Tiny Machine Learning Environment:

Enabling Intelligence on Constrained Devices

FOUAD SAKR

Department of Electrical, Electronics, Telecommunications Engineering and Naval Architecture (DITEN) - University of Genoa

&

School of Electronic Engineering and Computer Science (EECS) - Queen Mary University of London

This dissertation is submitted for the degree of

Doctor of Philosophy

JD ICE - Cycle XXXV

March 2023

Tiny Machine Learning Environment: Enabling Intelligence on Constrained Devices

Fouad SAKR

Joint Doctorate in Interactive and Cognitive Environments
JD ICE

Cycle XXXV

Acknowledgement

This dissertation was prepared within the framework and according to the rules of the Joint Doctorate in Interactive and Cognitive Environments (JD ICE) in collaboration with the following universities:

Università degli Studi di Genova (UNIGE)

DITEN - Dept. of Electrical, Electronic, Telecommunications Engineering and Naval Architecture

ELIOS - Electronics for the Information Society

Primary Supervisor: Prof. Francesco BELLOTTI
Secondary Supervisor: Prof. Riccardo BERTA

Queen Mary University of London (QMUL)

EECS - School of Electronic Engineering and Computer Science

CSI - Centre for Intelligent Sensing

Supervisor: Dr. Joseph DOYLE

This dissertation is dedicated to the memory of my uncle and namesake, Fouad Sakr. Although he inspired me to do my PhD, he was not able to see me graduate. There is no quotation that does him justice, nor is there a text sufficient to speak of him. This is for him, he is not forgotten..

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations.

FOUAD SAKR
March 2023

Acknowledgements

First and foremost, I am extremely grateful to my thesis advisors Prof. Francesco Bellotti, Prof. Riccardo Berta, and Dr. Joseph Doyle for their invaluable advice, continuous support, and patience throughout my doctoral studies. Their immense knowledge and rich experience encouraged me in all phases of my academic research and daily life. I would like to extend my sincere thanks to the laboratory director Prof. Alessandro De Gloria and the coordinators of the Joint Doctoral Program in Interactive and Cognitive Environments (JD ICE), Prof. Carlo Regazzoni, Prof. Lucio Marcenaro and Dr. Riccardo Mazon, for their support at every stage of the research project. Special thanks go to the independent assessor Dr. Gianni Antichi for the smooth running of the review process throughout the PhD stages. I would like to thank the University of Genoa and Queen Mary University of London for giving me the opportunity to complete my PhD studies. Finally, I am deeply grateful to my family and friends for their support, appreciation, encouragement and keen interest in my academic achievements.

Abstract

Running machine learning algorithms (ML) on constrained devices at the extreme edge of the network is problematic due to the computational overhead of ML algorithms, available resources on the embedded platform, and application budget (i.e., real-time requirements, power constraints, etc.). This required the development of specific solutions and development tools for what is now referred to as TinyML. In this dissertation, we focus on improving the deployment and performance of TinyML applications, taking into consideration the aforementioned challenges, especially memory requirements.

This dissertation contributed to the construction of the Edge Learning Machine environment (ELM), a platform-independent open source framework that provides three main TinyML services, namely shallow ML, self-supervised ML, and binary deep learning on constrained devices. In this context, this work includes the following steps, which are reflected in the thesis structure. First, we present the performance analysis of state of the art shallow ML algorithms including dense neural networks, implemented on mainstream microcontrollers. The comprehensive analysis in terms of algorithms, hardware platforms, datasets, pre-processing techniques, and configurations shows similar performance results compared to a desktop machine and highlights the impact of these factors on overall performance. Second, despite the assumption that TinyML only permits models inference provided by the scarcity of resources, we have gone a step further and enabled self-supervised on-device training on microcontrollers and tiny IoT devices by developing the Autonomous Edge Pipeline (AEP) system. AEP achieves comparable accuracy compared to the typical TinyML paradigm, i.e., models trained on resource-abundant devices and then deployed on microcontrollers. Next, we present the development of a memory allocation strategy for convolutional neural networks (CNNs) layers, that optimizes memory requirements. This approach reduces the memory footprint without affecting accuracy nor latency. Moreover, e-skin systems share the main requirements of the TinyML fields: enabling intelligence with low memory, low power consumption, and low latency. Therefore, we designed an efficient Tiny CNN architecture for e-skin applications. The architecture leverages the memory allocation strategy presented earlier and provides better performance than existing solutions. A major contribution of the thesis is given by CBin-NN, a library of functions for implementing extremely efficient

binary neural networks on constrained devices. The library outperforms state of the art NN deployment solutions by drastically reducing memory footprint and inference latency. All the solutions proposed in this thesis have been implemented on representative devices and tested in relevant applications, of which results are reported and discussed. The ELM framework is open source, and this work is clearly becoming a useful, versatile toolkit for the IoT and TinyML research and development community.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The Internet of Things (IoT) is a growing field that is driving the expansion of ubiquitous and interconnected devices in a variety of areas, including wearable, smart cities, and infrastructure [1]. The increasing market demand for IoT has led to its deep integration into our daily lives. With this proliferation, large volumes of heterogeneous devices are producing enormous amounts of data, with machine learning (ML) and deep learning (DL) algorithms playing a prominent role in processing them. For example, Gartner [7] estimated a total of 14.2 billion functioning IoT devices (also called things) by the end of 2020, while an estimate of 25 billion was made for the end of 2021. Comparing the time span, a significant increase in operational smart devices can be seen. In quantifying the generated data, [8] predicts a tremendous 73.1 zettabytes of IoT data by 2025. In the past, due to the severe resource and processing constraints of IoT devices, integrating cloud computing (CC) has been a promising approach, as CCs have significant resources to support resource-scarce devices in terms of processing, storage, and memory [9]. Nevertheless, the geographical shift between IoT devices and CC introduces delays that hinder applications with real-time constraints, thus affecting the quality of service.

In addition, the growing volume and complexity of data raises concerns about dependence on a particular provider, i.e., Vendor lock-in [10]. Moreover, a device whose intelligence is based on the cloud has all the problems that an Internet connection brings [11]. Aside from the time it takes for the request to reach the cloud and the response to be received, and the network bandwidth required to transfer this data, there are also privacy and security concerns. Therefore, there has been a paradigm shift to process data close to its source, i.e., smart objects. This new paradigm, known as edge computing, aims to overcome the disruptive challenges of CC by providing storage and processing within the Radio Access Network [12]. According to Shi *et al.* [13] edge computing acts as an intermediary between data sources and cloud computing. It can be inferred from this that there is a mutual dependency between

the two paradigms when data traffic is regularly transferred to the cloud in order not to burden edge resources with large volumes of data. One measure that could mitigate frequent access to the cloud is to integrate the processing techniques available in the cloud, such as ML/DL, at the edge. Providing local cognition will allow smart devices to demonstrate advanced capabilities even when they are not connected and show traces of intelligence; a kind of "Internet of Conscious Things" [14]. However, several technological barriers (e.g., lack of infrastructure and networks, cost per unit) hinder the broader potential impact of edge computing.

In exploring alternatives for expanding IoT networks, research has led to the use of the microcontroller unit (MCU) class of economical devices for processing at the absolute edge of the network [15]. According to [16], moving computations to the extreme edge, i.e., sensors and actuators, improves latency and autonomy. Figure 1.1 shows a clear comparison of resources, size, energy consumption, and latency in each of the aforementioned paradigms. In particular, research efforts are now focused on integrating Tiny Machine Learning (TinyML) as a workaround to alleviate some of the computational burden associated with pure-edge systems, which is critical to enabling intelligence on such constrained devices and mitigating the drawbacks of CC.

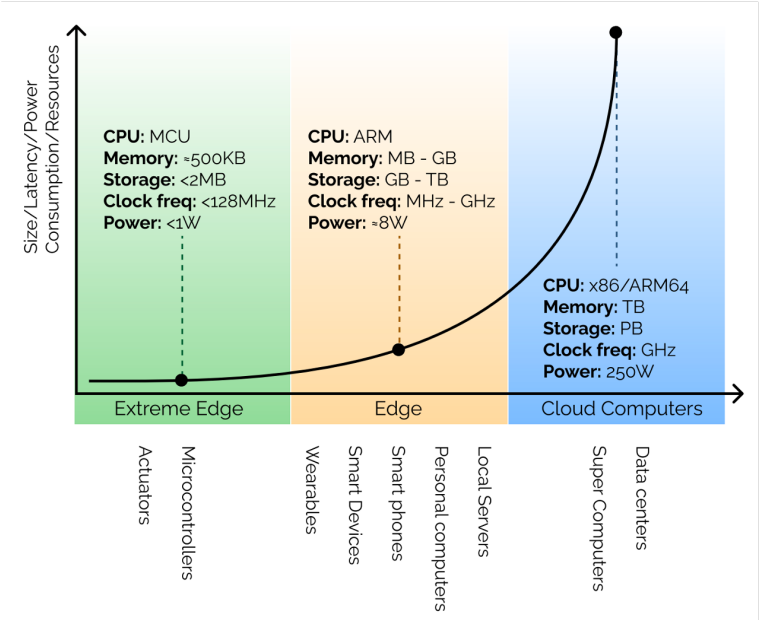

Fig. 1.1 Comparison of paradigms [1].

TinyML is an emerging technology and is deemed as the future of hyper-digitization [2], where a large number of MCUs are used to implement artificial intelligence (AI) at the extreme edge using interfacing systems like sensors and actuators. With approximately

300 billion microcontrollers [17], leveraging existing idle resources for cognitive tasks is a more cost-effective and energy-efficient way to reduce reliance on the cloud. For example, Warden *et al.* [18] note that most MCUs operate in the milli-Watt range, and the industry has agreed that similar power consumption arbitrarily means that an MCU can run for more than a year on a coin cell battery. Coupled with their inexpensiveness, TinyML enables a myriad of new, low-cost, battery-powered ML applications. Furthermore, there are a handful of solutions that contradict the stereotypical notion that ML/DL requires enormous resources for processing, paving the way for the use of TinyML on such constrained devices.

The TinyML paradigm is still in its early stages and requires appropriate adaptation to existing IoT frameworks. Pioneering research shows that the TinyML approach is critical for the development of smart IoT applications. However, the lack of supporting tools hinders the development and integration of TinyML into such applications. With this in mind, this dissertation aims to provide an efficient environment, namely Edge Learning machine (ELM), for hosting TinyML applications on resource-scarce devices. The environment features three main services. First, it includes modules for (1) training shallow ML algorithms, including dense neural networks, to find the best configurations and generate the best models, and (2) reading the generated models and performing inference on mainstream microcontrollers. Second, ELM hosts the Autonomous Edge Pipeline System for self-supervised ML. AEP performs resilient training and classification on the device and leverages the vast amounts of unlabeled IoT data. And third, ELM supports the implementation of the extremely efficient binarized neural network on tiny devices with the developed CBin-NN library. Overall, ELM also uses various compression techniques such as quantization and binarization, and proposes several optimization techniques such as memory replacement strategy and memory filtering to reduce the memory requirement and computational complexity appropriately.

1.1 Objectives and Contributions

The main goal of this dissertation is to enable the efficient deployment of various machine and deep learning applications on constrained devices by (i) investigating and improving established approaches, and (ii) integrating these techniques into an effective TinyML environment. This environment should support inferencing on low-resource devices, considering challenges such as preserving accuracy, memory requirements, computational complexity, and latency, etc. To achieve our objective, we contributed to the TinyML research community with several deployment services, which can be summarized as follows:

- A machine learning platform for constrained devices is proposed. It is the basic part of the ELM environment and consists of two modules, namely Desk-LM for training

and Micro-LM for inference. The framework has been extensively validated on various devices and datasets to answer a number of research questions investigating the performance of microcontrollers in typical ML Internet of Things (IoT) applications.

- An Autonomous Edge Learning and Inferencing Pipeline (AEP) system is presented. AEP is a software system that we developed for resource-constrained devices. In addition to inferencing capability, AEP enables autonomous on-device training. AEP demonstrates similar performance to models trained in the cloud on actual labels and then deployed on microcontrollers.
- A memory replacement strategy that reuses memory allocated to convolutional layers is proposed to reduce the overall memory consumption of the Convolutional Neural Network (CNN). The proposed strategy reduces memory consumption when applied to an existing TinyML library.
- A tiny embedded CNN architecture (TinyCNN) for e-skin applications is designed. The architecture is evaluated on a robotic touch modality classification task. Compared to existing solutions targeting the same problem, the TinyCNN requires a smaller number of parameters and FLOPS, while achieving a slight improvement in accuracy. TinyCNN also achieves real-time latency with low energy consumption.
- A CBin-NN inference engine is proposed. This is a library for executing binarized neural networks (BNNs) on resource-constrained devices, namely microcontroller units (MCUs). CBin-NN outperforms state-of-the-art TinyML deployment solutions in terms of memory footprint and latency at the expense of a slight decrease in accuracy.

1.2 Dissertation Outline

1.2.1 Background

This chapter explores the TinyML field, which focuses on optimizing machine learning workloads so that they can be processed on IoT devices. First, the benefits of such a computing paradigm are presented. We then discuss the limitations that hinder the deployment of machine and deep learning algorithms on such constrained devices and the optimization techniques that can mitigate these challenges. In addition, we introduce in this chapter the wide range of applications that are possible with TinyML. We also provide an overview of the available frameworks published by the research community and industry that support the

use of TinyML. Finally, we give an overview of the progress of TinyML in various research areas and applications.

1.2.2 Machine Learning on Mainstream Microcontrollers

This chapter presents the first framework in the Edge Learning Machine (ELM) environment. It is a baseline machine learning framework for constrained devices, which manages the training phase on a desktop computer and performs inferences on microcontrollers. The framework implements, in a platform-independent C language, three supervised shallow machine learning algorithms (Support Vector Machine (SVM) with a linear kernel, k-Nearest Neighbors (K-NN), and Decision Tree (DT)), and exploits STM X-Cube-AI to implement Artificial Neural Networks (ANNs) on STM32 Nucleo boards. We investigated the performance of these algorithms on six embedded boards (STM32 F0, F3, F4, F7, and L4) and six datasets (four classifications and two regression). Our analysis, which aims to provide a quantitative analysis of the performance of common ML algorithms on constrained devices, shows that the target platforms allow us to achieve the same performance score as a desktop machine, with similar latency. In general, several factors impact performance in different ways across datasets. This highlights the importance of such framework, which can train, compare, and deploy different algorithms.

1.2.3 Self-Learning Pipeline for Low-Energy Resource-Constrained Devices

In this chapter, the self-learning Autonomous Edge Learning and Inferencing Pipeline (AEP) is presented. AEP is a deployable system designed for devices with limited resources that can be used for periodic self-supervised on-device training and classification. AEP uses two complementary approaches: pseudo-label generation with a confidence measure using soft k-means clustering and periodic training of one of the supported classifiers, namely decision tree (DT) and k-nearest neighbor (k-NN), exploiting the pseudo-labels. We tested the proposed system on two IoT datasets. The AEP, running on the STM NUCLEO-H743ZI2 microcontroller, achieves comparable accuracy levels as same-type models trained in the cloud using the actual labels and then deployed on the microcontrooler. This chapter makes an in-depth performance analysis of the system, particularly addressing the limited memory footprint of embedded devices and the need to support remote training robustness.

1.2.4 Memory-Efficient CMSIS-NN with Replacement Strategy

In this chapter, we propose an in-place computation strategy to reduce the memory requirements of convolution neural network inference. The strategy exploits the MCU single-core architecture, with sequential execution. Experimental analysis using the CMSIS-NN library on the CIFAR-10 dataset shows that the proposed optimization method can reduce the memory required by a neural network model by more than 9%, without impacting the execution performance or accuracy. This can be further reduced in deeper network architectures.

1.2.5 A Tiny CNN for Embedded Electronic Skin Systems

This chapter presents a tiny Convolution Neural Network (TinyCNN) architecture suitable for deployment on an off-the-shelf commercial microcontroller in compliance with the e-skin requirements. The training, optimization, and implementation of the proposed CNN are presented. The TinyCNN implementation is optimized through layer fusion and buffer reuse strategies for efficient inference on constrained edge devices. Experimental analysis of a touch modality classification task demonstrates that the proposed CNN-based system is capable of processing tactile data in real-time while reducing the model size by up to 65% compared to comparable existing solutions.

1.2.6 CBin-NN: Inference Engine for Binarized Neural Networks on Constraint Devices

To support the effective deployment of Binarized Neural Networks (BNNs) on scarce devices, we propose the CBin-NN inference engine. It is a library of operators that enables the creation of simple yet flexible convolutional neural networks (CNNs) with binary weights and activations. CBin-NN is platform-independent, thus portable to virtually any software-programmable device. We tested the library by implementing a BNN for robotic touch classification on a Cortex-M7 microcontroller. Experimental results show the validity of the proposed library, with improvements over the state of the art in terms of accuracy (+2%), memory footprint (+15%), and latency (+25%). Moreover, further analysis of the CIFAR-10 dataset shows that using some specific optimizations, our library outperforms the state of the art in TinyML solutions based on 8-bit quantization, both in terms of speed and memory usage, but at the cost of a slight loss in accuracy.

1.3 List of Publications

This dissertation has resulted in several publications, which are listed below. An assignment of the corresponding publications to the individual chapters of the dissertation is shown in Table 1.1:

Table 1.1 Mapping of the related publications to the individual thesis chapters [6]

Chapter	Publication
Chapter 1	J1, J2, J3, C1, C2, BC1, BC2
Chapter 2	-
Chapter 3	J3
Chapter 4	J2, BC2
Chapter 5	C2
Chapter 6	BC1
Chapter 7	J1, C1
Chapter 8	J1, J2, J3, C1, C2, BC1, BC2

Journals:

J1. F. Sakr, R. Berta, J. Doyle, A. De Gloria, and F. Bellotti, "CBin-NN: inference engine for binarized neural networks on resource-constraint devices", IEEE Internet of Things Journal, (under review).

J2. F. Sakr, R. Berta, J. Doyle, A. De Gloria, and F. Bellotti, “Self-learning pipeline for low-energy resource-constrained devices,” Energies, vol. 14, no. 20, p. 6636, 2021.

J3. F. Sakr, F. Bellotti, R. Berta, and A. De Gloria, “Machine learning on mainstream microcontrollers,” Sensors, vol. 20, no. 9, p. 2638, 2020.

Conference Papers:

C1. F. Sakr, R. Berta, J. Doyle, H. Younes, A. De Gloria, and F. Bellotti, "Memory Efficient Binary Convolutional Neural Networks on Microcontrollers," in 2022 IEEE International Conference on Edge Computing and Communications (EDGE), Barcelona, Spain, 2022 pp. 169-177.

C2. F. Sakr, F. Bellotti, R. Berta, A. De Gloria, and J. Doyle, “Memory-efficient cmsis-nn with replacement strategy,” in 2021 8th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, 2021, pp. 299–303.

Book Chapters:

BC1. F. Sakr, H. Younes, R. Berta, J. Doyle, A. De Gloria, and F. Bellotti, "A tiny CNN for embedded electronic skin systems," in 6th International Conference on System-Integrated Intelligence (SysInt), Genoa, Italy. Springer, 2022.

BC2. F. Bellotti, R. Berta, A. D. Gloria, J. Doyle, and F. Sakr, "Exploring unsupervised learning on stm32 f4 microcontroller," in International Conference on Applications in Electronics Pervading Industry, Environment and Society. Springer, 2020, pp. 39–46.

Chapter 2

Background

Recently, the Internet of Things (IoT) has attracted a lot of attention as IoT devices are being deployed in various fields. As data generated by these devices continues to grow, finding IoT solutions that will reduce the burden of accessing and transferring data to the cloud is crucial. One remedy to circumvent these drawbacks is to integrate ML/DL on IoT devices, i.e. TinyML [19]. This is an effective tool that extends the device's data processing capabilities in constrained environments and eliminates the need for frequent access to already overloaded cloud services. TinyML enables constrained devices with intelligence, meaning that an MCU connected to sensors can locally run ML algorithms to make predictions based on sensor data and accelerate decision making. Despite the relatively under exploited field of machine learning integrated into MCUs, early successes incorporating TinyML into IoT devices demonstrate the field's potential for implementing ML at the deepest edge of the Internet of Things. The TinyML workflow in the IoT environment is shown in Figure 2.1.

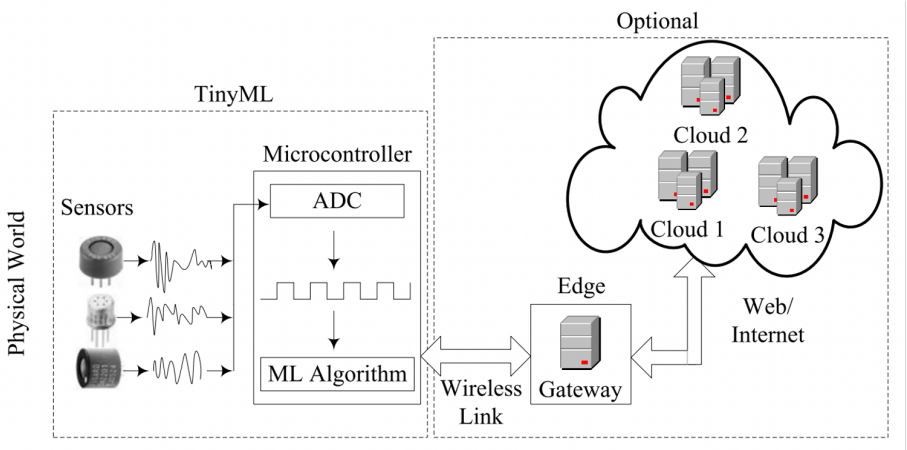

Fig. 2.1 TinyML workflow in IoT [2].

The aim of TinyML is not to replace cloud computing, but rather to provide the end device with certain computational capabilities [20]. Several IoT devices can be coupled together to achieve TinyML services, each performing its analysis as part of a larger system and regulating the information sent to the cloud. It is also possible to isolate the devices from the IoT ecosystem by assuming that each model operates independently without the need for any other IoT layers. In this context, the goal of this chapter is to provide an overview of the TinyML revolution in terms of devices, optimization techniques, frameworks, and applications to illustrate the state of the art and present development requirements.

2.1 Internet of Things

The Internet of Things refers to billions of devices connected to the Internet, which collect and share data from the field [21]. Typical IoT architectures consist of a set of technological layers, bringing scalability, modularity, and configurability (e.g., [22, 23]). Figure 2.2 illustrates the four-tier IoT architecture. The first layer (namely, the perception layer) consists of smart objects equipped with sensors, such as fiber optic sensors (FOS), micro-electromechanical systems (MEMS), sensors, radio frequency identification (RFID) sensors, etc. The second layer is the network layer. Current networks, often connected with very different protocols, have been deployed to support machine-to-machine (M2M) networks and their applications. There may be a local area network (LAN), such as Ethernet and Wi-Fi connections, or a personal area network (PAN) such as ZigBee, Bluetooth, and ultra wideband (UWB). Sensors that require low power and low data rate connectivity typically form networks commonly known as wireless sensor networks (WSNs). Besides the sensor aggregators, the connection to backend servers/applications in the cloud is made over wide area networks (WAN) such as GPRS, LTE, and 5G. These protocols are widely used for IoT, including constrained application protocol (CoAP), message queuing telemetry transport (MQTT), extensible messaging and presence protocol (XMPP), etc. The management layer includes data storage, management, and processing. The data management infrastructure includes common data stores such as relational or non-relational databases, distributed file systems such as the Hadoop Distributed File System (HDFS), etc. The data processing techniques are generally based on machine learning or artificial intelligence algorithms and pattern recognition. Finally, the application layer is responsible for providing services to the user using the IoT field data. The IoT application covers various fields, such as transportation, smart cities, agriculture, industry, healthcare, and environment.

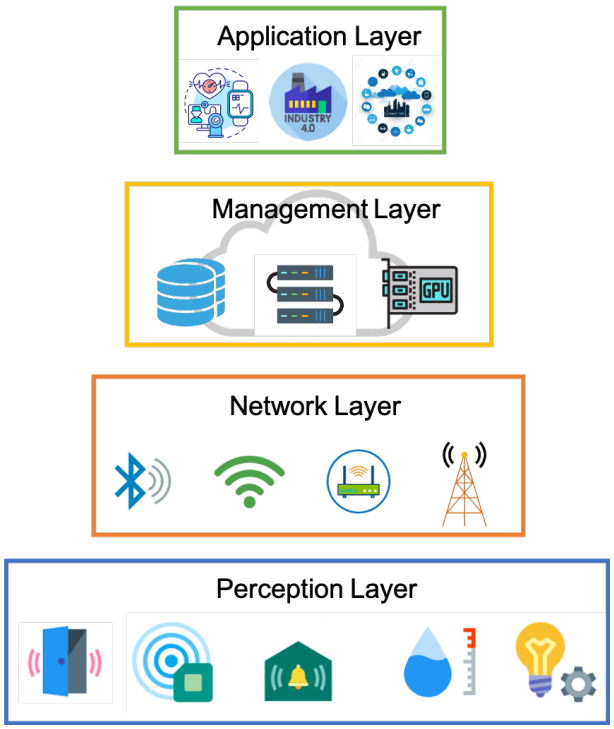

Fig. 2.2 Four-tier IoT architecture.

The overall application latency is typically dominated by the propagation delay between the edge and the data centers on the cloud. Another significant performance penalty factor is given by packet losses [24]. This, together with consideration about bandwidth, energy consumption, and privacy, suggests the importance of edge computing, thus keeping the computation as much as possible close to the information source [25]. Reference [26] singles out four main characteristics of IoT data in cloud platforms: multi-source high heterogeneity, huge scale dynamic, low-level with weak semantics, and inaccuracy. These characteristics are important, as they highlight key features that should be provided by an effective IoT data framework (e.g., source characterization, variety of source data configurations/aggregation, outlier computation) [27]. IoT data characteristics are largely dependent on delay, incompletion, and dynamic variation.

2.2 Machine Learning

Machine learning is a field of research concerned with understanding and developing methods that "learn," i.e., methods that use data to improve performance on a range of tasks. It is considered part of artificial intelligence. In Machine Learning, models are built based on sample data, referred to as training data, to make predictions or decisions without explicit

programming [28]. A key concept to remember when discussing the topic of ML is that ML is not a single algorithm. In order to create a ML model, we must select from a set of algorithms. Apart from the numerous projects and domains in which they have already been applied, we also cannot make any assumptions about how the ML algorithm will perform. The well-known no-free-lunch theorem [29] states that: *Even if we know that a particular algorithm has performed well in a particular domain, we cannot say that it will also perform well on our problem until we build and test a model.* Of all algorithms, the most commonly used/known are (i) deep learning algorithms, including but not limited to artificial neural networks (ANN), convolutional neural networks (CNN), recurrent neural networks (RNN), transformers, and (ii) machine learning algorithms such as support vector machines (SVM), nearest neighbors, Naive Bayes, and decision trees. It is also important to note that ML models rely heavily on data. During the learning phase (Subsection 2.2.1.1), the quality of the data we feed the model is crucial. Thus, the extraction and cleaning of data is the most time-consuming part of model development, as the quality of the data is critical to the performance of the algorithm. Model creation and optimization are only a small part of the entire process.

Two main blocks can be distinguished in the development of machine learning. Model building is the first step, where data is fed into the ML algorithm and a model is created. Inference is the second block, in which data is fed into the model and the model produces an output. These two blocks are not strictly independent, but neither are they dependent on each other. At the end of the model creation block, the algorithm provides a representation of the data. After that, the inference block uses the data representation to obtain the outputs. For the vast majority of ML algorithms, the inference results are generally not used to rebuild or improve the model, but there are some exceptions [30, 31]. Thus, the inference block can start after the model is created, and the inference results can rebuild the model, but there is a separation between the two blocks that allows them to run on different machines.

2.2.1 Model Building

Building a machine learning model can involve several steps, but we will focus only on the two most important ones: the training/learning phase and the validation/simulation phase. The learning phase is the most critical phase, as it is responsible for creating the data representation. As mentioned earlier, the data used in the learning phase must be of high quality to produce a meaningful data representation. In the learning phase, the computational resources required to achieve the goal are very high. The number of computations is enormous; the memory consumption is large (and increases with the complexity of the data); the processing power and the time to complete the learning phase are inversely proportional.

On the other hand, the performance of the model is demonstrated in the validation phase. During the construction of a ML model, the primary dataset is divided into two parts (usually 75/25), with 75% of the data used in the learning phase and 25% in the validation phase.

2.2.1.1 Learning Phase

An ML algorithm can be classified according to its learning nature. The nature of the learning describes whether the algorithm requires a primary dataset and whether the dataset must contain the expected output. If the algorithm is a supervised algorithm (supervised learning) [28, 32, 33], the algorithm requires a primary dataset, and all instances must have the corresponding output. If the algorithm does not need a primary dataset to have the outputs, and the algorithm intends to group the data into subsets, we say that the algorithm uses unsupervised learning [28, 32, 33]. However, sometimes it can be time consuming and costly to hire someone to label the data. To reduce the cost and time of labeling the data, it is possible to label only a portion of the data set and then have the algorithm divide the remaining data into each class. This method is called semi-supervised learning [28, 32, 33]. Another approach that is used to reduce the data labeling cost is self-supervised learning where a supervised task is created out of the unlabelled data. Moreover, the algorithm may not need a primary dataset; learning is incremental, and the machine learns by trial and error as it collects the data. Each time the machine does something wrong, it receives a penalty; each time it does something right, it receives a reward. The term that describes this technique is reinforcement learning [30]. Different algorithms have different learning dynamics. The learning dynamics of the algorithm provide information about its plasticity and the way it handles the primary dataset. If the algorithm learns in a single batch, then to improve or reconstruct the model, the developer must start from scratch. Thus, once the model is created, the algorithm cannot learn anything new; this is referred to as batch learning [34]. If the algorithm can learn on-the-fly, i.e., it can be built and rebuilt as it receives new data, this is referred to as online learning [34].

2.2.1.2 Validation Phase

In the validation phase of the model, the developer can determine how well the algorithm has processed the data. Until the end of this phase, we cannot make any assumptions about the performance of the algorithm. The standard metric for describing the performance of the algorithm is overall accuracy. This metric is the percentage of correct classifications versus incorrect classifications. However, using other metrics can provide better insight into the overall performance of the model. Some of these metrics are: Confusion matrix, area

under the ROC curve, precision, recall, f1-score, logarithmic loss, mean absolute/square error, coefficient of determination (R-squared), etc. [35, 36].

2.3 Tiny Machine Learning

Embedding machine learning in resource-constrained and battery-powered applications has some great advantages, but also comes with some limitations. Fortunately, several optimization techniques have been proposed in the literature to mitigate these constraints and contribute to the expansion of this field. Various applications are enabled by the use of these techniques, and extensive research and industry efforts are contributing to the development of tools that enable such applications. In the following subsections, we will discuss all of these points and provide a broad overview of the TinyML field.

2.3.1 Benefits

TinyML enables data analysis that was previously not possible in confined environments, close to the sensors. Because of its versatility, cost-effectiveness, and simplicity, TinyML has the potential to transform the entire IoT ecosystem. The key performance indicators that certify TinyML as an effective and essential tool can be characterized as follow [37, 38]:

- **Change from dumb to smart IoT devices:** Sensor systems generate large amounts of raw data that are challenging to process through cloud computing. As a result, most data is not transferred to the cloud but is lost at the edge of the network. TinyML provides a way to analyze this data in a low-resource environment by integrating ML algorithms into each IoT device [2]. For example, 5 GB of data is generated every second in the Boeing 787, and 1 GB per second in a smart car [2]. Using machine learning algorithms on each IoT device, rather than uploading all raw data to the cloud, these IoT devices could capture sensitive information and remove junk data.

- **Network Bandwidth** Data ingestion, information processing, and machine learning algorithms require a sensor-to-gateway network, gateways, and cloud services [11]. It is possible to redefine these requirements by reducing the ubiquity of the cloud and making other IoT layer services optional through innovative approaches to TinyML in resource-constrained environments. TinyML offers greater independence and lower transmission and bandwidth than standard IoT services. Raw data transmission in an IoT framework with dense devices has high bandwidth requirements [2]. As a result, pre-processing raw data at the edge of the network and limiting transmission

to important metadata can significantly reduce bandwidth requirements which on the other hand reduces the cost. This can be further illustrated using the Boeing 787 and smart car examples.

- **Security and Privacy** Data security is a factor that negatively impacts IoT adoption as large amounts of private data are transferred to the cloud [39, 40]. Using third-party providers for IoT services leaves the end user unclear as to where their personal information is stored and who owns it [2]. Additionally, malicious sources may intercept data when it is transferred. As a result, preventing data flow and restricting it to the device improves security and privacy. The data in TinyML is not transmitted (or is transmitted less), which makes it less vulnerable to attacks. Due to this, TinyML includes data security and privacy features by default.

- **Latency** The casual sequence of events in an IoT system starts with the transmission of sensor data from edge devices to cloud servers and ends with the reception of decisions/predictions calculated there [41]. Clearly, this approach introduces a huge delay and therefore requires analysis close to the device. An effective approach to solve this problem is to use TinyML. Furthermore, waiting for the cloud to make a decision in safety-critical systems such as healthcare (e.g., robotic surgery) and autonomous vehicles can be disastrous [2]. By reducing the reliance on external communications, TinyML will reduce the latency of delivering ML services in such situations.

- **Reliability** The main problem with a cloud-dependent system is that it relies on an Internet connection for data transfer [11]. Once this connection is interrupted, delayed, or stopped, the system will not function properly. Therefore, data-driven computation in sensor networks is a highly desirable feature of the IoT. There is great interest in using TinyML for tasks in the field, e.g., offshore and in rural areas without cellular or Internet connectivity [2, 42]. Of course, it is important that an edge device is first tested for reliability (i.e., reliability assessment) before it is deployed in the field [41, 43]. This change will in turn increase the reliability of IoT services.

- **Energy Efficiency** This is another important metric for TinyML in MCUs. In an IoT ecosystem, the majority of devices are always-on and powered by batteries. Typically, these devices are powered by a coin battery (e.g., CR2032), and are expected to stay on for several months and sometimes even years [2, 39, 44]. As a result, it is often necessary for the device to remain mostly idle (sleep mode), where the MCU can observe the data and wake up when necessary. Often, transferring data can consume

more energy than providing ML services locally. To overcome these challenges, TinyML proves to be a useful tool.

- **Potential Crash** Short Time Inoperability (STI) is a common issue that occurs due to interruptions in cloud services or crashes in an IoT layer [2]. This STI can lead to a potential crash of the IoT structure. Unless an acceptable level of inoperability is achieved, the chance of the IoT surviving in a risk-critical environment is very low, regardless of its readiness. By using TinyML in the IoT, it would be possible to prevent STI by keeping analytics in the IoT device.
- **Data filtering** By analyzing the data within the IoT device, it is possible to eliminate residue. Developing an intelligent support system at the extreme edge can offer significant advantages over traditional systems when used in high-traffic use cases. For instance, a surveillance system designed to detect anomalies consists of multiple cameras where most of the information collected by the cameras is redundant. In such situations, it would make more sense to filter out the redundant images within the device instead of transferring everything to the cloud [2].

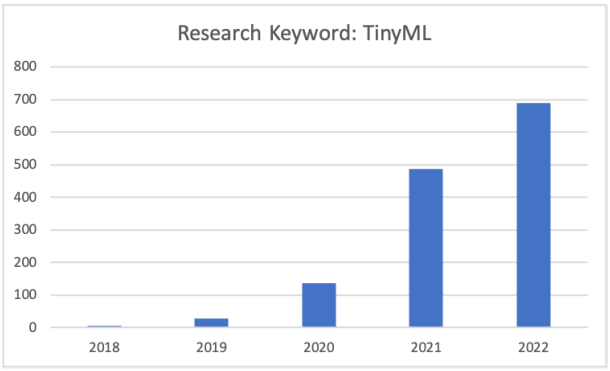

Fig. 2.3 Number of research papers for the keyword "TinyML" [3].

From this, we infer that TinyML solutions combined with cloud technology can increase the capacity of traditional cloud services and provide MCUs with new computational capabilities to improve the above performance indicators. TinyML is thus the new trend and has been heralded by many as the technology that will drive the AI revolution, as seen in Figure 2.3, which reflects the interest in TinyML.

2.3.2 Constraints

In the following section, we enumerate the two major problems in implementing a ML model in an MCU with scarce resources and briefly explain how they affect the performance and accuracy of the model:

- **Memory Footprint:** Limited memory capacity is one of the biggest challenges in implementing ML in resource-constrained MCUs. Most models need to store some data representations such as thresholds, hyperplanes, and data points; these data representations require large amounts of memory. Therefore, the developer must ensure that he can reduce the memory footprint without sacrificing accuracy. It is also important to know that there are different types of memory in modern MCUs. There is Flash, SRAM, DRAM, NVRAM and EEPROM memory. All of these memory types have advantages and disadvantages, but perhaps the most important is how much time it takes to access the data in them. SRAM and NVRAM are the fastest, but they are expensive; therefore, the size of these memory types is typically small to make the MCU affordable [45]. The lack of space hinders the adaptation of the models to the edge device and thus the growth of the TinyML field [1, 2, 46].

- **Processing Power:** Time is an obstacle in almost every system, whether it is real time or not. Even if a system is not real time it is likely it will have a deadline and users will characterize a service as having poor performance if this deadline is exceeded. This means that the program code needs to be executed in a time compatible with the application. One of the factors that make the algorithm run faster is the clock frequency. The clock frequency is what makes the hardware tick, so it determines how fast a processor can complete a task. However, we cannot estimate how much time an algorithm will take to run based on clock frequency alone. We also need to know how many clock cycles a machine cycle requires (Equation 2.1). A machine cycle is the time it takes the processor to execute these four steps: Fetch, Decode, Execute, and Store. Thus, one needs to consider the architecture and instructions used in addition to the clock speed as different architectures and instructions require different numbers of clock cycles for execution. For example, multiplication can take almost three times as long as a sum operation, and division usually takes even longer. Therefore, the mathematical operations performed during the inference of the model have a critical impact on execution time. We must not focus only on the arithmetic operations, but also take into account the number of branch instructions, data transfer operations, and virtual data allocations that may be responsible for increasing the execution time [45]. This can lead to the complex models at the edge not running efficiently [1, 2, 46].

$$\text{Execution Time} = \frac{\text{number of instructions} \times \text{average cycles per instruction}}{\text{clock frequency}} \tag{2.1}$$

TinyML hardware platforms are very heterogeneous, low-cost devices typically represented by MCUs. Table 2.1 shows a list of MCUs and development boards associated with articles describing ML implementations in low-resource devices.

Table 2.1 Comparison among hardware platforms supporting TinyML.

Hardware	Processor	CPU Clock	Flash	SRAM	Board Cost
TinyML Board [47]	Cortex-M0+	48 MHz	256 KB	32 KB	∼35$
STM32F0 Series [48]	Cortex-M0	48 MHz	up to 256 KB	up to 32 KB	∼10$
STM32F1 Series [49]	Cortex-M3	up to 72 MHz	up to 1 MB	up to 96 KB	∼11$
STM32F4 Series [50]	Cortex-M4	up to 180 MHz	up to 2 MB	up to 384 KB	∼13$
STM32L4 Series [51]	Cortex-M4	80 MHz	up to 1 MB	up to 320 KB	∼14$
STM32F7 Series [52]	Cortex-M7	216 MHz	up to 2 MB	up to 512 KB	∼23$
STM32H7 Series [53]	Cortex-M7	up to 550 MHz	up to 2 MB	up to 1.4 MB	∼27$
Arduino Nano 33 BLE Sense [54]	Cortex-M4	64 MHz	1 MB	256 KB	∼35$
SparkFun Edge [55]	Cortex-M4F	96 MHz	1 MB	384 KB	∼16$
OpenMV Cam H7 [56]	Cortex-M7	480 MHz	2 MB	1 MB	∼65$
GAP8 [57]	RISC-V	250 MHz (FC), 175 MHz (Cluster)	512 KB	80 KB, 8 MB SDRAM	∼15$

Many other options on the market that can also be evaluated for their suitability for developing TinyML applications. In Table 2.1, we compare the edge devices in terms of processor, CPU speed, flash memory, SRAM, and cost. We find that most hardware platforms operate at processor frequencies below 500 MHz and have, on average, less than 2 MB of flash and less than 1 MB SRAM. A few boards (e.g., GAP8, GAP9) are equipped with a hardware convolutional engine (HCE) to improve neural network computations at the edge. In addition, due to their limited resources, the cost of these devices is low, ranging from $10.00 to $35.00, with the exception of the OpenMV, which has a camera sensor.

2.3.3 Optimization Techniques

Several optimization techniques are used in literature to create efficient models that can run on MCUs while mitigating the aforementioned problems. The key enablers of TinyML can be formulated as quantization, pruning, knowledge distillation, and constrained neural architecture search.

- **Quantization:** The most widely used quantization technique for microcontrollers is the fixed-precision post-training quantization, in which a real number is mapped to a fixed-point representation after training via a scaling factor and a zero (offset)

[58, 59]. Variations include quantization of weights, weights and activations, or weights, activations, and inputs [60]. While post-training quantization (with 4, 8, and 16 bits) has been shown to reduce model size by a factor of four and speed up inference by a factor of two to three, quantization-aware training is recommended for microcontroller-class models to reduce layer-wise quantization error due to a wide range of weights across channels [58, 59]. This is achieved by simulated quantization operations, weight clamping, and fusion of special layers [61], resulting in equal or minor accuracy losses. To account for the different computational and memory requirements of the different layers, mixed-precision quantization assigns different bit widths to each layer for weights and activation [62]. Mixed-precision quantization enables up to 7x memory reduction [63], but is supported only by limited models of microcontrollers. Recently, binarized neural networks [64] have been ported to microcontroller-class hardware [65], where weights and activations are quantized to a single bit (-1 or +1). The binarized quantization can provide 8.5 to 19 times speedup and 8 times memory reduction [66, 67].

- **Pruning:** Pruning is usually an iterative process used to convert a dense neural network into a sparse network. Pruning can be divided into two types depending on the type of network component to be removed: Unstructured pruning removes individual weights or neurons, while structured pruning removes entire channels or filters. To create a pruned model, the network is first trained using a conventional training approach. Then, as proposed in [68], the important connections are determined by finding the weights that are above a certain threshold. Pruning is achieved by removing the number of weights that fall below the threshold. However, when the pruned network is tested, it does not achieve the same level of accuracy as the dense network. The poor performance of the pruned network limits its applicability in classification tasks; however, re-training the remaining weights improves accuracy. Moreover, the lost accuracy can be fully recovered even if 90% of the parameters are pruned if the pruning and retraining are performed iteratively [69]. It should be noted that the pruning technique not only helps in removing connections but also provides a systematic approach to prune neurons that have no input connections or no output connections [68]. These neurons are automatically eliminated during retraining and are therefore referred to as dead neurons. In [68], it is shown that pruning a network reduces the size of the network by up to 13 times without sacrificing accuracy. Although pruning provides a compressed version of the original model, the time complexity in deriving the acceptable model is very high and labour intensive.

- **Knowledge Distillation (KD):** Larger NN models are often better able to learn the structure of a complicated dataset. However, as discussed earlier, it may be infeasible to use large models on microcontrollers. One solution is to "distill" the knowledge of a large model into a smaller model. This is called knowledge distillation. The idea is to train a small model based not only on ground truths but also on the predictions of a larger model [70]. Consider the following example. We want to build a small model that can classify the content of an image. Normally, we would train the network to perform the same classification as the ground truths. In knowledge distillation, we first train a larger model on our data. The larger model then gives its classifications for all images in the dataset. We then train the small classifier not only to classify the ground truth data but also to make similar classifications to the larger model. This can be accomplished by modifying the loss function of the smaller model. Often the larger model is referred to as the "teacher" and the small model as the "student".

- **Constrained Neural Architecture Search (NAS):** NAS is the automated process of searching for the most optimal neural network within a neural network search space, given the target architecture and network architecture constraints, while achieving a balance between accuracy, latency, and energy consumption [71–73]. There are three key elements in a hardware-aware NAS pipeline, namely the search space formulation, the search strategy, and the cost function. The search space contains a set of ML operators, valid connection rules, and possible parameter values that the search algorithm can explore. The neural network search space can be represented as layer-wise, cell-wise, or hierarchical [71]. The search strategy involves sampling, training, and evaluation of candidate models from the search space to find the best performing model. This is done using reinforcement learning (RL), one-shot gradient-driven NAS, evolutionary algorithms (with weight sharing), or Bayesian optimization [74]. The cost function provides numerical feedback to the search strategy about the performance of a candidate network. Common parameters of the cost function include network accuracy, SRAM usage, flash usage, latency, and energy consumption. The goal of NA is to find a candidate network that finds the extrema of the cost function, i.e., the cost function can be viewed as a search for a Pareto-optimal configuration of network parameters. The constrained NAS can be assumed to be a multi-objective optimization problem that takes into account both the accuracy and hardware constraints (e.g., memory, processing delay) of the embedded devices so that the search space can be optimized for selecting appropriate deep neural networks [75–78].

2.3.4 Applications

There are many applications where TinyML models can be used to improve the performance of our system. The focus is on applications that have particular importance in the world of resource scarcity.

- **Computer Vision (CV):** This field deals with how computer systems see, recognize, and understand digital data, images, and video. This interdisciplinary field automates the key elements of human vision systems using sensors, intelligent computers, and machine learning algorithms. Computer vision is the technical theory underlying the ability of artificial intelligence systems to see - and understand - their surroundings. Computer vision has great potential to improve our daily lives and there are many applications and uses for it. Some examples include autonomous vehicles, smart doorbells, parking lots, and drones with connected cameras. All of these applications use intelligent video/image analytics based on AI and machine learning (ML) to observe video/images, make intelligent decisions, and then take action. The amount of data collected by these technologies is enormous, and the ML models used to make predictions are complex. Today, new advances in AI are making Deep Neural Networks (DNNs) faster, smaller, and more energy-efficient, helping to move AI from the cloud and data centers to edge devices. Implementing ML models that can fit into tiny MCUs and complex models that can process such volumes of data without relying on communications is a major milestone towards a robust CV application in the TinyML domain [41, 45, 79].

- **Natural Language Processing (NLP):** NLP uses artificial intelligence and machine learning to extract meaning from human speech as it is spoken. Depending on the programming of the natural language, the representation of this meaning can be in the form of plain text, a text-to-speech reading, or in a graphical representation or diagram. This is of interest in the TinyML domain because of the large number of applications that can be created using the flood of available spoken/written data. These applications include wake word recognition, sound event detection, fall detection, etc. TinyML allows such applications to function independently of an Internet connection, resulting in a solution that is highly accurate under a variety of conditions. In addition, attempts to issue the wake-up word or commands are never sent to the cloud, virtually eliminating privacy concerns [41, 79].

- **Communication:** An important application for ML in embedded systems is to improve communication or reduce traffic between the end device and upper network layers.

Several models can be used in this application area. Wireless networks are not static, and wireless signals have many variables that affect their strength, data rate, and latency (e.g., distance between nodes, propagation environment, collisions). Hardware reconfiguration can mitigate some of these issues. Reconfiguration is done by software, so a model that dynamically checks the network, understands the problem, and finds the correct solution can make communication more stable and reduce power consumption. However, if one of the main goals is to reduce traffic or/and database size, implementing the model in the end device can ensure this. If the end device sends only the result of the data inference, it will generate less traffic than sending the full data instance. Another option is to train our model to predict the importance of the collected data [80]. Much of the data collected by the terminals will be repetitive. Therefore, storing or sending this data may be unnecessary after a period of time.

- **Security and Privacy:** Another key application may be to enhance the security and privacy of the system. Improving privacy and security can be done by many ML models/algorithms. Algorithms based on feature extraction (e.g., ANN) create semantics that can only be understood by the algorithm itself. Thus, if the end device sends the data generated after the feature extraction layer, the attacker cannot use it even if the data is intercepted [81].

- **Healthcare:** Personal medical devices are subject to various physical limitations due to their proximity to the human body. In addition, some sensors are surgically implanted into the human body. Therefore, one of the most important limitations is the size of the device. To reduce the size of the device, the MCU must be small. Therefore, in some cases, the memory space is limited to the minimum required to run the program. Factors such as radio frequency interference from the equipment in medical facilities, radiation limitations, and small batteries place tight limits on communication [82]. The use of ML models in such devices can help predict health risks (e.g., stroke, diabetes, cardiac arrest, seizures) and improve patient outcomes. However, medical devices for monitoring human body activity are subject to strict limitations, harsh environments, and tight deadlines. Therefore, implementing ML models in these devices is a difficult but not impossible task and depends on overcoming the challenges discussed previously.

- **Industry 4.0:** Industry 4.0 is an application area where ML models, data analytics and IoT networks are useful and helpful tools. In production lines, it is critical to detect errors as quickly as possible, and automated ways to correct these errors are crucial to avoid production downtime. In addition, collecting data inside factories raises the

highest level of security and privacy concerns; therefore, sending data outside the local network may not be a viable approach [83]. Currently, Fog computing implementations are most prevalent in factories. Within factories, the edge device is responsible for the inference process and deciding what actions to take. The data does not leave the local network, but the machines, monitoring systems, and intelligent systems rely on a central system. Therefore, communication latency is still a problem, and centralization leads to a single point of failure and can be considered high risk from a security perspective.

- **Robotics:** An emerging application that tests the challenges of developing ML for edge devices is Tiny Robot learning, i.e., the use of ML on resource-constrained, low-cost autonomous robots [84]. These robots are lightweight and can operate in small spaces, making them a promising solution for applications ranging from emergency search and rescue [85, 86] to routine monitoring and maintenance of infrastructure and equipment [87].

2.3.5 Frameworks and Tools

Among recent advances in TinyML software tools, several frameworks have shown great promise for advancing ML research in MCUs. One of the main features attracting TinyML research is that most of the tools and frameworks are open-source and non-proprietary. Some of these tools are summarized in Table 2.

- **TensorFlow Lite Micro:** TensorFlow Lite Micro (TFLM) [88] is a specialized version of TFLite focused on optimizing TF models for Cortex-M and ESP32 MCU. TFLite Micro incorporates several embedded runtime design philosophies. TFLM avoids uncommon features, data types, and operations for portability. It also avoids specialized libraries, operating systems, or build system dependencies for heterogeneous hardware support and memory efficiency. TFLM avoids dynamic memory allocation to reduce memory fragmentation. TFLM interprets the neural network graph at runtime, rather than generating C++ code, to support simple paths for upgrades, multi-tenancy, multi-threading, and model replacement while sacrificing finite savings in memory. TFLM consists of three main components. First, the operator resolver associates only essential operations with the binary model file. Second, TFLM pre-allocates a contiguous memory stack called the arena that is used for the initialization and storage of runtime variables. TFLM uses a two-stack allocation strategy to discard initialization variables after their lifetime to minimize memory consumption. The space between the two stacks is used for temporary allocations during memory planning, and TFLM employs

bin packing to promote memory reuse and achieve an optimal, compacted memory layout during runtime. Finally, TFLM uses an interpreter to resolve the network graph at runtime, allocate the arena, and perform runtime computations.

Table 2.2 TinyML software suites for microcontrollers.

Framework	Supported Platforms	Supported Models	Supported Training Libraries	Open-Source	Free
TensorFlow Lite Micro (Google)	ARM Cortex-M, Espressif ESP32, Himax WE-I Plus	NN	TensorFlow	✓	✓
μTensor (ARM)	ARM Cortex-M (Mbed-enabled)	NN	TensorFlow	✓	✓
μTVM (Apache)	ARM Cortex-M		PyTorch, TensorFlow, Keras	✓	✓
EdgeML (Microsoft)	ARM Cortex-M, AVR RISC	NN, DT, k-NN, unary classifier	PyTorch, TensorFlow	✓	✓
CMSIS-NN (ARM)	ARM Cortex-M	NN	PyTorch, TensorFlow, Caffe	✓	✓
EON Compiler (Edge Impulse)	ARM Cortex-M, TI CC1352P, ARM Cortex-A, Espressif ESP32, Himax WE-I Plus, TENSAI SoC	NN, k-means, regressors (supports feature extraction)	TensorFlow, Scikit-Learn	✗	✓
STM32Cube.AI (STMicroelectronics)	ARM Cortex-M (STM32 series)	NN, k-means, SVM, RF, k-NN, DT, NB, regressors	PyTorch, Scikit-Learn, TensorFlow, Keras, Caffe, MATLAB, Microsoft Cognitive Toolkit, Lasagne, ConvnetJS	✗	✓
NanoEdge AI Studio (STMicroelectronics)	ARM Cortex-M (STM32 series)	Unsupervised learning	-	✗	✗
TinyEngine (MIT)	ARM Cortex-M	NN	PyTorch, TensorFlow	✓	✓
EloquentML	ARM Cortex-M, Espressif ESP32, Espressif ESP8266, AVR RISC	NN, DT, SVM, RF, XGBoost, NB, RVM, SEFR (feature extraction through PCA)	TensorFlow, Scikit-Learn	✓	✓
Sklearn Porter	-	NN (MLP), DT, SVM, RF, AdaBoost, NB	Scikit-Learn	✓	✓
EmbML	ARM Cortex-M, AVR RISC	NN (MLP), DT, SVM, regressors	Scikit-Learn, Weka	✓	✓
FANN-on-MCU	ARM Cortex-M, PULP	NN	FANN	✓	✓

- **μTensor:** μTensor [89] generates C++ files from TF models for Mbed-enabled boards and aims to generate models with a size of ≤ 2 kB. It is divided into two parts. The μTensor core provides a set of optimized runtime data structures and interfaces under computing constraints. The μTensor library provides standard error handlers, allocators, contexts, ML operations, and tensors built on top of the core. Basic data types include integral type, μTensor strings, tensor shape, and quantization primitives inherited from TFLite. Interfaces include the memory allocation interface, the tensor interface, tensor maps, and the operator interface. For memory allocation, μTensor uses the arena concept adopted from TFLM. In addition, μTensor has a set of optimized (CMSIS-NN under the hood), ordinary and quantized ML operators, consisting of activation functions, convolution operators, fully connected layers, and pooling.

- **μTVM:** micro-TVM [90, 91] extends the TVM compiler stack to run models on bare-metal IoT devices without requiring operating systems, virtual memory, or advanced programming languages. micro-TVM first generates a high-level and quantized computational graph (with support for complex data structures) from the model using the relay module. The functional representation is then fed into the TVM intermediate representation module, which generates C-code by performing operator and loop optimizations via AutoTVM and Metascheduler, procedural optimizations, and graph-level

modeling for whole program memory planning. AutoTVM consists of an automatic schedule explorer to generate promising and valid operator and inference optimization configurations for a given microcontroller and an XGBoost model to predict the performance of each configuration based on the characteristics of the lowered loop program. The developer can either specify the configuration parameters to be examined using a schedule template specification API or extract possible parameters from the hardware computation description written in the Tensor expression language. The generated code is integrated with the TVM C runtime, created, and transferred to the device. Inference is performed on the device using a graph extractor.

- **Microsoft EdgeML:** EdgeML provides a collection of lightweight ML algorithms, operators, and tools aimed towards deployment on Class 0 devices, written in PyTorch and TF. Included algorithms include Bonsai [92], ProtoNN [93], FastRNN [94], FastGRNN [94], ShallowRNN [95], EMIRNN [96], RNNPool [97], and DROCC [98]. EMI-RNN exploits the fact that only a small, tight portion of a time-series plot for a certain class contributes to the final classification while the remaining portions represent noise and are common among all classes. Shallow-RNN is a hierarchical RNN architecture that divides the time series signal into distinct blocks and passes them in parallel to multiple RNNs with shared weights and activation maps. RNNPool is a nonlinear pooling operator that can perform "pooling" on intermediate layers of a CNN with a downsampling factor much higher than 2 (4-8×) without losing accuracy while reducing memory usage and computational overhead. The Deep Robust One-Class Classifier (DROCC) is an OCC with limited negatives and an anomaly detector that does not require domain heuristics or hand-crafted features. The framework also includes a quantization tool called SeeDot [99].

- **CMSIS-NN:** Cortex Microcontroller Software Interface Standard-NN [100] is designed to transform TF, PyTorch and Caffe models for Cortex-M series MCUs. It generates C++ files from the model that can be included and compiled into the main program file. It consists of a collection of optimized neural network functions with fixed-point quantization, including fully connected layers, depth-wise separable convolution, partial image-to-column convolution, in-situ split x-y pooling, and activation functions (ReLU, sigmoid, and tanh, with the latter two implemented via lookup tables). It also provides a collection of auxiliary functions, including data type conversion and activation function tables (for sigmoid and tanh). CMSIS-NN provides a speedup of 4.6 times for execution and a reduction in energy consumption of 4.9 times.

- **Edge Impulse EON Compiler:** Edge Impulse [101] provides a complete end-to-end solution for deploying models to TinyML devices, starting with data collection with IoT devices, feature extraction, model training, and model deployment and optimization to TinyML devices. It uses the interpreter-less Edge Optimized Neural (EON) compiler for model deployment while supporting TFLM. The EON compiler compiles the network directly into C++ source code, eliminating the need to store unused ML operators (at the expense of portability). It has been shown that the EON compiler can run the same network with 25-55% less SRAM and 35% less flash than TFLM.

- **STM32Cube.AI** and **NanoEdge AI Studio:** X-Cube-AI from STMicroelectronics [102] generates STM32-compatible C code from a variety of deep learning frameworks (e.g., PyTorch, TensorFlow, Keras, Caffe, MATLAB, Microsoft Cognitive Toolkit, Lasagne, and ConvnetJS). It allows quantization (min-max), operator fusion, and the use of external flash or SRAM to store activation maps or weights. The tool also provides system performance and deployment accuracy measurement capabilities and suggests a list of compatible STM32 platforms based on model complexity. X-Cube-AI was shown to provide 1.3x memory reduction and 2.2x speedup compared to TLFM for gesture recognition and keyword spotting [103]. NanoEdge AI Studio [104] is another AutoML framework from STMicroelectronics for prototyping anomaly detection, outlier detection, classification and regression problems for STM32 platforms, including an embedded emulator.

- **TinyEngine:** The MIT MCUNet framework [75, 76] combines a neural architecture search algorithm (TinyNAS) that optimizes the search space with a lightweight inference engine (TinyEngine) that controls resource management in a similar way to an operating system. In other words, the TinyEngine provides the essential code to run the customized neural network of the TinyNAS. Moreover, TinyEngine adapts the memory scheduling according to the overall network topology rather than layer-wise optimization.

- **Eloquent MicroML** and **TinyML:** MicroMLgen ports decision trees, support vector machines (linear, polynomial, radial kernels, or single-class), random forests, XGboost, Naive Bayes, relevant vector machines, and SEFR (a variant of SVM) from SciKit-Learn to Arduino-style C code, storing model entities in Flash. It also supports onboard feature projection through PCA. TinyMLgen ports TFLite models to optimized C code using TFLite's code generator [105].

- **Sklearn Porter:** Sklearn Porter [106], generates C, Java, PHP, Ruby, GO and Javascript code from scikit-learn models. It supports the conversion of support vector machines, decision trees, random forests, AdaBoost, k-nearest neighbors, Naive Bayes, and multi-layer perceptrons.

- **EmbML:** Embedded ML [107] converts logistic regressors, decision trees, multilayer perceptrons, and support vector machines (linear, polynomial, or radial kernels) generated by Weka or Scikit-Learn into C++ code suitable for embedded hardware. It generates initialization variables, structures, and functions for classification, stores classifier data on flash to avoid high memory consumption, and supports the quantization of floating-point units. EmbML has been shown to reduce memory consumption by 31% and latency by 92% compared to Sklearn Porter.

- **FANN-on-MCU:** FANN-on-MCU [108] ports multilayer perceptrons generated by the Fast Artificial Neural Networks (FANN) library to Cortex-M series processors. It enables model quantization and generates an independent callable C function based on the specific instruction set of the MCU. It takes into account the memory of the target architecture and stores the network parameters either in RAM or in flash memory, depending on which memory does not overflow and is closer to the processor (e.g. RAM is closer than flash).

In this context, we presented the Edge Learning Machine (ELM) environment, a platform consisting of several frameworks and approaches to improve existing deployment solutions. Unlike the tools mentioned above, the initial ELM environment (i.e., the Desk-LM and Micro-LM modules) not only supports the use of various ML/DL algorithms on IoT devices (Micro-LM) but also searches for the best configuration for a variety of user-defined parameters during the training phase (Desk-LM). The ELM environment is then extended to support on-device training using unlabeled data (AEP), which is the case for most IoT-generated data. In addition, we have improved one of the existing implementation solutions, namely CMSIS-NN, by proposing a memory replacement strategy that reduces memory requirements while maintaining the time and energy efficiency offered by this library. In addition, ELM also hosts the CBin-NN inference engine for binarized neural networks. CBin-NN outperforms existing solutions such as TFLM, CMSIS-NN, μTVM and TinyEngine. Finally, the ELM environment runs on bare-metal devices and can be used in any microcontroller that supports C code to support the TinyML research community.

2.4 Advancement in TinyML Research

A growing number of articles have been published on the implementation of ML in embedded systems with scarce resources. To understand the optimizations and success of TinyML, we need to take a look at the advances in the field that have helped to accomplish this task.

Mansoureh Lord [109] uses a generic artificial neural network, an autoencoder, and a variational autoencoder on the Arduino Nano 33 BLE sense module. In the study, the model of a Kenmore top-loading washing machine is used to detect anomalies in the unbalanced dry spin cycle. The results show an accuracy of about 92% and a precision of 90%. In [110], a Deep Tiny Neural Network (DTNN) for numerical weather prediction (NWP) is presented. The framework uses STM32 MCU and X-CUBE-AI toolchain on Miosix operating system. Only 45.5 KB flash and 480 bytes on-board RAM are required to run the DTNN architecture. In [111], an ARM Cortex-M4F MCU is integrated with a low-power camera module (e.g., HM01B0) and a time-of-flight sensor (e.g., VL53L1X). When motion is detected, the MCU is triggered to wake up. In this setting, two convolutional layers are used with two MaxPool layers. Finally, three dense layers help detect faces. The work uses the CelebA dataset for training the face classifier based on the TensorFlow framework on the MCU. An accuracy of 97% is achieved in the wireless camera detection design mode. In [112], a few-shot KWS is used that employs only 5 training examples that can be used for each language. The method achieves an F1 score of 0.75 for 180 new keywords for 9 different languages. Such an embedding model can achieve a reasonable F1 score of 0.65 using only 5 shots. The 5-shot model is tested for its accuracy in the following two domains: Keyword Search and Keyword Spotting. It was found that the keyword spotting accuracy is 87.4% for 22 languages with 440 keywords. Capacitive sensing was considered to predict hand gestures for TinyML settings [113]. In this study, a wrist-worn embedded prototype with 4 capacitive sensing electrodes is developed. It uses only one hidden layer that acts as a classifier with 96.4% accuracy. The design uses an ARM Cortex-M4 MCU with 1 MB flash and 256 KB RAM. Finally, the on-board model used has a size of only 29.6 KB in float32 mode without quantization. The inference time is very low, namely 12 ms with a power consumption of 26.4 mW. In [114], an architecture for predicting American Sign Language is presented. The architecture can operate on an ARM Cortex-M7 MCU with a 496 KB of memory footprint. The implicit model size is only 185 Kb during the post-quantization phase and can infer 20 frames/s. The work uses four datasets, namely Sign MNIST, Kaggle ASL, OpenMV H7 aware dataset and ASL alphabet test dataset, all from Kaggle. The generalizability of 74.58% is achieved in this study. FastGRNN and FastRNN [94] are algorithms to implement recurrent neural networks (RNNs) and gated RNNs in tiny IoT devices and mitigate some of the challenges they face. Using these FastGRNNs can shrink some real-world application

models to fit into MCUs with scarce resources. Some models are shrunk to fit in MCUs with 2 Kbytes of RAM and 32 Kbytes of flash memory. The reduction of these models is achieved by adding residual connections,and using low-rank, sparse, quantized (LSQ) matrices. De Almeida *et al.* [115] implemented a multilayer perceptron (MLP) in an Arduino UNO that was able to identify intrusions. The system was able to detect intruders by reducing the number of features required through the use of feature extraction techniques. The neural network consisted of 26 input neurons, 9 hidden layers, and 2 output neurons.

In [116], the authors propose a DNN architecture pruning algorithm capable of finding a pruned version of a given DNN within a user-specified memory space. The algorithm is used in a memory-constrained wireless sensor network (WSN) to detect flooding events in urban rivers using semantic segmentation. The results show that the presented algorithm can find a pruned DNN model with the specified memory requirement with little to no degradation in its performance. In [117], the authors present a differentiable structured network pruning method for convolutional neural networks that integrates MCU-specific resource utilization of a model and parameter importance feedback to obtain highly compressed yet accurate classification models. Their method (a) improves the main resource usage of models by 80x; (b) iteratively prunes while a model is trained, resulting in little to no overhead or even improved training time; (c) produces compressed models with the same or improved resource utilization by 1.7x in less time compared to previous MCU-specific methods.

Sharma *et al.* [118] studied the application of Knowledge Transfer (KT) to edge devices and obtained good results transferring knowledge from both intermediate layers and the last layer of the teacher (original model) to a shallower student (target). The authors of [77] presented a neural architecture search system (NAS), called μNAS, to automate the design of small but powerful MCU-level networks. μNAS explicitly targets the three main aspects of MCU resource scarcity: the size of RAM, persistent memory, and processor speed. They show that μNAS can improve top-1 classification accuracy, reduce memory requirements, or reduce the number of multiplication accumulation operations for a variety of image classification datasets compared to existing MCU literature and resource-efficient models.

Recently, automated traffic planning has been investigated for possible integration into the TinyML framework to improve the real-time traffic system [119]. The implied method uses piezoelectric sensors embedded in many lanes of a road. Using the data from the piezoelectric sensors, a two-point time ratio method is aimed at detecting a vehicle. The vehicles are then classified for green light timing prediction. In the study, a random forest regressor (RFR) is used to predict the duration of the signal based on the input number of vehicles in each lane. The algorithm is deployed on an Arduino Uno enhanced with the

m2gen library and Scikit Learn support. The algorithm used requires only 1.754 KB. Szydlo *et al.* [120] have proposed a way to implement various ML models in an Arduino Mega and an ARM STM32F429. They managed to train the model using Python (on a PC) and export it to these MCUs. They were able to implement Naive Bayes, multi-layer perceptron, and decision tree classifiers. The classifiers were able to classify images (MNIST dataset) and iris flowers (Iris dataset). To fit the models into MCU memory, the authors used compiler optimizations such as those provided by GNU. Suresh *et al.* [121] used a k-NN model for activity classification that achieved 96% accuracy. They implemented the model in a Cortex-M0 processor, which helped reduce the data bytes to be sent from the terminal to the upper network layers in a LoRA network. They optimized the model by representing each class by a single point (centroid). ProtoNN [93] is a novel algorithm to replace k-NN in resource-poor MCUs. This algorithm stores the entire training dataset and uses it during the inference phase; this technique would not be feasible in memory-constrained MCUs. Since k-NN computes the distance between the new data point and the previous ones, ProtoNN has achieved optimization through techniques such as low-D spatial projection, prototyping, and joint optimization. Therefore, the model is constructed during optimization to match the maximum size instead of being pruned after construction. Bonsai [92] is a novel algorithm based on decision trees. The algorithm reduces the model size by learning a sparse, single shallow tree. This tree has nodes that improve prediction accuracy and can make nonlinear predictions. The final prediction is the sum of all predictions provided by each node.

2.4.1 Self-Learning Approaches

A large research effort has been devoted to learning from unlabeled data. We can identify three main research areas in the field: combining clustering with classification, semi-supervised learning, and self-supervised learning, as shown in this section.

Qaddoura *et al.* [122] propose a three-stage classification approach for IoT intrusion detection. In the first stage, the IoT training data is reduced by 10% after grouping the data using k-means clustering. Then, oversampling is performed to address the lack of minority class instances. In the final stage, the enlarged dataset is used to train a Single Hidden Layer Feed-Forward Neural Network. The authors claim that the proposed approach outperforms other approaches on the selected dataset. [123] propose a method to improve the classification accuracy by applying clustering techniques, namely k means and hierarchical clustering, to the dataset before the classification algorithms. Their experimental analysis shows that higher accuracy is achieved after applying the classification algorithms, namely Naïve Bayes and Neural Network, to the clustered data. The work in [124] presents a hybrid framework that uses network traffic to protect IoT networks from unauthorized device access. For each

device type in the dataset, a binary one-vs-rest random forest classifier is trained and the model is stored. During prediction, if all saved classifiers failed to classify the feature vector as a "known" device, the OPTICS clustering algorithm is used to group devices with similar behavior together. Experimental results on ten IoT device types (three types are provided without labels) show good accuracy levels, and reduced runtime and memory requirements due to the use of an autoencoder.

Semi-supervised approaches refer to training a specific classifier using a small amount of annotated data in conjunction with a large amount of unlabeled data [31]. The general workflow consists first in training a classifier using the small set of labeled training data. Then, the classifier is used to predict the outputs (pseudo-labels) of the unlabeled data. The final step is to train the model on the labeled and pseudo-labeled data. In [125], Zhou *et al.* designed a semi-supervised deep learning framework for human activity detection in IoT. The proposed method exploits two modules. The auto-labeling module is based on reinforcement learning and assigns labels to the input data before sending it to the LSTM-based classification module for training. The evaluation of the framework shows the effectiveness of the proposed method in the case of weakly labeled data (i.e., large amounts of unlabeled data and a small amount of labeled data). Ravi and Shalinie [126] propose a mechanism for distributed denial of service (DDoS) mitigation and detection based on an Extreme Learning Machine algorithm trained in a semi-supervised manner. They demonstrate the effectiveness of the proposed approach compared to other ML algorithms. Rathore *et al.* [127] propose a Fog-based framework that integrates a semi-supervised fuzzy C-Means with an Extreme Learning Machine classifier for efficient attack detection in IoT. The proposed framework achieves better performance and detection time when compared to cloud-based attack detection frameworks. The idea behind self-supervision is to find a surrogate task for the network to learn that does not require explicit labeling, but rather the inherent structure of the data that provide the labels.

Self-supervised learning is a branch of unsupervised training in which pseudo-labels are automatically generated from the data itself [128], then a model is trained through supervised learning exploiting such pseudo-labels (e.g., [129]). In this area, Wu *et al.* [130] propose a framework for on-device training of convolutional neural networks that consists of two approaches: a self-supervised early instance filtering of the data to select important samples from the input stream, and an error-map pruning algorithm to drop insignificant computations in the backward pass. The framework reduces the computational and energy costs of training with little loss of accuracy. Human activity detection with self-supervised learning is introduced in [131]. Signal transformation is performed as a pretext task for label generation, and various activity recognition tasks are performed on six public datasets. The results show that self-supervised learning enables the convolutional model to learn high-level

features. Saeed *et al.* [132] present a self-supervised method that learns representations from unlabeled multisensory data based on wavelet transform. The approach is evaluated on the datasets with sensory streams and achieves comparable performance to fully supervised networks.

2.4.2 Memory Optimization Strategies

Memory strategies have been investigated as well to overcome the memory limitations of edge devices. Gural and Murmann [133] present memory-optimal direct convolutions in presence of strict hardware memory constraints (on Arduino devices) at the expense of extra computation. Unlu [134] proposes two optimizations that provide memory savings for 2D convolutions and fully connected layers; the first is in-place max-pooling and the second is the use of ping-pong buffers between layers. The work focuses on memory optimization, achieving a 74% RAM utilization reduction compared to the CMSIS-NN implementation, but at the expense of a reduction in execution performance due to the latency caused by fetching data from flash and the lack of SIMD (single instruction multiple data) instructions. Liberis *et al.* [135] present a way of saving memory by changing the execution order of the network's operators within the inference software, which is orthogonal to other compression methods. They published a tool for reordering operators of TensorFlow Lite models and demonstrate its utility by sufficiently reducing the memory footprint of a CNN to deploy it on an MCU with 512KB SRAM.

2.4.3 Neural Network Binarization

Some TinyML applications have been implemented on edge devices using Binarized Neural Networks (BNNs). In the field of CV, Fasfous *et al.* [136] presented BinaryCoP, a BNN classifier implemented on an embedded FPGA accelerator for correct face mask wearing and positioning. Cerutti *et al.* [137] implemented a 7 layer BNN on the highly energy-efficient, RISC-V-based (8+1)-core GAP8 microcontroller. This BNN is trained on a sound event detection dataset and reaches a 77.9% accuracy. This is just 7% lower than the full-precision version. Compared to the performance of an ARM Cortex-M4 implementation, their system has a 10.3x faster execution time and 51.1x higher energy-efficiency. Danilo *et al.* [65] compares two industry frameworks used to train BNNs, namely Larq and QKeras. Two different models are implemented, one for human activity detection (computer vision) and the other for anomaly detection (industry). They report inference time on ARM Cortex-A and Cortex-M CPUs using custom C functions designed for binary layer execution. Younes

et al. [138] proposed a hybrid fixed-point CNN (H-CNN) implemented on an FPGA for robot touch modality classification.

2.5 Promising Future Trends

The field of TinyML is constantly evolving, with new and improved techniques being developed to overcome the limitations posed by constrained devices. In this section, we will delve into three of the most promising techniques in the TinyML field: Federated Learning, Online Learning, and Transformers. These techniques offer innovative solutions for efficient and effective model training and deployment, even in resource-constrained environments.

- **Federated Learning:** Federated Learning (FL) is a paradigm that has recently been developed primarily for privacy-preserving learning. In this paradigm, training is distributed to decentralized devices such as smart edge devices (i.e., mobile or IoT sensor devices) that generate data samples such that the data does not leave the local storage of the data source [139]. Ideal problems for federated learning have the following characteristics. First, training on real data from mobile devices provides a distinct advantage over training on proxy data, which is generally available in the data center. Second, this data is privacy-sensitive or very large (compared to the size of the model), so it is preferable not to store it in the data center solely for model training. And third, for supervised tasks, labels on the data can be derived naturally from user interaction. Many models that enable intelligent behavior on mobile devices meet the above criteria. As two examples, we consider image classification, which predicts, for example, which photos are most likely to be viewed or shared multiple times in the future, and language models, which can be used to improve speech recognition and text input on touchscreen keyboards by improving decoding, next word prediction, and even whole reply prediction [140]. The potential training data for these two tasks (all the photos a user takes and everything they type on their mobile keyboard, including passwords, URLs, messages, etc.) may be privacy sensitive. The distributions from which these examples are drawn are also likely to be significantly different from easily accessible proxy records: Language use in chats and text messages is generally very different from standard language corpora, e.g., Wikipedia and other web documents; the photos people take with their cell phones are likely to be very different from typical Flicker photos. Finally, the labels for these problems are directly available; input text is self-labelled to learn a language model, and photo labels can be defined by the user's natural interaction with their photo app (which photos are deleted, shared, or viewed). Both tasks are well suited for neural network learning.

To extend the widespread potential of TinyML, Grau *et al.* [141] and Kopparapu *et al.* [142] demonstrate the applicability of federated learning concepts in tandem with TinyML. Both implementations resort to Eager Learning approaches to improve local models. Grau, for instance, developed a Multi-class classification Key-Word Spotter. Training on a singular device, it was observed that the loss gradually degrades although the accuracy can fluctuate. Similarly, Kopparapu implemented Federated Transfer Learning (TinyFedTL) in a low resource setting, where they considered a binary classification problem (cat vs no-cat and dog vs no-dog) where the cat vs no-cat was treated as the Transfer Learning problem. Other fruitful areas that could be explored include the elevated scenario of non-IID data and federated learning training to better understand its ability to obtain versatile models when limited local training data is available. Communication costs are also an area for improvement. Although round-trip communication time per update is not unrealistic, this is a problem for edge devices in remote areas with unstable internet connection.

- **Online Learning:** Online learning is an important concept in the TinyML field as it enables real-time adaptation of machine learning models on resource-constrained devices. In this approach, models are trained incrementally on streaming data, allowing them to continuously update and improve their predictions as new data becomes available. This is particularly useful in the context of IoT devices, where data is generated in real-time and may change dynamically over time. Several studies have investigated the use of online learning in TinyML, with a focus on improving the efficiency and accuracy of models for various applications. For example, Ren *et al.* [143] introduce TinyOL, a system for online learning in TinyML, which enables microcontrollers to learn from data in real-time without relying on a cloud infrastructure. The study evaluates the performance of TinyOL in various scenarios, such as data processing, energy consumption, and latency, to determine its feasibility and suitability for real-world applications. The results of the evaluation demonstrate the potential of TinyOL as a solution for online learning in TinyML, especially for resource-constrained devices like microcontrollers. Another study [144] introduces a HW/SW platform for end-to-end continual learning (CL) based on a 10-core FP32-enabled parallel ultra-low-power (PULP) processor. The authors propose a modification to the Latent Replay-based Continual Learning (CL) [145] algorithm, which enables online serverless adaptation, by quantizing the frozen stage of the model and Latent Replays to reduce memory cost without sacrificing accuracy. The compression of the Latent Replays memory is shown to be almost lossless with 8-bit compression, while 7-bit compression incurs minimal accuracy degradation. The platform also includes optimized primitives and data tiling

strategies to maximize efficiency. On a 22nm prototype of their platform, called VEGA, the proposed solution performs 65 times faster and is 37 times more energy efficient compared to a low-power STM32 L4 microcontroller. Overall, online learning is a promising approach for TinyML, allowing models to continuously adapt to new data and changing environments in real-time, while minimizing resource requirements and data transfer overhead.

- **Transformers:** Transformers are a type of neural network architecture that have shown great success in the field of natural language processing (NLP). Recently, the use of transformers has expanded into the realm of TinyML. There has been a significant amount of research in this area, with numerous studies focusing on the deployment of transformers on edge devices. For example, in a recent study, researchers have proposed a lightweight transformer architecture called TinyBERT [146], which is specifically designed for deployment on small devices with limited computational resources. TinyBERT has been shown to achieve state-of-the-art results on NLP benchmarks, while still being small and fast enough to run on a microcontroller. In addition to NLP, transformers have also been applied to other tasks in the field of TinyML, such as image classification. For example, authors in [147] propose a new set of execution kernels tuned for MCUs, focusing on minimizing memory movements and maximizing data reuse in the attention layers. The results showed significant improvement in latency and energy consumption compared to previous state-of-the-art implementations. The authors also demonstrate that their TinyTransformer library can fit a 263 KB transformer on a GAP8 platform, outperforming a previous convolutional architecture in terms of accuracy and energy consumption. In conclusion, the use of transformers in TinyML is a rapidly growing field with many exciting research developments. From optimizing energy consumption to achieving state-of-the-art performance on small devices, there is a lot of potential for transformers to be used in a wide range of applications in the field of TinyML.

Despite these advances, there are still limitations to the widespread adoption of TinyML. One of the biggest challenges is the limited computational and memory resources of tiny devices, which can impact the accuracy and real-time performance of TinyML models. Another challenge is the lack of standardization and common software platforms, which makes it difficult for developers to compare and implement different TinyML models. Additionally, there are concerns around the privacy and security of data collected by TinyML devices, especially when the devices are connected to the cloud. These issues need to be addressed for the TinyML field to realize its full potential.

2.6 Conclusion

TinyML is a rapidly growing field that combines hardware, software, and machine learning algorithms to enable smart capabilities in tiny devices. In this chapter, we provided a comprehensive overview of TinyML and its advantages in the Internet of Things (IoT) domain. We discussed the challenges and limitations in adopting TinyML and the optimization techniques that help overcome these barriers. Additionally, we showcased various key applications of TinyML and highlighted the frameworks available for its implementation. Furthermore, we highlighted the recent advances and optimizations in TinyML research, as well as promising future trends in the field. Overall, this chapter aims to provide a comprehensive understanding of TinyML and its potential for transforming small devices into smart ones.

Chapter 3

Machine Learning on Mainstream Microcontrollers

3.1 Introduction

The trend of moving computation towards the edge is becoming ever more relevant, leading to performance improvements and the development of new field data processing applications [25]. This computation shift from the cloud (e.g., [148]) to the edge has advantages in terms of response latency, bandwidth occupancy, energy consumption, security and expected privacy (e.g., [13]). The huge amount, relevance and overall sensitivity of the data now collected also raise clear concerns about their use, as is being increasingly acknowledged (e.g., [149]), meaning that this is a key issue to be addressed at the societal level.

The trend towards edge computing also concerns machine learning (ML) techniques, particularly for the inference task, which is much less computationally intensive than the previous training phase. ML systems "learn" to perform tasks by considering examples, in the training phase, generally without being programmed with task-specific rules. When running ML-trained models, Internet of Things (IoT) devices can locally process their collected data, providing a prompter response and filtering the amount of bits exchanged with the cloud. ML on the edge has attracted the interest of industry giants. Google released the TensorFlow Lite platform, which provides a set of tools that enable the user to convert TensorFlow Neural Network (NN) models into a simplified and reduced version that can run on edge devices [69, 150]. EdgeML is a Microsoft suite of ML algorithms designed to work off the grid in severely resource-constrained scenarios [151]. ARM has published an open-source library, namely Cortex Microcontroller Software Interface Standard Neural Network (CMSIS-NN),

for Cortex-M processors [100]. Likewise, an expansion package, namely X-Cube-AI, has been released for implementing deep learning models on STM 32-bit microcontrollers [102].

While the literature is increasingly reporting on novel or adapted embedded machine learning algorithms, architectures and applications, there is a lack of quantitative analyses about the performance of common ML algorithms on state-of-the-art mainstream edge devices, such as ARM microcontrollers [152]. We argue that this has limited the development of new applications and the upgrading of existing ones through an edge computing extension.

In this context, we have developed the Edge Learning Machine (ELM), a framework that performs ML inference on edge devices using models created, trained, and optimized on a Desktop environment. The framework provides a platform-independent C language implementation of well-established ML algorithms, such as linear Support Vector Machine (SVM), k-Nearest Neighbors (k-NN) and Decision Tree. It also supports artificial neural networks by exploiting the X-Cube-AI package for STM 32 devices [102]. We validated the framework on a set of STM microcontrollers (families F0, F3, F4, F7, H7, and L4) using six different datasets, to answer a set of ten research questions exploring the performance of microcontrollers in typical ML Internet of Things (IoT) applications. The research questions concern a variety of aspects, ranging from inference performance comparisons (also with respect to a desktop implementation) to training time, and from pre-processing to hyper-parameter tuning.

3.2 Background

The Edge Learning Machine environment aims to provide an extensible set of algorithms and tools to perform inference on the edge. The initial implementation featured four well-established supervised learning algorithms, which we briefly introduce in the following subsections. They all support both classification and regression problems.

3.2.1 Artificial Neural Network

An Artificial Neural Network is a model that mimics the structure of our brain's neural network. It consists of a number of computing neurons connected in a three-layer system; one input layer, several hidden layers, and one output layer. Artificial Neural Networks (ANNs) can model complex and non-linear or hidden relationships between inputs and outputs [28]. This is one of the most powerful and well-known ML algorithms, which is used in a variety of applications, such as image recognition, natural language processing, forecasting, etc.

3.2.2 Linear Kernel Support Vector Machine

The SVM algorithm is a linear classifier that computes the hyperplane that maximizes the distance from it to the nearest samples of the two target classes. It is a memory-efficient inference algorithm and can capture complex relationships between data points. The downside is that the training time increases with huge and noisy datasets [153]. While the algorithm deals well with non-linear problems, thanks to the utilization of kernels that map the original data in higher dimension spaces, we implemented only the original, linear kernel [154] for simplicity of implementation into the edge device.

3.2.3 K-Nearest Neighbor

k-NN is a very simple algorithm based on feature similarity that assigns, to a sample point, the class of the nearest set of previously labeled points. k-NN's efficiency and performance depend on the number of neighbors K, the voting criterion (for K > 1) and the training data size. The training phase produces a very simple model (the K parameter), but the inference phase requires exploring the whole training set. Its performance is typically sensitive to noise and irrelevant features [153, 155, 156].

3.2.4 Decision Tree

This is a simple and useful algorithm, which has the advantage of clearly exposing the criteria of decisions that are made. In building the decision tree, at each step, the algorithm splits data to maximize the information gain, thus creating homogeneous subsets. The typical information gain criteria are Entropy and Gini. DT can deal with linearly inseparable data and can handle redundancy, missing values, and numerical and categorical types of data. It is negatively affected by high dimensionality and high numbers of classes, because of error propagation [153, 157, 158]. Typical hyper-parameters that are tuned in the model selection phase concern regularization and typically include depth, the minimum number of samples for a leaf, and the minimum number of samples for a split, the maximum number of leaf nodes and the splitter strategy (the best one, which is the default, or a random one, which is typically used for random forests), etc. Several DTs can be randomly built for a problem, to create complex but high-performing random forests.

3.3 Framework and Algorithm Understanding

The originally proposed Edge Learning Machine (ELM) framework consists of two modules, one working on the desktop (namely Desk-LM, for training and testing), and one on the edge (Micro-LM, for inferencing and testing), as sketched in Figure 3.1.

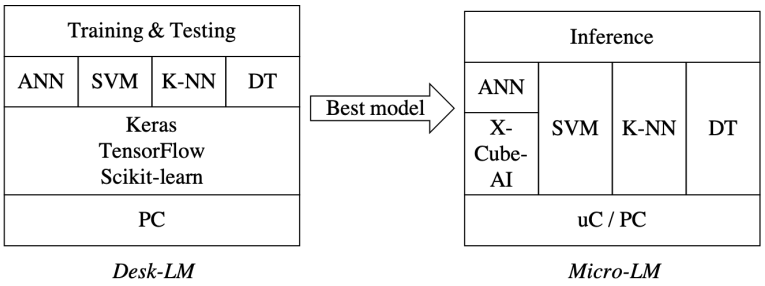

Fig. 3.1 Block diagram of the EdgeML system architecture.

1. **Desktop:** the Desk-LM module is implemented in python and works on a PC to identify the best models for an input dataset. The current implementation involves four algorithms for both classification and regression: artificial neural networks (ANN), linear support vector machines (SVM), K-Nearest Neighbors (k-NN), and Decision Tree (DT) algorithms. For each algorithm, Desk-LM identifies the best model through hyper-parameter tuning, as is described later in Table 3.2. Desk-LM relies on the scikit-learn python libraries [159] and exploits the TensorFlow [160] and Keras [161] packages for ANNs;

2. **Edge:** the Micro-LM module reads and executes the models generated by Desk-LM. It is implemented in platform-independent C language (for linear kernel SVM, k-NN, DT) and can run on both microcontrollers and desktops, to perform inferences. ANNs are deployed using the X-Cube-A expansion package for stm32 microcontrollers (TensorFlow and Keras on desktops).

   The tool has been designed to support a four-step workflow, as shown in Figure 3.2

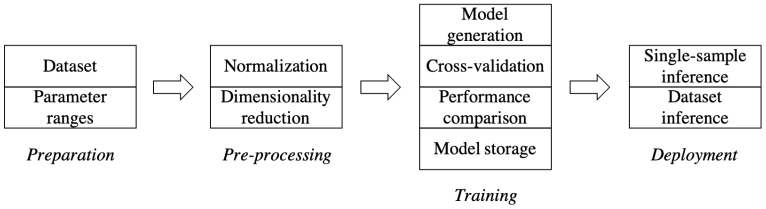

Fig. 3.2 Supported workflow.

1. **Preparation:** in this first phase, the user provides the dataset and defines the range of the parameters to be investigated for each algorithm. The parameters are listed in Tables 3.1 and 3.2 (common and algorithm-specific, respectively—these common parameters are used by all the algorithms, even if they have different values);

Table 3.1 Common configuration parameters.

Common parameters
Algorithm type (SVM, K-NN, DT, ANN)
Dataset
Content format (dataset start and end column, target column, etc.)
Number of classes (if classification)
Testing set size
Regression (True or False)
PCA (a specific number of features or MLE algorithm)
Normalization (standard or minmax)
K-fold cross-validation
Accuracy metrics (accuracy, R2)

Table 3.2 Algorithm-specific configuration parameters.

Algorithm-Specific Configuration			
ANN	**Linear SVM**	**K-NN**	**DT**
Layer Shape	C	K (number of neighbors)	Splitting criterion
Activation Function		Training set size for targets	max_depth
Dropout			min_samples_split
Loss metrics			min_samples_leaf
Number of epochs			max_leaf_nodes
Batch size			
Number of repeats			

2. **Pre-processing:** in this phase, data goes through the scaling and dimensionality reduction steps, which are important to allow optimal processing by the prediction algorithms [162]. The type of algorithm used for this step is one of the common parameters set by the user (Table 3.1);

3. **Model generation:** in this phase, all the configurations resulting from combining the values of the user-specified parameters (both common and algorithm-specific; see Tables 3.1 and 3.2, respectively) are evaluated through cross-validation, and their k values are, again, used as the parameters (Table 3.1). Most of the parameters (the algorithm's hyper-parameters) can be assigned a list of values, each one of which is evaluated (scikit-learn exhaustive grid search), to allow for the selection of the best values. At the end of this step, the best model is saved in the disk, to be deployed on the edge. Desk-LM also saves the pre-processing parameters and, if needed for

performance assessment purposes, the testing set (or a reduced version of it). All these files are then compiled in Micro-LM for the processing of data on the edge;

4. **Deployment:** in this final phase, the Micro-LM module loads the model prepared on the desktop. The deployment process for our tests on microcontrollers is done using the STM32CubeIDE integrated development environment, which exploits the X-Cube-AI pack for ANNs. The software output by our framework supports both single-sample inference and whole dataset inference, for performance analysis purposes. In the latter case, Micro-LM exploits the testing set file produced by Desk-LM.

As anticipated, the initial version of the ELM framework features four well-established supervised learning algorithms, of which, in the following subsections, we briefly describe the implementation on both the desktop and edge side.

3.3.1 Artificial Neural Network

In Desk-LM, ANNs are implemented through the TensorFlow [160] and its wrapper Keras [161] packages. As an optimizer, we use adaptive moment estimation ('adam') [163]. At each execution run, the Desk-LM module performs the hyper-parameter tuning by analyzing different ranges of parameters (Table 3.1 and the first column of Table 3.2 specified by the user. The ANN model hyper-parameters include layer shape (number and size of input, hidden, and output layers), activation function for the hidden layers (Rectified Linear Unit (ReLU), or Tangent Activation Function (Tanh)), number of epochs, batch size, number of repeats (to reduce result variance), and dropout rate. The best selected model is then saved in the high-efficiency Hierarchical Data Format 5 (HDF5) compressed format [164]. For the edge implementation, Desk-LM relies on the STM X-Cube-AI expansion package, which is supported by STM32CubeIDE and allows for its integration in the application of a trained Neural Network model. The package offers the possibility of compressing models up to eight times, with an accuracy loss that is estimated by the package. The tool also provides an estimation of the complexity, through the Multiply and Accumulate Operation (MACC) figure, and of the Flash and RAM memory footprint [102].

3.3.2 Linear Support Vector Machine

As anticipated, for the simplicity of the implementation of the edge device, we implemented only the original, linear kernel SVM [154]. The linear model executes the $y = w * x + b$ function, where w is the support vector and b is the bias. Model selection concerns the C

regularization parameter [165] (Table 3.2). As an output model, Desk-LM generates a C source file containing the w and b values.

3.3.3 K-Nearest Neighbor

For simplicity of implementation, we used a Euclidean distance criterion and majority voting (for $k > 1$). The training phase produces a very simple model (the K parameter), but deployment also requires the availability of the whole training set (Table 3.2).

3.3.4 Decision Tree

To cope with the limited resources of edge devices, our framework allows us to analyze different tree configurations in terms of depth, leaf size, and the number of splits. Concerning the splitting criterion, for simplicity of implementation on the target microcontrollers, we implemented only the "Gini" method.

3.4 Experimental Analysis and Result

We conducted the experimental analysis using six ARM Cortex-M microcontrollers produced by STM, namely F091RC, F303RE, F401RE, F746ZG, H743ZI2, and L452RE. The F series represents a wide range of microcontroller families in terms of execution time, memory size, data processing and transfer capabilities [166], while the H series provides higher performance, security, and multimedia capabilities [167]. L microcontrollers are ultra-low-power devices used in energy-efficient embedded systems and applications [168]. All listed MCUs have been used in our experiments with their STM32CubeIDE default clock values, which could be increased for a faster response. Table 3.3 synthesizes the main features of these devices. In the analysis, we compare the performance of the embedded devices with that of a desktop PC hosting a 2.70 GHz Core i7 processor, with 16 GB RAM and 8 MB cache.

The performance of the selected edge devices was evaluated using six benchmark datasets that are representative of various IoT applications (as listed in Table 3.4). These datasets were carefully selected to cover different types of applications, including binary classification, multiclass classification, and regression. The University of California Irvine (UCI) heart disease dataset [169] is a well-known medical dataset that was included as a representative of the medical field. The Virus dataset [170–172], developed by the University of Genova, was selected to represent data traffic analysis. The Sonar dataset [173, 174] represents readings from a sonar system that analyzes materials, allowing for the distinction between rocks

and metallic materials. The Peugeot 207 dataset [175] was chosen as it includes various parameters collected from cars and was used to predict either the road surface or traffic (two labels were considered in our studies: label_14: road surface and label_15: traffic). The EnviroCar dataset [176–178] records various vehicular signals through the onboard diagnostic (OBDII) interface to the Controller Area Network (CAN) bus, and was included to represent vehicular data. The air quality index (AQI) dataset [179] was selected as it measures air quality in Australia over the course of a year and represents environmental data. All datasets were pre-processed and converted to float32 to match the target execution platform. These datasets were chosen for their importance in evaluating the performance of edge devices, as they cover a range of real-world IoT scenarios, allowing for a comprehensive analysis of the devices' capabilities. By including datasets from various domains, the experimental analysis provides a thorough evaluation of the selected edge devices, making the results generalizable to a wider range of IoT applications.

Table 3.3 Microcontroller specifications.

Microcontroller	Flash Memory	SRAM	Processor Speed Used (MHz)	Unit Cost ($)
F091RC	256 Kbytes	32 Kbytes	48 (max: 48)	4,5
F303RE	512 Kbytes	80 Kbytes	72 (max: 72)	7.3
F401RE	512 Kbytes	96 Kbytes	84 (max: 84)	6,1
F746ZG	1 Mbyte	360 Kbytes	96 (max: 216)	11.3
H743ZI2	2 Mbytes	1 Mbyte	96 (max: 480)	10,8
L452RE	512 Kbytes	160 Kbytes	80 (max: 80)	7,3

Table 3.4 Datasets specifications.

Dataset	Samples Features	Type
Heart	303 x 13	Binary Classification
Virus	24736 x 13	Binary Classification
Sonar	209 x 60	Binary Classification
Peugeot207*	8615 x 14	Multiclass Classification
EnviroCar	47077 x 5	Regression
AQI	367 x 8	Regression

* For Peugeot 207, we considered two different labels.

Our analysis was driven by a set of questions, synthesized in Table 3.5, aimed at investigating the performance of different microcontrollers in typical ML IoT contexts. We are also interested in comparing the inference performance of microcontrollers vs. desktops. The remainder of this section is devoted to the analysis of each research question. In a few cases, when the comparison is important, results are reported for every tested target platform. On the other hand, in most of the cases, when not differently stated, we chose the F401RE device as the reference for the embedded targets.

Table 3.5 Research questions.

Research Questions (RQ)	Description
1. Performance	Score (accuracy, R2) and inference time
2. Scaling	Effect of scaling data
3. PCA	Effect of dimensionality reduction on score
4. ANN Layer Configuration	Different layer shapes (depth and thickness)
5. ANN Activation Function	Effect of different neuron activation functions
6. ANN Batch Size	Effect of batch size on score and time
7. ANN Epochs	Effect of number of epochs in training
8. ANN Dropout	Effect of a regularization technique to avoid overfitting
9. SVM Regularization Training Time	SVM training time with different values of the "C" regularization parameter
10. DT Parameters	Tuning the decision tree

3.4.1 Performance

The first research question concerns the performance achieved both on desktop and edge. For SVM, k-NN and DT on desktops, we report the performance of both our C implementation and the python scikit-learn implementation, while for ANN we have only the TensorFlow Keras implementation. The following set of tables show, for each algorithm, the obtained score, which is expressed in terms of accuracy (in percent, for classification problems), or coefficient of determination, R-Squared (R2, for regression problems). R2 is the proportion of the variance in the dependent variable that is predictable from the independent variable(s). The best possible score for R2 is 1.0. In scikit-learn, R2 can assume negative values because the model can be arbitrarily worse. The second performance we consider is the inference time.

In the following (Tables 3.6 - 3.13), we report two tables for each algorithm. The first one provides the best performance (in terms of score) obtained in each dataset. The second shows the hyper-parameter values of the best model.

Table 3.6 Artificial Neural Network (ANN) performance.

ANN									
Dataset		**Performance**	**Desktop**	**MCUs**					
Type	**Name**	**Metrics**	**Python**	**F0**	**F3**	**F4**	**F7**	**H7**	**L4**
Binary	Heart	Acc.	84%	*	84%	84%	84%	84%	84%
		Inf. Time	<1 ms	*	3 ms	1 ms	<1 ms	<1 ms	1 ms
	Virus	Acc.	99%	*	99%	99%	99%	99%	99%
		Inf. Time	<1 ms	*	5 ms	3 ms	1 ms	1 ms	4 ms
	Sonar	Acc.	87%	*	87%	87%	87%	87%	87%
		Inf. Time	<1 ms	*	16 ms	8 ms	3 ms	3 ms	10 ms
Multiclass	Peugeot_Target 14	Acc.	99%	*	99%	99%	99%	99%	99%
		Inf. Time	<1 ms	*	2 ms	1 ms	<1 ms	<1 ms	1 ms
	Peugeot_Target 15	Acc.	99%	*	99%	99%	99%	99%	99%
		Inf. Time	<1 ms	*	18 ms	10 ms	4 ms	4 ms	12 ms
Regression	Enviro Car	R2	0.99	*	0.99	0.99	0.99	0.99	0.99
		Inf. Time	<1ms	*	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	AQI	R2	0.86	*	0.86	0.86	0.86	0.86	0.86
		Inf. Time	<1 ms	*	4 ms	2 ms	1 ms	1 ms	3 ms

*: F0 not supported by the STM X-Cube-AI package.

Table 3.7 ANN corresponding configurations.

Dataset	**Best Configuration (Table 3.1, Table 3.2)**				
	AF	**LC**	**PCA**	**Dropout**	**Scaling**
Heart	Tanh	[500]	30%	0	StandardScaler
Virus	Tanh	[100,100,100]	None	0	StandardScaler
Sonar	ReLU	[300,200,100,50]	30%	0	MinMaxScaler
Peugeot_Target 14	ReLU	[500]	None	0	StandardScaler
Peugeot_Target 15	Tanh	[300,200,100,50]	None	0	StandardScaler
EnviroCar	Tanh	[50]	mle	0	MinMaxScaler
AQI	ReLU	[300,200,100,50]	None	0	MinMaxScaler

Activation Function (AF), Layer Configuration (LC), Principal Component Analysis (PCA).

1. ANN: Remarkably, all the embedded platforms were able to achieve the same score (accuracy or R2) as the desktop python implementation. None of the chosen datasets required the compression of the models by the STM X-Cube-AI package. ANN performed well in general, except for the Heart and Virus datasets, where the accuracy is under 90%. The inference time is relatively low in both desktop and MCUs (with similar values, in the order of ms and sometimes less). However, there is an exception in some cases—especially for Peugeot_Target_15 and Sonar—when using the F3 microcontroller.

Table 3.8 Linear Support Vector Machine (SVM) performance.

Linear SVM										
Dataset		**Performance**	**Desktop**		**MCUs**					
Type	**Name**	**Metrics**	**Python**	**C**	**F0**	**F3**	**F4**	**F7**	**H7**	**L4**
Binary	Heart	Acc.	84%	84%	84%	84%	84%	84%	84%	84%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Virus	Acc.	94%	94%	94%	94%	94%	94%	94%	94%
		Inf. Time	<1 ms	<1 ms	1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Sonar	Acc.	78%	78%	78%	78%	78%	78%	78%	78%
		Inf. Time	<1 ms	<1 ms	3 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Multiclass	Peugeot_Target 14	Acc.	91%	91%	91%	91%	91%	91%	91%	91%
		Inf. Time	<1 ms	<1 ms	2 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Peugeot_Target 15	Acc.	90%	90%	90%	90%	90%	90%	90%	90%
		Inf. Time	<1 ms	<1 ms	2 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Regression	EnviroCar	R2	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		Inf. Time	<1 ms	<1 ms	5 ms	3 ms	<1 ms	<1 ms	<1 ms	<1 ms
	AQI	R2	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73
		Inf. Time	<1 ms	<1 ms	5 ms	3 ms	<1 ms	<1 ms	<1 ms	<1 ms

Table 3.9 Linear SVM corresponding configurations.

Dataset	**Best Configuration (Table 3.1, Table 3.2)**		
	C	**PCA**	**Scaling**
Heart	0.1	30%	StandardScaler
Virus	1	None	StandardScaler
Sonar	0.01	None	StandardScaler
Peugeot_Target 14	0.1	mle	StandardScaler
Peugeot_Target 15	10	mle	StandardScaler
EnviroCar	0.1	None	StandardScaler
AQI	1	mle	StandardScaler

SVM regularization parameter (C), Principal Component Analysis (PCA).

2. Linear SVM: As with ANN, for the linear SVM, we obtained the same score across all the target platforms, and relatively short inference times (again, with almost no difference between desktop and microcontroller implementations). However, we obtained significantly worse results than ANN for more than half of the investigated datasets. Table 3.9 stresses the importance of tuning the C regularization parameter, which implies the need for longer training times, particularly in the absence of normalization. We explore this in more depth when analyzing research question 9.

Table 3.10 K-Nearest Neighbors (K-NN) performance.

K-NN										
Dataset		**Performance**	**Desktop**		**MCUs**					
Type	**Name**	**Metrics**	**Python**	**C**	**F0**	**F3**	**F4**	**F7**	**H7**	**L4**
Binary	Heart	Acc.	83%	83%	83%	83%	83%	83%	83%	83%
		Inf. Time	<1 ms	<1 ms	366 ms	71 ms	7 ms	4 ms	4 ms	8 ms
	Virus	Acc.	99%	99%	95%	95%	95%	95%	95%	95%
		Inf. Time	<1 ms	<1 ms	199 ms	38 ms	4 ms	3 ms	3 ms	4 ms
	Sonar	Acc.	92%	92%	76%	92%	92%	92%	92%	92%
		Inf. Time	<1 ms	<1 ms	329 ms	140 ms	14 ms	10 ms	10 ms	14 ms
Multiclass	Peugeot_Target 14	Acc.	98%	98%	88%	88%	88%	88%	98%	88%
		Inf. Time	<1 ms	<1 ms	205 ms	39 ms	4 ms	3 ms	200 ms	4 ms
	Peugeot_Target 15	Acc.	97%	97%	86%	86%	86%	86%	97%	86%
		Inf. Time	<1 ms	<1 ms	204 ms	39 ms	4 ms	3 ms	200 ms	4 ms
Regression	EnviroCar	R2	0.99	0.99	0.97	0.97	0.97	0.97	0.97	0.97
		Inf. Time	<1 ms	<1 ms	125 ms	30 ms	3 ms	2 ms	2 ms	4 ms
	AQI	R2	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73
		Inf. Time	<1 ms	<1 ms	414 ms	85 ms	9 ms	6 ms	6 ms	10 ms

Table 3.11 K-NN corresponding configurations.

Dataset	**Best Configuration (Table 3.1, Table 3.2)**			
	K	**PCA**	**Scaling**	**Notes**
Heart	10	mle	StandardScaler	This configuration fits all targets
Virus	1	None	StandardScaler	In all MCUs K = 1, training set cap = 50
Sonar	1	None	MinMaxScaler	In F0 K = 1, training set cap = 50
Peugeot_Target 14	1	mle	MinMaxScaler	In F0, F3, F4, F7, L4 K = 4; training set cap = 100
Peugeot_Target 15	1	mle	MinMaxScaler	In F0, F3, F4, F7, L4 K = 3; training set cap = 100
EnviroCar	3	mle	MinMaxScaler	In all MCUs K = 2, training set cap = 100
AQI	1	mle	StandardScaler	This configuration fits all targets

Number of neighbors (K), Principal Component Analysis (PCA).

3. K-NN: Notably, in some cases, the training set cap needed to be set to 100, because the Flash size was a limiting factor for some MCUs. Hence, for different training sets, we also had a different number of neighbors (K). Accordingly, the accuracy is also affected by the decrease in training set size, since the number of examples used for training is reduced. This effect is apparent for Sonar with an F0 device. This dataset has sixty features, much more than the others (typically 10–20 features). The inference time varies a lot among datasets, microcontrollers and in comparison with the desktop implementations. This is because the k-NN inference algorithm always requires the exploration of the whole training set, and thus its size plays an important role in performance, especially for less powerful devices. In the multiclass problems, k-NN exploits the larger memory availability of H7 well, outperforming SVM, and reaching a performance level close to that of ANN. It is important to highlight that the

Sonar labels were reasonably well predicted by k-NN compared to ANN and SVM (92% vs. 87% and 78%). In general, k-NN achieves performance levels similar to ANN but requires a much larger memory footprint, which is possible only on the highest-end targets.

Table 3.12 Decision Tree (DT) performance.

DT										
Dataset		**Performance**	**Desktop**		**MCUs**					
Type	**Name**	**Metrics**	**Python**	**C**	**F0**	**F3**	**F4**	**F7**	**H7**	**L4**
Binary	Heart	Acc.	78%	78%	78%	78%	78%	78%	78%	78%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Virus	Acc.	99%	99%	99%	99%	99%	99%	99%	99%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Sonar	Acc.	76%	76%	76%	76%	76%	76%	76%	76%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Multiclass	Peugeot_Target 14	Acc.	99%	99%	99%	99%	99%	99%	99%	99%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
	Peugeot_Target 15	Acc.	98%	98%	98%	98%	98%	98%	98%	98%
		Inf. Time	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Regression	EnviroCar	R2	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		Inf. Time	<1 ms	<1 ms	2 ms	2 ms	<1 ms	<1 ms	<1 ms	<1 ms
	AQI	R2	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65
		Inf. Time	<1 ms	<1 ms	2 ms	2 ms	<1 ms	<1 ms	<1 ms	<1 ms

Table 3.13 DT corresponding configurations.

Dataset	**Best Configuration (Table 3.1, Table 3.2)**					
	Max Depth	**Min Sample Split**	**Min Sample Leaf**	**Max Leaf Nodes**	**PCA**	**Notes**
Heart	7	2	10	80	mle	*
Virus	None	5	1	80	None	*
Sonar	7	2	10	80	mle	*
Peugeot_Target 14	None	2	3	80	None	*
Peugeot_Target 15	None	2	1	200	None	*
EnviroCar	None	2	1	5000	None	Max leaf nodes = 1000 in F3, F4, L4; max leaf nodes = 200 for F0
AQI	None	10	3	80	None	*

Principal Component Analysis (PCA). * This configuration fits all targets.

4. DT: When processing the EnviroCar dataset, the DT algorithm saturated the memory in most of the targets. We had to reduce the leaf size for all MCU families, apart from F7 and H7. However, this reduction did not significantly reduce the R2 value. In addition, DT performs worse than the others in two binary classification datasets, Heart and Sonar, and in the AQI regression dataset as well, but performs at the same level as the ANNs for the multiclass datasets and in the EnviroCar regression problem.

Notably, DT achieves the fastest inference time among all algorithms, with F0 and F3 performing worse than the others, particularly in the regression problems.

As a rough summary of the first research question, we can conclude that ANN and, surprisingly, k-NN, had the highest accuracy in most cases, and Decision Tree had the shortest response time, but accuracy results were quite dependent on the dataset. The main difference between ANN and k-NN results is represented by the fact that high performance in ANN is achieved by all the targets (but not F0, which is not supported by the STM X-Cube-AI package), while k-NN poses much higher memory requirements. Concerning the timing performance, microcontrollers perform similarly to desktop implementations on the studied datasets. The only exception is found in k-NN, for which each inference requires the exploration of the whole dataset, and the corresponding computational demand penalizes the performance, especially on low-end devices. When comparing the edge devices, the best time performance is achieved by F7 and H7 (and we used default clock speeds, which can be significantly increased). Unsurprisingly, given the available hardware, F0 performs worse than all the others. Considering the score, we managed to train all the edge devices to achieve the same level of performance as the desktop in each algorithm, with the exception of k-NN in the multiclass tests (Peugeot), where only H7 can perform like a desktop, but with a significant time performance penalty. On the other hand, F0 performs significantly worse than the other edge devices in the k-NN Sonar binary classification.

3.4.2 Scaling

Feature pre-processing is applied to the original features before the training phase, with the goal of increasing prediction accuracy and speeding up response times [28]. Since the range of values is typically different from one feature to another, the proper computation of the objective function requires normalized inputs. For instance, the computation of the Euclidean distance between points is governed by features with a broader value range. Moreover, gradient descent converges much faster on normalized values [180].

We considered three cases that we applied on ANN, SVM, and k-NN: no scaling, MinMax Scaler, and Standard Scaler (Std) [181]. The set of tables below (Tables 3.14 - 3.18) show the accuracy of R2 for all datasets under various scaling conditions. Most common DT algorithms are invariant to monotonic transformations [182], so we did not consider DT in this analysis.

1. ANN:

Table 3.14 Performance and configuration of ANN with no scaling.

ANN				
Dataset	**None**	**Configuration**		
		AF	**LC**	**PCA**
Heart	78%	ReLU	[500]	mle
Virus	74%	Tanh	[100,100,100]	mle
Sonar	85%	ReLU	[100,100,100]	mle
Peugeot_Target 14	95%	Tanh	[500]	mle
Peugeot_Target 15	92%	ReLU	[100,100,100]	mle
EnviroCar	0.97	Tanh	[50]	mle
AQI	0.7	ReLU	[300,200,100,50]	None

Activation Function (AF), Layer Configuration (LC), Principal Component Analysis (PCA).

Table 3.15 Performance and configuration of ANN with MinMax scaling.

ANN				
Dataset	**MinMax**	**Configuration**		
		AF	**LC**	**PCA**
Heart	80%	Tanh	[300,200,100,50]	30%
Virus	99%	ReLU	[100,100,100]	None
Sonar	87%	ReLU	[300,200,100,50]	30%
Peugeot_Target 14	99%	ReLU	[100,100,100]	mle
Peugeot_Target 15	98%	Tanh	[300,200,100,50]	mle
EnviroCar	0.99	ReLU	[50]	mle
AQI	0.86	ReLU	[300,200,100,50]	None

Activation Function (AF), Layer Configuration (LC), Principal Component Analysis (PCA).

Table 3.16 Performance and configuration of ANN with StandardScaler normalization.

ANN				
Dataset	**Std**	**Configuration**		
		AF	**LC**	**PCA**
Heart	84%	Tanh	[500]	30%
Virus	99%	Tanh	[100,100,100]	None
Sonar	86%	ReLU	[100,100,100]	mle
Peugeot_Target 14	99%	ReLU	[500]	None
Peugeot_Target 15	99%	Tanh	[300,200,100,50]	None
EnviroCar	0.99	Tanh	[50]	mle
AQI	0.84	ReLU	[300,200,100,50]	None

Activation Function (AF), Layer Configuration (LC), Principal Component Analysis (PCA).

2. SVM:

Table 3.17 Performance and configuration of SVM for different scaling techniques.

Linear SVM									
Dataset	None	Configuration		MinMax	Configuration		Std	Configuration	
		C	PCA		C	PCA		C	PCA
Heart	78%	0.01	mle	79%	1	mle	84%	0.1	30%
Virus	71%	0.01	mle	94%	100	None	94%	1	None
Sonar	77%	0.1	mle	73%	0.1	None	78%	0.01	None
Peugeot_Target 14	50%	0.1	mle	91%	10	mle	91%	0.1	mle
Peugeot_Target 15	76%	0.01	mle	90%	10	mle	90%	10	mle
EnviroCar	0.97	Slow	mle	0.98	0.1	None	0.99	0.1	None
AQI	0.7	100	mle	0.62	100	30%	0.73	1	mle

SVM regularization parameter (C), Principal Component Analysis (PCA).

3. K-NN:

Table 3.18 Performance and configuration of SVM for different scaling techniques.

K-NN									
Dataset	None	Configuration		MinMax	Configuration		Std	Configuration	
		K	PCA		K	PCA		K	PCA
Heart	63%	3	mle	75%	3	None	83%	10	mle
Virus	95%	1	mle	99%	1	None	99%	1	None
Sonar	77%	1	mle	92%	1	None	87%	1	None
Peugeot_Target 14	91%	1	mle	98%	1	mle	97%	1	mle
Peugeot_Target 15	89%	2	mle	97%	1	mle	97%	1	None
EnviroCar	0.98	5	mle	0.99	3	mle	0.99	3	None
AQI	0.73	4	mle	0.7	3	mle	0.73	6	mle

Number of neighbors (K), Principal Component Analysis (PCA).

These results clearly show the importance across all the datasets and algorithms of scaling the inputs. For instance, MinMax scaling allowed ANNs to reach 99% accuracy in Virus (from a 74% baseline), and Peugeot 14 (from 95%) and 0.86 R2 (from 0.70) in AQI. The application of MinMax allowed SVM to achieve 94% accuracy in Virus (form 71%) and 91% accuracy in Peugeot 14 (from 50%). Standard input scaling improved the k-NN accuracy of Heart from 63% to 83%. For large regression datasets, especially with SVM (see also research question 9), input scaling avoids large training times.

3.4.3 Principal Component Analysis (PCA)

Dimensionality reduction allows us to reduce the effects of noise, space and processing requirements. One well-known method is Principal Component Analysis (PCA), which performs an orthogonal transformation to convert a set of observations of possibly correlated

variables into a set of values of linearly independent variables, which are called principal components [183]. We tried different values of PCA dimension reduction: none, 30% (i.e., the algorithm selects a number of components such that the amount of variance that needs to be explained is greater than 30%), and automatic maximum likelihood estimation (mle) [184], whose results are shown in Tables 3.19 – 3.26.

1. ANN:

Table 3.19 ANN performance and configuration for PCA = None.

ANN				
Dataset	**Score**	**Configuration**		
		AF	**LC**	**Scaling**
Heart	81%	Tanh	[100,100,100]	Std
Virus	99%	Tanh	[100,100,100]	Std
Sonar	84%	ReLU	[50]	Std
Peugeot_Target 14	99%	ReLU	[500]	Std
Peugeot_Target 15	99%	Tanh	[300,200,100,50]	Std
EnviroCar	0.99	Tanh	[50]	MinMax
AQI	0.86	ReLU	300,200,100,50]	MinMax

Activation Function (AF), Layer Configuration (LC).

Table 3.20 ANN performance and configuration for PCA = 30%.

ANN				
Dataset	**Score**	**Configuration**		
		AF	**LC**	**Scaling**
Heart	84%	Tanh	[500]	Std
Virus	98%	ReLU	[100,100,100]	Std
Sonar	87%	ReLU	[300,200,100,50]	MinMax
Peugeot_Target 14	92%	ReLU	[50]	MinMax
Peugeot_Target 15	90%	ReLU	[50]	MinMax
EnviroCar	0.98	ReLU	[50]	MinMax
AQI	0.71	ReLU	[100,100,100]	MinMax

Activation Function (AF), Layer Configuration (LC).

Table 3.21 ANN performance and configuration for PCA = mle.

ANN				
Dataset	**Score**	**Configuration**		
		AF	**LC**	**Scaling**
Heart	83%	Tanh	[300,200,100,50]	Std
Virus	99%	Tanh	[100,100,100]	Std
Sonar	86%	ReLU	[100,100,100]	Std
Peugeot_Target 14	99%	Tanh	[100,100,100]	Std
Peugeot_Target 15	99%	Tanh	[300,200,100,50]	Std
EnviroCar	0.99	Tanh	[50]	MinMax
AQI	0.82	Tanh	[300,200,100,50]	MinMax

Activation Function (AF), Layer Configuration (LC).

2. SVM:

Table 3.22 SVM performance and configuration for various PCA values.

SVM									
	PCA = None			PCA = 30%			PCA = mle		
Dataset	Score	Configuration		Score	Configuration		Score	Configuration	
		C	Scaling		C	Scaling		C	Scaling
Heart	78%	0.01	Std	84%	0.1	Std	79%	0.1	Std
Virus	99%	1	Std	86%	100	MinMax	94%	0.1	Std
Sonar	78%	0.01	Std	75%	100	Std	77%	0.01	Std
Peugeot_Target 14	91%	10	MinMax	83%	0.1	MinMax	91%	0.1	Std
Peugeot_Target 15	90%	10	MinMax	86%	10	MinMax	90%	10	Std
EnviroCar	0.99	0.1	Std	0.94	0.1	MinMax	0.98	0.1	MinMax
AQI	0.73	10	Std	0.71	100	Std	0.73	1	Std

SVM regularization parameter (C).

3. K-NN:

Table 3.23 K-NN performance and configuration for various PCA values.

K-NN									
	PCA = None			PCA = 30%			PCA = mle		
Dataset	Score	Configuration		Score	Configuration		Score	Configuration	
		K	Scaling		K	Scaling		K	Scaling
Heart	77%	3	Std	76%	13	Std	83%	10	Std
Virus	99%	1	Std	99%	1	Std	99%	1	Std
Sonar	92%	1	MinMax	84%	1	MinMax	97%	1	Std
Peugeot_Target 14	98%	1	MinMax	92%	3	MinMax	98%	1	Std
Peugeot_Target 15	97%	1	MinMax	90%	6	MinMax	97%	1	Std
EnviroCar	0.99	3	MinMax	0.99	9	Std	0.99	3	MinMax
AQI	0.73	6	Std	0.57	3	MinMax	0.73	6	Std

Number of neighbors (K).

4. DT:

Table 3.24 DT performance and configuration for PCA = None.

DT					
Dataset	Score	Configuration			
		Max Depth	Min Sample Split	Min Sample Leaf	Max Leaf Nodes
Heart	67%	7	2	10	80
Virus	99%	None	5	1	80
Sonar	65%	7	2	10	80
Peugeot_Target 14	99%	None	2	3	80
Peugeot_Target 15	98%	None	2	1	200
EnviroCar	0.99	None	2	1	5000
AQI	0.65	None	10	3	80

Table 3.25 DT performance and configuration for PCA = 30%.

DT					
Dataset	**Score**	**Configuration**			
		Max Depth	**Min Sample Split**	**Min Sample Leaf**	**Max Leaf Nodes**
Heart	62%	3	2	1	80
Virus	96%	None	2	1	1000
Sonar	75%	7	2	3	80
Peugeot_Target 14	84%	None	2	1	200
Peugeot_Target 15	87%	7	2	10	80
EnviroCar	0.98	None	2	10	1000
AQI	0.62	7	10	1	80

Table 3.26 DT performance and configuration for PCA = mle.

DT					
Dataset	**Score**	**Configuration**			
		Max Depth	**Min Sample Split**	**Min Sample Leaf**	**Max Leaf Nodes**
Heart	78%	7	2	10	80
Virus	99%	None	2	1	200
Sonar	76%	7	2	10	80
Peugeot_Target 14	93%	None	5	1	80
Peugeot_Target 15	92%	None	10	1	80
EnviroCar	0.98	None	2	10	5000
AQI	0.49	7	5	1	80

The results reported in the above tables are quite varied. The 30% PCA value is frequently too low, except for the Sonar dataset, which has 60 features, much more than the others, and thus looks less sensitive to such a coarse reduction. For SVM, PCA does not perform better than or equal to mle, while the opposite is true for k-NN. Moreover, in ANNs, mle tends to provide better results, except AQI. In DT, there is a variance of outcomes. Mle does not perform any better than the other algorithms in Heart (78% vs. 67% accuracy) and Sonar (76% vs. 65%), while performance decreases for Peugeot 14 (93% vs. 99%) and AQI (0.49 vs. 0.65). For AQI, PCA never improves performance. The opposite is true for Heart and (except SVM) Sonar.

3.4.4 ANN Layer Configuration

To answer this question, we investigated performance among four ANN hidden-layer configurations, as follows:

1. One hidden layer of 50 neurons;
2. One hidden layer of 500 neurons;

3. Three hidden layers of 100 neurons each;

4. Four hidden layers with 300, 200, 100 and 50 neurons, respectively.

Tables 3.27 – 3.29 indicate the highest performance for each layer shape.

Table 3.27 Layer configuration LC = [50] and LC = [500] results. We have omitted columns with Dropout values of all zeros.

Dataset	LC = [50]	Configuration			LC = [500]	Configuration		
		AF	PCA	Scaling		AF	PCA	Scaling
Heart	83%	Tanh	30%	Std	84%	Tanh	30%	Std
Virus	98%	ReLU	None	Std	98%	ReLU	None	Std
Sonar	84%	ReLU	None	Std	81%	ReLU	mle	Std
Peugeot_Target 14	98%	ReLU	mle	Std	99%	ReLU	None	Std
Peugeot_Target 15	98%	ReLU	mle	Std	98%	ReLU	None	Std
EnviroCar	0.99	Tanh	mle	MinMax	0.99	ReLU	mle	MinMax
AQI	0.76	Tanh	mle	MinMax	0.78	ReLU	mle	MinMax

Activation Function (AF), Principal Component Analysis (PCA).

Table 3.28 Layer configuration LC = [100,100,100] results.

Dataset	LC = [100,100,100]	Configuration			
		AF	PCA	Scaling	Dropout
Heart	84%	Tanh	30%	Std	0
Virus	99%	Tanh	None	Std	0
Sonar	87%	ReLU	30%	Std	0
Peugeot_Target 14	99%	ReLU	mle	Std	0
Peugeot_Target 15	98%	Tanh	mle	Std	0.1
EnviroCar	0.99	Tanh	mle	MinMax	0
AQI	0.80	ReLU	mle	MinMax	0

Activation Function (AF), Principal Component Analysis (PCA).

Table 3.29 Layer configuration LC = [300,200,100,50] results.

Dataset	LC = [300,200,100,50]	Configuration		
		AF	PCA	Scaling
Heart	84%	Tanh	30%	Std
Virus	99%	ReLU	None	Std
Sonar	87%	ReLU	30%	MinMax
Peugeot_Target 14	99%	ReLU	mle	MinMax
Peugeot_Target 15	99%	Tanh	None	Std
EnviroCar	0.99	Tanh	mle	MinMax
AQI	0.86	ReLU	None	MinMax

Activation Function (AF), Principal Component Analysis (PCA).

By observing the results, we can see that deepening the network tends to improve the results, but only up to a certain threshold. For the Heart dataset, which has the lowest overall

accuracy, we tried additional, deeper shapes beyond those reported in Tables 3.27 – 3.29, but with no better results. On the other hand, widening the first layer provides only slightly better results (and in one case worsens them).

3.4.5 ANN Activation Function

Another relevant design choice concerns the activation function in the hidden layers. Activation functions are attached to each neuron in the network and define its output. They introduce a non-linear factor in the processing of a neural network. Two activation functions are typically used: Rectified Linear Unit (ReLU) and Tangent Activation Function (Tanh). On the other hand, for the output layer, we used a sigmoid for binary classification models as an activation function, and a softmax for multiclassification tasks. For regression problems, we created an output layer without any activation function (i.e., we use the default "linear" activation), as we are interested in predicting numerical values directly, without transformation. Tables 3.30 and 3.30 show the highest accuracy achieved in hidden layers for each function, alongside its corresponding configuration.

Table 3.30 ReLU activation function results.

Dataset	ReLU	Best Configuration		
		LC	PCA	Scaling
Heart	83%	[500]	30%	Std
Virus	99%	[100,100,100]	mle	Std
Sonar	87%	[300,200,100,50]	30%	MinMax
Peugeot_Target 14	99%	[500]	None	Std
Peugeot_Target 15	99%	[300,200,100,50]	None	Std
EnviroCar	0.99	[50]	mle	MinMax
AQI	0.86	[300,200,100,50]	None	MinMax

Layer Configuration (LC), Principal Component Analysis (PCA).

Table 3.31 Tanh activation function results.

Dataset	Best tanh	Best Configuration			
		LC	PCA	Scaling	Dropout
Heart	84%	[500]	30%	Std	0
Virus	99%	[100,100,100]	None	Std	0
Sonar	81%	[50]	30%	MinMax	0
Peugeot_Target 14	99%	[100,100,100]	mle	Std	0
Peugeot_Target 15	99%	[300,200,100,50]	None	Std	0
EnviroCar	0.99	[50]	mle	MinMax	0
AQI	0.82	[300,200,100,50]	mle	MinMax	0.1

Layer Configuration (LC), Principal Component Analysis (PCA).

The results are similar, with a slight prevalence of ReLU, with a valuable difference for Sonar (+7% accuracy) and AQI (+5% R2).

3.4.6 ANN Batch Size

The batch size is the number of training examples processed in one iteration before the model being trained is updated. To test the effect of this parameter, we considered three values, one, 10, and 20, keeping the number of epochs fixed to 20. Table 3.32 shows the accuracy of each dataset for various batch sizes.

Table 3.32 Performance on different Batch sizes.

Dataset	Epoch=20		
	Batch size = 1	Batch size = 10	Batch size = 20
Heart	84%	84%	84%
Virus	99%	99%	99%
Sonar	87%	87%	84%
Peugeot_Target 14	98%	99%	98%
Peugeot_Target 15	97%	99%	99%
EnviroCar	0.99	0.99	0.99
AQI	0.63	0.86	0.76

The results show that the value of 10 provides optimal results in terms of accuracy. The difference becomes relevant only in the case of AQI. A batch size equal to one poses an excessive time overhead (approximately 30% slower than the batch size of 10), while a batch size of 20 achieves a speedup of about 40%.

3.4.7 ANN Accuracy vs. Epochs

ANN training goes through several epochs, where an epoch is a learning cycle in which the learner model sees the whole training data set. Figures 3.3 and 3.4 show that the training of ANN on all datasets converges quickly within 10 epochs.

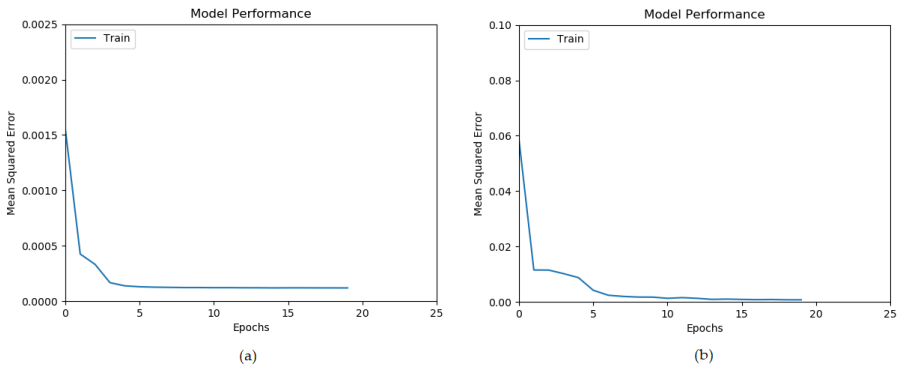

Fig. 3.3 Mean Squared Error vs. Epochs for (a) EnviroCar, and (b) air quality index (AQI).

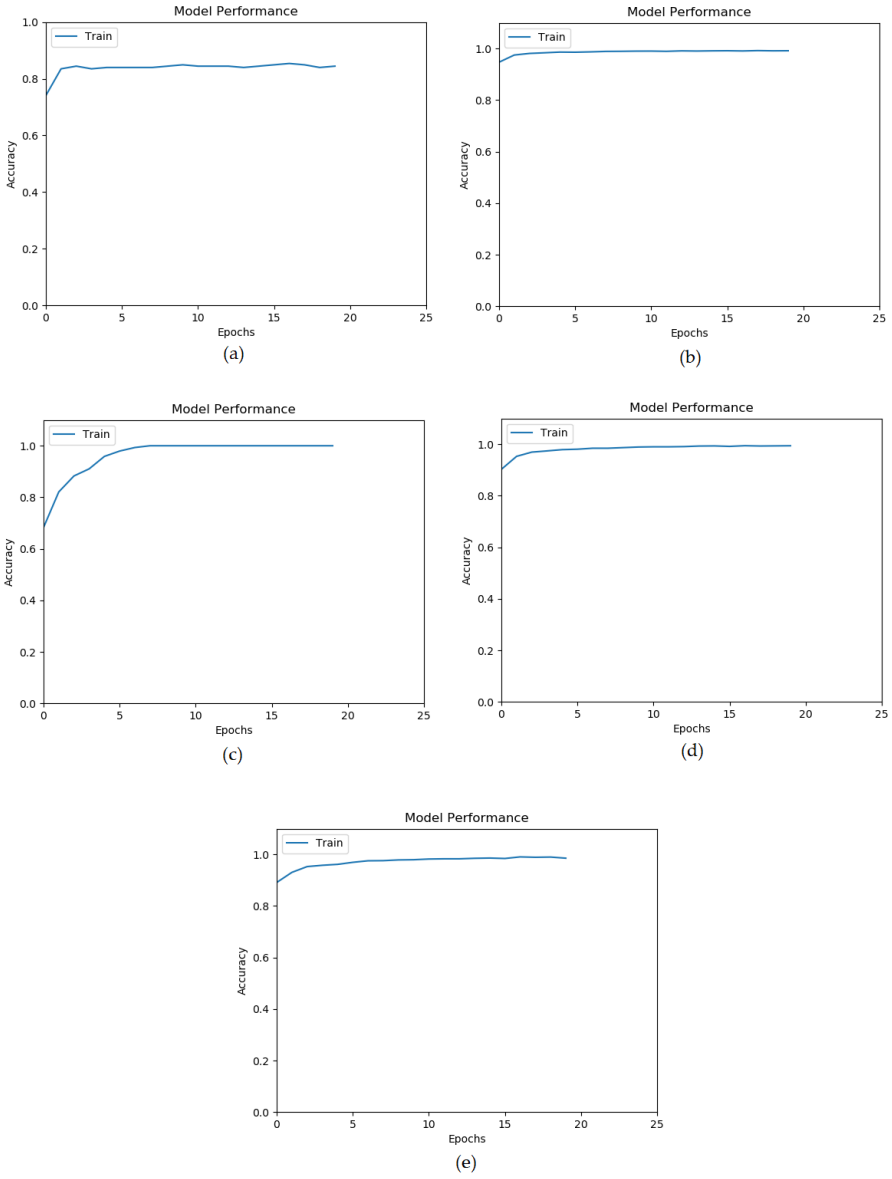

Fig. 3.4 Accuracy vs. Epochs for (a) Heart, (b) Virus, (c) Sonar, (d) Peugeot target 14, and (e) Peugeot target 15

3.4.8 ANN Dropout

Dropout is a simple method to prevent overfitting in ANNs. It consists of randomly ignoring a certain number of neuron outputs in a layer during the training phase.

The results in Table 3.33 show that this regularization step provides no improvement in the considered cases, but has a slight negative effect in a couple of datasets (Sonar and AQI).

Table 3.33 Dropout effect on ANN.

Dataset	Dropout	
	0	0.1
Heart	84%	84%
Virus	99%	99%
Sonar	87%	83%
Peugeot_Target 14	99%	99%
Peugeot_Target 15	99%	99%
EnviroCar	0.99	0.99
AQI	0.86	0.82

3.4.9 SVM Regularization Training Time

In SVM, C is a key regularization parameter, that controls the tradeoff between errors of the SVM on training data and margin maximization [154, 185]. The classification rate is highly dependent on this coefficient, as confirmed by Tables 3.8 and 3.9. Desk-LM uses the grid search method to explore the C values presented by the user, which require long waiting times in some cases. To quantify this, we measured the training latency time in a set of typical values (C = 0.01, 0.1, 1, 10, and 100), with the results provided in Table 3.34.

Different values of the C parameter have an impact on the training time. The table shows that higher C values require higher training time. We must stress that the above results represent the training time for the best models. In particular, when no normalization procedure was applied, the training time using large values of C became huge (also up to one hour), especially for regression datasets.

Table 3.34 Training time for different values of the C parameter.

Dataset	C parameter				
	0.01	0.1	1	10	100
Heart	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Virus	100 ms	300 ms	500 ms	600 ms	600 ms
Sonar	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
Peugeot_Target 14	<1 ms	100 ms	200 ms	300 ms	400 ms
Peugeot_Target 15	<1 ms	100 ms	300 ms	300 ms	300 ms
EnviroCar	100 ms	100 ms	100 ms	100 ms	100 ms
AQI	<1 ms	<1 ms	<1 ms	<1 ms	300 ms

3.4.10 DT Parameters

Tuning a decision tree requires us to test the effect of various hyper-parameters, such as *max_depth*, *min_simple_split*. Figure 3.5 shows the distribution of the tested parameter values for the best models in the different datasets (see also Table 3.13 to see the best results).

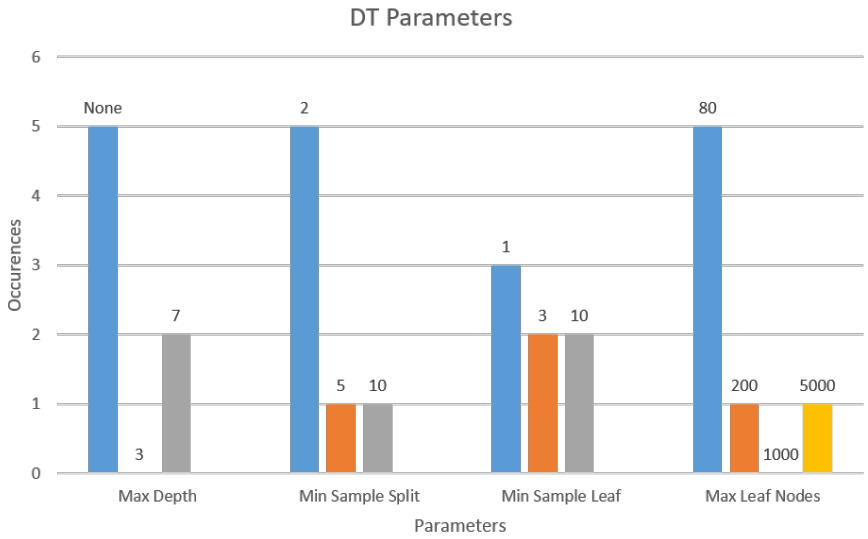

Fig. 3.5 Number of occurrences of each DT parameter.

3.5 Conclusion

This chapter presented the Edge Learning Machine (ELM), a machine learning platform for edge devices. It performs training on desktop computers, exploiting TensorFlow, Keras, and scikit-learn, and makes inferences on microcontrollers. Initially, ELM implements, in platform-independent C language, three supervised machine learning algorithms (Linear SVM, k-NN, and DT), and exploits the STM X-Cube-AI package for implementing ANNs on STM32 Nucleo boards. The training phase on Desk-LM searches for the best configuration across a variety of user-defined parameter values. To investigate the performance of these algorithms on the targeted devices, we posed ten research questions (RQ 1–10, in the following) and analyzed a set of six datasets (four classifications and two regressions). To the best of our knowledge, this is the first work presenting such an extensive performance analysis of edge machine learning in terms of datasets, algorithms, configurations, and types of devices.

Our analysis shows that, on a set of available IoT data, we managed to train all the targeted devices to achieve, with at least one algorithm, the best score (classification accuracy or regression R2) obtained through a desktop machine (RQ1). ANN performs better than the other algorithms in most cases, without differences among the target devices (apart from F0, which is not supported by STM X-Cube-AI). k-NN performs similarly to ANN, and in one case even better, but requires that all the training sets are kept in the inference phase, posing a significant memory demand, which penalizes time performance, particularly on

low-end devices. The performance of the Decision Tree varied widely across datasets. When comparing edge devices, the best time performance is achieved by F7 and H7. Unsurprisingly, given the available hardware, F0 performs worse than all the others.

The prep-rocessing phase is extremely important. Results across all the datasets and algorithms show the importance of scaling the inputs, which lead to improvements of up to 82% in accuracy (SVM Virus) and 23% in R2 (k-NN Heart) (RQ2). The applications of PCA have various effects across algorithms and datasets (RQ3).

In terms of the ANN hyper-parameters, we observed that increasing the depth of a NN typically improves its performance, up to a saturation level (RQ4). When comparing the neuron activation functions, we observed a slight prevalence of ReLU over Tanh (RQ5). The batch size has little influence on the score, but it does have an influence on training time. We established that 10 was the optimal value for all the examined datasets (RQ6). In all datasets, the ANN training quickly converges within 10 epochs (RQ7). The dropout regularization parameter only led to some slight worsening in a couple of datasets (RQ8).

In SVM, the C hyper-parameter value selection has an impact on training times, but only when inputs are not scaled (RQ9). In most datasets, the whole tree depth is needed for DT models, and this does not exceed the memory available in the microcontrollers. However, the values of Max_Leaf_Nodes usually require a low threshold value (80) (RQ10).

As synthesized above, in general, several factors impact performance in different ways across datasets. This highlights the importance of a framework like ELM, which can test different algorithms, each one with different configurations. The framework is released on an open-source basis (ELM), to support researchers in designing and deploying ML solutions on edge devices. ELM has been extended by further tools for the TinyML area, as will be shown in the following chapters.

Chapter 4

Self-Learning Pipeline for Low-Energy Resource-Constrained Devices

4.1 Introduction

The maturation of machine learning (ML) is enabling embedding intelligence in Internet of Things (IoT) devices [45]. Various effective machine and deep learning models have been deployed on resource-constrained devices to gain insights from collected data. Field devices, such as microcontrollers, consume significantly less energy than larger devices, especially if their memory footprint is minimized and huge edge-cloud raw data transmission can be spared by only sending high-level information.

Typically, ML models are trained on high-performance computers in the cloud and then deployed on IoT devices to perform the target inference task. However, in real-world applications, statically trained models are not able to adapt to the environment, which could reduce the accuracy of new samples. Training a ML model on the device, instead, can learn from the physical world and update the system locally. This allows for lifelong incremental learning [186] updating its knowledge, as well as personalizing the device thus improving performance by learning the characteristics of the specific deployment context.

Moreover, as millions of new IoT devices are produced every year [187], vast amounts of unlabeled data in various domains (e.g., industrial, healthcare, environmental, etc.) are generated. Designing systems based solely on supervised learning, which requires a labeled dataset, is not ideal for IoT scenarios. Creating labeled IoT datasets is expensive and impractical [188]. Therefore, it is of utmost importance to smartly leverage the quantity of unlabeled data produced by the end devices deployed in the field.

Semi-supervised and self-supervised learning techniques have shown encouraging results in learning from unlabeled data. The former approach exploits a partially labeled dataset [31], while the latter trains a model using automatically generated pseudo-labels [128]. Fully automated solutions look particularly suited for applications deployable in hostile environments, possibly due to difficulties in communications or the need to keep energy consumption and/or costs low. Target applications may involve, for instance, remote agriculture, farming, mining, manufacturing, emergency, or the military.

We believe that for ML-driven extreme IoT to reach its full potential, it is crucial to experiment and invest in solutions that can be improved over time with minimal human intervention. Therefore, in this chapter, we aim to investigate the feasibility of a self-learning system in an embedded environment and deepen the performance analysis by addressing three research questions. First, we investigated the possibility of increasing the robustness of a sample selection for automatic training, since the training data in such an autonomous system should be as compact and as noise free as possible to avoid performance degradation. We are interested in sample selection rather than feature selection (another method of dataset reduction) because we assume that the sensor configuration of a device used in the field is already optimized for the target application. Second, we explored the potential benefits of a memory management algorithm for field devices with limited resources to minimize memory usage associated with the continuous flow of samples that can be used for the training set. Third, we performed timing analysis on a state-of-the-art microcontroller to evaluate the ability of the overall system to meet real-time performance requirements.

We have performed this analysis on the Self-Learning Autonomous Edge Learning and Inferencing Pipeline (AEP) system. AEP is a software system we have developed for resource-constrained devices within the ELM environment. Besides the inference capability, the AEP allows autonomous field training utilizing a two-stage pipeline, involving label generation with a confidence measure step and on-device training. This chapter presents an in-depth evaluation of the proposed system on two publicly available datasets for IoT applications, using the NUCLEO-H743ZI2 microcontroller.

4.2 Autonomous Edge Pipeline (AEP) Algorithms

Before presenting the AEP system architecture and the supported workflow, we give a short overview of the underlying algorithms. Particularly, we use k-means clustering for unsupervised learning, which provides the binary labels for training a decision tree or a k-NN classifier (supervised learning), which is used for the actual classification of the samples.

- **Unsupervised Learning** is a branch of ML that derives interesting patterns and insights directly from unlabeled datasets. Clustering consists of the unsupervised classification of (unlabeled) samples into groups or clusters such that data points in the same group are similar to each other and different from data points in other clusters [189]. In the AEP implementation, we use one of the most common clustering methods, namely k-means clustering. This algorithm works iteratively to partition a set of observations into k (predetermined) distinct and non-overlapping clusters based on feature similarity. It first selects k random data points as initial centroids. In the next step, each data sample is assigned to the closest centroid based on a certain proximity measure. Once all the data points are assigned and the clusters are formed, the centroids are updated with the mean of all the data samples belonging to the same cluster. The algorithm iteratively repeats these two steps until a convergence criterion is met [21]. One drawback of the standard k-means is its sensitivity to the initial placement of centroids. Therefore, our AEP implementation uses the k-means++ algorithm [190], which combines the standard k-means with a smarter initialization of the centroids. k-means++ first chooses a random point from the data as the first centroid. Then, for each instance in the dataset, k-means++ computes the distance to the nearest centroid previously selected. Then, it selects the next centroid from the remaining data points such that it has the largest distance to the closest centroid previously selected. These steps are repeated until k centroids are chosen. As we will see in Subsection 4.3.2 ("Development challenges"), the AEP also exploits the soft k-means enhancement, which provides a confidence value when assigning samples to clusters, thus allowing an increase in the robustness of the system.

- **Supervised Learning** is the most common branch of ML and is implemented by our AEP system to classify the samples based on the pseudo-labels generated through the unsupervised learning module. Supervised learning is the process of deriving a function that maps an input to an output, based on labeled training data. Each training example is a pair consisting of an input vector (features) and a desired output (class or value, for classification and regression, respectively) [191]. The AEP system currently features two supervised learning algorithms, namely K-Nearest Neighbor and Decision Tree (see Sections 3.2.3 and 3.2.4 for definition).

We have selected these three algorithms (one for labeling and two alternatives for classification), at least as an initial choice, keeping into account the limitations on memory and computational resources of the targeted edge devices. The k-means method is efficient with guaranteed convergence, and fast when running on small processors with low capabilities

[21]. K-NN is a simple algorithm with good performance and requires no training [156]. Finally, DTs require little to no data pre-processing effort and can effectively handle missing values in the data [158].

4.3 Autonomous Edge Pipeline

4.3.1 Framework Overview

The self-learning AEP (Figure 4.1) is an iterative pipeline that alternates between clustering/training and classification. Particularly, the k-means clustering is periodically executed on the input stream and provides the pseudo-labeled clustered data. The labeling results are then evaluated by a confidence algorithm (if enabled by the user), presented in Subsection 4.3.2, which makes a binary decision whether to keep or discard each instance. Once important samples with their corresponding pseudo-labels are selected, the training process is executed (this applies to the DT case only, since K-NN does not involve a training phase). The resulting classifier then continuously classifies the incoming samples. Given the resource limitations of the target devices, a memory management strategy is implemented to prevent memory overflow, which is explained in the following subsection.

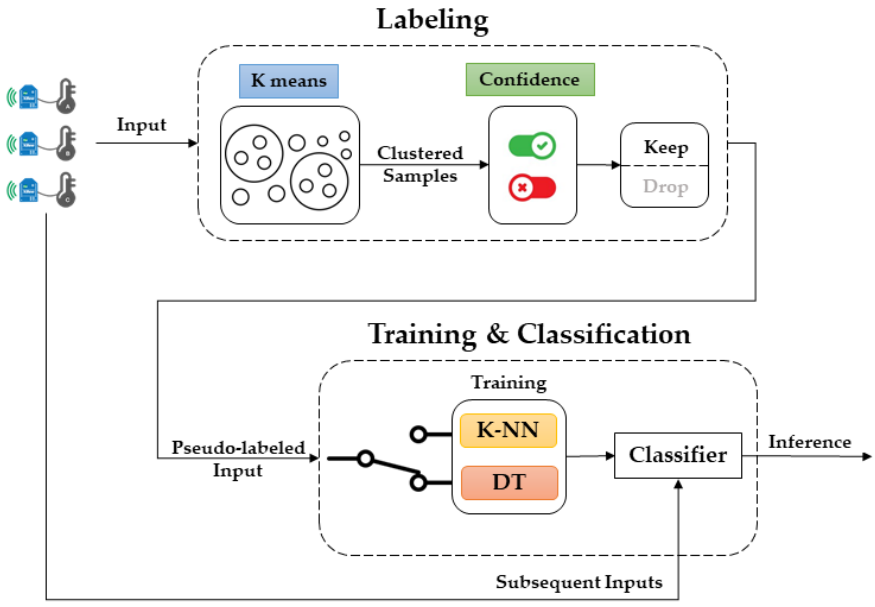

Fig. 4.1 Overview of the AEP.

4.3.2 Development Challenges

In the following, we discuss the main challenges we faced in developing the AEP system: filtering of samples to be used for the training; dataset management on the edge and DT training on the edge.

- Filtering of samples to be used for the training. k-means is a hard clustering algorithm, as the data points are assigned exclusively to one cluster. However, in some situations (e.g., in our case in which clustering is used to define labels to train a classifier), it is important to find out how confident the algorithm is in each one of its decisions. The soft k-means improvement calculates a weight that determines the extent to which each data sample belongs to each cluster [26]. Higher values indicate a certain or strong assignment, and lower weights indicate a weak assignment. The soft k-means weights can also be computed a posteriori, using cluster centers obtained by a hard k-means, exploiting Equation 4.1:

$$w_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|}\right)^{\frac{2}{m-1}}} \tag{4.1}$$

  where x_i is a data point, c_j is the coordinates of the j^{th} cluster center, and m is a parameter that controls the fuzziness of the algorithm, typically set to 2 [26]. Once the soft k-means weights for each sample are computed (using the Euclidean distance), a Confidence threshold can be set (e.g., with a value of 0.9), so to remove all samples that cannot be assigned to a cluster with a higher weight than the threshold. This approach is beneficial in terms of removing outliers or data points in uncertain regions and can improve clustering results and overall system performance, particularly robustness;

- Dataset memory management on the edge. In resource-constrained scenarios, there is a need to manage the amount of data to prevent memory overflow caused by continuous sampling and consequent increase of the training set. Additionally, the K-NN algorithm is also very memory hungry as it needs to store the entire training set for the inference phase. To tackle these challenges, we implemented three memory management algorithms: First In, First Out (*FIFO*), which removes the older samples, when the memory is full; Random Memory Filtering (*RND*); and *CONF* Memory Filtering, which retains in memory samples having higher confidence values. The impact on the performance of these memory management strategies is compared in the experimental analysis section.

- Training the decision tree on the edge. The DT training algorithm used in the AEP is implemented from scratch in C language, as we could not find a publicly avail-

able version for Cortex-M microcontrollers. The implemented training algorithm is Classification and Regression Trees (CART), which constructs binary trees using the features and thresholds that yield the largest information gain at each node [156]. Our implementation allows the user to specify two hyper-parameters for tree configuration, maximum depth of the tree and a minimum number of samples required to split an internal node. To simplify the implementation on the target device, we implemented only the "Gini" splitting criterion which is less computationally intensive than the entropy criterion.

4.3.3 AEP Workflow and Memory Management

The workflow supported by the AEP is sketched in Figure 4.2. In the picture, the term AutoDT refers to a decision tree classifier trained on the pseudo-labeled data (obtained by the k-means clustering algorithm running on the field device). Similarly, AutoKNN is the alternative K-NN classifier, which requires no training, but directly exploits the pseudo-labeled input dataset.

At the start-up, the AEP has no knowledge or data. The initial phase thus consists of filling the Memory with data samples up to a user-specified initial threshold (i.e., *INITIAL_THR*). Once the threshold is reached, the k-means clustering is run on the recorded samples and returns their pseudo-labels. If the user sets the AutoDT option, the DT training algorithm is executed. As anticipated, the user can set the max *tree-depth* and min *split-samples* hyper-parameters for this algorithm. On the other hand, if AutoKNN is selected, the collected samples together with their corresponding pseudo-labels are ready to be used by the K-NN classifier, and the only user-selectable hyper-parameter is the number of neighbors, k.

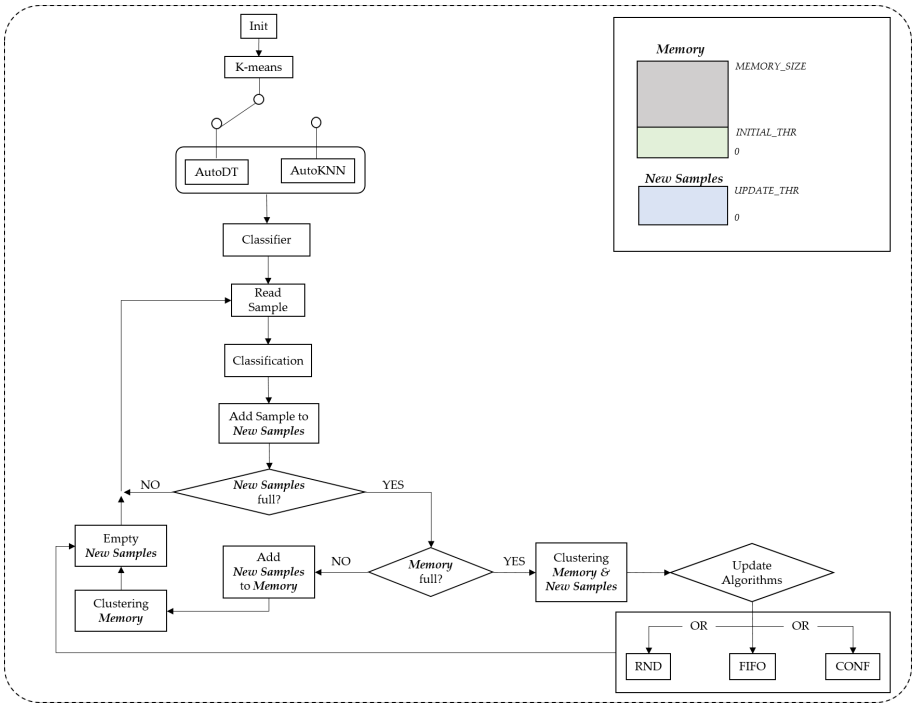

Fig. 4.2 The AEP workflow.

After this start-up, the operation loop can begin. The trained model is used to classify the subsequent sample stream. To improve (or adapt) the classifier performance, the new samples can also be stored in a dedicated memory section, called "New Samples", until an update threshold specified by the user is reached (e.g., *UPDATE_THR*). Then, k-means clustering is performed again for all samples in Memory, providing the new set of pseudo-labels and updating the classifier accordingly. If the Memory is full, one of the three data filtering algorithms available (i.e., *RND*, *FIFO*, or *CONF* – cfr. Previous sub-section) is run to prevent Memory overflow. We should emphasize that the samples are considered independent and identically distributed (iid). In machine learning theory, the iid assumption is made for datasets to imply that all their samples have the same probability distribution and are mutually independent (e.g., the data distribution does not change over time or space).

The AEP is written in platform-independent C language (i.e., it does not use native OS libraries), which enables code mobility across different platforms. The proposed pipeline is integrated with the Edge Learning Machine (ELM) framework and can be used also on various types of microcontrollers and resource-constrained devices.

4.4 Experimental Analysis and Result

The experimental analysis was performed using a NUCLEO-H743ZI2 board with an Arm Cortex-M7 core running at 480 MHz, with 2 MB flash memory and 1 MB SRAM. The STM32 H7 series is a family of high performance microcontrollers offering higher security and multimedia features [167].

To evaluate the performance of the Autonomous Edge Pipeline (AEP), we have selected two binary classification datasets that are representative of important IoT domains, such as health and industry. The Pima Indians Diabetes Database is a well-established dataset that is commonly used in the field of machine learning and data analysis for the identification and forecasting of diabetes. The database comprises 768 observations and 8 key features, including demographic and medical information, such as age, blood pressure, skin thickness, insulin levels, and others. As per the research paper by Saiteja *et al.* [192], the database provides valuable insights into the diagnostic measurements that are crucial in determining the presence of diabetes. The Semiconductor Manufacturing (SECOM) dataset represents a comprehensive collection of data collected from a semiconductor manufacturing process. This dataset encompasses information from various stages of the manufacturing process, such as the production of wafers, the application of materials, and the evaluation of the final product. The purpose of this dataset is to enhance the efficiency and quality of the semiconductor manufacturing process. With 1567 observations, each observation is a combination of 590 sensor measurements and a pass/fail test label, the SECOM dataset is a valuable resource for the field [193]. The selection of these two datasets was made with the intention of assessing the AEP in a wide range of field data typologies. The diabetes dataset represents a major application domain in the health sector, while the SECOM dataset represents an application domain in the industrial sector. By including datasets from different domains, we aim to demonstrate the generalizability of the AEP and its ability to handle diverse data inputs, which is crucial in the TinyML field where edge devices are deployed in various environments and application domains. The choice of binary classification datasets also aligns with the current trends in the field where the focus is on efficient and accurate autonomous decision making on the edge.

To make the dataset fit into the MCU memory and mimic a field device environment, we reduced both the datasets to only four features, by applying feature selection using the scikit-learn library f_classif [194]. f_classif is used only for categorical targets and based on the Analysis of Variance (ANOVA) statistical test. To evaluate the effect of this dataset feature reduction (DFR), we tested the performance (accuracy) on the selected datasets of the two classifiers used in this work, namely DT and K-NN, before and after DFR, as shown in Table 4.1. We emphasize that the results in Table 4.1 are also the comparison reference for

the various AEP implementations we will analyze later. Scikit-learn exhaustive grid search cross-validation [195] was used to select the best hyper-parameter values. In all experiments, the accuracy is measured with the same 20% test set.

Table 4.1 Classifiers performance (accuracy) before and after DFR.

Dataset	DT		K-NN	
	Original	After DFR	Original	After DFR
Diabetes	85.0%	81.2%	83.1%	77.9%
SECOM	94.3%	92.2%	95.8%	92.2%

The results show a 3% and 5% performance decrease in the Diabetes dataset for DT and K-NN respectively, while a 2% and 3% decrease was observed in SECOM. We thus argue that the applied DFR does not significantly affect the classifiers' performance, and the resulting datasets can be used to evaluate the proposed AEP. Of course, in the AEP experiments, we did not use the actual labels in the training phase, but only as ground truth for comparing the classification results.

4.4.1 Learning Curves

Learning curves (LCs) are a commonly used diagnostic tool in ML for algorithms that learn incrementally from a training dataset, such as in our case described in Subsection 4.3.3. Learning curves are used to represent the predictive generalization performance as a function of the number of training samples. It is a visualization technique that can show how much a model benefits from adding more training examples [196]. As a first step, for each dataset, Figure 4.3 plots the learning curves for the two classifiers trained on the pseudo-labels obtained by k-means clustering (namely, AutoDT and AutoKNN), as a function of the size of the training set, which varies from 10% to 100% (out of the whole 80% training set size). For comparison, we also present the performance of the two classifiers trained on actual labels (DT and K-NN), as well as the performance of the k-means clustering only (Clustering).

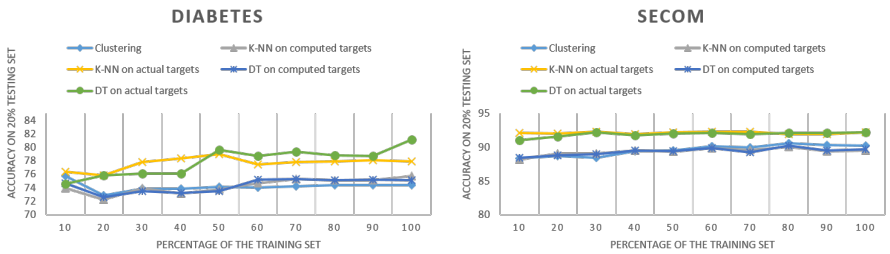

Fig. 4.3 Learning Curves on Diabetes and SECOM datasets.

In Figure 4.3 we can see that for the Diabetes and SECOM datasets, the AEP (AutoDT and AutoKNN) can achieve accuracy levels comparable to the models of the same type (DT and K-NN) trained on actual labels. Performance on the SECOM dataset quickly saturates with a small percentage of training data. The "Clustering" approach almost always had similar accuracy levels as AEP. This is reasonable, as the latter exploits the labels generated by the k-means clustering algorithm to train its classifiers in the field.

AEP generally achieves fairly similar performance to the original supervised approach trained with actual labels (DT and K-NN on actual labels). However, AEP has limitations as the data in high-density areas affects the k-means clustering performance. In the following subsections, we go more in depth with the analysis of the AEP, considering the three research questions reported in the introduction, analyzing the various steps and alternatives of the workflow described in Subsection 4.3.3.

4.4.2 Robustness in Sample Selection for Automated Training

The first research question investigates the possibility of making a self-supervised learning system robust in sample selection for automated training in the field. We thus explore the effect of not including in the training memory samples having a low confidence value, which is obtainable from the soft k-means algorithm.

4.4.2.1 AEP: AutoDT on the NUCLEO-H743ZI2

We evaluated the AEP with and without the Confidence feature of the soft k-means algorithm (Subsection 4.3.2), with a confidence threshold of 0.9. We empirically identified this value as a good trade-off, after several tests. A lower threshold means that more samples are retained, increasing the likelihood that points are in an uncertain (or dense) region, thus reducing the label robustness, while higher values may cause some representative data points to be discarded. To keep into account the memory limitations, we use the RND filtering strategy. For the performance analysis on the target device, we set the memory threshold values (*MEMORY_SIZE*, *INITIAL_THR*, and *UPDATE_THR*) to 200, 50, and 100 data samples, respectively. The best hyper-parameters for the AutoDT classifier are reported in Table 4.2, after multiple tests on the AEP. The learning curves for AutoDT are shown in Figure 4.4.

Table 4.2 Decision Tree training hyper-parameters for the AutoDT.

Dataset	Max Depth	Min Samples
Diabetes	3	10
SECOM	2	10

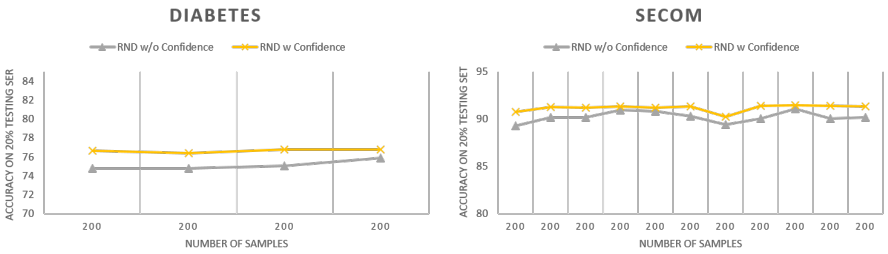

Fig. 4.4 Learning Curves of AutoDT on Diabetes and SECOM datasets.

As described in the workflow in Subsection 4.3.3, the AEP starts learning incrementally until it reaches the *MEMORY_SIZE* threshold (200). At this point, the filtering strategy starts executing and the AEP is then only trained on *MEMORY_SIZE* samples. Overall, the learning curves in Figure 4.4 show that the Confidence algorithm leads to a certain accuracy increase in both datasets (+1.9% for Diabetes, +1.0% for SECOM).

For better result presentation, the bar graphs in Figure 4.5 show the average accuracy and standard deviation of AutoDT on both the datasets in the *MEMORY_SIZE* occupation level case.

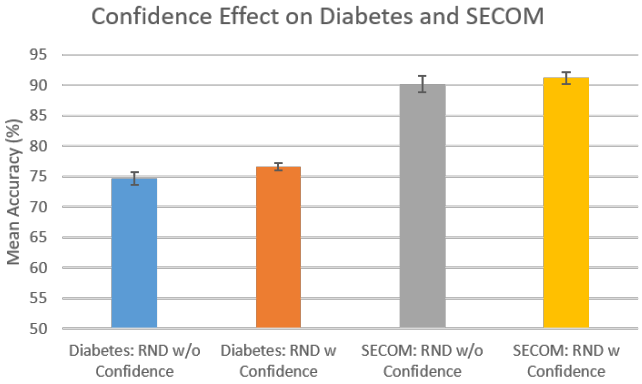

Fig. 4.5 Mean accuracy with standard deviation bars of AutoDT on Diabetes and SECOM datasets.

When we compare the above results with the DT classifier from Table 4.1 (after DFR), we can observe the following:

1. Diabetes: the AutoDT model with Confidence enabled achieves a 76.6% average accuracy, compared to 81.2% of the baseline DT model.

2. SECOM: the AutoDT model with Confidence achieves a 91.2% average accuracy compared to 92.2% of the baseline DT model.

4.4.2.2 AEP: AutoKNN on the NUCLEO-H743ZI2

For the memory part, the procedure for the AutoKNN case is the same as for AutoDT, which is described in the previous sub-section. The best hyper-parameter values for the AutoKNN classifier are reported in Table 4.3, after multiple tests on the AEP. The learning curves for AutoKNN are reported in Figure 4.6.

Table 4.3 Number of Neighbors for the AutoKNN.

Dataset	N Neighbors
Diabetes	5
SECOM	5

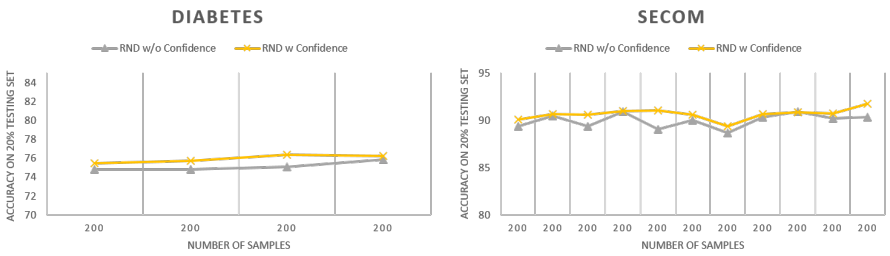

Fig. 4.6 Learning Curves of AutoKNN on Diabetes and SECOM datasets.

Overall, the AutoKNN learning curves for the Diabetes and SECOM datasets show a limited improvement introduced by enabling the soft k-means Confidence algorithm (+0.8% for Diabetes, +0.7% for SECOM), but it should be noted that embedded performance is anyways close to the reference desktop implementation (Table 4.1). To better illustrate the results, Figure 4.7 shows the mean accuracy and standard deviation of Au-toKNN on the test datasets in the *MEMORY_SIZE* occupation level case.

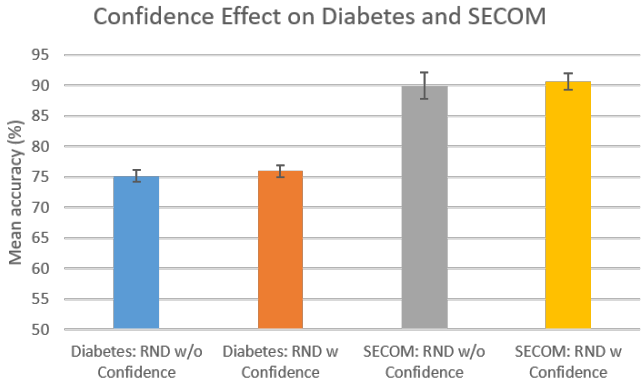

Fig. 4.7 Mean accuracy with standard deviation bars of AutoKNN on Diabetes and SECOM datasets.

Comparing the bar charts result with the baseline K-NN classifier from Table 4.1 (after DFR), we can observe the following:

1. Diabetes: AutoKNN with Confidence achieves a 76.07% average accuracy compared to 78.0% of the baseline K-NN model.

2. SECOM: AutoKNN with Confidence achieves a 90.7% average accuracy compared to 92.2% of the baseline K-NN model.

4.4.3 Effect of Memory Management

In this section, we address the second research question, by comparing the performance of the three implemented memory filtering strategies, namely *RAND*, *FIFO* and *CONF*. Given the iid nature of our test dataset samples, we expect that the results of the RND and FIFO strategies should be very similar. An improvement is expected with the *CONF* strategy since it should incrementally improve the dataset by retaining the samples with higher confidence values.

4.4.3.1 AEP: AutoDT on the NUCLEO-H743ZI2

The procedure for this analysis step is the same as for the first research question. As an additional comparison term, we compare the memory filtering strategies with a "one-shot" implementation of the AEP, where clustering and training are performed only the first time the threshold *MEMORY_SIZE* samples (200 samples in this case) is reached, without any further update of the classifier. Given its added value, soft k-means Confidence thresholding is enabled in all the reported test cases, except for the one-shot scenario (as samples are never removed in this case). Results are shown in the bar graphs in Figure 4.8.

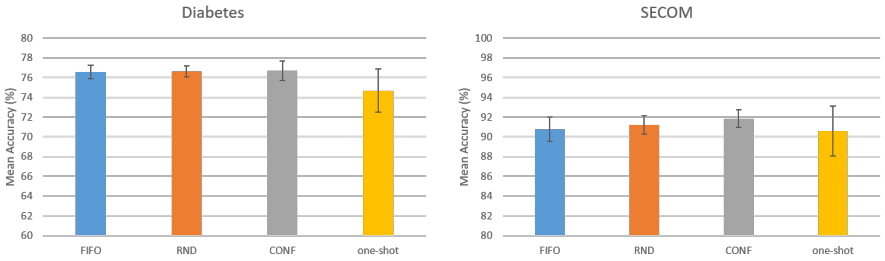

Fig. 4.8 Mean accuracy with standard deviation bars of AutoDT on Diabetes and SECOM datasets with different memory approaches.

Observing the bar chart results, we can see that the three filtering strategies have similar accuracy for the diabetes dataset, with the CONF approach being slightly superior. However,

the “one-shot” implementation had the lowest accuracy because the Confidence algorithm is disabled and only the first *MEMORY_SIZE* samples are used. Also, for the SECOM dataset, the CONF memory filtering had the highest accuracy level, and the “one-shot” approach had the lowest one.

Comparing the above results with the baseline DT classifiers from Table 4.1 (after DFR), we can observe the following:

1. Diabetes: the AutoDT model with CONF memory filtering and Confidence enabled achieves a 76.7% average accuracy, compared to 81.2% of the baseline DT model.

2. SECOM: the AutoDT model with CONF memory filtering and Confidence enabled with 91.8% average accuracy, compared to 92.2% of the baseline decision tree model.

4.4.3.2 AEP: AutoKNN on the NUCLEO-H743ZI2

The same procedures as for AutoDT were applied for the performance assessment of the AutoKNN module as well. The resulting bar charts are shown in Figure 4.9:

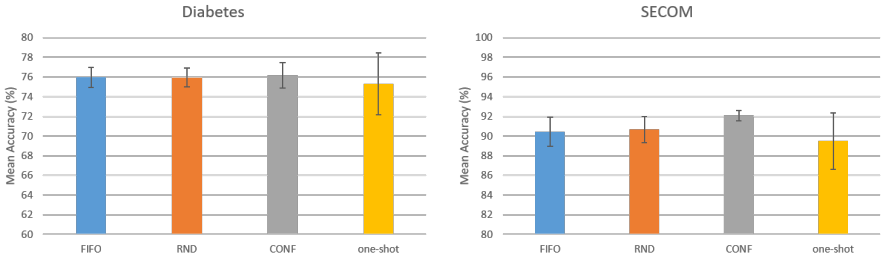

Fig. 4.9 Mean accuracy with standard deviation bars of AutoKNN on Diabetes and SECOM datasets with different memory approaches.

For the Diabetes dataset, the three filtering strategies had approximately equal accuracy levels with a slight superiority of the *CONF* strategy. Again, the “one-shot” implementation showed the lowest accuracy due to Confidence disabling. The effect of the *CONF* filtering strategy is more pronounced in the SECOM dataset, with about 2% higher accuracy.

Comparing the bar chart result with the baseline K-NN classifiers from Table 4.1 (after DFR), we can observe the following:

1. Diabetes: the AutoKNN model with CONF memory filtering and Confidence enabled reaches a 76.2% average accuracy, compared to 77.9% of the baseline K-NN model.

2. SECOM: the AutoKNN model with CONF memory filtering and Confidence enabled reaches a 92.1% average accuracy, compared to 92.2% of the baseline K-NN model.

Similar to AutoDT results, the *CONF* strategy had the highest accuracy levels for both datasets.

4.4.4 Timing Performance Analysis

In the following subsections, we address the third research question by analyzing the timing performance of each AEP component (e.g., clustering, filtering, training, and classification) on the NUCLEO-H743ZI2 MCU running at 480 MHz (max).

4.4.4.1 k-means Clustering Time

Tables 4.4 and 4.5 show the timing performance for the k-means clustering module of the AEP on the datasets with and without the Confidence algorithm enabled, respectively. This latency is important because periodically affects the inference process, which is stopped (assuming a single-thread system) for the classifier to get the updated pseudo-labels.

Table 4.4 k-means Clustering time performance on the edge device with Confidence algorithm enabled.

Dataset	**Clustering Time (ms)**											
	Number of Samples											
	50				**150**				**200**			
	Min	**Mean**	**Max**	**Stdev**	**Min**	**Mean**	**Max**	**Stdev**	**Min**	**Mean**	**Max**	**Stdev**
Diabetes	52	52.6	53	0.6	156	156.6	157	0.6	261	300.5	315	26.1
SECOM	52	53.0	54	1	158	159.3	161	1.5	262	314.8	327	16.7

Table 4.5 k-means Clustering time performance on the edge device with Confidence algorithm disabled.

Dataset	**Clustering Time (ms)**											
	Number of Samples											
	50				**150**				**200**			
	Min	**Mean**	**Max**	**Stdev**	**Min**	**Mean**	**Max**	**Stdev**	**Min**	**Mean**	**Max**	**Stdev**
Diabetes	52	52.0	52	0.0	143	148.0	152	4.6	220	288.9	319	43.5
SECOM	50	52.0	54	2.0	157	158.3	160	1.5	262	312.2	324	15.8

The AEP first performs clustering on 50 samples and then, repeatedly, on 200 filtered samples (see Subsection 4.3.3). We thus report minimum, maximum, average, and standard deviation (3 different runs under the same operating conditions) in the cases of: 50 samples (*INITIAL_THR*), 150 samples (*INITIAL_THR* + *UPDATE_THR*), and 200 samples, which represents the steady state condition and the most critical case (also because clustering and training with fewer samples happens only at the beginning of the operation). The minimum

time for clustering given in Tables 4.4 and 4.5 is achieved in the case of 50 samples and the maximum time is achieved in the case of 200 samples. The comparison between Table 4.4 and Table 4.5 shows that enabling Confidence results in slower clustering, which is due to the computational overhead (running the Confidence algorithm on the clustering results). Moreover, slight variations in the clustering time occurred because of the stopping criterion of the k-means clustering adopted in the AEP, which stops the algorithm once the centroids of the newly formed clusters stop changing. This criterion depends on the choice of the initial centroids, which affects the convergence speed. In all AEP tests, the maximum number of iterations of the k-means algorithm is set to 50 (the stopping criterion is met before this number in all tests).

4.4.4.2 Filtering Time

This latency is important because filtering is performed at regular intervals and affects the inference process. Table 4.6 shows the timing performance for the three filtering strategies used in the AEP: FIFO, RND, and CONF.

Table 4.6 Filtering time performance on the edge device with different strategies.

Datasets	Filtering Time (ms)				
	FIFO w/o Confidence	FIFO w Confidence	RND w/o Confidence	RND w Confidence	CONF
Diabetes	1	<1	1	<1	9
SECOM	1	<1	1	<1	10

The filtering is performed repeatedly to keep only *MEMORY_SIZE* samples (200 samples in this case). From the above results, for both datasets, we can see that *FIFO* and *RND* have the same timing performance with a slightly faster filtering time when Confidence is enabled. This is because the Confidence algorithm removes some uncertain samples after clustering, resulting in a smaller number of samples that need to be filtered (e.g., instead of 200 out of 300 samples, 200 out of 280 samples would be filtered). The *CONF* strategy takes more time because of the execution of the sorting algorithm needed to identify the *MEMORY_SIZE* samples having higher Confidence.

4.4.4.3 Decision Tree Training Time

Another significant AEP system latency factor is given by the DT training time, which gets summed with the clustering time and filtering time discussed in the previous sub-sections. Results in Table 4.7 illustrate the minimum, maximum, and average training latency on the datasets using the hyper-parameter configuration reported in Table 4.2.

Table 4.7 Decision Tree Training time on the edge device.

Dataset	DT Training Time (ms)											
	Number of Samples											
	50				150				200			
	Min	Mean	Max	Stdev	Min	Mean	Max	Stdev	Min	Mean	Max	Stdev
Diabetes	5	6.0	7	1.0	37	43.3	50	6.5	104	127.5	139	11.2
SECOM	10	10.6	11	0.6	86	90.6	94	4.2	71	133.3	198	36.5

While starting from an empty dataset that is continuously updated, the training time varies with the number of samples, so three cases were considered: the case with 50 samples (*INITIAL_THR* samples), the case with 150 samples (*INITIAL_THR* + *UP-DATE_THR*), and the case with 200 samples (steady state condition). Confidence is enabled in all cases and the *CONF* filtering method is used for the 200-sample case (the temporal effects of the filtering strategies are analyzed in the previous section). The minimum training time, reported in Table 4.7, is achieved in the 50 sample case and the maximum time is achieved in the 200 samples case. The average AutoDT training time and standard deviation analysis are also shown in Table 4.7 after 3 different runs in the same operational conditions.

4.4.4.4 Inference Time

The time for inference with the DT and the K-NN classifiers in the AEP is shown in Table 4.8. We always consider the 200 samples case, which is the steady-state condition.

Table 4.8 Inference time performance on the edge device for the two supported classifiers.

Dataset	Inference Time (ms)	
	DT	KNN
Diabetes	<1	1
SECOM	<1	1

The inference time with the DT classifier is relatively low (less than 1 ms) for both datasets. This is because the DT algorithm produces a simple model despite the size of the training set. For the K-NN classifier, the inference time is slightly higher (1 ms for both datasets) because the K-NN inference algorithm requires the exploration of the entire training set and thus its size plays an important role in the timing performance.

4.4.5 AEP vs Full Supervised Scenario on The Edge

As a final experiment, we compared the performance of the baseline models from Table 4.1 in a fully supervised scenario (i.e., training on the cloud and then deploying the model) with the performance of AEP (accuracy and latency), and the results are shown in Table 4.9. The

base models are deployed with the Micro-LM module of the ELM framework from Chapter 3.

Table 4.9 Performance evaluation between base models and AEP on the NUCLEO-H743ZI2 board.

Dataset	DT		AutoDT		K-NN		AutoKNN	
	Accuracy	Latency	Accuracy	Latency	Accuracy	Latency	Accuracy	Latency
Diabetes	81.16%	<1 ms	76.67% (-5.53%)	<1 ms	77.90%	4 ms	76.17% (-0.22%)	1 ms
SECOM	92.22%	<1 ms	91.83% (-0.42%)	<1 ms	92.22%	7 ms	92.07% (-0.16%)	1 ms

From the results, we can see that the AEP accuracy drop wrt the baseline models is up to 5.5% and 0.4% for Diabetes and SECOM respectively. The inference time is relatively short in all cases, with partially better performance on the AEP (since fewer samples are processed, *MEMORY_SIZE* = 200 samples).

However, it is important to emphasize that the inference time performance reported in Table 8 does not take into account the specific case of the samples coming every *UPDATE_THR* samples (100 new samples in this case study) when a new clustering and training run is performed by the AEP. For such samples, the latency is thus given by Equations 4.2 and 4.3, for AutoDT and AutoKNN respectively. Table 10 shows the computed inference time for this sample for both datasets:

$$AutoDT_{Latency@UT} = IT_{DT} + AEP_{CT} + AEP_{FT} + AEP_{DT\_TT} \tag{4.2}$$

$$AutoKNN_{Latency@UT} = IT_{KNN} + AEP_{CT} + AEP_{FT} \tag{4.3}$$

where *IT*, *CT*, *FT*, and *TT* are the DT and K-NN inference time from Table 4.8, the clustering time, the filtering time, and the DT training time, respectively.

Table 4.10 Inference time for the first sample after each *UPDATE_THR*.

Dataset	Inference Time (ms)	
	AutoDT	AutoKNN
Diabetes	437.2	310.7
SECOM	458.2	325.9

The "one-shot" AEP option avoids the periodical update, at the cost of an accuracy drop with respect to the full AEP system reported in Table 4.9. The comparison is provided in table 4.11. For the diabetes dataset, a 2% decrease in accuracy is observed when using AutoDT,

while a marginal loss in accuracy of less than 1% is observed when using AutoKNN. For the SECOM dataset, a degradation in accuracy by 1.25% and 2.59% is observed for AutoDT and AutoKNN, respectively.

Table 4.11 "One-shot" AEP performance accuracy (and accuracy drop with respect to the full AEP system) on the NUCLEO-H743ZI2 board.

Dataset	**Accuracy**		**Accuracy drop**	
	AutoDT	**AutoKNN**	**AutoDT**	**AutoKNN**
Diabetes	74.67%	75.32%	2.00%	0.85%
SECOM	90.58%	89.48%	1.25%	2.59%

4.4.6 Summative Considerations

As a summary, we can state that the AEP (AutoDT and AutoKNN) provides accuracy levels comparable to the models of the same type (DT and K-NN) trained on actual labels and with the full dataset. This stresses the feasibility and effectiveness of the proposed self-learning pipeline. Our experience has also shown that not including in the training memory samples having a soft k-means Confidence value under a certain threshold is a good option to enhance overall system robustness, especially in the case of overlapping clusters, and leads to performance improvement in all cases. Robustness – which is particularly critical in remote, autonomous systems - can be further increased by implementing a sample filtering strategy (e.g., *CONF*, which outperforms *FIFO* and *RND*), which allows to incrementally improve the dataset (while not increasing its size) by retaining the samples with higher confidence values. In our tests, the achieved accuracy improvements are relatively limited (under 3%) but allow getting further close to the reference values of the supervised learning implementation. We also argue that the test datasets are already optimized, while actual field operation may deal with noisy measurements, in which case the robustness enhancing solutions presented in this chapter should be more relevant.

Considering the timing performance, the inference time is relatively low in all cases, but the overhead incurred by periodic clustering and training may be unacceptable in some real-time application cases, because of the interruption of the usual (inference) flow. To avoid this temporal overhead, a simple "one-shot" execution implementation is available from the AEP, at the cost of a slight decrease in accuracy. A better solution could be achieved by parallelizing the inference and clustering/training tasks in a real-time embedded operating system, provided that the inference task does not already consume the whole CPU time if the system is not multi-core.

4.5 Conclusion

With the rapid development of IoT technologies, end-devices deployed in the field are getting ever more powerful and able to process data before delivery to the cloud, thus reducing transmission rates, bandwidth and energy consumption. Processing data through ML models typically requires supervised training, that needs labeled datasets. Manual labeling of datasets is a bottleneck, that is addressed in literature through the development of semi-supervised learning and self-supervised learning techniques.

The advancement over state of the art aimed by this work is twofold. First, we have proposed the Autonomous Edge Pipeline (AEP), a system that combines unsupervised clustering, supervised learning and inferencing to autonomously classify binary samples on the edge. The system is implemented in pure C, which makes it available for any microcontroller and resource constrained device. The system also features training memory management strategies, to deal with the limited footprint of embedded devices and keep the energy demand low. To support the research activity in the field, the AEP is made available open source in the context of the ELM platform.

Second, we have explored the performance of the system to get a quantitative characterization of self-learning in an embedded environment. An experimental analysis of two publicly available IoT datasets shows that the AEP achieves similar performance levels as the corresponding models trained on actual labels and with the full dataset. Our experience stresses the importance of filtering some samples, to select the most effective samples for building the training set. This can be achieved by setting a proper threshold for the Confidence value provided by the soft k-means algorithm. Classification performance can be slightly increased over time by keeping in the training memory only the samples with higher Confidence values and periodically executing new training sessions. In terms of timing performance, the inference time is very short, but the overhead incurred by periodic clustering and training penalizes the timing performance. This extra time overhead can be avoided by a "one-shot" implementation, where clustering and training are performed only once when the threshold *MEMORY_SIZE* samples is reached, resulting in a slight drop in accuracy.

The AEP system is completely application independent. Potential benefits are expected to be significant in several application domains, as it would allow the introduction of checks that are currently not foreseeable because of the need for manual sample labeling. For example, it could be possible to extend quality control in industrial plants by applying the AEP to detect defects and anomalies in several stages of a manufacturing chain. This application scenario is also close to the one represented by the SECOM dataset studied in this chapter. Of course, the loss in accuracy wrt a supervised-learning-based system should be carefully considered,

from a complete system/product design perspective, particularly for health/safety-related applications, for which unsupervised learning looks critical. More generally, human factor aspects, such as privacy, invasiveness, and dignity, must be always taken into account when designing real-world applications.

Chapter 5

Memory-Efficient CMSIS-NN with Replacement Strategy

5.1 Introduction

Convolutional Neural Networks (CNN) are well established machine learning (ML) models, frequently used for image recognition and classification. Recent advances in edge computing have made it possible to move the inference task from the cloud to embedded devices on the edge. Edge computing offers advantages in terms of latency, bandwidth, energy, privacy [25]. However, edge devices have limited computational power and on-chip memory. Several solutions have been devised to overcome the memory size limitation. For example, model compression techniques such as parameter pruning and quantization [197], binarization [198], low-rank factorization [199], and knowledge distillation [200] are widely used to reduce the model size. Further alternatives involve changing the execution order of the network's operations by requiring the inference software to follow a specific order [201].

A common class of edge devices is represented by microcontroller units (MCUs) [202], that are widely available, cheap, and energy efficient [201]. Unlike multicore CPUs and GPUs, which can perform multiple computations simultaneously [203], MCUs typically have single-core processors and sequential execution. Such an execution mode requires less memory than parallel execution because only one block of the network can be executed per operation. In addition, operations in a CNN convolution layer are local, as they only depend on the input of that layer. Once the input of the layer has been processed the memory can be replaced. Thus, the memory is reusable among layers.

In this chapter, we proposed a memory replacement strategy that reuses the memory assigned to convolutional layers to reduce the overall memory usage of the CNN. The

proposed strategy relies on the use of a single input/output buffer which is shared among all the layers in the inference phase of a CNN, exploiting the sequential execution paradigm of single-core processors.

5.2 Background

In this section, we briefly introduce the convolutional neural network class of deep learning, the software kernels utilized, and the deep learning framework used for training.

5.2.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a deep learning algorithm that is widely used in pattern recognition and computer vision tasks. It consists of several building blocks such as convolutional layers, pooling layers and fully connected layers. A CNN is designed to learn spatial hierarchies of features through a backpropagation algorithm [204]. This model is particularly suited to image classification as it extracts useful features from the image by observing patterns in the dataset. Other deep learning models are much less efficient in this task, requiring a much higher number of trainable parameters.

5.2.2 Caffe

Convolutional Architecture for Fast Feature Embedding or CAFFE [205] is a deep learning framework developed at the University of California, Berkeley. It is an open-source framework written in C++ and released under the BSD 2-Clause license. Caffe supports different types of deep learning architectures (CNN, RCNN, LSTM, and fully connected neural networks) focused on image segmentation and image classification. It also supports both GPU and CPU-based acceleration computational kernel libraries (NVIDIA, cuDNN, Intel MKL). In our implementation, we used the same model as CMSIS-NN for comparison purposes. The model is based on a built-in example provided by the Caffe framework.

5.3 Optimization

Our goal is to optimize memory efficiency in convolutional layer computation in CNNs through a replacement strategy. State of the art frameworks, such as CMSIS-NN, already do in-place (or in-situ) computation in the pooling layer, but they allocate memory space for each output feature map for each layer. The resulting memory footprint is the sum of the

output sizes of the feature maps in all the convolutional layers. The deeper a network, the higher the sum. Our idea is to allocate a single buffer (we call it total buffer) sized as the maximum needed capacity.

The idea stems from the fact that convolutions are local operations; the output feature maps of a convolutional layer depend only on the input features of that layer. Therefore, after a layer has computed its operations, its memory can be reused to store output feature maps of subsequent layers. This requires that there is no out of order instruction execution, which guarantees that only the feature maps in the currently active layer are used in each layer operation. In-order execution is the case of single-core processors, that cover the vast majority of microcontrollers.

The maximum needed capacity is computed as the maximum, among all the couples of adjacent layers, of the needed buffer size for each couple of adjacent layers (Equation 5.1). Table 5.1 reports the needed buffer size for the two most common types of layers of CNNs.

$$\text{total buffer size} = min_{\,i \in 1, N-1}(\text{needed\_buff\_size}\,(L_i, L_i + 1)) \tag{5.1}$$

A pooling layer can be filled completely in place since each pooling operation produces a single value and destroys at least one value. Convolutions, on the other hand, process the same input area one time for each filter, thus results cannot replace the values in the current layer, but should be put in a new buffer. Consequently, the required size for a couple of layers of which the second one is a convolutional one is given by the sum of their respective sizes.

Table 5.1 Needed buffer size for a couple of adjacent layers.

	Next Layer	
	Convolutional	**Pooling**
	Sum of the sizes of current and next layer's feature maps	Sum of current layer's feature maps

Figure 5.1 shows a typical application case of our optimization. 3-D tensors are used to represent data and weights, as is common in machine learning frameworks. Please note that targeting an embedded environment, the CMSIS-NN implementation (atop of which we built), differs from this representation. Particularly, the weight tensors are transposed (reordered) and flattened into a 1-D array with num. of input channels x filter height x filter width x num. of output channels elements. Similarly, each layer's output is stored in a 1-D array with {num. of output channels x feature map height x feature map width} elements.

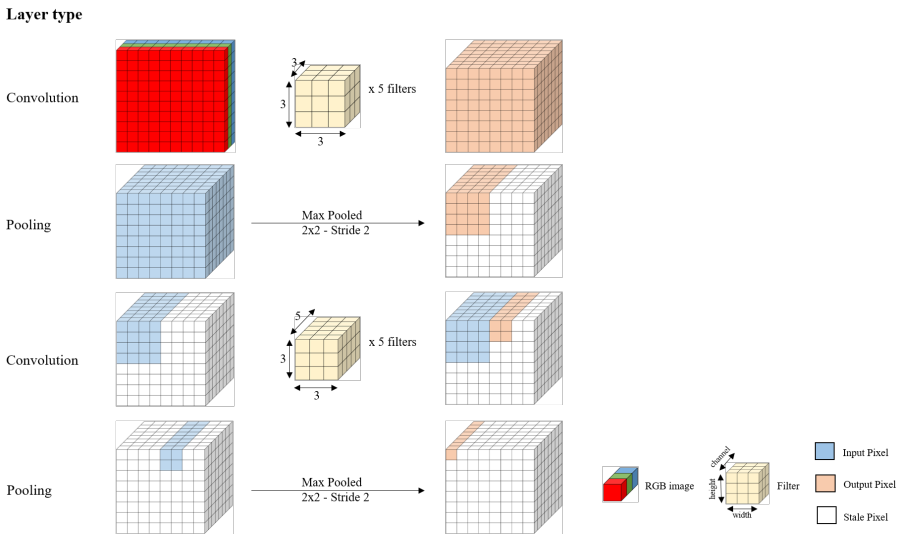

Fig. 5.1 Example of pixels arrangement in memory with the proposed replacement strategy.

The example uses two convolutional layers interleaved with max-pooling on a $10x10$ RGB image. Here we focus on 3D convolutions with odd square filters (3×3, 5×5, etc.), same padding, and stride of 1, which covers the most common cases for CNNs in memory-limited applications [133, 100, 134].

The first convolutional layer consists of 5 filters of shape $3 \times 3 \times 3$ and stride of 1 (in the CMSIS-NN implementation, this input is a 1-D array of shape $3 \times 3 \times 3 \times 5$). The output size is $8 \times 8 \times 5$, which corresponds to the feature map with width = (input width $-$ (filter width $-$ 1)), height $=$ (input height $-$ (filter height $-$ 1)), and channels $=$ (number of input filters). From Equation 5.1 and Table 5.1 it appears that the total buffer size is equal to the one of the first convolutional layer: $8 \times 8 \times 5$, whose feature maps thus completely occupy the total buffer. The first 2×2 max-pooling layer takes as input these first output feature maps and results in a $4 \times 4 \times 5$ second output feature maps. The pooling operation is destructive on the input, thus no additional memory is needed to store the output. Therefore, the second output feature maps are saved in place of the input, resulting in many pixels being freed, which we call stale pixels ($8 \times 8 \times 5 - 4 \times 4 \times 5 = 240$ pixels out of 320 allocated for the total buffer are stale). The third layer consists of 5 filters of shape $3 \times 3 \times 5$ and stride of 1. We should emphasize that we used the same number of filters for a better graphical representation of our methodology, while using a different number is also possible, as in our case study in Section 5.4. The input pixels in this layer need to be preserved during the convolution operation, therefore the output pixels of size $2 \times 2 \times 5$ (third output feature maps) should be stored next to (not in place of, as it happened for the pooling operation) the input pixels, occupying part of the stale pixels. In the last layer, the third output feature maps are the only input needed

for this operation. The output size of this operation is $1 \times 1 \times 5$ (fourth output feature maps) and is stored from the starting position given the destructive nature of the pooling operation. We can observe that, for each layer, there is always enough space to store its input and output in the total buffer, thus reducing the overall memory footprint.

Two main changes were needed to upgrade the CMSIS-NN implementation with our memory replacement feature. First, in the code generated by CMSIS-NN, instead of allocating the sum of the output sizes of the convolutional layers as scratch buffers before starting the inference, just one buffer is allocated (namely, the *total buffer*), sized as the maximum of the output feature maps, is allocated. Second, a pointer is created to refer to the first free slot in this buffer (namely, the *p_buffer*). As the inference algorithm progresses, the pointer is updated to place the output of the current convolutional layer in its correct position. The implementation can be found here: Efficient-CMSIS-NN.

5.4 Case Study

We tested our approach by implementing our memory optimization strategy atop CMSIS-NN kernels on a CNN trained on the CIFAR-10 dataset [206]. The latter consists of 60,000 32×32 RGB images divided into 10 classes. For the assessment, we adopted the network architecture that was used to test CMSIS-NN [100], which is based on a built-in example in Caffe and whose topology is shown in Table 5.2. This architecture is limited in size and provided good performance on edge devices. The model, pre-trained by Caffe, is quantized to 8-bit integers (one byte/parameter) with 79.9% accuracy (like the CMSIS-NN model), and then translated into source and header files for deployment on the microcontroller.

Table 5.2 Parameters of layers.

Operation Layer	Filter Size	Stride	Padding	Output Shape
Convolution	3x5x5x32	1	Same	32x32x32
Pooling ReLU	-	2	-	32x16x16
Convolution ReLU	32x5x5x32	1	Same	32x16x16
Pooling	-	2	-	32x8x8
Convolution ReLU	32x5x5x64	1	Same	64x8x8
Pooling	-	2	-	64x4x4
Fully Connected	64x4x4x10	-	-	10

As explained in the previous Section, we first need to allocate the *total_buffer*. Since the activations in our example are quantized to int8, each output value corresponds to one byte. Thus, the size of the buffer is $32 \times 32 \times 32$ or 32,768 bytes, given by the largest output

feature maps, resulting from the first convolutional layer. Second, the p_buffer pointer is initialized, to place the output of the convolutional layer in the correct position after each pooling layer.

Figure 5.2 shows the memory replacement strategy in our case study. We are using a 1-D array buffer to store all parameters during computation (coherently with the modality in which CMSIS-NN saves the output, as said in the previous section):

- Convolution 1: the output of this layer is equal to $32 \times 32 \times 32 = 32,768$ bytes (largest output feature maps) and occupies all the memory allocated to the *total_buffer*.

  Free space in total_buffer: 0 bytes

- Pooling 1: the output of this layer is equal to $32 \times 16 \times 16 = 8,192$ bytes. The nature of the pooling operation allows the output to be stored in place of the input starting from address 0 of *total_buffer*.

  Next p_buffer offset: 8,192

  Free space in total_buffer: $32,768 - 8,192 = 24,576$ *bytes*

- Convolution 2: the output of this layer is equal to $32 \times 16 \times 16 = 8,192$ bytes. The blue sector represents the previously stored 8,192 bytes resulting from the pooling operation and they are only needed in the current layer during the convolution operation. Thus, the output activations of this convolutional layer are stored from

  p_buffer offset 8,192

  Free space in total_buffer: $32,768 - 8,192 - 8,192 = 16,384$ *bytes*

- Pooling 2: the output of this layer is equal to $32 \times 8 \times 8 = 2,048$ bytes. As indicated in the previous step, the blue sector is no longer needed in this layer, so the output of the pooling operation can be stored starting from address 0 of *total_buffer*.

  Next p_buffer offset: 2,048

  Free space in total_buffer: $32,768 - 2,048 = 30,720$ *bytes*

- Convolution 3: the output of this layer is equal to $64 \times 8 \times 8 = 4,096$ bytes. The blue sector represents the previously stored 2,048 bytes resulting from the pooling operation and they are only needed in the current layer during the convolution operation. Thus, the output activations of this convolutional layer are stored from *p_buffer* offset 2,048.

  Free space in total_buffer: $32,768 - 2,048 - 4,096 = 26,624$ *bytes*

- Pooling 3: the output of this layer is equal to $64 \times 4 \times 4 = 1,024$ bytes. As indicated in the previous step, the blue sector is no longer needed in this layer, so the output of the pooling operation can be stored starting from address 0 of p_buffer.

  Next p_buffer offset (if additional layers exist): 1,024

  free space in total_buffer: $32,768 - 2,048 = 30,720$ *bytes*

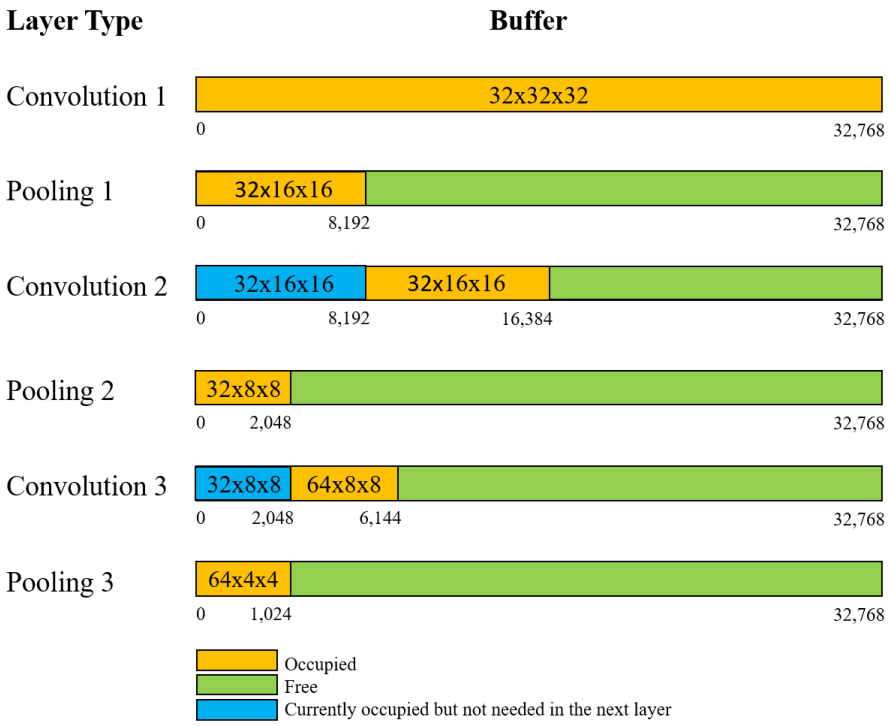

Fig. 5.2 Buffer utilization to save output activations throughout the layers.

As our work focuses only on reducing the required memory for the activations, the model parameters (weights and biases) in both cases (plain and enhanced CMSIS-NN) are the same (their size is equal to the sum of the input shape column in Table 5.3, i.e., 89,440 bytes). On the other hand, the memory needed for the activation (feature maps) in the original CMSIS-NN case is equal to the sum of the output shape column in Table 5.2, without adding the rows resulting from the pooling layers as pooling operations are performed in-place (45,056 bytes). Overall results (Table 5.3) show that our replacement strategy reduces the feature maps' memory footprint by 20%. Considering the total NN model occupancy (i.e., parameters + activations) is reduced by slightly more than 6%.

Table 5.3 Feature maps' memory utilization comparison.

	CMSIS-NN	Proposed Memory Replacement Strategy (bytes)	Reduction
Activations	45,056	32,768	27.27%

Considering the timing performance, we measured the inference time using a NUCLEO-H743ZI2 board with an Arm Cortex-M7 core running at 216 MHz (max 480 Mhz) [167], which is the processing speed used in the testing experiment on the CMSIS-NN kernels [100]. The results show that the runtime is not affected by this optimization and is stable at 10.1 FPS. This is reasonable, as our solution relies on the efficient CMSIS-NN kernels, and the memory required for the activations is allocated before the execution starts, in both cases. Thus, our proposed memory replacement strategy does not affect execution performance.

5.4.1 Experiment With Deeper Networks

In this section, we are interested in exploring the performance of our proposed optimization in the case of deeper architectures, with a higher number of filters. Deeper networks are capable of learning more abstract and powerful patterns of the input and are increasingly being employed.

We thus tested two other CNN architectures, such as the following ones:

- The first architecture (Model 2) has four convolutional layers each one followed by a 2x2 max-pooling layer and a fully connected layer:
  1. 32 3×3 kernels, stride $= 1$, padding $=$ same
  2. 64 3×3 kernels, stride $= 1$, padding $=$ same
  3. 128 3×3 kernels, stride $= 1$, padding $=$ same
  4. 256 3×3 kernels, stride $= 1$, padding $=$ same
- Another deeper architecture (Model 3) has five convolutional layers each one followed by a max-pooling layer, and a fully connected layer:
  1. 32 1×1 kernels, stride $= 1$, padding $=$ same
  2. 64 1×1 kernels, stride $= 1$, padding $=$ same
  3. 128 1×1 kernels, stride $= 1$, padding $=$ same
  4. 256 1×1 kernels, stride $= 1$, padding $=$ same
  5. 512 1×1 kernels, stride $= 1$, padding $=$ same

The results using our memory optimization method are shown in the bar chart of Figure 5.3, which also includes the first model in our case study (Model 1).

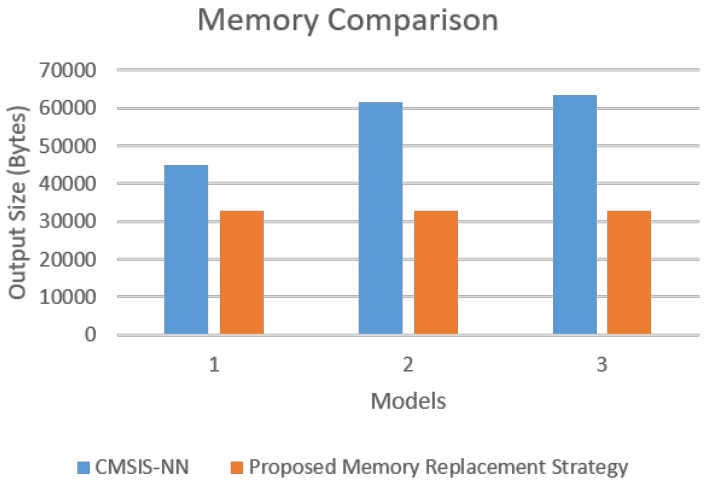

Fig. 5.3 Memory comparison bar chart.

Overall, the results show that using our approach significantly reduces the amount of memory needed to store feature maps during computation. Moreover, we can observe that as the architecture becomes deeper, the reduction percentage increases significantly: 46% and 48% for Model 2 and Model 3, respectively, compared to the first model (27%).

5.5 Conclusion

The ever-evolving field of embedded deep learning presents several challenges and opportunities. Using microcontrollers to run machine learning applications has proven to be a viable solution, but is limited by memory constraints. To alleviate this issue, we have proposed a memory replacement strategy that minimizes memory footprint during inference, while maintaining the same performance offered by the state of the art CMSIS-NN kernels. Our experiments show that we were able to achieve a significant reduction in the required memory for feature maps compared to the CMSIS-NN implementation (27%, corresponding to a 9% reduction of the total NN model size), and the reduction is even higher for deeper networks. This is achieved at no expense in terms of accuracy or timing performance.

Chapter 6

A Tiny CNN for Embedded Electronic Skin Systems

6.1 Introduction

Tiny Machine Learning is paving the way to the deployment of novel applications using battery-powered devices without the energy intensive processing support from the cloud [19]. To this extent, TinyML has been adopted in several application domains such as healthcare, smart cities, industrial control, etc. [207]. However, new application domains could join the rapid adoption of TinyML, specifically those with memory and power constraints such as the design of an efficient Electronic Skin (e-skin) system. These systems are widely used in wearable, prosthesis, and robotics applications.

An e-skin system, sketched in Figure 6.1, consists of a set of distributed tactile sensors integrated with an embedded electronic system for tactile data decoding [208]. An array of tactile sensors detects and convert a mechanical stimuli into electrical signals. Then, the electronics interface applies signal conditioning and analog to digital conversion. Finally, an embedded processing unit (EPU) decodes the received tactile data using an intelligent mechanism. ML algorithms are the main candidate for the decoding process as they can extract meaningful information such as texture, pattern, etc. However, the computational complexity of ML algorithms, accompanied by the design requirements of an e-skin, poses a challenge for EPUs to support real-time operation with very low energy consumption. In this context, we propose a Tiny Convolution Neural Network (TinyCNN) model to be employed inside an EPU based on a Cortex-M MCU devoted to tactile data processing. The main contributions of this chapter are thus:

Fig. 6.1 Electronic Skin: Blocks and Functionality

- A Tiny Convolution Neural Network (TinyCNN) is proposed for processing tactile data. This model provides up to 10% higher classification accuracy compared to existing solutions with a reduced number of parameters and floating point operations (FLOPs).
- A platform-independent C implementation for the proposed TinyCNN is provided for embedded processing. The implementation supports layer fusion and buffer reuse to increase memory and latency efficiency.
- The proposed CNN architecture is deployed on Cortex-M MCU and verified for touch modality classification. The obtained results lead to a real-time operation [209] with 0.5 ms and relatively low energy consumption of 65.36 μJ.

6.2 Machine Learning for Tactile Data Processing

6.2.1 Related Work

A variety of ML algorithms have been adopted in literature for tactile data processing in applications such as: surface texture, slip detection, object recognition, and touch modality classification. A tactile sensory array has been employed on a humanoid for object structure identification (e.g. soft, hard, etc.) [210]. The identification has been performed using a k-Nearest Neighbor (k-NN) classifier. Support Vector Machine (SVM) has been used to allow another humanoid to categorize ten objects (glass, sponge, etc.) using their surface texture in [211]. Another SVM classifier has been utilized in [212] that allows a robot to move objects from one place to another with a stable slip and grasp detection mechanism within 30 ms.

For the touch modality classification task presented in [213], different types of models have been employed, such as SVM based on tensorial kernel [213], k-NN [214], and a deep CNN [215, 216]. Authors in [217] have proposed an SVM accelerator on Virtex-7 Field Programmable Gate Array (FPGA) that can classify an input touch modality of size $4 \times 4 \times 20$ within 250 ms while consuming 285 mJ. A k-NN accelerator implemented on the Zynqberry platform capable of classifying a touch input of size 4×4 within 26 μs, recording a power consumption of 236 mW has been presented in [218]. For an input touch modality of

size $4 \times 4 \times 100$, a 1D-CNN deployed on Arduino Nano BLE 33 managed to offer real-time classification in 26 ms and 128 ms for float and int8 implementations respectively [219].

6.2.2 Tiny CNN for Touch Modality Classification

6.2.2.1 Dataset

Particularly, the system has to classify three modalities of touch (actions) by a user of the outer surface of a robot electronic skin's piezoelectric sensor array, namely: sliding the finger, brushing a paintbrush and rolling a washer [213]. The dataset collection involved 70 participants. Each participant performed both a horizontal and vertical modality on a 4×4 piezoelectric sensor for 10 seconds. Samples were obtained at the 3 kHz sampling frequency, thus it can be represented as a tensor $\phi = 4 \times 4 \times 30,000$. Such input size is impractical for the deployment of existing ML algorithms on hardware platforms [220]. Hence, we performed a couple of pre-processing operations on the original dataset including: (1) sub-sampling to obtain a reduced tensor size of $\phi = (4 \times 4 \times 8)$. Such tensor size has been adopted for machine learning training targeting the same classification problem [138, 213, 221]. The process involves the removal of noisy readings and null values by truncating each tactile sample from 10s to 3.5s followed by averaging across the time axis and (2) data augmentation to increase the size of the dataset to 1680 samples by applying a 90 degree rotation for each sample.

6.2.2.2 CNN Design

The structure of the designed CNN is shown in Figure 6.2. The network consists of two convolutional layers, with the second followed by the ReLU activation function, a maxpool layer followed by a ReLU layer, two fully connected layers, with the first followed by a ReLU layer, and the softmax output layer. The number of layers and the size of each layer (i.e. number of filters) has been determined based on a best-model approach that offers an acceptable trade-off between complexity and classification accuracy with respect to existing solutions. For the relatively small input dimension 4×4, the adequate filter size used in convolution layers is 3×3. Moreover, in order not to lose the whole input after a convolution layer, convolution layers with padding are adopted. Similarly, only one max pooling layer is used to limit the reduction of the input information. The output layer has three neurons, corresponding to the three touch modalities mentioned above.

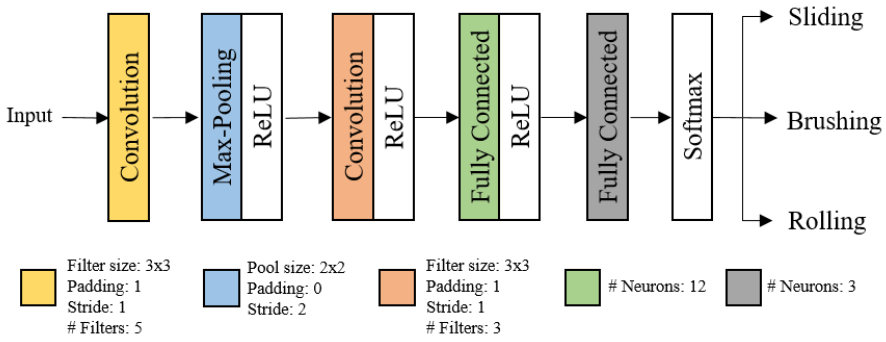

Fig. 6.2 Proposed TinyCNN architecture.

6.2.3 Training Procedure

The CNN is modeled using TensorFlow and trained for 300 epochs using a batch size of 64. Adam [163] has been chosen as an optimizer with a learning rate of 0.01 and a weight decay of 0.001. These hyper-parameters have been obtained through manual fine-tuning. The training has been performed using 5-fold cross validation, and the reported accuracy is the average among the folds; this method is used to be coherent with related works. Training data is normalized. For an input X, the normalized value is obtained as $X_{norm} = (X - \mu)/\sigma$, where μ and σ are the mean and standard deviation on the training set, respectively.

6.3 Embedded CNN Implementation

6.3.1 CNN Layers

To realize the neural network presented in the previous section, we have implemented a set of components that provide the needed functions for convolutional networks. These components are written in platform-independent C language (i.e., they do not use native OS libraries), which makes them seamlessly employable in most software-programmable edge devices, including but not limited to microcontrollers. Figure 6.3 shows how such components are integrated for the different layers' architecture of TinyCNN and how this is optimized for deployment on Cortex-M devices.

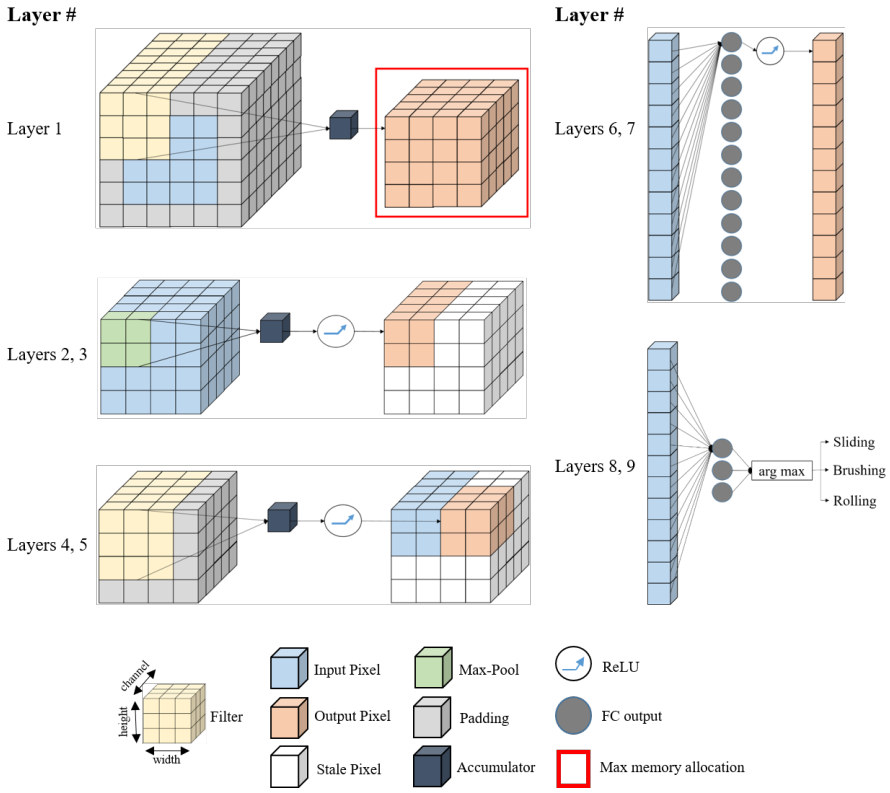

Fig. 6.3 TinyCNN layers and implementation.

A convolutional component performs 2D floating-point convolution using Multiply Accumulate (MAC) instructions. The convolution of each patch of the image is stored in an accumulator before the results are stored in an output buffer with a size equal to the maximum memory buffer requirement, which is pre-computed for the whole network (the block framed in red in Figure 6.3). This memory buffer hosts the values of all the needed neuron activations following a convolutional or a pooling layer, as proposed in Chapter 5. The implementations of the inference phase of neural networks in literature allocate a memory space equal to the sum of the output size of the convolutional layers (e.g., CMSIS-NN [100]). The approach presented in Chapter 5, instead, pre-computes and pre-allocates a maximum value of needed memory, without compromising correctness nor performance, under the hypothesis of sequential execution by single-core processors, which corresponds to the vast majority of microcontrollers. The pre-allocated memory block is shown within a red frame in Figure 6.3. The technique exploits the fact that convolutions are local operations; the feature maps of a convolutional layer depend only on the input features of that layer. Therefore, after a layer has computed its operations, its memory can be reused to store the features of subsequent layers. The convolutional component supports also layer fusion for the activation function. The activation function is applied directly on the accumulator value before it is

written to memory. This (applied, e.g., in layers 4, 5) avoids the extra reads and writes that would occur if we treated the activation layer as a separate one.

As anticipated, the memory optimization is applied to the max-pooling function as well, which extracts the maximum value in the area it convolves. Our implementation supports in-place computation. The pooling operation is destructive to the input, as each pooling operation results in a single value and allows destroying at least one value since at least one value will never be used again. This is the founding observation of the memory optimization introduced in Chapter 5, which allows directly passing from the fully occupied memory block output by layer 1 to a one quarter occupation after layer 2, using only the accumulator as additional temporary memory (i.e., one floating point cell). In the various inference steps, the pre-allocated memory block may feature also some stale (i.e. unused) pixels, as shown in layers 2, 3, 4 and 5, because of the lower space's need for some layers. Layer fusion is applied to the output of the max-pooling function as well. The stale pixels resulting from this layer are used to store the output of layer 5, based on the memory replacement strategy proposed in Chapter 5.

Finally, the implemented fully connected layer performs vector-to-matrix multiplication using MAC instructions in floating-point representation. Also, this layer supports layer fusion for the activation function (layers 6, 7 in Figure 6.3). For the inference phase, the Softmax function used as the activation function for the multi-class classification problems is replaced by a simpler "argmax" as shown in layers 8 and 9. This function operates on a vector and converts every value to zero except the maximum value, which returns 1. This is suitable for the inference phase, which is intended to simply return only one predicted class, not the probabilities of each possible class.

6.3.2 Embedded Device and Performance Metrics

In line with the literature on embedded systems for processing tactile data, we assessed the efficiency of the proposed TinyCNN in deployment on an MCU. In this context, the most relevant metrics are accuracy, area (which, for MCUs, corresponds to the allocated memory space in Flash/RAM), latency, and energy consumption.

6.3.2.1 Embedded Device

The implemented functions are tested on the STM32 H743ZI2 commercial MCU, hosted on a NUCLEO-144 board. The MCU is an Arm Cortex-M7 core running at 480 MHz, with 512 KB SRAM and 2 MB flash memory.

6.3.2.2 Latency

Time latency is defined as the time interval it takes the edge device to output a classification decision after the hosted ML model receives an input. We measured the latency on the microcontroller by using the *HAL_GetTick()* function provided by the STM32 hardware abstraction layer's (HAL), which returns the number of elapsed milliseconds.

6.3.2.3 Energy Consumption

The energy consumption of our implementation is calculated as $E = P \times T$ where P is the power consumption and T is the latency time. The power consumption is calculated as $P = V \times I$ where the average current and voltage values are provided by the STM32CubeIDE.

6.4 Results and Discussion

Table 6.1 presents the main performance aspects of the proposed TinyCNN compared to the solutions presented in the literature to address the same touch modality classification task. The reported accuracy is derived from related works using the average among 5-folds. The proposed model achieves the highest average classification accuracy of 81.5% among all existing models. Compared to Regularized Least Square (RLS) and SVM in [213], the proposed CNN offers up to 10% accuracy improvement, with a substantial decrease in the number of parameters and FLOPs. Compared to Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU-n; n is the number of neurons)[222], a 17.2% and 26% decrease is achieved in the number of parameters and FLOPs respectively. The reported decrease is with respect to the smallest model (i.e., GRU-10). A 7% accuracy increase is noticed compared to LSTM, which offers the highest accuracy among the solutions presented in [222].

Compared to the 1D-CNN model [219], although the accuracy improvement obtained by our proposed model is less than 2% on average, its size is about 65% smaller for the same input dimensions, which is a key advantage for resource constrained devices.

Table 6.1 Comparison between different ML algorithms for touch modality classification.

	RLS [213]	SVM [213]	LSTM [222]	GRU-10 [222]	GRU-12 [222]	1D-CNN [219]	Ours
Number of Parameters	-	67.2K	1113	843	1083	1963	698
kFLOPS	-	545K	23.6	17.82	23.6	530K^*	13.2
Average Accuracy (%)	73.7	71	74.51	73.1	73.43	79.78	81.5

* The reported number of FLOPS includes both pre-processing and inference phases.

The proposed TinyCNN has been deployed on the STM32 H743ZI2 MCU. The TinyCNN requires 2.73 KB for model parameters and 320 bytes for output activations. These requirements permit the allocation of the model to the RAM. This design choice significantly accelerates the inference (compared to Flash allocation), during which an input touch is classified within 0.5 while consuming 65.36 μJ operating under a 3V power source. Although not all ML algorithms listed in Table 6.1 have been deployed on embedded systems, Table 6.2 lists the performance of EPUs that have been designed for the same touch modality classification task as ours. As they target different scenarios, each solution differs in terms of ML algorithm, target hardware, pre-processing technique, and expected input size, the comparison is not straightforward. As a common factor, all such EPUs provide real-time classification in less than 50 ms [209].

Table 6.2 Synopsis of solutions for the touch modality classification task

Reference	ML Algorithm	Input Size	pre-processing Technique	Hardware Target	Time Latency	Energy Consumption
[217]	SVM	8x8x20	Subsampling	Virtex-7	250 ms	285 mJ
		4x4x20	Subsampling	PULP	3.3 s	69.3 mJ
[221]	CNN	4x4x30000	None	Jetson TX2	75 ms	55.5 mJ
[223]	k-NN	4x4	Time truncation	Zynqberry	25.7 us	6 μJ
[138]	H-CNN	4x4x8	Time truncation and Subsampling		0.8 ms	42.4 μJ
Ours	TinyCNN			STM32 H743ZI2	0.5 ms	65.36 μJ

Each one of the listed embedded ML algorithms is tailored to the targeted e-skin application. For instance, in a prosthetic application, a person should be able to tolerate the weight of the complete sensory feedback system, thus the adoption of hybrid-CNN or TinyCNN implemented on Zynqberry and MCU, respectively, is more suited, with the final choice based on the trade-off between latency and energy consumption. In an industrial application (e.g. automobile manufacturing), a system with a large number of sensors should be able to process a huge amount of data at once. Hence, higher processing power is required, and an embedded SVM implemented on a Virtex-7 (FPGA), or a CNN implemented on a Jetson

GPU, are the main candidates. On the one hand, the latter offers a faster processing time while consuming lower energy. On the other hand, the choice could go to the FPGA if a higher value of frames per second in one watt (fps/W) is targeted. Table 6.2 also shows that an embedded k-NN implemented on Zynqberry offers the fastest classification time and energy consumption. However, k-NN does not create a model during training and has a linear complexity as the number of training samples increases, which makes it only suitable for small scale applications. Table 6.2 also mentions pre-processing techniques, as they affect the input dimension, which in turn affects the complexity of the employable ML algorithms. For instance, in SVM, the input tensor is transformed into three matrices, where each matrix undergoes Singular Value Decomposition (SVD). Thus, to reduce the complexity of SVM, it is important to reduce the input size through subsampling. Moreover, there are silent intervals in the tactile data (i.e., regions of zero voltage are detected within the 10 seconds interval[224]), thus time truncation is applied to remove such regions. In this way, the complexity of the adopted ML algorithm is reduced and the classification accuracy is improved, which leads to reductions in time latency and energy consumption of the embedded system.

6.5 Conclusion

This chapter presents a tiny embedded CNN architecture that we have implemented on a Cortex-M microcontroller for e-skin applications. The tiny CNN features buffer reuse and layer fusion. Compared to existing solutions in the touch modality classification task, the tiny CNN requires a lower number of parameters and FLOPS, while achieving a slight improvement in accuracy. For inference, the embedded TinyCNN achieves real-time classification of tactile data with an energy consumption of 65.36 μJ. We thus argue that our TinyCNN on MCU solution would be advantageous for applications that have area constraints but still require low time latency and energy consumption. The promising results obtained now suggest the importance of validating the proposed TinyCNN and design decisions in other tactile-based tasks such as object recognition and slip detection.

Chapter 7

CBin-NN: Inference Engine for Binarized Neural Networks on Constraint Devices

7.1 Introduction

Shifting Deep Learning (DL)-based computer vision from cloud data centers to Internet of Things (IoT) devices is expected to offer benefits, including lower latency, reduced network requirements, and fewer privacy issues [45, 225]. Currently, the most widely used edge devices are the microcontrollers, which are typically characterized by energy efficiency and low costs [108]. The main challenges with these deployments are related to their computational and memory limitations. To cope with such limitations, techniques have been studied and solutions deployed, aimed at reducing the computational and memory requirements of DL models. A common paradigm is quantization, which reduces the number of bits used to represent the weights and the activations. Binarized Neural Networks (BNNs) are an extreme type of DL model quantization, with binary values, typically {-1, 1}. This leads to a significant reduction in memory requirements (i.e., up to 32x [226]) and enables highly efficient inference with XNOR and PopCount operations for binary multiplication and accumulation (i.e., up to 58x faster inference [227]), at the price of a drop in accuracy which is acceptable for some target applications [226, 227].

Deployment of BNNs in R&D is still limited. We argue that one of the reasons for this is that they are not yet adequately supported by publicly available libraries. As the resource limitation of typical edge devices clashes with the large model size and huge computational overhead of DL models [228], specialized solutions and development tools have been developed for what is now called TinyML (i.e., machine learning on embedded IoT devices) [88]. TinyML concerns specific model architectures, training and inference. Example tools

include Google TensorFlow Lite Micro [88], ARM CMSIS-NN [100], MicroTVM [90], MIT TinyEngine [75] and the STMicroelectronics STM32Cube.AI [102]. These tools concern floating-point or 8-bit quantized networks.

The goal of this chapter is to present an inference engine specifically devoted to BNNs, to favor the development of embedded AI applications on the edge and enable deployment on extremely constrained devices, such as the Cortex-M0 family.

This chapter introduces the CBin-NN inference engine. This is a framework for running BNNs on resource-constrained devices, namely microcontroller units (MCUs). CBin-NN primarily targets 32-bit Cortex-M devices (ARM). However, the library is written in platform-independent[1] C [229] and can run BNN models on bare-metal devices, so it is intended to be seamlessly portable to most software-programmable edge devices.

7.2 Binary Neural Network

State-of-the-art Convolutional Neural Networks (CNNs) are often unsuitable for embedded systems with limited resources due to their large model size and computational cost. Quantization is an optimization technique useful for deploying CNNs on resource-constrained devices. Binarization is the extreme case of quantization. By restricting both activations and weights to {-1,+1}, a 32 times memory saving is achieved compared with 32-bit floating-point networks. The *Sign* function shown in Equation 7.1 is normally used to binarize the activations and weights in the forward propagation:

$$Sign(x) = \begin{cases} +1, & \text{if } x \geq 1 \\ -1, & \text{otherwise} \end{cases} \tag{7.1}$$

Not only does binarization reduce memory requirements but also simplifies the computational logic. Unlike CNNs, which use expensive Multiplication and Accumulation (MAC) operations for convolutional layers, binary convolution can be implemented using XNOR and PopCount operations, as outlined in Figure 7.1. X_R, W_R, X_B, and W_B are the real and binary values of activations and weights, respectively. To avoid the use of two bits, -1 is encoded as 0. The binary weights of a BNN must first be learned through backpropagation. Training BNNs using the traditional gradient descent algorithm is not possible because the derivative of the sign function is 0, where defined. This problem has been solved using the straight-through estimator (STE) technique [226], as shown in Figure 7.2. STE can be

[1]For platform independence we refer to the lack of need for operating system libraries. The system only uses the C standard libraries.

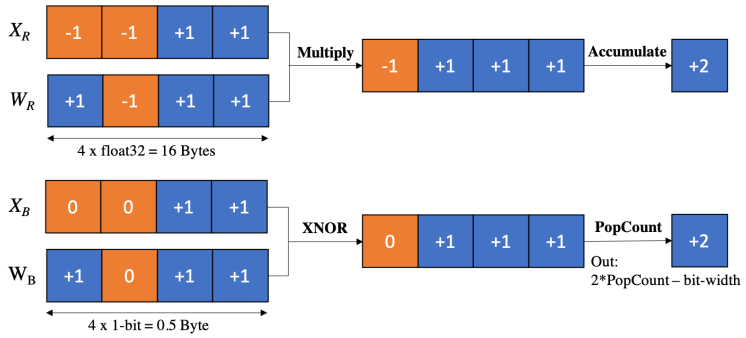

Fig. 7.1 Example of MAC operation in float vs binary representation.

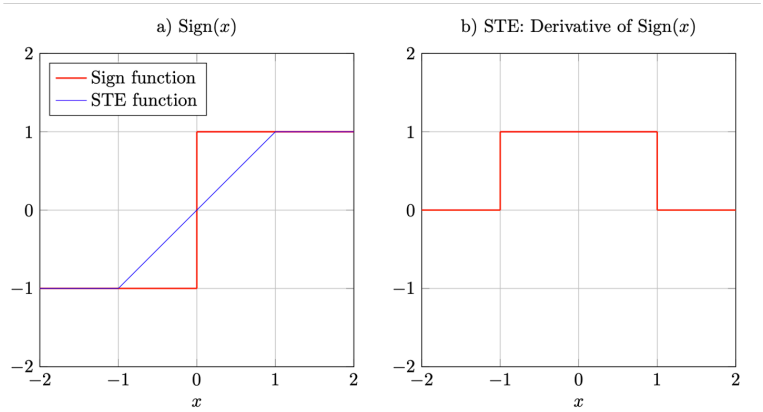

Fig. 7.2 The sign function and the use of the STE to enable gradient [4].

expressed as a clipped identity function:

$$\frac{\partial X_B}{\partial X_R} = 1_{|X_R \leq 1|} \tag{7.2}$$

where X_R and X_B are the real-valued input and binarized output of the sign function, respectively, and $1_{|X_R \leq 1|}$ takes the value 1 if $|X_R| \leq 1$ and 0 otherwise (see Figure 7.2). The gradient of the cost function C with respect to the real-valued weights using the chain rule can be written as follows:

$$\frac{\partial C}{\partial X_R} = \frac{\partial C}{\partial X_B}\frac{\partial X_B}{\partial X_R} = \frac{\partial C}{\partial X_B} * 1_{|X_R \leq 1|} \tag{7.3}$$

Also, a clip function 7.4 is introduced to truncate the update range of the latent weights (i.e., full-precision weights), so that they do not become too large without affecting the binary weights.

$$clip(W_R, -1, 1) = max(-1, min(1, W_R)) \tag{7.4}$$

Therefore, BNNs can be trained with the same gradient descent algorithms as real-valued CNNs using the techniques described above. In recent years, many optimization solutions have been proposed and have proven to be successful [230]. However, BNNs still suffer from accuracy degradation due to the severe information loss caused by parameter binarization, and the gap with their real-valued counterparts is still significant, in several cases.

7.3 CBin-NN Inference Engine

We present CBin-NN, an open source BNN inference engine for constrained devices. CBin-NN provides an optimized binarized implementation of all the basic CNN layer operators. The library is written in platform-independent C language, so to ensure seamless portability to most software-programmable edge devices. The following subsections introduce the techniques used to convert a real-value model for the inference phase, the operators implemented to execute the inference on the edge, as well as the optimizations made to increase inference efficiency.

7.3.1 Conversion to Inference Model

The smallest data type in the C language occupies 1 byte (8 bits). In the case of BNN, each parameter occupies 1 bit due to binarization. If BNN parameters were allocated as a single variable, 1 byte would thus be allocated for each of them, which would cancel out the memory savings compared to 8-bit quantized models. Bit-packing is a common procedure for BNNs in which N elements are binarized into 1-bit and then packed into an N-bit vector. In this way, XNOR can be performed directly between binarized vectors. CBin-NN uses bit-packing so that the actual in-memory implementation achieves the ideal gain of an 8x improvement over quantized 8-bit networks and a 32x memory saving over the full-precision ones. To store the model parameters, bit packing is performed offline using a Python script that extracts the network parameters and converts them to a compatible format for the inference phase. At the moment, the script accepts an h5 model trained with the Larq framework [66], with the goal of supporting other frameworks that support BNN training such as PyTorch. As mentioned earlier, BNN weights are restricted to $W \in \{-1,1\}$. To avoid using two bits, -1 is encoded as 0. The weights are then packed over the input channels to a multiple of 32, or padded to a multiple of 32 if less, to achieve optimal memory access patterns on MCUs. Common BNNs already have a multiple of 32 channels in all layers, so no padding is done in practice. During inference computation, CBin-NN binarizes the activations by extracting the sign bit, which is then packed into a multiple of 32 over the input channels to make optimal use of

memory. Padding is applied in cases where the size of the input is not a multiple of 32. Thus, CBin-NN operates on multiples of 32 input channels.

7.3.2 CBin-NN Operators

The CBin-NN library provides a binarized implementation of the fundamental CNN layer operators, as described in the following (Table 7.1 provides an outlook).

A common practice in BNNs, for performance reasons, is to keep the first and last layers in full precision [231]. Therefore, the inputs to the network are not binary and must be handled by a separate operator. Thus, the first two operators that will be presented (i.e., QBConv2D and QQConv2D) are designed to be the first network layer, and it is up to the user to choose among them, depending on requirements and results.

7.3.2.1 QBConv2D

Quantized Input Binarized Kernel Convolution. *QBConv2D* receives quantized inputs and binary kernel weights and writes the corresponding bit-packed outputs. The convolution operation in this function is computed using a comparator. A weight of -1 (encoded as 0) leads to a negative accumulation of the inputs, while a weight of 1 leads to a positive accumulation of the inputs. In addition, *QBConv2D* supports layer fusion for Batch Normalization (BN) according to Equation 7.5:

$$y_i = \gamma(\frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}) + \beta \tag{7.5}$$

where x_i and y_i are the input and output of the batch normalization, and γ, β, μ, σ, and ε are parameters obtained from the training phase. BN is applied after convolution and has been shown to be essential in BNNs to maintain classification accuracy. Equation 7.5 can be rewritten as follows:

$$y_i = \alpha_{2i}(x_i - \alpha_{1i}) \tag{7.6}$$

where $\alpha_{i1} = (\mu - \frac{\sqrt{\sigma^2+\varepsilon}}{\gamma}\beta)$ and $\alpha_{i2} = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}$. This fusion is applied to the accumulator values before they are written to memory. This avoids additional reads and writes, that would occur if the BN was treated as a separate layer. The resulting output activations are then binarized using the sign function, and finally, the binary activations are bit-packed as described in Subsection 7.3.1.

7.3.2.2 QQConv2D

Quantized Input Quantized Kernel Convolution. This operator receives quantized inputs and quantized weights. It writes bit-packed outputs. Unlike QBCon2D, the weights are not

binary, but 8-bit quantized, to increase accuracy. The weights are stored using the "int8_t" data type, which occupies 1 byte of memory which is the smallest allocatable memory in C. *QQConv2D* uses the classic MAC instructions to compute the convolution operation. It is more efficient than the comparator used in *QBConv2D* in terms of inference latency, but this comes at the cost of a slight increase in model size. Likewise, BN fusion is supported in this function and follows the same steps as in *QBConv2D*. Lastly, the final outputs are binarized and bit-packed.

7.3.2.3 BBConv2D

Binary Input Binary Kernel Convolution. This operator accepts bit-packed inputs and weights. It writes bit-packed outputs. Binary convolution is implemented with XNOR and PopCount operations. Since Cortex-M processors do not support the XNOR operator, the XOR operator is used instead and the result of PopCount is inverted as in Equation 7.7:

$$y_i = \sum_{i=0}^{\frac{c_{in}}{32}} N - 2 * PopCount(x_i \; XOR \; w_i) \tag{7.7}$$

where x_i, y_i and w_i are the input, output, and weight of a convolutional step over the input channels, respectively. In other words, a convolutional step for a $5 \times 5 \times c_{in}$ kernel is the first column and row of the kernel across channels $1 \times 1 \times c_{in}$ convolved over a $1 \times 1 \times c_{in}$ receptive field simultaneously. N is the bit width, and c_{in} is the number of input channels. N is equal to 32 because the weights and activations are packed into multiples of 32 across the input channels. If the number of input channels is less than 32, N would be equal to the number of input channels before padding. Finally, this operator would perform up to 32 MACs with only 4 instructions namely *XOR*, *PopCount*, *Multiply*, and *Subtract*.

For padded convolutions, which are very common, the padded pixels are skipped in the calculation because they would distort the results. This is because a padding value, which is usually 0, would be treated as -1 according to the bit-packing specifications. Similarly, *BBConv2d* supports BN fusion. The BN layer can be approximated by adding an integer bias *W'*, which can be computed after training by using the BN parameters according to [232]. The resulting activations are then binarized by extracting the sign bit, and bit-packed.

7.3.2.4 BBPointwiseConv2D

Binary Input Binary Kernel Pointwise Convolution. This operator has the same specifications as *BBConv2D*, with one minor change. The loops for kernel height and width are removed

since they are equal to 1. This reduces the latency that would result from unnecessary loop branching.

7.3.2.5 BMaxPool2D

Binary Max-Pooling. This function is applied to bit-packed inputs and simply performs a bitwise *OR* to efficiently compute the binary max pool and reduce overall complexity. The operator *OR* is executed over the input channels rather than bitwise. This further accelerates the computation by running a 32-bit vector *OR* 32-bit vector instead of a 1-bit *OR* 1-bit.

7.3.2.6 BBFC

Binary Input Binary Weights Fully Connected. This operator expects bit-packed inputs and weights. It writes bit-packed outputs. The *XOR* and *PopCount* operators are used to implement the vector-matrix multiplication and perform 32 MAC at once. This is possible because the inputs to this layer are multiples of 32, as are the weights. This approach is more efficient than using a comparator that must check every bit in the input vector, and speeds up performance. *BBFC* supports BN fusion as in *BBConv2D* by adding an integer bias *W'*, followed by binarization and bit-packing.

7.3.2.7 BBQFC

Binary Input Binary Weights Quantized Output Fully Connected. Similar to *BBFC*, this operator accepts bit-packed inputs and weights but writes quantized outputs for the classification layer. This layer uses the same approach as for *BBFC* to compute the vector-matrix multiplication. Similarly, this operator merges the BN layer by adding *W'* to the accumulator. The accumulated activations are then written to memory without binarization.

7.3.3 Operator Optimization

Several optimization techniques are used to increase inference speed and improve operators' performance, particularly loop unrolling and memory caching.

7.3.3.1 Loop Unrolling

Loop unrolling is a well-known optimization that can be used to efficiently execute convolutional loops. When a loop is unrolled by a factor of K, the loop body is repeated K times and the loop's iteration space is reduced (or eliminated if the loop is unrolled completely).

Table 7.1 CBin-NN Operators

Operator	Input	Weights	Output	Notes
*QBConv2D**	8-bit quantized	32-bit packed	32-bit packed	Comparator + BN Fusion
*QBConv2D_Optimized**	8-bit quantized	32-bit packed	32-bit packed	Comparator + LU + BN Fusion
*QBConv2D_Optimized_PReLU**	8-bit quantized	32-bit packed	32-bit packed	Comparator + LU + BN & PReLU Fusion
*QQConv2D**	8-bit quantized	8-bit quantized	32-bit packed	Comparator + BN Fusion
*QQConv2D_Optimized**	8-bit quantized	8-bit quantized	32-bit packed	Comparator + LU + BN Fusion
*QQConv2D_Optimizd_PReLU**	8-bit quantized	8-bit quantized	32-bit packed	Comparator + LU + BN & PReLU Fusion
BBConv2D	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + BN Fusion
BBConv2D_Optimized	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + LU + BN Fusion
BBConv2D_Optimized_PReLU	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + LU + BN & PReLU Fusion
BBPointwiseConv2D	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + BN Fusion
BBPointwiseConv2D_Optimized	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + LU + BN Fusion
BBPointwiseConv2D_Optimized_PReLU	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + LU + BN & PReLU Fusion
BMaxPool2D	32-bit packed	-	32-bit packed	bit-wise OR
BMaxPool2D_Optimized	32-bit packed	-	32-bit packed	32-bit Simultaneous OR
BBFC	32-bit packed	32-bit packed	32-bit packed	Comparator + BN Fusion
BBFC_Optimized	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + BN Fusion
BBFC_Optimized_PReLU	32-bit packed	32-bit packed	32-bit packed	XOR & PopCount + BN & PReLU Fusion
BBQFC§	32-bit packed	32-bit packed	quantized± (8, 16, 32 bit)	Comparator + BN Fusion
BBQFC_Optimized§	32-bit packed	32-bit packed	quantized± (8, 16, 32 bit)	XOR & PopCount + BN Fusion
BBQFC_Optimized_PReLU§	32-bit packed	32-bit packed	quantized± (8, 16, 32 bit)	XOR & PopCount + BN & PReLU Fusion

LU stands for Loop Unrolling.
* Usable as the first layer of a network.
§ Usable as the last layer of a network for classification.
± The cumulative output for the classification layer. It is an integer that can be 8, 16, or 32 bits long.

This technique increases the degree of parallelism in the loop body as the number of operations increases. We use this technique to optimize the operators presented in the previous subsection. For the convolution operators such as*QBConv2D, QQConv2D, BBConv2D, and BBPointwiseConv2D*, the first loop of the convolution corresponding to the output channels is unrolled by a factor of 32. We unrolled this loop by this factor because network architectures commonly use multiples of 32 filters. This allows 32 filters to be processed simultaneously, rather than one at a time. Moreover, we have completely unrolled the loop that iterates over the input channels in the operators *QBConv2D and QQConv2D*. It is possible to eliminate this loop since these operators correspond to the first network layer. Thus, the number of iterations is fixed before execution and corresponds to the number of channels in the input images. This is not possible with the other intermediate convolution operators (*BBConv2D and BBPointwiseConv2D*), because the number of input channels is variable. Moreover, the bit-packing approach already reduces the computation of this loop by a multiple of 32, as a 32-bit binary weight vector is simultaneously convolved over a 32-bit binary receptive field.

7.3.3.2 Memory Caching

A cache is a small and fast memory component in the computer/microcontroller that is inserted between the CPU and main memory to reduce the average cost (time and/or energy) of accessing data from main memory. The STM32H7 series devices include an optional 16 Kbytes of L1-cache for instructions (I-cache) and 16 Kbytes of L1-cache for data (D-cache) [233]. An L1-cache stores data or a set of instructions near the CPU, so the CPU does

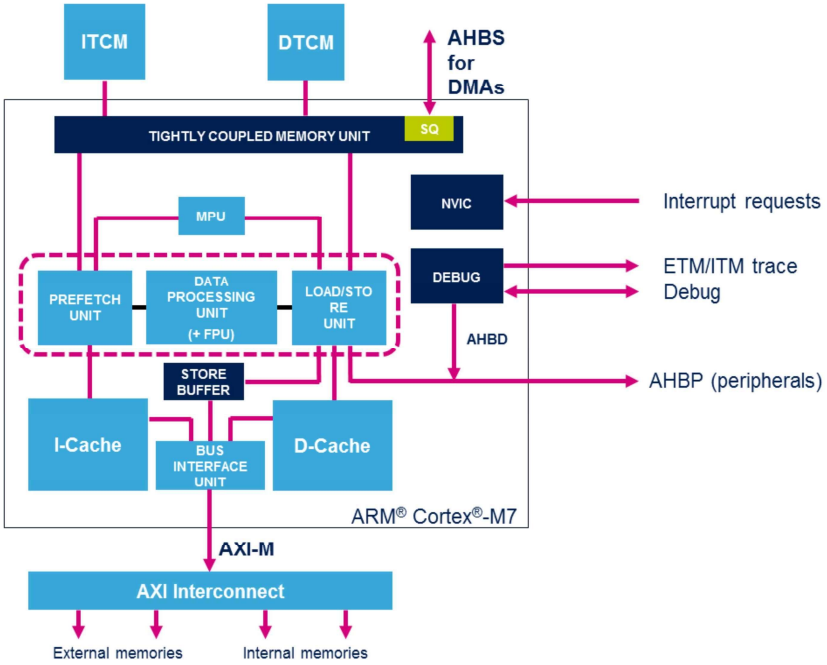

Fig. 7.3 STM32H7 series system architecture [5].

not have to keep fetching the same data that is used repeatedly. Figure 7.3 illustrates the system architecture of the STM32H7 series. Since memory access to the subsystem can take several cycles (especially for the external memory interfaces with multiple wait states), the caches are meant to speed up the read/write operations to the memory. Moreover, as the core frequency increases, the performance penalty for loading instructions and data from slow external memories (Flash/SRAM) increases. The idea is that both operations (read/write) can be optimized if the data is available locally (in an area that requires only one cycle to access). Bus accesses to subsystem memory, which require more than one CPU cycle to execute, are different from the CPU pipeline instruction stream execution. This allows for a large jump in performance. A cache is usually implemented with sets of lines, where a line is just a short segment of memory. The number of lines in a set is called x-way associative. This property is determined by the hardware design.

The speed of the binarized convolution also depends on how efficiently the CPU cache can be used, which is where binarized layers have an advantage. For instance, a binary neural network model that is small enough can often fit entirely in the L1 cache, unlike the float or 8-bit equivalents. Therefore, the CPU I-cache and D-cache were enabled for a performance improvement in terms of latency.

7.4 Case Studies

7.4.1 Robotic Touch Modality Classification

As a first use case to test the CBin-NN inference engine, we have chosen the processing of tensorial tactile data, namely the classification of touch modality, which has already been treated in the literature using binarization. Processing tactile data near its source (at the edge) is of great importance in many applications such as touch recovery and human-robot interactions.

7.4.1.1 Dataset

The robotic touch modality dataset presented in Chapter 6, Subsection 6.2.2.1 is used first to test the CBin-NN inference engine. As described previously, the dataset contains three touch modalities, namely sliding the finger, brushing a paintbrush, and rolling a washer. In addition, we performed the same pre-processing steps as before, namely sub-sampling and data augmentation.

7.4.1.2 BNN design and training

The topology we designed for our BNN test is shown in Figure 7.4. The model includes most of the generic operators that are described in Section 7.3.2. The model consists of two convolutional layers, a max-pooling layer, two fully connected layers, and a softmax layer. The first layer has a hybrid structure, where the inputs are quantized and the kernels are binarized since this layer is intended to extract as many features as possible. Full binarization is performed on all other layers. Batch normalization is applied in each layer, not only to speed up training time and convergence but also to compensate for the unbalanced output due to the loss of precision caused by binarization [232]. The output layer consists of three neurons, representing the three touch modalities of the classification task [213]. The proposed architecture is similar to the one presented (with an FPGA implementation) in [138], but with two key differences: (i) a larger number of filters for the convolutional layers (32 compared to 7) and (ii) a larger number of neurons for the fully connected layer (32 compared to 15). This could be obtained thanks to the 32-bit parameter packing memory management strategy presented in the previous section.

We used the Larq framework [66] to model and train the BNN on the tensorial tactile data. Larq is an open-source Python library for training NNs with extremely low-precision weights and activations, such as BNNs. The network is trained for 300 epochs with 5-fold

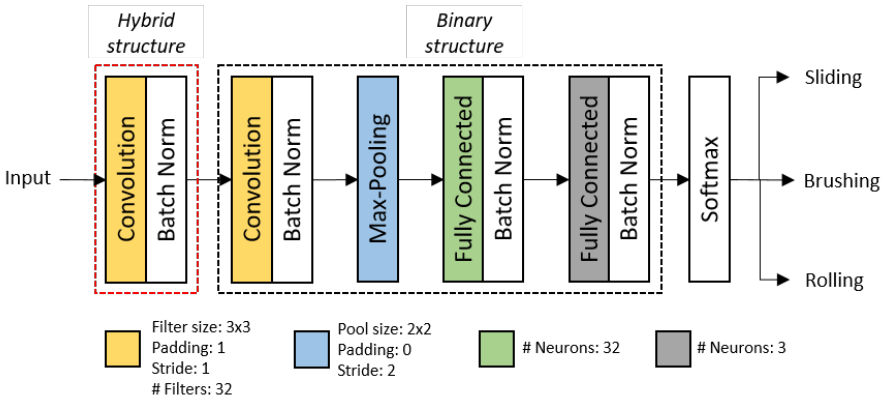

Fig. 7.4 Topology of the designed BNN.

cross-validation and a batch size of 64. A learning rate of 0.01 is used with Adam optimizer and "ste_sign" as input and kernel quantizer and "weight_clip" for kernel constraint.

7.4.1.3 Experimental Results

This subsection presents the test results of our BNN model (called BNN_Micro) trained on the tactile data presented in the previous sub-section. The model has been implemented on the STM32 H734ZI2 commercial microcontroller unit (MCU), hosted on a NUCLEO-144 board. The MCU is an Arm Cortex-M7 core running at 480 MHz, with 512 KB SRAM and 2 MB flash memory. Results are reported in Table 7.2.

Table 7.2 Comparison with similar solutions

Algorithm	Accuracy	Model Size	Latency
H-CNN [138]	77%	2.7 KB	0.8 ms
SVM [221]	72.8%	52.2 KB	$3.3x10^3$ ms
SVM [234]	70.9%	1.7 KB	28 ms
BNN_Micro	**78.84%**	2.3 KB	**0.6 ms**

- **Accuracy:** Compared to existing solutions targeting the same touch modality classification problem, our model achieves a higher classification accuracy of 78.84%. Figure 7.5 shows the training and test accuracy (y-axis) over 300 training epochs (x-axis). A similar approach using BNNs (i.e., H-CNN) achieves 77% accuracy. We argue that the improvement of our model is due to the larger number of filters and neurons used in the convolutional and fully connected layers, respectively. Other approaches, using the SVM algorithm, achieve quite lower accuracy (6% less in [221], 8% less in [234]).

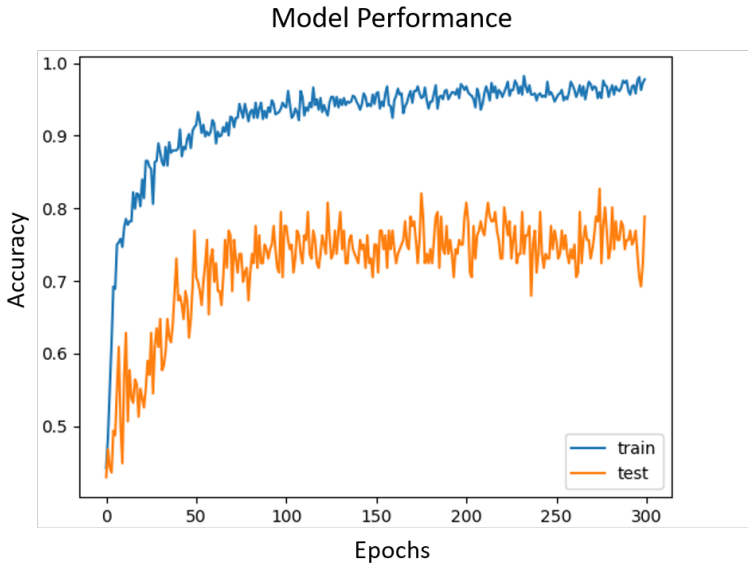

Fig. 7.5 Train and test accuracy.

- **Memory Footprint:** Regarding the size of the model, the only one BNN in Table 7.2 (namely, the H-CNN) requires 2.7 KB, while our implementation requires 2.3 KB (15% improvement). We could achieve this reduction (while using a larger architecture in terms of the number of neurons and filters), because of the efficient 32-bit packing for the weights, while the H-CNN accelerator is designed with a 16-bit fixed-point representation. Considering non-BNN models, BNN_Micro requires around 96% less memory compared to the SVM in [221] and 26% more than the SVM in [234], but with an 8% increase in accuracy.

- **Latency:** The comparison in terms of latency is not straightforward, since each model is implemented on a different hardware platform with a different processing speed. However, the proposed BNN_Micro provides 5,500x speedup compared to the SVM implementation on Ultra Low Power Processor (PULP) [221] and 47x speedup compared to the SVM implementation on ZedBoard [234]. Moreover, compared to the 0.8 ms FPGA-based H-CNN implementation with Zynqberry [138], BNN_Micro achieves a 25% (0.6 ms) speedup even with a deeper architecture. We argue that this speedup is due to the optimization techniques discussed in Section 7.3.3, namely loop unrolling and the enabled cache. The 0.6 ms latency reached by BNN_Micro meets the real-time constraints for the targeted robotic touch classification task. Table 7.3 shows the increase in inference efficiency obtained utilizing the optimization techniques discussed in Section 7.3.3. The unoptimized BNN version has an inference time of

4 ms. The loop unrolling technique results in a 1.9x speedup, while enabling CPU I-cache and D-cache results in a 4x speedup. This last performance increase is due more to the I-cache than to the D-cache. This is because fetching data from the D-cache is only slightly faster than from the SRAM (where the data is stored), while fetching instructions from Flash (where the code is stored) is much slower than from the I-cache, thus the improvement is more significant when enabling CPU I-cache. The concomitant employment of both the loop unrolling and the caching strategies achieved a 0.6 ms. latency, with an 85% latency reduction.

Table 7.3 Effect of optimization techniques on latency

Optimization	Latency
None	4 ms
Loop Unrolling (LU)	2.1 ms
I-cache + D-cache	1 ms
LU + D-cache	2 ms
LU + I-cache	1.2 ms
LU + I-cache + D-cache	0.6 ms

- **CBin-NN vs Baseline Implementation:** To highlight the advantages of using the proposed library functions, we consider the performance of the baseline implementation (namely, BNN_Micro Baseline), always running on an STM32 H734ZI2 MCU, but without the optimized operators. Particularly, the baseline lacks the bit packing approach, according to which each weight is stored in a 8-bit representation ("int8_t"), which occupies the smallest allocatable space in C. Thus, each parameter is stored in one byte, even if it represents 1 bit. Also, the binary operations (XOR and popcount) are replaced by the normal MAC instructions. The comparison results are shown in Table 7.4. The baseline implementation has the same accuracy since the exact same model (also in terms of quantized parameters and operations) is used. However, considering memory usage and latency, the BNN_Micro model is 7x smaller and 5.5x faster than the baseline.

Table 7.4 Comparison with baseline model

Algorithm	Model Size	Latency
BNN_Micro Baseline	15.7 KB	3.3 ms
BNN_Micro	2.3 KB	0.6 ms

7.4.2 Computer Vision Application

To prove the applicability of the CBin-NN inference engine in other TinyML domains, we selected a computer vision application represented by the CIFAR-10 [206] dataset.

7.4.2.1 Dataset

The CIFAR-10 dataset consists of 60,000 color images of size 32×32, divided into 10 classes, with 6,000 images per class. 50,000 images are used for training and 10,000 for testing. The dataset is smaller than others dedicated to computer vision, in terms of the number of both samples and classes, but we argue that it is a better representative of the TinyML embedded vision domain [235].

7.4.2.2 BNN Design and Training

As reference architecture for the tests, we used SmallCifar (see Figure 7.6), a small network used in literature [100, 91] for the CIFAR dataset. It takes as input an image of size 32×32, that is passed through three convolutional layers with a kernel size of 5×5, each followed by a max-pooling layer. As mentioned earlier, the BN layer is essential for maintaining accuracy. Therefore, a BN layer is added after each convolutional layer. The output channels (i.e. number of filters) are 32, 32, and 64, respectively. The final feature maps are flattened and passed through a linear layer of weight 1024×10 to obtain the class. The model is quite small and well suited to embedded devices.

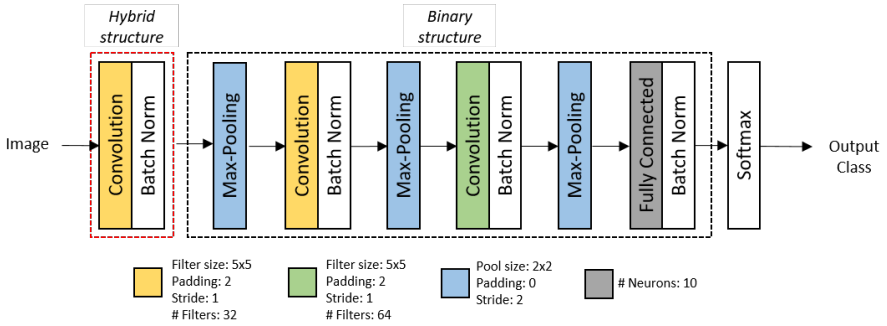

Fig. 7.6 SmallCifar topology.

To model and train the BNN we also used the Larq framework [66]. The network is trained for 500 epochs and a batch size of 50. To achieve high performance, we found it useful to resort to two main solutions concerning the binary optimizer and the activation functions:

1. Binary Optimizer: For BNN training, we used the latent-free Binary Optimizer (Bop) introduced by [236]. Bop is specifically designed for BNNs and binary weight networks (BWN). Bop has only one action available, which is to flip weights by changing the sign. It maintains an exponential moving average of gradients controlled by the adaptive

rate γ. When this average exceeds a threshold τ, a weight is flipped. We set γ to $1e^{-4}$ and τ to $1e^{-8}$ during training.

2. PReLU: [237] argue that an unbalanced distribution of activations can improve the accuracy of BNNs. They show that using an additional activation function (between the binary convolution layer and the following BN layer) makes the activation distribution unbalanced and thus improves accuracy. For this reason, we used an additional activation function after each convolutional and fully connected layer in the SmallCifar architecture. Results are discussed in the following subsection. We should emphasize that the operators described in the 7.3.2 subsection also support layer fusion for the PReLU activation function according to Equation 7.8:

$$PReLU(x_i) = \begin{cases} a_i x_i, & \text{if } x \leq 0 \\ x_i, & \text{if } x > 0 \end{cases} \tag{7.8}$$

where x_i is the output activation and a_i is a learnable array with the same shape as x_i. To avoid excessive memory consumption[2] for the convolutional layers, we have shared the parameters over the entire space, so that (i.e., each filter has only one parameter) [238].

7.4.2.3 Experimental Results

This section presents the test results comparing our inference framework with other existing frameworks. Comparison is in terms of accuracy, latency and memory requirements. For model deployment and library testing, we employed the STM32F746 commercial, which is housed on a NUCLEO-144 board. The MCU is an Arm Cortex-M7 core running at 216 MHz, with 320 KB SRAM and 1 MB flash memory. We compare CBin-NN and the related frameworks using this well established hardware platform, so to have a fair performance analysis.

- **Accuracy:** Table 7.5 shows the SmallCifar accuracy results using different inference engines. All available libraries are tailored to quantized models (typically 8-bit quantization), while CBin-NN is specialized for BNNs. The same 8-bit quantized model is deployed using state of the art libraries [88, 91, 100, 75], reaching a 79.9% accuracy. The basic CBin-NN implementation has lower accuracy because of the binarization (12.6% less). Replacing the Adam optimizer with the Bop yields an accuracy of 71.9%.

[2] As the number of output activations x_i increases, also the number of activation function parameters a_i to be stored increases. This can lead to considerable memory overhead.

Adding the PReLU activation function after the convolutional and fully connected layers leads to 72.5%. Another performance improvement is obtained by keeping the weights of the first network layer in a quantized 8-bit representation instead of binarization (76.1%). Combining all these optimizations, we achieve a 77.4% accuracy, which is only 2.5% less than the 8-bit quantized model.

Table 7.5 SmallCifar accuracy using different inference engines

Engine	Accuracy	Notes
TF-Lite Micro [88]	79.9%	-
MicroTVM [91]	79.9%	-
CMSIS-NN [100]	79.9%	-
TinyEngine [75]	79.9%	-
CBin-NN	67.3%	Adam Optimizer*
CBin-NN	71.9%	Bop Optimizer*
CBin-NN	72.53%	Bop + PReLU*
CBin-NN	76.12%	Bop + QQConv2D±
CBin-NN	**77.42%**	Bop + PReLU + QQConv2D±

* First layer weights are binarized and QBConv2D is used.
± First layer weights are 8-bit quantized and QQConv2D is used.

- **Memory Footprint** Comparison on model size are reported in Table 7.6. Our initial implementation is 7.5x smaller than existing frameworks (11.6 KB vs. 87.3 KB). This is mainly due to the binarization, where each parameter occupies only 1 bit compared to 8 bits in the quantized representation. Adding the PReLU activation slightly increases the model size. The PReLU layer requires storing additional parameters (cfr. Subsection 7.4.2.2). To improve performance, the weights of the first layer are quantized instead of binarized (see Table 7.5). This increases the size of the model, as each parameter in the first layer now occupies 1 byte instead of 1 bit. This approach does not significantly affect the model size, since the first layer usually has only 3 channels corresponding to the number of channels in the input images. Thus, the size of the largest model (quantized first layer and PReLU) increases to 14.2 KB. This, on the other hand, improves the accuracy by approximately 5%.

Table 7.6 SmallCifar size using different frameworks

Engine	Accuracy	Size	Notes
TF-Lite Micro	79.9%	87.34 KB	-
MicroTVM	79.9%	87.34 KB	-
CMSIS-NN	79.9%	87.34 KB	-
TinyEngine	79.9%	87.34 KB	-
CBin-NN	71.9%	**11.58 KB**	Bop*
CBin-NN	72.53%	12.12 KB	Bop + PReLU*
CBin-NN	76.12%	13.63 KB	Bop + QQConv2D±
CBin-NN	**77.42%**	14.17 KB	Bop + PReLU + QQConv2D±

* First layer weights are binarized and QBConv2D is used.
± First layer weights are 8-bit quantized and QQConv2D is used.

In terms of peak memory (input and output activations [76] for the peak layer/block), CBin-NN reduces memory requirements by up to 28.8 times. This is mainly due to binarization, where activations are 32-bit packed. Figure 7.7 shows the difference between the available inference engines in terms of peak memory usage. CBin-NN is 12.8x and 28.8x more memory efficient compared to the TF-Lite Micro and MicroTVM libraries, respectively. Moreover, our proposed library requires 13.4x and 9.2x less memory than the CMSIS-NN and TinyEngine frameworks.

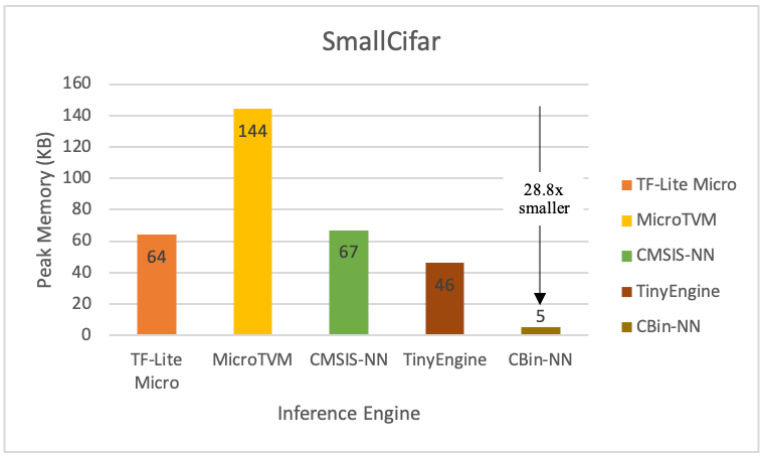

Fig. 7.7 SmallCifar memory footprint using various inference engines.

It is important to note that the achieved memory reductions enable the use of some NN models also on extremely constrained, low cost devices, This is the case of SmallCifar on the STM32F091RC, which has an Arm Cortex-M0 core running at 48 MHz, with 32 KB SRAM and 256 KB flash memory. Testing the most accurate model (77.4%) on the F0 MCU yields a latency of 925 ms, which may also meet the real-time requirements of some applications.

- **Latency:** The bar chart in Figure 7.8 shows that the CBin-NN achieves higher inference efficiency than the other inference engines. Our library is 3.6x and 1.4x faster than TF-Lite Micro and MicroTVM, respectively. Compared to the CMSIS-NN library and the MIT TinyEngine, CBin-NN provides 20% and 15% lower inference latency, respectively. This latency result is obtained with the highest accuracy CBin-NN configuration (last row in Table 7.6).

  Table 7.7 analyzes the effects on latency and accuracy of the single optimization approaches. The inference time of the initial model, where the weights also of the first layer are binarized, is 128 ms. In this case, the QBConv2D operator is used to compute the convolution. This operator uses a comparator, which increases the time complexity

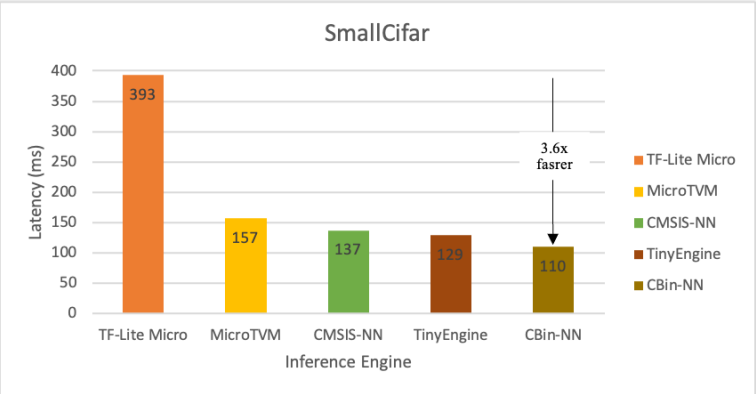

Fig. 7.8 SmallCifar latency using various inference engines.

due to its if-else instructions. The addition of the PReLU activation function comes at the expense of higher latency (2 ms slower) due to the additional computations from Equation 7.8. On the other hand, when the weights of the first layer are 8-bit quantized rather than binary, the QQConv2D operator is used during inference, which is more efficient than the QBConv2D operator since the classical MAC instructions are faster than comparators. This approach improves both accuracy and latency (76.12% and 107 ms, respectively) with a small increase in model size (see above). Similarly, the inference latency rises to 110 ms when a PReLU activation function is added, which is 3 ms slower than without PReLU.

Table 7.7 SmallCifar performance with different optimizations

Optimization	Accuracy	Latency
Bop*	71.9%	128 ms
Bop + PReLU*	72.53%	130 ms
Bop + QQConv2D±	76.12%	**107 ms**
Bop + PReLU + QQConv2D±	**77.42%**	110 ms

* First layer weights are binarized and QBConv2D is used.
± First layer weights are 8-bit quantized and QQConv2D is used.

7.5 Conclusion

BNNs radically reduce the memory footprint compared to 8-bit quantized models and full precision models, without a huge accuracy drop. They also reduce the computational cost thanks to simple XNOR and PopCount operations. To support the effective deployment of BNNs, we have proposed a library of layer operators that facilitates simple yet flexible CNNs with binary weights and activations. CBin-NN, has been developed in platform-independent

C, supporting seamless porting to any software-programmable device, including affordable but extremely memory-limited devices such as Cortex-M0.

We tested the library by implementing a 6-layer BNN for robotic touch classification on the STM32H743ZI2 MCU. The results show that the accuracy is improved by 2%, while the memory consumption is reduced by 15% compared to the state-of-the-art, with a latency of 0.6 ms, which is 25% lower than a dedicated-hardware FPGA implementation, and able to satisfy the real-time application constraint. Further experimental analysis on the STM32F746 MCU using the CIFAR-10 dataset shows that our library, employing also some specific optimizations (e.g., channel-wise max pooling, Bop optimizer, PReLU additional activations, loop unrolling, differentiated quantization for input layers), outperforms state of the art TinyML deployment solutions. In fact, CBin-NN speeds up inference by 3.6 times and reduces the memory required to store model weights and activations by 7.5 times and 28 times, respectively, but at the cost of slightly lower accuracy (2.5%).

CBin-NN is an open-source inference engine within the ELM environment, available on GitHub at CBin-NN. We hope it can become a useful versatile toolkit to deploy binarized models.

As limitations, we cite two aspects: the toolkit does not focus on very large datasets, for which binarization typically involves a significant accuracy loss [239]. Also, architectures specifically designed for mobile and embedded devices, such as MobileNet, suffer from performance degradation during binarization due to their use of depth-wise convolutional layers [240].

We believe that a major goal for future research is the design of a dedicated optimizer for BNNs, to mitigate the performance degradation due to the gradient mismatch problem [241]. We expect that this should allow an architecture designed specifically for mobile/edge devices to achieve comparable accuracy to its 8-bit, full-precision counterparts. Finally, we aim to further optimize the available operators to make inference even more efficient, by using Single Instruction Multiple Data (SIMD) in QQConv2D, since both the inputs and the weights are in 8-bit quantization representation.

Chapter 8

Conclusion

The use of machine learning algorithms on constrained devices is a current area of research that has led researchers and industry giants to the concept known as TinyML. This field focuses on optimizing ML workloads so that they can be processed on embedded devices. However, most of these devices are severely limited in terms of processing power, available memory, battery power, etc. With this in mind, this dissertation aims to analyze how various modifications to established approaches can be used to improve the performance of intelligent algorithms on such devices.

For a variety of IoT applications in the TinyML field, we have developed and evaluated an environment, namely Edge Learning Machine (ELM). ELM provides three main TinyML services, namely shallow ML, self-supervised ML, and binary deep learning on constrained devices, in addition to other independent approaches to improve the state of the art in deployment solutions. The initial implementation of ELM consists of two modules, one of which, Desk-LM, supports training of various algorithms using TensorFlow, Keras, and Skicit-Learn, seeking the best configuration for a variety of user-defined parameters. The second module, Micro-LM, performs inference on microcontrollers using models generated and optimized by the Desk-LM module. Extensive analysis in terms of hardware platforms, algorithms, datasets, and configurations show the impact of these factors and the importance of such a framework for the TinyML domain. We then went a step further to enable on-device training on microcontrollers and tiny IoT devices leveraging the vast amounts of unlabeled data and allowing for lifelong incremental learning. The ELM environment is then extended to include the Self-Learning Autonomous Edge Learning and Inferencing Pipeline (AEP) system. Analysis of the AEP shows interesting results and, in short, achieves similar levels of performance to models trained in the cloud on actual labels and with a full dataset. Moreover, the CBin-NN inference engine has been added to the ELM environment. This library enables the implementation of binarized neural networks on constrained devices. CBin-NN can be

considered an important milestone in this field, as BNNs offer significant memory reduction and high inference efficiency, which is crucial for the TinyML domain. This inference engine was evaluated for robotics and computer vision applications and outperformed existing solutions and inference libraries for the TinyML domain in terms of memory requirements (up to 28.8x smaller) and inference speed (up to 3.6x faster), but at the cost of a slight reduction in accuracy (2.5%). In addition to the ELM environment, we also proposed a memory replacement strategy for an existing TinyML inference library that reduced memory requirements by up to 9% without compromising performance. We have also proposed a new solution in the form of a tiny CNN architecture on microcontrollers for e-skin applications. This design, which also benefits from the proposed memory replacement strategy, improves accuracy with much lower FLOPs and parameters, and achieves real-time latency with low power consumption compared to existing solutions for the same application.

ELM is an actively developed open-source project intended to be used in any microcontroller that supports C code and is available on GitHub at ELM. We hope it can become a useful versatile toolkit for the IoT and TinyML R&D community.

In the future, we aim to further enhance and optimize the environment and in particular the CBin-NN inference engine. To improve the overall performance and also mitigate the degradation of accuracy that BNNs suffer from, the following techniques can be used:

- Single Instruction Multiple Data (SIMD) may be included in some of the supported operators for parallel processing. In addition, convolutional loops can be optimized by using loop tiling to further increase parallelization.
- The development of a dedicated optimizer for BNNs would mitigate the performance degradation due to binarization (i.e., the gradient mismatch problem).
- Constrained Neural Architecture Search (Constrained NAS) can be used to find the best architecture for an application that improves performance while considering the constraints that would allow porting these networks to embedded devices.

Finally, Transformer networks [242] have become state of the art for many tasks such as NLP and fill the gap for other tasks such as computer vision [243]. However, Transformer models were previously considered too large and too complex to run on constrained platforms such as MCUs. Therefore, in the future, we intend to pave the way for running Transformers on tiny devices [147], especially for computer vision, exploiting available optimization techniques.

8.1 Lessons Learned

The presented thesis offers insightful and valuable lessons derived from extensive research and analysis in the emerging field of TinyML. These lessons not only provide a deeper understanding of the field, but also serve as a guide for future work and advancements in this rapidly growing area:

1. The importance of optimizing ML workloads for constrained devices due to the limitations of processing power, memory, and battery power.
2. The performance of ML/DL algorithms on constrained devices is greatly influenced by hardware platforms, algorithms, datasets, and configurations.
3. On-device training plays a pivotal role in reducing reliance on cloud computing resources and enabling incremental learning.
4. A Binary Neural Network inference engine is a crucial element in enhancing the efficiency of TinyML applications and reducing memory usage.
5. Innovations in memory replacement strategies and compact architecture designs hold tremendous potential in mitigating memory constraints, increasing accuracy, and reducing power consumption for specific TinyML applications.

References

[1] V. Rajapakse, I. Karunanayake, and N. Ahmed, "Intelligence at the extreme edge: A survey on reformable tinyml," *CoRR*, vol. abs/2204.00827, 2022.

[2] D. L. Dutta and S. Bharali, "Tinyml meets iot: A comprehensive survey," *Internet of Things*, vol. 16, p. 100461, 2021.

[3] "Google scholar," accessed: 2022-03-29. [Online]. Available: https://scholar.google.com/schhp?hl=en&as_sdt=2005&sciodt=0,5

[4] F. de Putter and H. Corporaal, "How to train accurate bnns for embedded systems?" 2022, unpublished. [Online]. Available: https://arxiv.org/abs/2206.12322

[5] STMicroelectronics, "STM32H7-System-ARM Cortex M7 (M7)," STMicroelectronics, Tech. Rep., 2020.

[6] "Fouad sakr - google scholar," accessed: 2022-08-9. [Online]. Available: https://scholar.google.com/citations?user=x3TEgPQAAAAJ&hl=en

[7] "Gartner. 2022. top 10 strategic iot technologies and trends | gartner," accessed: 2022-08-9. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-trends

[8] "Idc. 2020. iot growth demands rethink of long-term storage strategies, says idc." accessed: 2022-08-9. [Online]. Available: https://www.idc.com/

[9] M. A. Marotta, L. R. Faganello, M. A. K. Schimuneck, L. Z. Granville, J. Rochol, and C. B. Both, "Managing mobile cloud computing considering objective and subjective perspectives," *Computer Networks*, vol. 93, pp. 531–542, 2015, cloud Networking and Communications II.

[10] "What is vendor lock-in? | vendor lock-in and cloud computing," accessed: 2022-08-9. [Online]. Available: https://www.cloudflare.com/en-gb/learning/cloud/what-is-vendor-lock-in/

[11] M. Nadeski, "Bringing machine learning to embedded systems," Texas Instruments, Tech. Rep., 2019.

[12] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2018.

[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, oct 2016.

[14] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review," *Sensors*, vol. 20, no. 9, 2020.

[15] J. Portilla, G. Mujica, J.-S. Lee, and T. Riesgo, "The extreme edge at the bottom of the internet of things: A review," *IEEE Sensors Journal*, vol. 19, no. 9, pp. 3179–3190, 2019.

[16] J. S. Preden, K. Tammemäe, A. Jantsch, M. Leier, A. Riid, and E. Calis, "The benefits of self-awareness and attention in fog and mist computing," *Computer*, vol. 48, no. 7, pp. 37–45, 2015.

[17] "Ic insights. 2020. mcus expected to make modest comeback after 2020 drop." accessed: 2022-08-9. [Online]. Available: https://www.icinsights.com/news/bulletins/MCUs-Expected-To-Make-Modest-Comeback-After-2020-Drop--/

[18] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020.

[19] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities," *IEEE Circuits Syst. Mag.*, vol. 20, no. 3, pp. 4–18, 2020.

[20] "Arm-nn," accessed: 2022-08-9. [Online]. Available: https://github.com/ARM-software/armnn

[21] S. Kumar, P. Tiwari, and M. Zymbler, "Internet of things is a revolutionary approach for future technology enhancement: a review," *Journal of Big Data*, vol. 6, pp. 1–21, 12 2019.

[22] C. Arcadius Tokognon, B. Gao, G. Y. Tian, and Y. Yan, "Structural health monitoring framework based on internet of things: A survey," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 619–635, 2017.

[23] K. K. Patel, S. M. Patel, and P. G. Scholar, "Internet of things-iot: Definition, characteristics, architecture, enabling technologies, application & future challenges," *International Journal of Engineering Science and Computing*, 2016.

[24] A. Althoubi, R. Alshahrani, and H. Peyravi, "Delay analysis in iot sensor networks," *Sensors*, vol. 21, no. 11, 2021.

[25] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1584–1607, 2019.

[26] S. K. Sharma and X. Wang, "Live data analytics with collaborative edge and cloud processing in wireless iot networks," *IEEE Access*, vol. 5, pp. 4621–4635, 2017.

[27] R. Berta, A. Kobeissi, F. Bellotti, and A. De Gloria, "Atmosphere, an open source measurement-oriented data framework for iot," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 1927–1936, 2021.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[29] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Int. Res.*, vol. 4, no. 1, p. 237–285, may 1996.

[31] J. E. van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Machine Learning*, vol. 109, pp. 373–440, 2 2020.

[32] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[33] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2012.

[34] S. Raschka, *Python Machine Learning*. Packt Publishing - ebooks Account, 2015.

[35] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, rOC Analysis in Pattern Recognition.

[36] Y. Liu, W. Chen, P. Arendt, and H. Huang, "Toward a better understanding of model validation metrics," *Journal of Mechanical Design - Transactions of the ASME*, vol. 133, no. 7, 2011.

[37] "Tinyml as a service and the challenges of machine learning at the edge," accessed: 2022-08-9. [Online]. Available: https://www.ericsson.com/en/blog/2019/12/tinyml-as-a-service

[38] "What is the cloud? how does it fit into the internet of things (iot)?" accessed: 2022-08-9. [Online]. Available: https://www.iotforall.com/what-is-the-cloud#:~:text=The%20cloud%20is%20a%20huge,for%20businesses%20and%20for%20people

[39] V. Rajapakse, I. Karunanayake, and N. Ahmed, "Intelligence at the extreme edge: A survey on reformable tinyml," *CoRR*, vol. abs/2204.00827, 2022.

[40] D. Chen and H. Zhao, "Data security and privacy protection issues in cloud computing," in *Proceedings of the 2012 International Conference on Computer Science and Electronics Engineering - Volume 01*, ser. ICCSEE '12. USA: IEEE Computer Society, 2012, p. 647–651.

[41] P. P. Ray, "A review on tinyml: State-of-the-art and prospects," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1595–1623, 2022.

[42] W. Bao, C. Wu, S. Guleng, J. Zhang, K.-L. A. Yau, and Y. Ji, "Edge computing-based joint client selection and networking scheme for federated learning in vehicular iot," *China Communications*, vol. 18, no. 6, pp. 39–52, 2021.

[43] S. Moore, C. Nugent, S. Zhang, and I. Cleland, "Iot reliability: a review leading to 5 key research directions," *CCF Transactions on Pervasive Computing and Interaction*, Aug. 2020.

[44] D. Fernandes, A. G. Ferreira, R. Abrishambaf, J. Mendes, and J. Cabral, "Survey and taxonomy of transmissions power control mechanisms for wireless body area networks," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1292–1328, 2018.

[45] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey," *Electronics (Switzerland)*, vol. 8, no. 11, 2019.

[46] J. Morales-García, A. Bueno-Crespo, R. Martínez-España, J.-L. Posadas, P. Manzoni, and J. M. Cecilia, "Evaluation of edge computing platforms through tinyml workloads," 2022, unpublished. [Online]. Available: https://assets.researchsquare.com/files/rs-1849666/v1_covered.pdf?c=1658421433

[47] "Tinyml board - syntiant," accessed: 2022-03-29. [Online]. Available: https://www.syntiant.com/tinyml

[48] "Stm32 f0 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series.html

[49] "Stm32 f1 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f1-series.html

[50] "Stm32 f4 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html

[51] "Stm32 l4 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html

[52] "Stm32 f7 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html

[53] "Stm32 h7 series," accessed: 2022-03-29. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html

[54] "Nano 33 ble sense," accessed: 2022-03-29. [Online]. Available: https://docs.arduino.cc/hardware/nano-33-ble-sense

[55] "Sparkfun edge development board - apollo3 blue," accessed: 2022-03-29. [Online]. Available: https://www.sparkfun.com/products/15170

[56] "Openmv cam h7 plus," accessed: 2022-03-29. [Online]. Available: https://openmv.io/products/openmv-cam-h7

[57] "Gap8," accessed: 2022-03-29. [Online]. Available: https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1_9.pdf

[58] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[59] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *ArXiv*, vol. abs/1806.08342, 2018.

[60] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors (Basel, Switzerland)*, vol. 21, 2021.

[61] S. M. Siddegowda, M. Fournarakis, M. Nagel, T. Blankevoort, C. Patel, and A. Khobare, "Neural network quantization with ai model efficiency toolkit (aimet)," *ArXiv*, vol. abs/2201.08442, 2022.

[62] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8604–8612.

[63] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 871–875, 2020.

[64] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016.

[65] D. Pau, M. Lattuada, F. Loro, A. De Vita, and G. Domenico Licciardo, "Comparing industry frameworks with deeply quantized neural networks on microcontrollers," in *2021 IEEE International Conference on Consumer Electronics (ICCE)*, 2021, pp. 1–6.

[66] "Getting Started - Larq." [Online]. Available: https://docs.larq.dev/larq/

[67] T. Bannink, A. Hillier, L. Geiger, T. de Bruin, L. Overweel, J. Neeven, and K. Helwegen, "Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks," in *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, A. Smola, A. Dimakis, and I. Stoica, Eds. mlsys.org, 2021.

[68] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 1135–1143.

[69] "Tensorflow lite for mobile and edge," accessed: 2022-03-29. [Online]. Available: https://www.tensorflow.org/lite/

[70] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *ArXiv*, vol. abs/1503.02531, 2015.

[71] P. Ren, Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, no. 4, may 2021.

[72] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations*, 2017.

[73] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *International Conference on Learning Representations*, 2017.

[74] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[75] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

[76] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021, pp. 2346–2358.

[77] E. Liberis, L. Dudziak, and N. D. Lane, "μNAS: Constrained neural architecture search for microcontrollers," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, ser. EuroMLSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 70–79.

[78] I. Fedorov, R. Adams, M. Mattina, and P. Whatmough, "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[79] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors Journal*, vol. 22, no. 22, pp. 21 362–21 390, November 2022.

[80] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 199–213.

[81] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[82] P. Gerrish, E. Herrmann, L. Tyler, and K. Walsh, "Challenges and constraints in designing implantable medical ics," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 435–444, 2005.

[83] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, 2017.

[84] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabbour, A. Faust, G. C. de Croon *et al.*, "Tiny robot learning: Challenges and directions for machine learning in resource-constrained robots," *arXiv preprint arXiv:2205.05748*, 2022.

[85] B. P. Duisterhof, S. Li, J. Burgu'es, V. J. Reddi, and G. C. de Croon, "Sniffy bug: A fully autonomous swarm of gas-seeking nano quadcopters in cluttered environments," *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9099–9106, 2021.

[86] K. N. McGuire, C. D. Wagter, K. Tuyls, H. J. Kappen, and G. de Croon, "Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment," *Science Robotics*, vol. 4, 2019.

[87] B. Goldberg, N. Doshi, K. Jayaram, J. Zhou, and R. J. Wood, "Inverted and vertical climbing of a quadrupedal microrobot using electroadhesion," *Science Robotics*, vol. 3, 2018.

[88] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proc.s of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[89] "utensor: Tinyml ai inference library," accessed: 2022-08-9. [Online]. Available: https://github.com/uTensor/uTensor

[90] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, USA, 2018, p. 579–594.

[91] "Tinyml - how tvm is taming tiny." [Online]. Available: https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny

[92] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb ram for the internet of things," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, p. 1935–1944.

[93] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "Protonn: Compressed and accurate knn for resource-scarce devices," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 1331–1340.

[94] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 9031–9042.

[95] D. Dennis, D. A. E. Acar, V. Mandikal, V. S. Sadasivan, V. Saligrama, H. V. Simhadri, and P. Jain, "Shallow rnn: Accurate time-series classification on resource constrained devices," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[96] D. Dennis, C. Pabbaraju, H. V. Simhadri, and P. Jain, "Multiple instance learning for efficient sequential data classification on resource-constrained devices," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[97] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, "Rnnpool: Efficient non-linear pooling for ram constrained inference." *CoRR*, vol. abs/2002.11921, 2020.

[98] S. Goyal, A. Raghunathan, M. Jain, H. V. Simhadri, and P. Jain, "DROCC: Deep robust one-class classification," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 3711–3721.

[99] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling kb-sized machine learning models to tiny iot devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 79–95.

[100] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *ArXiv*, vol. abs/1801.06601, 2018.

[101] "Edge impulse," accessed: 2022-08-9. [Online]. Available: https://www.edgeimpulse.com/

[102] "X-cube-ai - ai expansion pack for stm32cubemx - stmicroelectronics." [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[103] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, "Tinyml platforms benchmarking," in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds. Cham: Springer International Publishing, 2022, pp. 139–148.

[104] "Nano edge ai studio," accessed: 2022-08-9. [Online]. Available: https://www.st.com/en/development-tools/nanoedgeaistudio.html

[105] "Eloquentarduino," accessed: 2022-08-9. [Online]. Available: https://github.com/eloquentarduino

[106] "Sklearn porter," accessed: 2022-08-9. [Online]. Available: https://github.com/nok/sklearn-porter

[107] L. T. da Silva, V. M. A. Souza, and G. E. A. P. A. Batista, "Embml tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 1633–1637.

[108] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4403–4417, 2020.

[109] M. Lord, "Tinyml, anomaly detection," Master's thesis, California State University, Northridge, 5 2021.

[110] F. Alongi, N. Ghielmetti, D. Pau, F. Terraneo, and W. Fornaciari, "Tiny neural networks for environmental predictions: An integrated approach with miosix," in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2020, pp. 350–355.

[111] M. Giordano, P. Mayer, and M. Magno, "A battery-free long-range wireless smart camera for face detection," in *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ser. ENSsys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 29–35.

[112] M. Mazumder, C. R. Banbury, J. Meyer, P. Warden, and V. J. Reddi, "Few-shot keyword spotting in any language," in *Interspeech*, 2021.

[113] S. Bian and P. Lukowicz, "Capacitive sensing based on-board hand gesture recognition with tinyml," in *Adjunct Proceedings of the 2021 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2021 ACM International Symposium on Wearable Computers*, ser. UbiComp '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 4–5.

[114] A. J. Paul, P. Mohan, and S. Sehgal, "Rethinking generalization in american sign language prediction for edge devices with extremely low memory footprint," *2020 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pp. 147–152, 2020.

[115] F. de Almeida Florencio, E. D. Moreno, H. Teixeira Macedo, R. J. P. de Britto Salgueiro, F. Barreto do Nascimento, and F. A. Oliveira Santos, "Intrusion detection via mlp neural network using an arduino embedded system," in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2018, pp. 190–195.

[116] F. E. Fernandes Junior, L. G. Nonato, C. M. Ranieri, and J. Ueyama, "Memory-based pruning of deep neural networks for iot devices applied to flood detection," *Sensors*, vol. 21, no. 22, 2021.

[117] E. Liberis and N. D. Lane, "Differentiable network pruning for microcontrollers," *arXiv preprint arXiv:2110.08350*, 2021.

[118] R. Sharma, S. Biookaghazadeh, B. Li, and M. Zhao, "Are existing knowledge transfer techniques effective for deep learning with edge devices?" in *Proceedings - 2018 IEEE International Conference on Edge Computing, EDGE 2018 - Part of the 2018 IEEE World Congress on Services*. Institute of Electrical and Electronics Engineers Inc., sep 2018, pp. 42–49.

[119] A. N. Roshan, B. Gokulapriyan, C. Siddarth, and P. Kokil, "Adaptive traffic control with tinyml," in *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, 2021, pp. 451–455.

[120] T. Szydlo, J. Sendorek, and R. Brzoza-Woch, "Enabling machine learning on resource constrained devices by source code generation of the learned models," in *Computational Science – ICCS 2018: 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2018, p. 682–694.

[121] V. M. Suresh, R. Sidhu, P. Karkare, A. Patil, Z. Lei, and A. Basu, "Powering the iot through embedded machine learning and lora," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 349–354.

[122] R. Qaddoura, A. M. Al-Zoubi, I. Almomani, and H. Faris, "A multi-stage classification approach for iot intrusion detection based on clustering with oversampling," *Applied Sciences*, vol. 11, no. 7, 2021.

[123] K. S. Yaswanth Kumar Alapati, "Combining clustering with classification: A technique to improve classification accuracy," in *International Journal of Computer Science Engineering (IJCSE)*, 2016.

[124] J. Bao, B. Hamdaoui, and W.-K. Wong, "Iot device type identification using hybrid deep learning approach for increased iot security," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, 2020, pp. 565–570.

[125] X. Zhou, W. Liang, K. I.-K. Wang, H. Wang, L. T. Yang, and Q. Jin, "Deep-learning-enhanced human activity recognition for internet of healthcare things," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6429–6438, 2020.

[126] N. Ravi and S. M. Shalinie, "Learning-driven detection and mitigation of ddos attack in iot via sdn-cloud architecture," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3559–3570, 2020.

[127] S. Rathore and J. H. Park, "Semi-supervised learning based distributed attack detection framework for iot," *Applied Soft Computing*, vol. 72, pp. 79–89, 2018.

[128] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (2019, O'reilly)*. O'Reilly Media, 2017.

[129] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon, "A survey on contrastive self-supervised learning," *Technologies*, vol. 9, no. 1, 2021.

[130] Y. Wu, Z. Wang, Y. Shi, and J. Hu, "Enabling on-device cnn training by self-supervised instance filtering and error map pruning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3445–3457, 2020.

[131] A. Saeed, T. Ozcelebi, and J. Lukkien, "Multi-task self-supervised learning for human activity detection," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 3, no. 2, jun 2019.

[132] A. Saeed, F. D. Salim, T. Ozcelebi, and J. Lukkien, "Federated self-supervised learning of multisensor representations for embedded intelligence," *IEEE Internet of Things Journal*, vol. 8, no. 2, pp. 1030–1040, 2021.

[133] A. Gural and B. Murmann, "Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 4454–4472, 2019.

[134] H. Unlu, "Efficient neural network deployment for microcontroller," *ArXiv*, vol. abs/2007.01348, 2020.

[135] E. Liberis and N. D. Lane, "Neural networks on microcontrollers: saving memory at inference via operator reordering," *ArXiv*, vol. abs/1910.05110, 2019.

[136] N. Fasfous, M.-R. Vemparala, A. Frickenstein, L. Frickenstein, M. Badawy, and W. Stechele, "Binarycop: Binary neural network-based covid-19 face-mask wear and positioning predictor on edge devices," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 108–115.

[137] G. Cerutti, R. Andri, L. Cavigelli, E. Farella, M. Magno, and L. Benini, "Sound event detection with binary neural networks on tightly power-constrained iot devices," in *Proc.s ACM/IEEE Int.l Symp. on Low Power Electronics and Design*, ser. ISLPED '20. New York, NY, USA: ACM, 2020, p. 19–24.

[138] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Hybrid Fixed-point/Binary Convolutional Neural Network Accelerator for Real-time Tactile Processing," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. Dubai, United Arab Emirates: IEEE, Nov. 2021, pp. 1–5.

[139] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[140] "Computer, respond to this email." accessed: 2022-03-29. [Online]. Available: https://ai.googleblog.com/2015/11/computer-respond-to-this-email.html

[141] M. M. Grau, R. P. Centelles, and F. Freitag, "On-device training of machine learning models on microcontrollers with a look at federated learning," in *Proceedings of the Conference on Information Technology for Social Good*, ser. GoodIT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 198–203.

[142] K. Kopparapu, E. Lin, J. G. Breslin, and B. Sudharsan, "Tinyfedtl: Federated transfer learning on ubiquitous tiny iot devices," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 79–81.

[143] H. Ren, D. Anicic, and T. A. Runkler, "Tinyol: Tinyml with online-learning on microcontrollers," in *2021 international joint conference on neural networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[144] L. Ravaglia, M. Rusci, D. Nadalini, A. Capotondi, F. Conti, and L. Benini, "A tinyml platform for on-device continual learning with quantized latent replays," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 789–802, 2021.

[145] L. Pellegrini, G. Graffieti, V. Lomonaco, and D. Maltoni, "Latent replay for real-time continual learning," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 10 203–10 209.

[146] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[147] A. Burrello, M. Scherer, M. Zanghieri, F. Conti, and L. Benini, "A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes," in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, 2021, pp. 1–6.

[148] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, sep 2013.

[149] S. Zuboff, *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*, 1st ed. Public Affairs, New York, 2018.

[150] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," 2019.

[151] "Edgeml machine learning algorithms for resource constrained devices," accessed: 2022-03-29. [Online]. Available: https://microsoft.github.io/EdgeML/

[152] Y. Bai, *Practical microcontroller engineering with ARM® technology*. Wiley-IEEE Press, December 2015.

[153] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[154] V. N. Vapnik, "Pattern recognition using generalized portrait method," *Automation and Remote Control*, vol. 24, pp. 774–780, 1963.

[155] G. Shakhnarovich, T. Darrell, and P. Indyk, "Nearest-neighbor methods in learning and vision: Theory and practice," *Neural Information Processing*, 2005.

[156] K. Taunk, S. De, S. Verma, and A. Swetapadma, "A brief review of nearest neighbor algorithm for learning and classification," in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 2019, pp. 1255–1260.

[157] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. FL, USA: Chapman and Hall/CRC: Boca Raton, 1984.

[158] L. Rokach and O. Maimon, *Decision Trees*. Springer, Boston, MA, 5 2005.

[159] "scikit-learn: machine learning in python — scikit-learn 0.24.1 documentation." [Online]. Available: https://scikit-learn.org/stable/

[160] "TensorFlow." [Online]. Available: https://www.tensorflow.org/

[161] "Keras: the python deep learning api." [Online]. Available: https://keras.io/

[162] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, *Scaling Learning Algorithms toward AI*. MIT Press, 2007, pp. 321–359.

[163] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, dec 2015.

[164] "The HDF5® Library & File Format - The HDF Group." [Online]. Available: https://www.hdfgroup.org/solutions/hdf5

[165] "Support vector machines (svms)." [Online]. Available: https://www.svms.org/

[166] "STM32 High Performance Microcontrollers (MCUs) - STMicroelectronics." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

[167] "STM32H7 - Arm Cortex-M7 and Cortex-M4 MCUs (480 MHz) - STMicroelectronics." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html

[168] "STM32L4 - ARM Cortex-M4 ultra-low-power MCUs - STMicroelectronics." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html

[169] "Heart Disease UCI | Kaggle." [Online]. Available: https://www.kaggle.com/ronitf/heart-disease-uci/kernels

[170] L. Boero, M. Cello, M. Marchese, E. Mariconti, T. Naqash, and S. Zappatore, "Statistical fingerprint-based intrusion detection system (SF-IDS)," *International Journal of Communication Systems*, vol. 30, no. 10, p. e3225, jul 2017.

[171] A. Fausto and M. Marchese, "Implementation details to reduce the latency of an SDN Statistical Fingerprint-Based IDS," in *2019 International Symposium on Advanced Electrical and Communication Technologies, ISAECT 2019*. Institute of Electrical and Electronics Engineers Inc., nov 2019.

[172] V. Falbo, T. Apicella, D. Aurioso, L. Danese, F. Bellotti, R. Berta, and A. De Gloria, "Analyzing Machine Learning on Mainstream Microcontrollers," in *Lecture Notes in Electrical Engineering*, vol. 627. Springer, sep 2020, pp. 103–108.

[173] "Benchmark datasets used for classification: comparison of results." [Online]. Available: http://fizyka.umk.pl/kis-old/projects/datasets.html{#}Sonar

[174] A. Parodi, F. Bellotti, R. Berta, and A. De Gloria, "Developing a machine learning library for microcontrollers," in *Lecture Notes in Electrical Engineering*, vol. 550, no. 9783030119720. Springer Verlag, sep 2019, pp. 313–317.

[175] "Traffic, Driving Style and Road Surface Condition | Kaggle." [Online]. Available: https://www.kaggle.com/gloseto/traffic-driving-style-road-surface-condition

[176] "enviroCar - Datasets - the Datahub." [Online]. Available: https://old.datahub.io/dataset/envirocar

[177] R. Massoud, S. Poslad, F. Bellotti, R. Berta, K. Mehran, and A. De Gloria, "A Fuzzy Logic Module to Estimate a Driver's Fuel Consumption for Reality-Enhanced Serious Games," *International Journal of Serious Games*, vol. 5, no. 4, pp. 45–62, dec 2018.

[178] M. Magno, L. Cavigelli, P. Mayer, F. V. Hagen, and L. Benini, "Fanncortexm: An Open Source Toolkit for Deployment of Multi-layer Neural Networks on ARM Cortex-M Family Microcontrollers : Performance Analysis with Stress Detection," in *IEEE 5th World Forum on Internet of Things, WF-IoT 2019 - Conference Proceedings*. Institute of Electrical and Electronics Engineers Inc., apr 2019, pp. 793–798.

[179] "Search for and download air quality data | NSW Dept of Planning, Industry and Environment."

[180] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[181] "6.3. Preprocessing data — scikit-learn 0.24.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html

[182] "Decision Trees: How To Optimize My Decision Making Process? | Experfy." [Online]. Available: https://www.experfy.com/blog/bigdata-cloud/decision-trees-how-to-optimize-my-decision-making-process/

[183] "Aws greengrass machine learning inference - amazon web services." [Online]. Available: https://aws.amazon.com/greengrass/ml/

[184] "sklearn.decomposition.PCA — scikit-learn 0.24.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[185] C. G. Sentelle, G. C. Anagnostopoulos, and M. Georgiopoulos, "A Simple Method for Solving the SVM Regularization Path for Semidefinite Kernels," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 4, pp. 709–722, apr 2016.

[186] D. A. Ross, J. Lim, R.-S. Lin, M.-H. Yang, D. A. Ross, J. Lim, M.-H. Yang, and R.-S. Lin, "Incremental learning for robust visual tracking," *International Journal of Computer Vision 2007 77:1*, vol. 77, pp. 125–141, 8 2007.

[187] "Cisco Annual Internet Report - Cisco." [Online]. Available: https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html

[188] Y. Roh, G. Heo, and S. E. Whang, "A survey on data collection for machine learning: A big data - ai integration perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1328–1347, 2021.

[189] M. E. Celebi and K. Aydin, *Unsupervised learning algorithms*. Springer International Publishing, 1 2016.

[190] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, USA, 2007, p. 1027–1035.

[191] P. Cunningham, M. Cord, and S. J. Delany, *Supervised learning*. Springer Verlag, 2008.

[192] S. P. Chatrati, G. Hossain, A. Goyal, A. Bhan, S. Bhattacharya, D. Gaurav, and S. M. Tiwari, "Smart home health monitoring system for predicting type 2 diabetes and hypertension," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 3, pp. 862–870, 2022.

[193] "Uci machine learning repository: Isolet data set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/isolet

[194] "sklearn.feature_selection.f_classif - scikit-learn 1.1.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html

[195] "sklearn.model_selection.gridsearchcv - scikit-learn 1.1.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[196] C. Perlich, "Learning curves in machine learning," in *Encyclopedia of Machine Learning*, 2010.

[197] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[198] L. Hou, Q. Yao, and J. T.-Y. Kwok, "Loss-aware binarization of deep networks," *ArXiv*, vol. abs/1611.01600, 2017.

[199] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 2148–2156.

[200] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *ArXiv*, vol. abs/1503.02531, 2015.

[201] E. Liberis and N. D. Lane, "Neural networks on microcontrollers: saving memory at inference via operator reordering," *ArXiv*, vol. abs/1910.05110, 2019.

[202] "Cortex-M – Arm Developer." [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m

[203] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks," *Future Internet*, vol. 12, no. 7, p. 113, 2020.

[204] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," in *Insights into Imaging*, vol. 9, no. 4. Springer Verlag, aug 2018, pp. 611–629.

[205] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, pp. 675–678, 2014.

[206] A. Krizhevsky, "Learning multiple layers of features from tiny images," Canadian Institute For Advanced Research, Tech. Rep., 2009.

[207] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current Progress, Research Challenges, and Future Roadmap," *Proceedings - Design Automation Conference*, vol. 2021-December, pp. 1303–1306, Dec. 2021.

[208] R. Mukherjee and R. Dahiya, "Life Cycle Assessment of Energy Generating Flexible Electronic Skin," in *2021 IEEE International Conference on Flexible and Printable Sensors and Systems (FLEPS)*. Manchester, United Kingdom: IEEE, Jun. 2021, pp. 1–4.

[209] R. S. Johansson and J. R. Flanagan, "Coding and use of tactile signals from the fingertips in object manipulation tasks," *Nat Rev Neurosci*, vol. 10, no. 5, pp. 345–359, May 2009.

[210] T. Bhattacharjee, J. M. Rehg, and C. C. Kemp, "Haptic classification and recognition of objects using a tactile sensing forearm," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vilamoura-Algarve, Portugal: IEEE, Oct. 2012, pp. 4090–4097.

[211] M. Kaboli, P. Mittendorfer, V. Hugel, and G. Cheng, "Humanoids learn object properties from robust tactile feature descriptors via multi-modal artificial skin," in *2014 IEEE-RAS International Conference on Humanoid Robots*. Madrid, Spain: IEEE, Nov. 2014, pp. 187–192.

[212] J. Schill, J. Laaksonen, M. Przybylski, V. Kyrki, T. Asfour, and R. Dillmann, "Learning continuous grasp stability for a humanoid robot hand based on tactile sensing," in *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*. Rome, Italy: IEEE, Jun. 2012, pp. 1901–1906.

[213] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "Computational Intelligence Techniques for Tactile Sensing Systems," *Sensors*, vol. 14, no. 6, pp. 10 952–10 976, Jun. 2014.

[214] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Data Oriented Approximate K-Nearest Neighbor Classifier for Touch Modality Recognition," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*. Lausanne, Switzerland: IEEE, Jul. 2019, pp. 241–244.

[215] M. Alameh, A. Ibrahim, M. Valle, and G. Moser, "DCNN for Tactile Sensory Data Classification based on Transfer Learning," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*. Lausanne, Switzerland: IEEE, Jul. 2019, pp. 237–240.

[216] M. Alameh, Y. Abbass, A. Ibrahim, and M. Valle, "Smart Tactile Sensing Systems Based on Embedded CNN Implementations," *Micromachines*, vol. 11, no. 1, p. 103, Jan. 2020.

[217] A. Ibrahim and M. Valle, "Real-Time Embedded Machine Learning for Tensorial Tactile Data Processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 11, pp. 3897–3906, Nov. 2018.

[218] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "An Efficient Selection-Based kNN Architecture for Smart Embedded Hardware Accelerators," *IEEE Open J. Circuits Syst.*, vol. 2, pp. 534–545, 2021.

[219] C. Gianoglio, E. Ragusa, R. Zunino, and M. Valle, "1-D Convolutional Neural Networks for Touch Modalities Classification," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. Dubai, United Arab Emirates: IEEE, Nov. 2021, pp. 1–6.

[220] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "A tensor-based pattern-recognition framework for the interpretation of touch modality in artificial skin systems," *IEEE Sensors Journal*, vol. 14, no. 7, pp. 2216–2225, 2014.

[221] M. Osta, A. Ibrahim, M. Magno, M. Eggimann, A. Pullini, P. Gastaldo, and M. Valle, "An Energy Efficient System for Touch Modality Classification in Electronic Skin Applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. Sapporo, Japan: IEEE, May 2019, pp. 1–4.

[222] M. Alameh, Y. Abbass, A. Ibrahim, G. Moser, and M. Valle, "Touch Modality Classification Using Recurrent Neural Networks," *IEEE Sensors J.*, vol. 21, no. 8, pp. 9983–9993, Apr. 2021.

[223] A. Ibrahim, H. Younes, M. Alameh, and M. Valle, "Near Sensors Computation based on Embedded Machine Learning for Electronic Skin," *Procedia Manufacturing*, vol. 52, pp. 295–300, 2020.

[224] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "A shallow neural network for real-time embedded machine learning for tensorial tactile data processing," *IEEE Trans. Circuits Syst. I*, vol. 68, no. 10, pp. 4232–4244, October 2021.

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

[226] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[227] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[228] Y. Li, Y. Bao, and W. Chen, "Fixed-sign binary neural network: An efficient design of neural network for internet-of-things devices," *IEEE Access*, vol. 8, pp. 164 858–164 863, 2020.

[229] "Standards (using the gnu compiler collection (gcc))." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Standards.html

[230] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, 2020.

[231] A. Bulat, B. Martínez, and G. Tzimiropoulos, "Bats: Binary architecture search," in *ECCV*, 2020.

[232] H. Yonekawa and H. Nakahara, "On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA," *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, pp. 98–105, 2017.

[233] "Arm Cortex-M7 Processor Technical Reference Manual r1p2." [Online]. Available: https://developer.arm.com/documentation/ddi0489/f/memory-system/l1-caches

[234] M. Saleh, A. Ibrahim, F. Menichelli, Y. Mohanna, and M. Valle, "Efficient machine learning algorithm for embedded tactile data processing," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2021-May, 2021.

[235] N. N. Alajlan and D. M. Ibrahim, "Tinyml: Enabling of inference deep learning models on ultra-low-power iot edge devices for ai applications," *Micromachines*, vol. 13, no. 6, p. 851, May 2022.

[236] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder, "Latent weights do not exist: Rethinking binarized neural network optimization," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

[237] H. Kim, J. Park, C.-H. Lee, and J.-J. Kim, "Improving accuracy of binary neural networks using unbalanced activation distribution," *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7858–7867, 2021.

[238] "Prelu layer." [Online]. Available: https://keras.io/api/layers/activation_layers/prelu/

[239] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020.

[240] H. Phan, D. Huynh, Y. He, M. Savvides, and Z. Shen, "Mobinet: A mobile binary network for image classification," in *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2020, pp. 3442–3451.

[241] D. D. Lin and S. S. Talathi, "Overcoming challenges in fixed point training of deep convolutional networks," *ArXiv*, vol. abs/1607.02241, 2016.

[242] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[243] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2021.