrat* and *Pedraforca* (two iconic mountains of Catalonia), and a third type to classify silence. The keywords are spoken to the device in alternate order. Figure 7.5 shows the result of the training process. The erratic short-scale behaviour is produced by the online learning approach, where each epoch is using a single audio recording, and therefore, the accuracy obtained can vary significantly between one word and the next. However, the important observation is that loss progressively decays over the epochs.

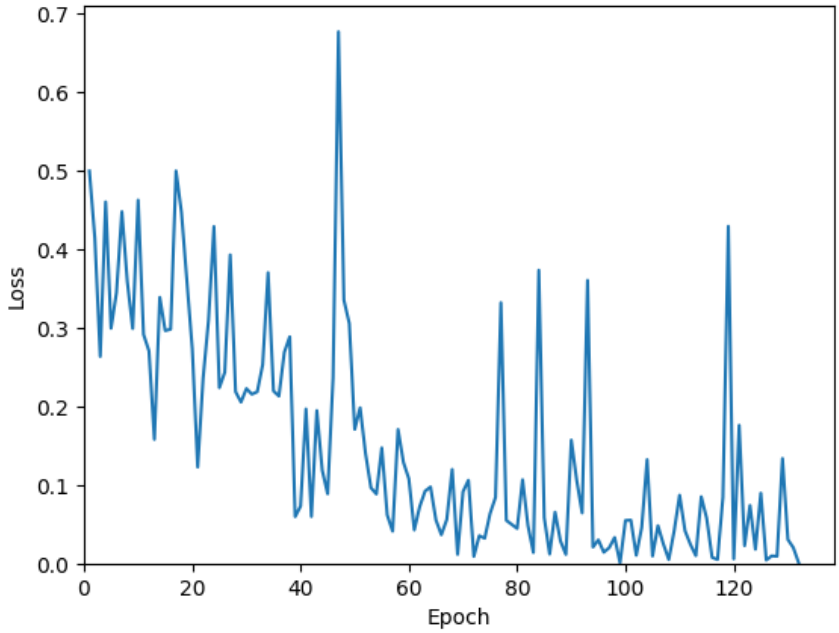

FIGURE 7.5: Loss vs. epochs during the training of the three keywords (*Montserrat*, *Pedraforca* and silence). Number of observations is 130, learning rate 0.1, momentum 0.9.

In the previous experiment, we have set a learning rate of 0.1. With this learning rate, we needed around 80 epochs for the loss to converge. However, 80 training epochs could suppose too much time for the user to train the model, since for each epoch the user have to say a keyword. Therefore, in this second experiment, we increased the learning rate up to 0.3. Figure 7.6 shows the result with the updated value. Again, we see that the loss progressively decays over the epochs, but this time it only needs 30 epochs to converge.

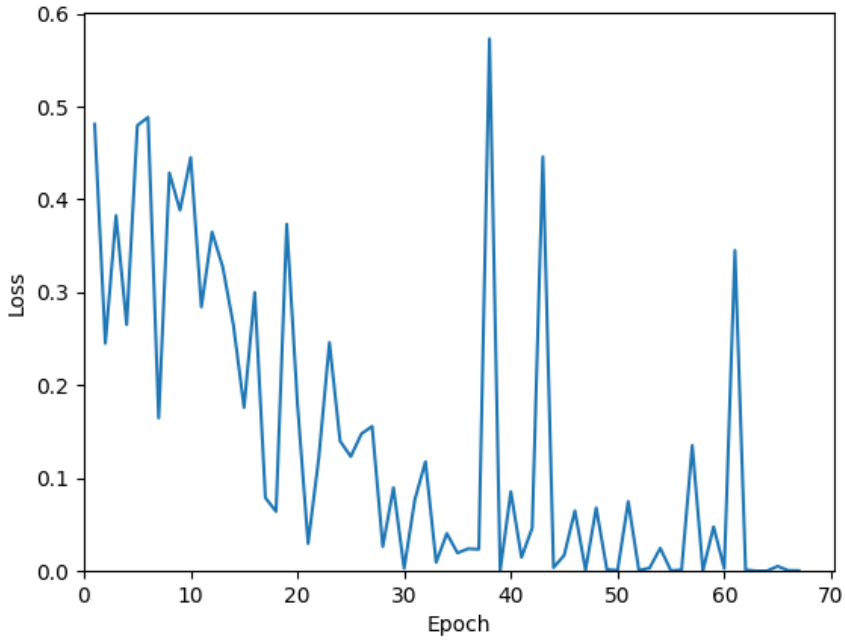

FIGURE 7.6: Loss vs. epochs during the training of the three keywords (*Montserrat*, *Pedraforca* and silence). Number of observations is 70, learning rate 0.3, momentum 0.9.

Before, we have trained the model using three keywords: two words (*Montserrat*, *Perdraforca*) and a silence class. In this third experiment, we have trained the model using only the words *Montserrat* and *Pedraforca*. The result is similar to the previous experiment, but we obtained a much more consistent accuracy in the last epochs (Figure 7.7). Altought we are not able to classify the silence, when no word is spoken the model shows a low probability for both keywords. Therefore, we could set a threshold to classify also the silence or unknown words.

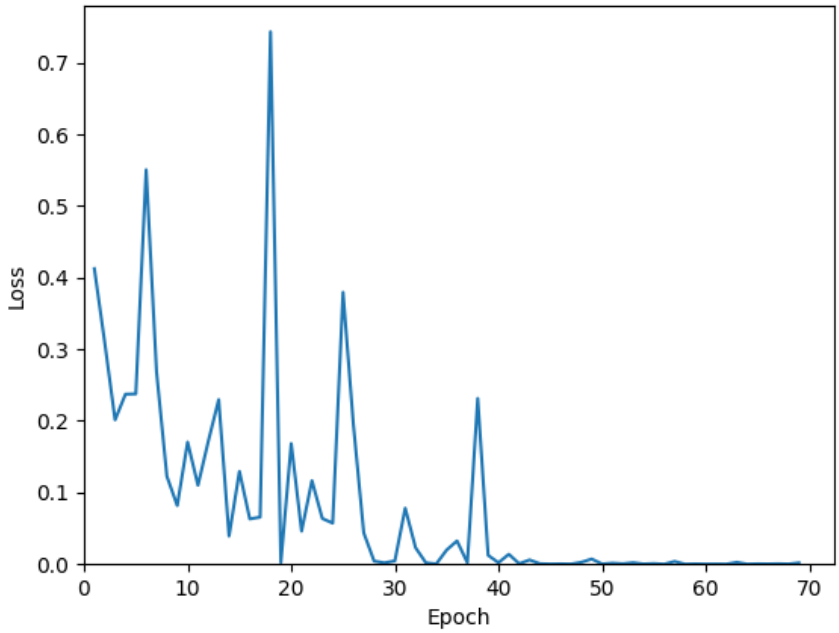

FIGURE 7.7: Loss vs. epochs during the training of the two keywords (*Montserrat* and *Pedraforca* (no silence)). Number of observations is 70, learning rate 0.3, momentum 0.9.

In this fourth experiment, the model is trained with the three keywords (*Monserrat*, *Pedraforca* and silence) and a learning rate of 0.1. However, we removed the momentum parameter. The result is shown in Figure 7.8. With this setup it takes more than 200 training epochs for the loss to converge. After analyzing the previous experiments, we have decided to set a default learning rate of 0.3 and a momentum of 0.9.

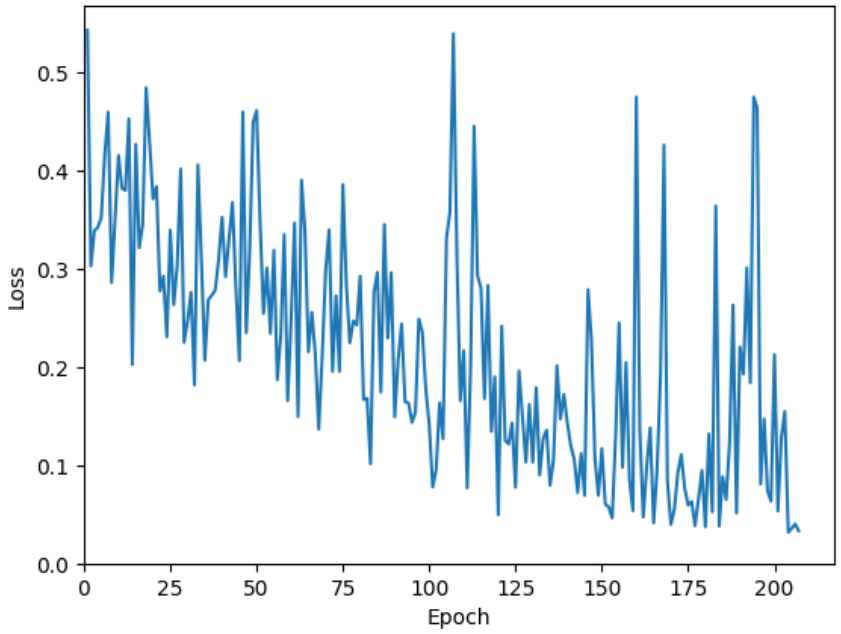

FIGURE 7.8: Loss vs. epochs during the training of the three keywords (*Montserrat* and *Pedraforca* and silence). Number of observations is 200, learning rate 0.1, no momentum.

To have a more reliable evaluation, we will perform a last experiment with a different set of keywords. This set of keywords contains the words *vermell*, *verd* and *blau* (which stand for red, green and blue, in Catalan). Figure 7.9 shows the results of the training with the default hyperparameters (0.3 learning rate, 0.9 momentum). Again, it can be seen how the loss decays over the epochs, proving that the application works with different sets of keywords.

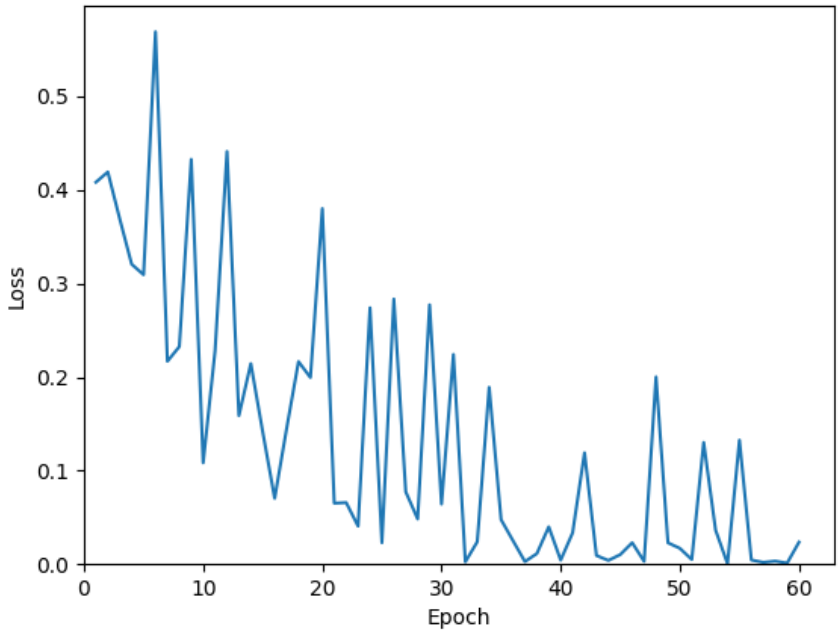

FIGURE 7.9: Loss vs. epochs during the training of the three keywords (*vermell*, *verd* and *blau*). Number of observations is 60. learning rate 0.3, momentum 0.9.

The results are very positive. It has been proven the feasibility of training a neural network model in a microcontroller, at least, for a problem as complex as keyword spotting. It is important to notice that the application is not fully optimized. In fact, there are several upgrades that could be done in order to improve the performance. For example, we could add a convolutional layer to the model to process the input image (Figure 7.3). This layer would reduce the short-scale behaviour. We could also tweak the number of hidden layers and neurons. Also, we could use a more complex Fourier transform, or more coefficients in the MFCC algorithm for processing the audio. However, we are satisfied with the results obtained, and we leave the possibility to implement these upgrades in forthcoming applications.

8 Federated Learning with Microcontrollers

8.1 Federated Learning

Over the last few years, federated learning has raised the interest of the research community, as it provides a means to train machine learning models on distributed devices without sharing the local training data [19]. In federated learning, instead of training a single model with a centralized dataset, local models are trained with local datasets and then merged into a global model. The inconvenience of having less data on each device can be compensated by the capacity of the global model built upon the local ones. Federated learning is also seen as a solution to train with data which, for privacy reasons, cannot be sent to the cloud, such as medical records [20]. Therefore, the main advantages of federated learning over traditional machine learning is the security it brings from data leakage, and the personalized experience that is achieved by training a model with local data.

Federated learning can be orchestrated using a centralized, decentralized or heterogenous strategy. In centralized federated learning there is a central server that is responsible for coordinating the clients (edge nodes) in order to create the global model (Figure 8.1). The central server receives the local models (or the gradients) from the clients, aggregates them to update the global model, and sends back the global model to be used and trained by the clients. In contrast, decentralized federated learning does not use any central server. The clients have to coordinate in a distributed scenario in order to generate the global model. This approach is much more complex, but it has the advantage that does not depend on any central server. Lastly, the heterogenous strategy is a new federated learning approach that enables

the training of heterogenous local models while still producing a single global model [21]. In this approach, devices with different capabilities can be used together, even if they can only train models of different architectures.

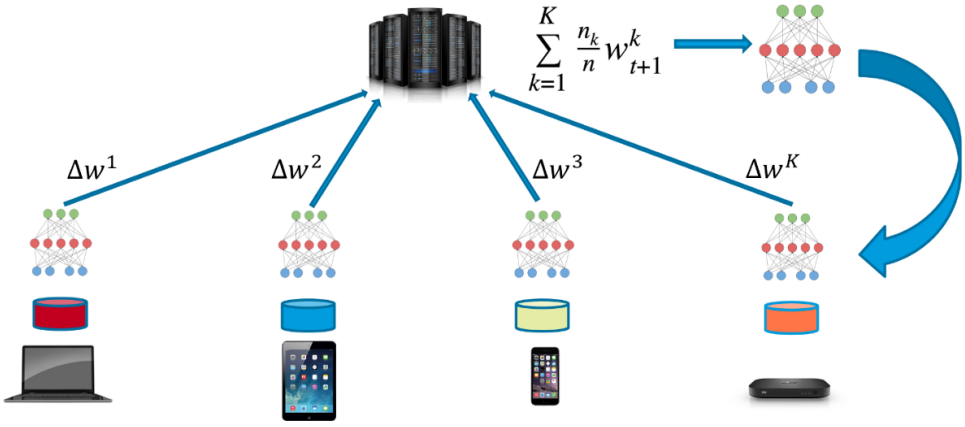

FIGURE 8.1: Centralized federated learning diagram (source: KD-nuggets).

The two most popular aggregation methods are Federated Stochastic Gradient Descent (FedSGD) and Federated Averaging (FedAVG). In FedSGD, the clients calculate the gradient vector using their local data, but instead of backpropagating the error to the model, the clients just send back to the server this gradient. Then, the server will average all the gradient vectors and update the global model. In contrast, with FedAVG, the clients train their local model. Therefore, the error is backpropagated, and the model's parameters (the weights and biases) are updated. Then, the clients send back all the parameters to the server, which are averaged (and weighted) to create the global model. The advantage of using FedAVG over FedSGD is that with FedAVG we are able to train multiple times the model before sending the updated parameters to the server. Instead, with FedSGD it is required to send the gradient vector in each training iteration, and therefore, it has a much higher communication cost. However, FedSGD can guarantee the convergence, while FedAVG cannot [22].

Some commercial federated learning applications are Google's Smart Keyboard and Apple's Siri. Google's Smart Keyboard is used on smartphones in order to help the user by suggesting the next word to type. Depending on the user response, accepting or not the suggestion, the model will be updated to improve the accuracy of future

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $m \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of m clients)
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*
$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
for each local epoch i from 1 to E **do**
  for batch $b \in \mathcal{B}$ **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

FIGURE 8.2: Federated Averaging algorithm [23].

suggestions [24]. Apple's Siri is a voice assistant that works in iPhone devices. When a user says "Hey Siri", the Siri assistant wakes up to respond any of the user's query. In order to personalize the application, they have implemented a federated learning approach so the Siri assistant will only respond to the voice from the iPhone's owner [25]. The expectations for new federated learning applications are very high.

Although federated learning offers many possibilities, it also has several limitations. During the learning process, it requires frequent communication between the server and the clients. Depending on the aggregation methods, it may be necessary to exchange several times the model's parameters. Therefore, the communication network can get saturated and easily become the bottleneck of the training phase. Moreover, the clients involved in federated learning may be unreliable (and drop out from the training phase), since they commonly rely on less powerful communication media (i.e. Wi-Fi, Bluetooth) and usually depend on batteries. Lastly, the local data probably will not be independent and identically distributed (IID), and the amount of local data may span several orders of magnitude between the edge devices. This

can be detrimental to the model training.

8.2 Federated Learning Application

This section shows the development of a TinyML application[1] that demonstrates the feasibility of (centralized) federated learning with microcontrollers. This application will be based on the previous keyword spotting application implemented in On-Device Model Training. The training phase is identical from the device point of view: The user can train a keyword spotting model (using three buttons) to recognize three different keywords and test the performance (with a fourth button). However, this new application instead of training a model with a single device, it will use several devices following a federated learning approach. Therefore, we will also implement a server that will coordinate all the clients (devices), and update a global model.

8.2.1 Device Setup

The application will be deployed on several Arduino Nano 33 BLE Sense boards. Each of these boards should have a similar setup as the one shown in Figure 7.1. The three leftmost buttons are used to train the model, and the fourth button is used to test it. Another requirement is to connect all the boards to the computer that will be running the server script. This connection should be through the serial port, since we are using the pySerial library from Python. Figure 8.3 shows the application diagram with two boards (clients).

[1]The program can be found on the Gihub page `https://github.com/MarcMonfort/TinyML-FederatedLearning`

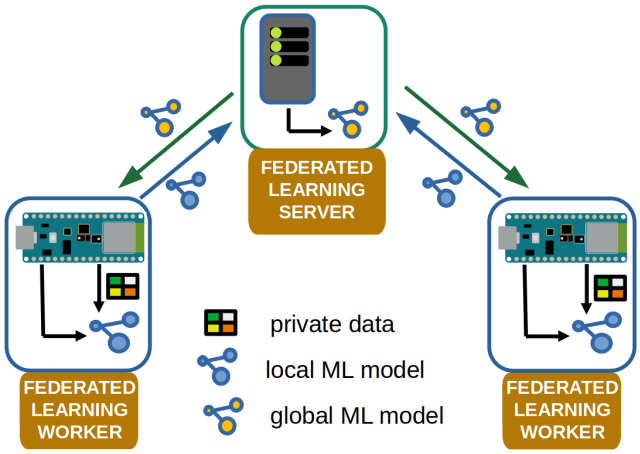

FIGURE 8.3: Federated learning diagram with a server and two clients.

8.2.2 Workflow

The application workflow differs from the previous one. To start the application, we have to flash the firmware to all the Arduino boards used as clients. Then, we should run and configure the server (a Python script). The server will ask the number of clients and the port where they are connected (e.g., COM5, tty5). After the configuration, the server will create a neural network model and initialize it with random parameters (see Artificial Neural Network). This model will be sent to all the clients. As soon as the clients received the model, they can start training it with their local data. The local data will be generated by the user saying the keywords. Of course, all the clients must use the same three keywords. Meanwhile, the server will set a ten second countdown to start the first federated learning iteration. When the countdown ends, the server will try to connect with all the clients and ask them to send their improved model. The clients will have five seconds to accept the request, or they will be discarded for this iteration. The models that are received from the connected clients will be aggregated to create a global model. This global model will be sent back to the connected devices to be further trained. Finally, the server will restart the countdown for the next iteration. Figure 8.4 shows the sequence diagram of the application workflow.

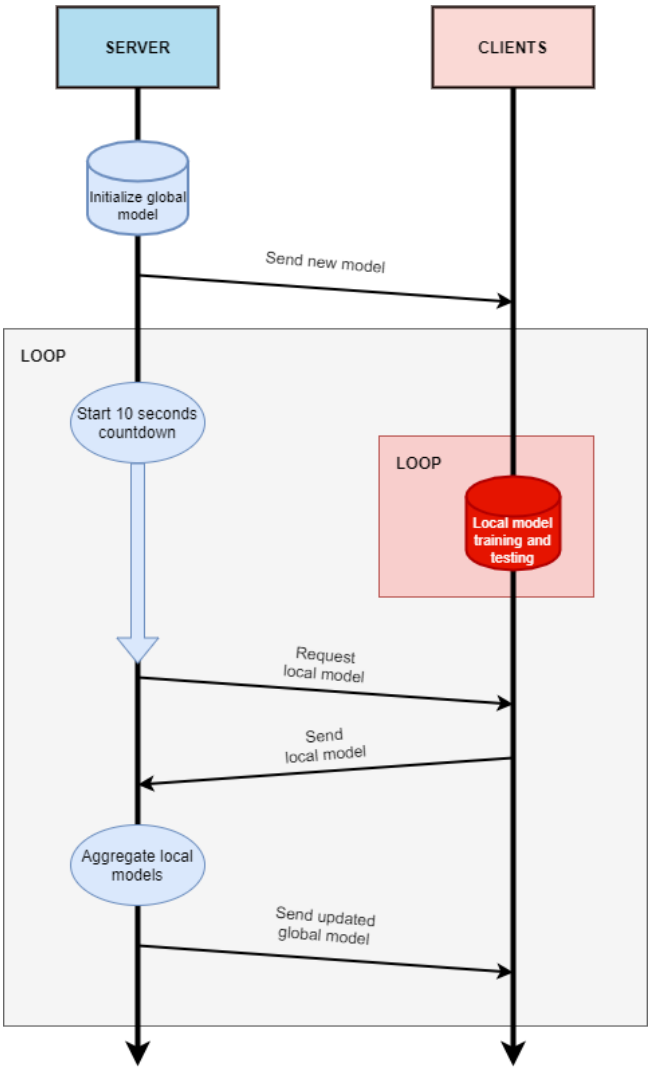

FIGURE 8.4: Sequence diagram of the federated learning application (own creation).

8.2.3 Communication and Data Transmission

The centralized federated learning approach requires frequent communication and high bandwidth. The Python server uses the pySerial library to communicate and exchange data with the Arduino boards. When the server starts, the first thing it does is to open a communication channel through the ports used by the clients. Then, using the open channels, it is possible to receive and send bytes. The bytes received on the server are the bytes sent by the Arduino through the Serial output buffer. The bytes sent from the server are received on the Arduino Serial input buffer. The input buffer of the Arduino only holds 64 bytes. However, the pySerial input buffer can hold as many bytes as the RAM allows.

The training phase requires to exchange the models between the server and the clients multiple times. First, the initialized model is sent from the server to all the clients. And then, in an iterative way, the clients send the local model to the server, and the server sends back the global model to the clients. As calculated in Artificial Neural Network, the neural network model has a size of 41868 kB. This size does not suppose any issue for the server when receiving the models, since it only has to read the parameters from the computer buffer. However, sending the model from the server to the Arudino can be problematic due to the small size of the Arduino's input buffer (64 bytes). To avoid a buffer overflow, the server only sends four bytes at once, before checking that the board has received them. Therefore, the sending time (from server to client) is significantly longer (3 sec) than the receiving time (1 sec).

8.2.4 Model Aggregation

Probably, the most important part of the federated learning is the aggregation of the models. The technique used in this application is FedAvg, which stand for federated average. In contrast to FedSGD (stochastic gradient descent) where the aggregation is done on the gradients, with FedAvg, the aggregation is done on the model's parameters (weights and biases). After the server receives all the models from the clients, the parameters are weighted by the number of training epochs done on the local model since the last aggregation. Then, the weighted parameters are simply averaged in order to produce the global model. Although the aggregation of the local models is the heart of the federated learning, the implementation is straightforward. In the Python script the aggregation comprises a single line of code (using Numpy):

```
global_model = np.average(local_models, axis=0, weights=num_epochs)
```

8.2.5 Results

To evaluate the application, we deployed the scenario shown in Figure 8.3. We used two Arduino Nano 33 BLE Sense boards with identical hardware setup as previously shown in Figure 7.1. The boards are connected via the serial port to a PC, where the

federated learning server runs. We run three experiments with different strategies to train a global keyword spotting model.

Federated learning with IID data: In the first experiment, we performed 10 training epochs on each client before sending the local model to the server. Then, the updated global model is sent back to both clients for the next local training round. Two keywords are used for training, *Montserrat* and *Pedraforca*. The two keywords are spoken in alternating order to both clients (nodes), to represent the scenario of training with independent and identically distributed (IID) data. Figure 8.5 shows the obtained results for the training loss. It can be seen that, in both nodes, the loss decreases over the training epochs.

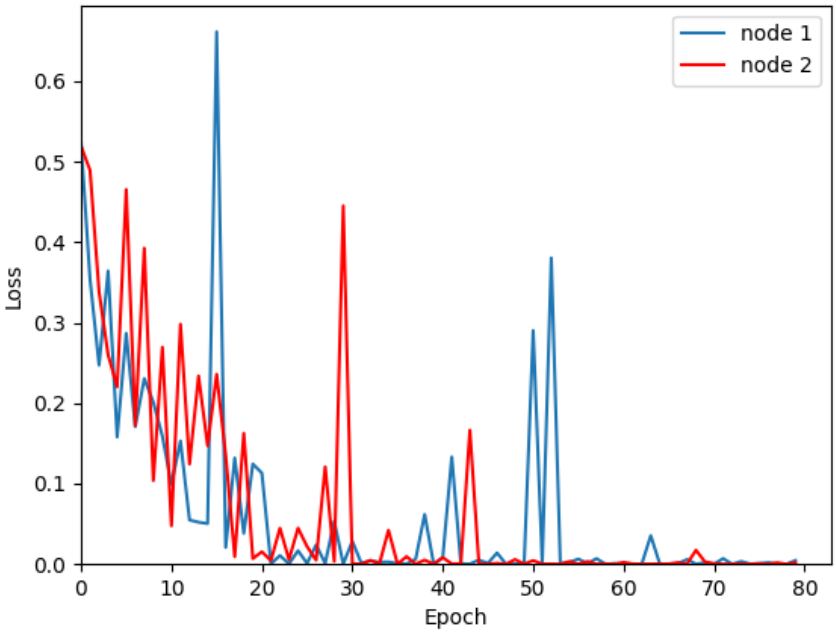

FIGURE 8.5: Loss vs. epochs during training with federated learning in IID data scenario (10 training epochs per aggregation).

Federated learning with non-IID data: In the second experiment, the two keywords are split among the clients (nodes) (i.e., each keyword is spoken to only one of the nodes). This setup aims at presenting the scenario of training with non-IID data. It can be seen in Figure 8.6 how, after averaging the model every 10 epochs, loss increases. This can be explained by the fact that the model averaging merges the

characteristics of the two models, which are trained with different keywords. However, the long-term trend is that the loss of the global model is decreasing over the training epochs.

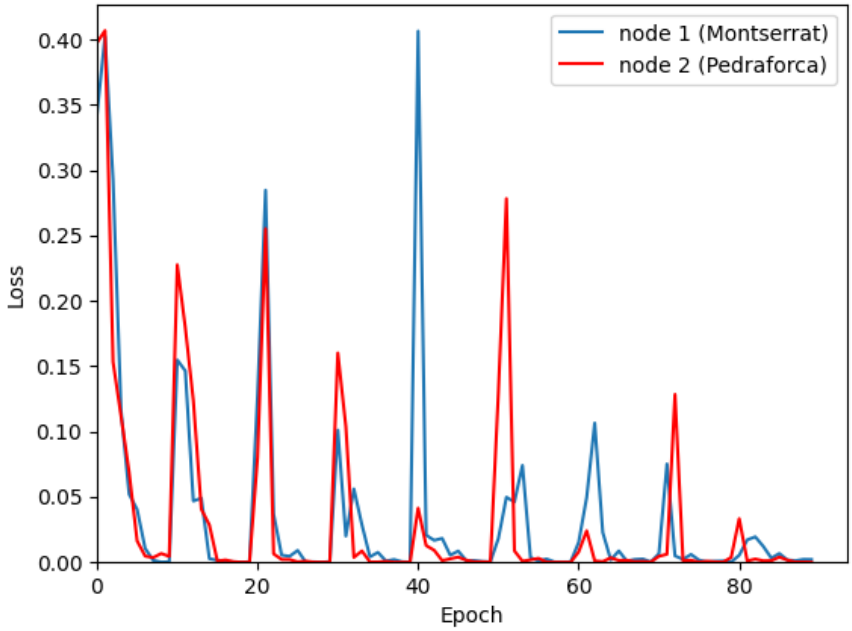

FIGURE 8.6: Loss vs. epochs during training with federated learning in non-IID data scenario (10 training epochs per aggregation).

Federated learning with non-IID data (1 training epoch per aggregation) In the third and last experiment we kept the two keywords split among the clients, like in the previous experiment, but instead of performing 10 training epoch, we only performed one training epoch on each client before sending the local model to the server. Figure 8.7 shows once again that the loss decrease over the training epochs, and therefore, it proves that averaging the local models helps to produce an improved global model, even with non-IID local data.

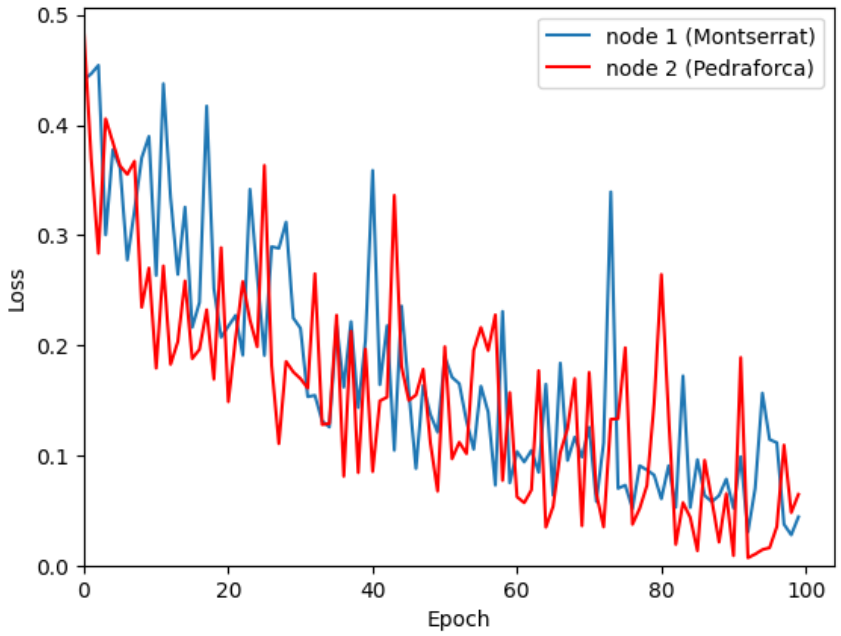

FIGURE 8.7: Loss vs. epochs during training with federated learning in non-IID data scenario (1 training epoch per aggregation).

The results proved the feasibility of using microcontrollers in a federated learning application. We could train a global keyword spotting model by aggregating local models trained with distributed devices. However, we could make several improvements to the application, in addition to those already mentioned for the training phase (see On-Device Model Training). For example, we could start with a pre-trained model in order to reduce the training time. Also, we could improve the communication protocol between the server and the clients to reduce the transmission time and make the application more robust to avoid the devices dropping out of the training phase. It would be very interesting to try different aggregation methods, like FedSVG or the more advance FedMA. Finally, we encourage anyone to adapt this application into a decentralized orchestration, which will bring up more challenges to overcome.

9 Sustainability Analysis

9.1 Matrix of Sustainability

A sustainability report is a common requirement for any IT project. In this chapter, we will analyze the impact of our project using the matrix of sustainability. The analysis is divided into three blocks identified by the matrix columns: **project put into production (PPP)**, **exploitation** and **risks**. For each block we will evaluate the environmental, economic and social impact. Finally, based on the analysis, we will assign a value to each cell of the matrix.

TABLE 9.1: Matrix of Sustainability

	PPP	Exploitation	Risks
Environmental	Consumption of the design	Ecologial footprint	Environmental
Economic	Invoice	Viability plan	Economic
Social	Personal impact	Social impact	Social

9.2 Project put into Production

This block includes the evaluation of the planning, development and implementation of the project.

9.2.1 Environmental

The environmental impact of undertaking the project has been quantified based on the energy consumption (kWh). This project has been carried out by the author and has been supervised by the director and co-director. Therefore, we required three computers. Considering that a computer typically uses about 50 watts of electricity and that the project has required 474 hours of use of a single computer and 20 hours of use of three computers, the amount of energy is:

$$50W * (474h + 20h * 3computers) = 26.7kWh$$

In addition, the average Arduino board power consumption is 0.2 W. Considering 10 hours of use, the amount of energy is:

$$0.2W * 10h = 2Wh$$

Training a machine learning model can involve large amounts of energy. However, TinyML models are relatively small, and therefore, the energy used is negligible. We will consider this energy as part of the energy used by the computers.

In order to reduce the energy, we have reused all three computers, and the microcontroller boards, from previous projects. For this reason, we have not taken into account the energy cost in the manufacturing process. To reduce the model training cost, we have considered using the transfer learning technique, which consists in reusing pre-trained models. With this technique we could save a lot of energy when training very big models. However, as stated above, the models used in TinyML are very small, and transfer learning would not be noticed in the total energy consumption. Therefore, from the previous equations, the total energy consumed is:

$$26.7kWh + 0.002Kwh = 26.702kWh$$

9.2.2 Economic

The estimated cost of undertaking the project is analyzed on Chapter 3. The final estimated cost was about 16158.88€. However, this cost has undergone some changes during the project. The task "Decentralized Federated Learning" was cancelled (as already foreseen in Section 2.4), and therefore, the final cost has been reduced to 16002.88€.

9.2.3 Social

The completion this project has allowed me to demonstrate the knowledge I have acquired during my degree. It has also helped me to improve my writing skills. And finally, I have gained a very deep understanding of machine learning and embedded systems. However, this project has taken me many hours to complete. At times I have felt frustrated and unmotivated due to the heavy workload. But overall, I believe that having done this work will help me in my career and future projects.

9.3 Exploitation

9.3.1 Environmental

Today, machine learning applications running on edge devices need to be connected to a data center to have sufficient computing power. However, the transmission of large amounts of data requires a lot of energy. The solution proposed in this project is to run the machine learning algorithms on the edge device without having to send the data over the internet. Therefore, the TinyML application can save energy, as computation requires less energy than the transmission of large amounts of data.

9.3.2 Economic

As mentioned above, TinyML applications do not have to send data over the internet (like previous solutions) and would therefore be less expensive to operate. In addition, microcontrollers (the devices used in TinyML) are very cheap and do not require a large investment in them. Therefore, a rapid growth of TinyML applications is feasible.

9.3.3 Social

The development of TinyML will enable smart devices everywhere. New applications could be developed to improve the quality of life, such as medical applications or home automation devices.

9.4 Risks

9.4.1 Environmental

Microcontrollers are very small and cheap, so from a selfish perspective, it may be preferable to throw them away rather than reuse or recycle them.

9.4.2 Economic

It could occur that the production of microcontrollers will be reduced, and the cost will increase, making TinyML applications less economically viable. However, this is highly unlikely due to the worldwide dependence on microcontrollers and the low cost of production.

9.4.3 Social

TinyML applications are likely to be part of our lives in the near future. However, it will take a long time for poorer countries to see the benefits of these applications. In addition, the training of machine learning models may be biased against discriminated or marginalized groups. For example, a keyword spotting application might have more difficulty recognising an unusual accent. Another social problem of the massive use of TinyML applications is the dependency it will create towards this technology, which will be difficult to replace.

9.5 Weighted Matrix

Based on the above analysis, we have assigned a value to each cell of the sustainability matrix in order to quantify its impact. Figure 9.2 shows the matrix with the assigned values, where 1 means a very negative impact (or very high risk) and 10 a very positive impact (or very low risk).

TABLE 9.2: Weighted Sustainability Matrix

	PPP	Exploitation	Risks
Environmental	Consumption of the design = 10	Ecologial footprint = 7	Environmental = 3
Economic	Invoice = 9	Viability plan = 8	Economic = 9
Social	Personal impact = 9	Social impact = 9	Social = 5

10 Conclusion

This project had two objectives: The **first objective** aimed to develop a basic TinyML application. However, before starting the development process, we had to identify the common techniques used in TinyML (Section 4.1). We noticed that the most popular TinyML applications were based on either keyword spotting, visual wake words or anomaly detection. We found several commercial applications that are using these techniques in very different scenarios (e.g., domestic, office and industrial).

After identifying these techniques, we looked into the tools and frameworks that are currently being used for developing TinyML applications (Section 4.2). It was no surprise to discover that TensorFlow is the most popular approach for developing machine learning solutions. However, for TinyML we needed something more compact, since the memory is very limited. Thankfully, the TensorFlow Lite Micro is a very small version (16 kB) specifically designed to be deployed into microcontrollers.

For this project we had two microcontroller boards available: the Arduino Nano 33 BLE Sense, and the Espressif ESP32 (with LoRa). Both boards are supported by TensorFlow Lite Micro. At first, we had the intention to use both boards, depending on the requirements of each application to be developed. However, in the whole project we have only used the Arduino Nano 33 BLE Sense. Although the ESP32 board have a better microcontroller, the Arduino board comes with several sensors (e.g., microphone, IMU, temperature, etc.) that made it easier to set up the device (Chapter 5). Anyway, the Arduino board has had enough power to run all the TinyML applications we developed.

After identifying the techniques, the tools and the microcontroller specifications, we started to develop a basic keyword spotting application able to recognize the words

vermell, *verd* and *blau* (red, green and blue in Catalan). To develop the application, we followed two approaches (Chapter 6). In the first approach we analyzed the machine learning pipeline, and then we implemented the application based on the *micro speech* example from TensorFlow. In the second approach, we used the Edge Impulse platform. In both approaches, we obtained a very high accuracy when training the model (above 94%), however, the performance after deploying the application was significantly lower. This low performance was due to the quality of the dataset and the different alignments of the recorded keywords. We conclude that for creating a typical TinyML application, it is much easier and faster to use the Edge Impulse platform (or any similar), as it does not require deep knowledge of machine learning nor embedded systems, and most of the work is automatized.

The **second objective** aimed to identify advanced research directions of TinyML, and then, to develop some advanced application. Our first goal was to discover the feasibility of on-device model training (Chapter 7). The training phase usually is the most computationally expensive, and therefore, typical TinyML applications prefer to perform this phase before deploying the model into the microcontroller. However, to personalize and improve the application, it is necessary to train the model once it is already deployed. Therefore, we developed a program that is able to train a keyword spotting model in a microcontroller following an online learning approach. We had to use a custom neural network interface because the TFLite Micro framework is not able to train models. The results were very positives considering the online learning approach. The loss function dropped below 0.05 after few training epochs.

Another research direction we explored was federated learning with microcontrollers (Chapter 8). After analyzing this approach, we decided to develop a federated learning application that would be able to train the same keyword spotting model, but using multiple microcontrollers. The application followed a centralized orchestration and the FedAVG aggregation. Therefore, in addition to the Arduino firmware, we also had to implement the server that would coordinate the devices in the training process. The most complex part to implement was the communication between the server and the clients in order to exchange the models' parameters. The results

of the Independent and Identically Distributed (IID) scenario were very positive, obtaining a loss value similar to the previous application. For the non-IID scenario, the loss values of the local models increase greatly with each aggregation. However, in the long term, the loss value of the global model decreases.

We conclude that it is feasible to successfully deploy advanced machine learning applications on microcontrollers, and therefore, we expect that TinyML will be very important for the future of machine learning. We hope that this thesis has helped to understand the field of TinyML and the applications that can be developed. Future work will further explore the raised scenario of non-IID data in federated learning to understand deeper the capacity to obtain versatile models with limited local training data. In addition, decentralized orchestration in federated learning with microcontrollers using wireless communication, and different aggregation techniques to update the global model will be explored.

Bibliography

[1] Tim Stack. *Internet of Things (IoT) Data Continues to Explode Exponentially. Who Is Using That Data and How? - Cisco Blogs*. 2018. URL: https://news-blogs.cisco.com/datacenter/internet-of-things-iot-data-continues-to-explode-exponentially-who-is-using-that-data-and-how.

[2] Ben Lutkevich. *Microcontroller (MCU)*. 2019. URL: https://internetofthingsagenda.techtarget.com/definition/microcontroller.

[3] Vijay Janapa Reddi, Laurence Moroney, and Pete Warden. *Tiny Machine Learning (TinyML)*. 2021. URL: https://www.edx.org/professional-certificate/harvardx-tiny-machine-learning.

[4] Karen Hao. *What is machine learning?* 2018. URL: https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/#:~:text=What%20is%20the%20definition%20of,into%20a%20machine%2Dlearning%20algorithm..

[5] PayScale. *Average Project Manager, Information Technology (IT) Salary in Spain*. URL: https://www.payscale.com/research/ES/Job=Project_Manager%2C_Information_Technology_(IT)/Salary.

[6] PayScale. *Average Research Scientist Salary in Spain*. URL: https://www.payscale.com/research/ES/Job=Research_Scientist/Salary.

[7] PayScale. *Average Software Developer Salary in Spain*. URL: https://www.payscale.com/research/ES/Job=Software_Developer/Salary.

[8] Agencia Tributaria. *Tabla de coeficientes de amortización lineal*. URL: https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml.

[9] TensorFlow. *TensorFlow Lite Guide*. URL: https://www.tensorflow.org/lite/guide.

[10] Wikipedia. *Single-board microcontroller*. URL: https://en.wikipedia.org/wiki/Single-board_microcontroller.

[11] Pete Warden. *Extract Loudest Section*. 2018. URL: https://github.com/petewarden/extract_loudest_section.

[12] Kelvin. *Model Compression via Pruning*. 2020. URL: https://towardsdatascience.com/model-compression-via-pruning-ac9b730a7c7b.

[13] FPL. *FlatBuffers*. URL: https://google.github.io/flatbuffers/.

[14] Marla Rosner. *Transfer Learning & Machine Learning: How It Works, What It's Used For, and Where it's Taking Us*. 2018. URL: https://www.sparkcognition.com/transfer-learning-machine-learning/.

[15] Ralph Heymsfeld. *A neural network for arduino*. URL: http://robotics.hobbizine.com/arduinoann.html.

[16] Wikipedia. *Online machine learning*. URL: https://en.wikipedia.org/wiki/Online_machine_learning.

[17] Jason Brownlee. *Understand the Impact of Learning Rate on Neural Network Performance*. 2020. URL: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[18] Mircea Diaconescu. *Optimize Arduino Memory Usage*. 2015. URL: https://web-engineering.info/node/30.

[19] Yuang Jiang. *Model Pruning Enables Efficient Federated Learning on Edge Devices*. 2020. URL: https://arxiv.org/abs/1909.12326.

[20] Olivia Choudhury et al. *Differential Privacy-enabled Federated Learning for Sensitive Health Data*. 2020. URL: https://arxiv.org/abs/1910.02578.

[21] Enmao Diao, Jie Ding, and Vahid Tarokh. *HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients*. 2020. URL: https://arxiv.org/abs/2010.01264.

[22] Yuan Ko. *Federated Learning Aggregate Method (1): FedSGD v.s. FedAVG*. 2020. URL: https://medium.com/disassembly/federated-learning-aggregate-method-1-c2f96bc03f59.

[23] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. 2017. arXiv: 1602.05629 [cs.LG].

[24] Brendan McMahan and Daniel Ramage. *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. 2017. URL: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[25] Karen Hao. *How Apple personalizes Siri without hoovering up your data*. 2019. URL: https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/.

[26] ODSC - Open Data Science. *What is Federated Learning?* 2020. URL: https://medium.com/@ODSC/what-is-federated-learning-99c7fc9bc4f5.

[27] ODSC - Open Data Science. *What is Federated Learning?* 2020. URL: https://medium.com/@ODSC/what-is-federated-learning-99c7fc9bc4f5.

[28] Ashwani Gupta. *How Federated Learning is going to revolutionize AI*. 2019. URL: https://towardsdatascience.com/how-federated-learning-is-going-to-revolutionize-ai-6e0ab580420f.

[29] Nicola Rieke et al. *The future of digital health with federated learning*. 2020. URL: https://www.nature.com/articles/s41746-020-00323-1#citeas.