MicroIA : Intelligence Artificielle Embarquée pour la Reconnaissance d'Activités Physiques sur Lunettes Intelligentes

Pierre-Emmanuel Novac

Laboratoire d'Électronique, Antennes et Télécommunications (LEAT)

Présentée en vue de l'obtention
du grade de docteur en électronique
d'Université Côte d'Azur
Dirigée par : Alain Pégatoquet
Co-encadrée par : Benoît Miramond
Soutenue le : 9 décembre 2022

Devant le jury, composé de :
Vincent Gripon, IMT Atlantique
Vincent Huard, Dolphin Design
Michele Magno, ETH Zürich
Benoît Miramond, Université Côte d'Azur
Alain Pégatoquet, Université Côte d'Azur
Frédéric Pétrot, Université Grenoble Alpes
Michel Riveill, Université Côte d'Azur
Gilles Sassatelli, CNRS

MicroIA : Intelligence Artificielle Embarquée pour la Reconnaissance d'Activités Physiques sur Lunettes Intelligentes

MicroAI: Embedded Artificial Intelligence for Human Activity Recognition on Smart Glasses

Résumé

Le vieillissement de la population pose de nouveaux défis à notre société en termes de services de soins aux personnes âgées. L'un des aspects est la prévention de la chute, dont la mise en œuvre reste difficile. Dans ce contexte, l'intelligence artificielle peut aider à détecter un risque accru. Pour atteindre cet objectif, il est possible de suivre le comportement du patient pour déceler un changement indicateur d'une dégradation de la mobilité grâce à la reconnaissance d'activités physiques. Notre approche repose sur l'utilisation d'une centrale inertielle embarquée dans des lunettes intelligentes. Ce dispositif est moins invasif que d'autres appareils tels qu'un téléphone mobile ou un appareil dédié, en particulier pour les personnes âgées qui portent souvent déjà des lunettes. D'autre part, les réseaux de neurones profonds ont montré leur capacité à fournir un taux de reconnaissance satisfaisant pour des tâches de classification telles que la reconnaissance d'activités physiques, de mots clés ou de panneaux de signalisation routière. Pour des raisons de confidentialité, de connectivité et de latence, le traitement des données est effectué directement par l'électronique embarquée des lunettes. Cependant, le calcul intensif et la quantité de mémoire requise pour le traitement des réseaux de neurones artificiels sont difficilement compatibles avec les contraintes temps réel, mémoire et énergétique de ces appareils. Dans cette thèse, nous étudions la faisabilité du déploiement de réseaux de neurones artificiels pour la reconnaissance d'activités physiques sur le microcontrôleur embarqué dans les lunettes intelligentes, en mettant en avant le compromis entre les performances de prédiction, la consommation énergétique et l'occupation mémoire. Nous proposons un nouvel outil logiciel appelé MicroAI, publié sous licence libre, dont le but est d'automatiser l'apprentissage, la quantification et le déploiement d'un réseau de neurones artificiel sur microcontrôleur de bout en bout. Nous appliquons une quantification sur 8 et 16 bits en virgue fixe pour plusieurs cas d'usages. Cette quantification et l'exécution à virgule fixe permettent de réduire l'empreinte mémoire, le temps d'exécution et donc l'énergie consommée. Nous fournissons des résultats comparatifs d'empreinte mémoire et d'efficacité énergétique avec d'autre moteurs d'inférence sur différents microcontrôleurs. En outre, nous présentons un nouveau jeu de données nommé UCA-EHAR collectées à partir des lunettes de la société Ellcie Healthy. UCA-EHAR est composé de données brutes étiquetées provenant de l'accéléromètre, du gyroscope et du baromètre des lunettes pour huit classes d'activités physiques réalisées par vingt sujets. Ce jeu de données est utilisé pour entrainer un réseau de neurones artificiels qui est par la suite déployé sur le microcontrôleur des lunettes intelligentes grâce à notre outil MicroAI. Grâce à ce réseau de neurones artificiels, une application de reconnaissance d'activités physiques à l'aide de lunettes intelligentes est réalisée. L'empreinte mémoire, la consommation et l'autonomie résultante d'une telle application fonctionnant sur les lunettes intelligentes Ellcie Healthy sont évaluées. Enfin, nous proposons l'utilisation d'une méthode d'apprentissage non supervisée en ligne afin de spécialiser un réseau de neurones artificiels pour la reconnaissance des activités d'un sujet en particulier. En effet, durant leur durée de vie, les lunettes ne seront portées que par un unique sujet, il convient donc d'essayer d'améliorer le taux de reconnaissance pour ce sujet. L'apprentissage non supervisé ne nécessite pas que les données soient étiquetées, et l'apprentissage en ligne permet une adaptation continue dans l'environnement d'utilisation.

Mots clés : systèmes embarqués ; intelligence artificielle ; apprentissage machine ; réseau de neurones artificiels ; IA embarquée ; apprentissage non supervisé ; quantification ; consommation énergétique ; microcontrôleurs ; reconnaissance d’activité physique ; lunettes intelligentes ; capteurs portables ; e-santé

Abstract

With the growth of the senior population, elderly care becomes an important topic in the society. One aspect of elderly care is fall prevention, which is still a challenging task depending on the subject's health condition. In this context, artificial intelligence can be leveraged to notify about an increased risk. To achieve this goal, a solution consists in monitoring the subject's behaviour to detect some changes that could indicate a degradation of their mobility. Human activity recognition (HAR) can be used for that purpose. Our approach is based on an inertial measurement unit (IMU) embedded in smart glasses. Smart glasses are less invasive than some other devices such as dedicated IMU devices or even smartphones, especially for elderly for whom wearing glasses is common. On the other hand, deep neural networks have shown their capability to provide good recognition accuracy for human activity recognition, among other classification tasks such as keyword spotting or traffic sign recognition. However, embedding deep neural networks onto low power devices remains a challenging task. Real-time, memory and power constraints do not cope well with the heavy computation and memory requirements of these algorithms. Therefore, in this thesis, we study the feasibility of deploying neural networks on the smart glasses' microcontroller for human activity recognition purposes, while optimizing the compromise between prediction performance, power consumption and memory. To do so, we propose a new open-source software framework, called MicroAI, for end-to-end deep neural network training, quantization and deployment. We provide some comparative results using different microcontrollers and alternative inference engines. Results are compared in terms of memory footprint and energy efficiency for various artificial neural networks with different accuracies. We propose to apply 8-bit and 16-bit quantization methods on multiple use cases in order to perform the computation with fixed-point numbers. Fixed-point computation leads to a reduction of the memory footprint, the execution time and therefore the energy consumption. It is also then possible to use a microcontroller without a hardware floating-point unit. Furthermore, we present a new dataset called UCA-EHAR with data collected from Ellcie Healthy's smart glasses. Our dataset provides labelled raw data collected from an accelerometer, a gyroscope and a barometer for eight classes of activity performed by twenty subjects. Therefore, the artificial neural network performing the classification task is executed on the smart glasses' microcontroller using our MicroAI framework. This work is used as a foundation for a real-world application of human activity recognition using smart glasses. Memory footprint, power consumption and battery lifetime are analyzed for this application running on the Ellcie Healthy smart glasses. Finally, we propose an unsupervised online learning method in order to specialize a general model onto a specific subject in the context of human activity recognition. Indeed, during their lifetime, the smart glasses are only worn by a single subject. Thus, our objective is to improve the accuracy for this subject. Unsupervised learning does not make use of labels for the training phase and online learning allows continuous adaptation in the environment.

Keywords: embedded systems; artificial intelligence; machine learning; artificial neural network; embedded AI; unsupervised learning; quantization; power consumption; microcontrollers; human activity recognition; smart glasses; wearable sensing; eHealth

Contents

List of Figures

List of Tables

List of Publications

Journal

[1] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoît Miramond, and Vincent Gripon. “Quantization and Deployment of Deep Neural Networks on Microcontrollers.” In: *Sensors* 21.9 (2021). DOI: 10.3390/s21092984.

[2] Pierre-Emmanuel Novac, Alain Pegatoquet, Benoît Miramond, and Christophe Caquineau. “UCA-EHAR: A Dataset for Human Activity Recognition with Embedded AI on Smart Glasses.” In: *Applied Sciences* 12.8 (2022). DOI: 10.3390/app12083849.

Conference

[3] Pierre-Emmanuel Novac, Adrien Russo, Benoît Miramond, Alain Pegatoquet, François Verdier, and Andrea Castegnetti. “Toward unsupervised Human Activity Recognition on Microcontroller Units.” In: *Proceedings of the 23rd Euromicro Conference on Digital System Design (DSD 2020)*. 2020, pp. 542–550. DOI: 10.1109/DSD51259.2020.00090.

[4] Pierre-Emmanuel Novac, Andres Upegui, Diego Barrientos, Claudio Sousa, Laurent Rodriguez, and Benoît Miramond. “Dynamic Structural and Computational Resource Allocation for Self-Organizing Architectures.” In: *Proceedings of the 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2020)*. 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294983.

[5] Edgar Lemaire, Loïc Cordone, Andrea Castagnetti, Pierre-Emmanuel Novac, Jonathan Courtois, and Benoît Miramond. “An Analytical Estimation of Spiking Neural Networks Energy Efficiency.” In: *29th International Conference on Neural Information Processing*. ICONIP 2022. Accepted. 2022.

Dataset

[6] Pierre-Emmanuel Novac, Alain Pegatoquet, Benoît Miramond, and Christophe Caquineau. *UCA-EHAR, a dataset for human activity recognition using smart glasses*. 10.5281/zenodo.5659336. Version 1.0. 2021.

Software

[7] Pierre-Emmanuel Novac, Alain Pegatoquet, and Benoît Miramond. *MicroAI, a software framework for end-to-end deep neural networks training, quantization and deployment onto embedded devices*. 10.5281/zenodo.5507397. Version 1.0. 2021.

[8] Pierre-Emmanuel Novac, Alain Pegatoquet, and Benoît Miramond. “Déclaration d’Invention DI-14851-01 du logiciel MicroAI.” DI-14851-01 (France). Apr. 2021. URL: https://hal.archives-ouvertes.fr/hal-03595177.

Chapter 1

Introduction

Contents

With the growth of the senior population, elderly care becomes an important topic in the society. Indeed, Figure 1.1 shows the trend of an increase of people aged 60 and older since the 1970s, while the share of people aged 59 and younger is decreasing. One aspect of elderly care is fall prevention, which is still a challenging task according to the subject's health condition. In this context, artificial intelligence can be leveraged to notify about an increased risk of fall. To do so, a solution consists in monitoring the subject's behaviour to detect some changes that could indicate a degradation of their mobility. Human activity recognition (HAR) can be used for that purpose. Our approach is based on an inertial measurement unit (IMU) embedded in smart glasses.

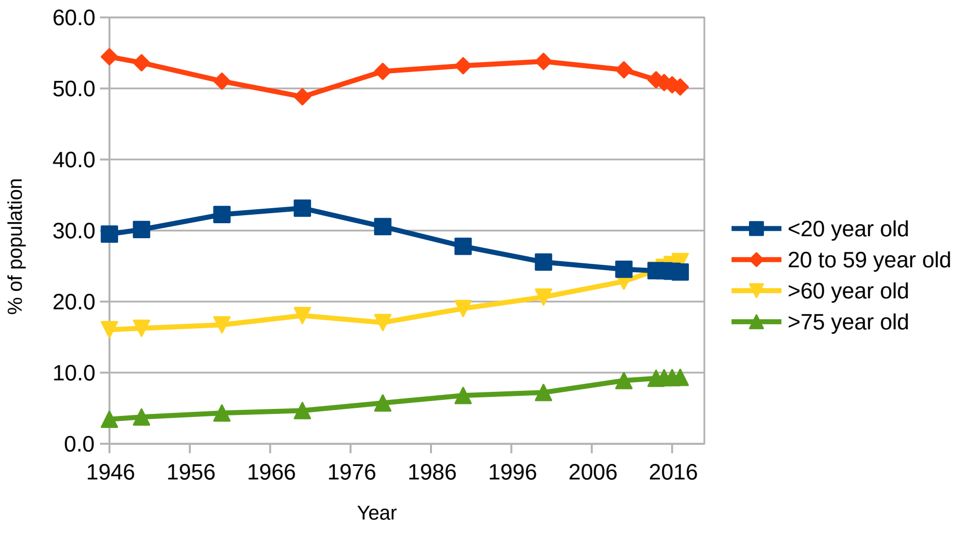


Figure 1.1: Population per groups of age in France from 1949 to 2020[1].

1.1 Artificial Intelligence with Neural Networks

Artificial intelligence is one of the main area of interest for the evolution of the society in the 2020s, whose goal is to emulate cognitive functions using machines. Machine learning is a field of artificial intelligence about the methods and algorithms associated with the learning process of an artificially intelligent system. More specifically, the goal is to solve a task or a set of tasks by learning a transformation of the input data to the output data. Furthermore, it is generally expected from the machine learning algorithm to be able to generalize the knowledge gained from a set of training data to a set of previously unseen data. This allows solving various tasks such as classification and clustering among others.

A machine learning problem can be solved by various type of algorithms such as decision trees or support-vector machines. However, since the 2010s, artificial neural networks have proved to be much more efficient to solve complex tasks in computer vision and natural language processing. Artificial neural networks are inspired from the biological brain, providing a computational model around the neuron. An artificial neuron processes multiple inputs through a transfer function and produces an output.

More specifically, deep learning where an artificial neural network is built with multiple layers of neurons has shown to be a versatile and powerful method to solve these complex tasks. Deep neural networks are widely used to solve a range of problems, including classification. Deep neural networks can classify all sorts of data such as audio, images or accelerometer samples for tasks such as speech recognition, object recognition or human activity recognition. Deep neural networks are often trained in a supervised manner, especially for classification purposes.

Supervised learning requires providing labels for the input data, as opposed to unsupervised learning which processes unlabelled data.

Both supervised learning and unsupervised learning are typically performed offline, with direct access to the entire dataset, or at least a significant subset of it at a time. Online learning, in contrast, processes the input vectors as they come sequentially, with the assumption that previously seen vectors are not all permanently stored in memory and therefore cannot be processed again. Online learning can enable an artificial neural network to adapt to a slightly different distribution of the input data, without having to store all the collected data. Combined with unsupervised learning, it does not require human intervention. Both of these characteristics in turn enable on-target lifelong learning directly on the embedded system.

A well-known downside of artificial neural networks is their high energy consumption requirement. In particular, the training phase is usually based on a large amount of data processed by costly algorithms. Although the inference phase requires less processing power, it is still a costly process, especially for embedded systems. Therefore, high-performance systems often perform such a computation in the cloud [2].

However, cloud computing requires transmitting the collected data to a network server to process it and fetch the result, thus requiring permanent connectivity, causing privacy concerns as well as non-deterministic latency. As an alternative to cloud computing, artificial neural network computation can be done at the edge on the device itself. By doing so, data do not need to be sent by the device to the cloud anymore. However, running artificial neural networks on resource-constrained devices is still a challenging task [3, 4, 5].

1.2 Embedded Systems

Embedded systems are indeed characterized by tight constraints regarding memory, latency, power consumption, cost and physical dimensions. Moreover, reliability and safety can also be a concern for embedded systems. However, we do not take these aspects into account in this work.

Such embedded systems can be architectured around various platforms containing embedded CPUs (Central Processing Units), GPUs (Graphics Processing Units) or FPGAs (Field Programmable Gate Arrays). However these components are often costly and do not always target ultra-low power consumption.

Instead, MCUs (MicroController Units) are often the component of choice for low-power, low-cost applications. Microcontrollers embed a processor core(s), memory and peripheral interfaces in a single integrated circuit. With microcontrollers, the focus is not on the computing performance, but rather on the cost and embeddability. As such, these devices only contain a very low amount of memory and are much slower than a personal computer. Both of these contraints are limitations to implementing artificial intelligence algorithms in embedded devices. Furthermore, some of these devices are powered by batteries. In this case, the energy consumption must be kept as low as possible.

Nonetheless, microcontrollers are found in many consumer and industrial electronics products. The demand continues growing and the global market size is expected to double in less than a decade, reaching USD 58.20B by 2030 after being valued USD 25.75B in 2021[6]. Therefore, with both the growth of the microcontroller market and the growth of artificial intelligence use cases, combining the two became a hot topic in the early 2020s. Wearables are one of the product category designed around microcontrollers that could benefit from being augmented with artificial intelligence.

1.3 Smart Glasses and Wearable Devices

Wearable devices are a category of embedded devices designed to be worn by the user all day long, as an accessory or even a cloth. In consequence, their physical shape and weight are very constrained, thus limiting the size of the batteries that can be used. Miniaturization as well as energy efficiency are key to ensure good wearability and autonomy of the device.

Wearable devices integrated into clothes are not yet popular. However accessories such as smart watches have become common in the 2010s. Smart glasses are another type of wearable accessory where the frame is equipped with electronics. Smart glasses are less invasive than some other devices such as dedicated sensor devices or even smartphones, especially for elderly for whom wearing glasses is common. The smart glasses can contain various sensors such as an inertial measurement unit. They can also embed signaling devices such as a buzzer and LEDs to report alerts to the user. As an example, Ellcie Healthy's smart connected glasses are shown in Figure 1.2.

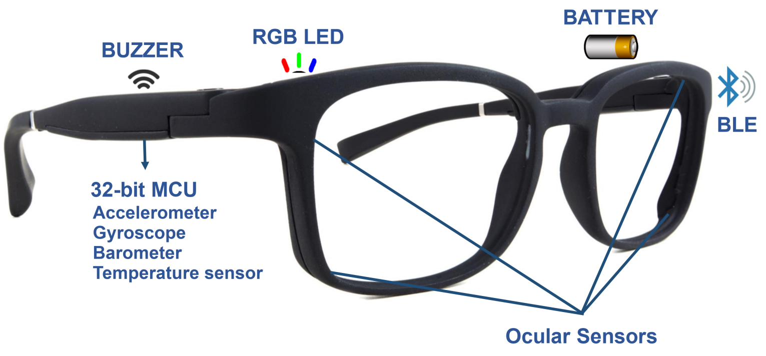


Figure 1.2: Ellcie Healthy Smart Glasses.

Wearable devices have many applications related to entertainment or health, as well as having a fashion aspect to them. In terms of health applications, they can typically provide information related to physical activity such as steps walked and heart rate. The inertial measurement unit found in most of these devices also enables more advanced applications such as human activity recognition.

1.4 Human Activity Recognition

Human activity recognition is a classification problem trying to predict the activity performed by a user among a set of known activities. Such activities can be activities of daily living, for example walking, sitting or various self-care activities. Human activity recognition can then be used to monitor a subject and their health condition. Indeed, statistics on the activities performed throughout the day can, for example, indicate if a subject lacks physical activities. Furthermore, an analysis over a longer period of time could help to detect a deterioration of the subject's mobility.

Human activity recognition can be performed from various modalities. It is possible to make use of computer vision on data coming from cameras placed in the environment to recognize the subject movements. However, this requires setting up the environment prior to performing the recognition as well as a large amount of processing power.

Alternatively, body-worn sensors such as inertial measurement units can be used. Accelerometers and gyroscopes are commonly embedded in smartphones, but they can also be found in smart watches and smart glasses.

1.5 Goals and Challenges

The main goal of this thesis is to propose a method to integrate artificial intelligence into embedded systems. We focus on deploying artificial neural networks on microcontrollers for inference, targeting the specific use-case of performing human activity recognition on smart glasses. Existing works (presented in Chapter 2) have shown that running artificial neural networks on microcontroller is possible. However, it is still a challenging task due to tight embedded constraints being a hindrance to the deployment of large neural networks. The trade-off between predicition performance, latency, power consumption and memory must therefore be optimized.

Our objective is also to propose a solution to perform unsupervised on-device fine-tuning. On-device learning is a step ahead on-device inference, and as of the early 2020s still has no universal solution. The training of a deep neural network is indeed much more demanding in terms of processing power and memory than the inference. Embedded fine-tuning can be considered as a practicable solution since the neural network is not trained from scratch, and fine-tuning can be applied to only a small part of the neural network. However, we add another challenge which is unsupervised learning: the fine-tuning process should be able to work without labels. So far, there is no universal solution to unsupervised learning either and existing methods are known to perform worse than traditional supervised learning.

1.6 Contributions

In this thesis, we propose three main contributions.

- Quantization and deployment of deep neural networks on microcontrollers:
  - the development of an open-source end-to-end software framework for deep neural network training, quantization, and deployment on microcontrollers,
  - an evaluation of deep neural networks quantization for three applications (human activity recognition, keyword spotting, traffic-sign recognition),
  - an evaluation of the embedded execution of these neural networks on microcontrollers with memory footprint, inference time, and energy consumption metrics.
- Human activity recognition on smart glasses:
  - the publication under an open-access policy of a dataset for human activity recognition with smart glasses and under the authorization of Université Côte d'Azur Ethics Committee,
  - a prototype of live human activity recognition with smart glasses,
  - an analysis of the memory footprint, energy consumption and autonomy when performing live human activity recognition on the smart glasses.
- Semi-supervised learning and unsupervised fine-tuning:
  - an evaluation of semi-supervised learning on human activity recognition datasets,
  - an unsupervised fine-tuning method,
  - an evaluation of subject-by-subject fine-tuning applied to human activity recognition and keyword spotting datasets,
  - an evaluation of self-organizing map quantization for human activity recognition.
  - the implementation of on-device unsupervised learning on a microcontroller,
  - an analysis of the memory footprint and inference time when performing on-device unsupervised learning.

1.7 Outline

In Chapter 2, an overview of the existing works regarding embedded artificial intelligence is provided.

Then, in Chapter 3, we propose a new open-source software framework, called MicroAI, for end-to-end deep neural network training, quantization and deployment. We provide some comparative results using different microcontrollers and inference engines. Results are compared in terms of memory footprint, latency and energy efficiency. Quantization of the artificial neural network is performed to convert the real numbers to a fixed-point representation. This quantization helps to lower the memory usage and to perform faster computation. The effect of quantization on the accuracy is evaluated from a memory efficiency perspective.

In Chapter 4, we present a new dataset, called UCA-EHAR, with data collected from Ellcie Healthy's smart glasses. This dataset provides raw data collected from an accelerometer, a gyroscope and a barometer for 8 classes of activity performed by 20 subjects. For privacy, connectivity and latency reasons, all the data processing related to human activity recognition is performed directly on the smart glasses. Therefore, the machine learning algorithm performing the classification task is executed on the smart glasses' microcontroller using our MicroAI framework. This work is used as a foundation for a real-world application of human activity recognition using smart glasses in the context of elderly care.

In Chapter 5 we propose an unsupervised online fine-tuning approach to specialize a general model onto a specific subject in the context of human activity recognition. Unsupervised learning does not make use of labels during the training phase and online learning allows continuous adaptation in the environment.

Finally, Chapter 6 describes the implementation of the unsupervised online fine-tuning approach on a microcontroller, making it an on-device learning method. The cost of such a method is evaluated in terms of memory and latency on a microcontroller.

Chapter 7 concludes this work and proposes new avenues to explore in order to improve on the power consumption and the unsupervised on-device fine-tuning method.

Chapter 2

State of the Art

Contents

2.1 Introduction

In this chapter, we present an overview of the main topics related to this thesis. First, artificial neural networks are presented in Section 2.2 as the subfield of artificial intelligence we focus on to perform various recognition tasks. Then, embedded systems are defined in Section 2.3 with some examples, and the field of study is narrowed down to low-power microcontrollers. However, the embedded constraints of low-power devices and the resource-intensive computation of artificial neural network are often seen as incompatible. Therefore, in Section 2.4, we present several existing methods and tools to enable the deployment of artificial neural networks on low-power devices. In Section 2.5, smart glasses devices are described, and some human activity recognition datasets and applications are presented. Lastly, in Section 2.6, we introduce the concept of unsupervised on-device fine-tuning, and take a closer look at unsupervised learning methods, online and continual learning, and on-device learning. Section 2.7 summarizes and explains the positioning of our work.

2.2 Artificial Neural Networks

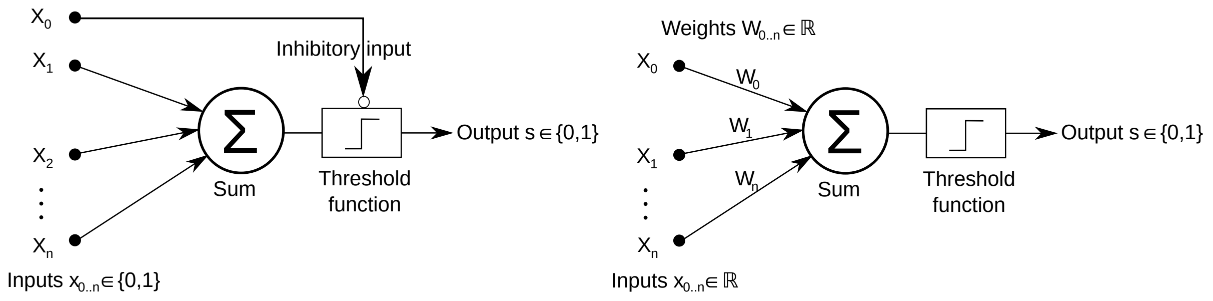
Artificial intelligence as a scientific field has already been studied for many decades, being put into practice early into the 1950s thanks to the development of programmable digital computers [7].

In 1943, McCulloch and Pitts proposed a computational model for an artificial neuron used to build a neural network, taking inspiration from the brain [8].

Similarly to a biological neuron, the artificial neuron is modeled as a unit that takes several inputs through its dendrites, aggregate the inputs in its body through a given transfer function, and produces an output through its axon. Artificial neurons are often bio-inspired rather than biomimetic, meaning that some of their properties are inspired by the biological neuron. However, they do not strive to behave identically. The computational models are a simplified representation of what happens in the brain, for multiple reason. As we still do not fully understand how the brain works, we are not able to reproduce it artificially. Furthermore, the complexity of the biological processes would require a lot of processing power to simulate. In any case, it is not necessary to reproduce all the properties of the biology to obtain the results sought for machine learning.

In the neuron modeled by McCulloch and Pitts, the inputs and the output are assumed to be binary, while the transfer function is a treshold over the sum of the inputs (Figure 2.1a). An input can be defined as inhibitory in order to prevent the neuron from activating when this input is present.

The perceptron described by Rosenblatt in 1958[9] is more flexible in that it allows real-valued inputs and introduces real-valued weights to apply on the input (Figure 2.1b). A single perceptron can act as a binary linear classifier over any set of inputs. However, instead of manually choosing the threshold and the behaviour of the inputs, the perceptron can have its weights learnt by an iterative algorithm in a supervised manner.


X_0
X_1
X_2
X_n
Inhibitory input
Sum
Threshold function
Output $s \in \{0,1\}$
Inputs $x_{0..n} \in \{0,1\}$
Weights $W_{0..n} \in \mathbb{R}$
W_0
W_1
W_n
Inputs $x_{0..n} \in \mathbb{R}$

(a) McCulloch and Pitts neuron. (b) Rosenblatt perceptron.

Figure 2.1: Early neuron models.

However, the perceptron itself can only model a linear function. This can be overcome by stacking multiple perceptron with an intermediate non-linear activation function. Instead of mapping the inputs directly to the output neurons, additional hidden layers of neurons are introduced inbetween the inputs and the output neurons. It is then possible to model any non-linear function given enough neurons. This gives birth to the multi-layer perceptron, which is a kind of deep neural network.

Training a multi-layer perceptron is significantly more difficult than simply adjusting the coefficients of a linear equation. One solution to this problem is backpropagation [10]. The backpropagation algorithm enables an efficient computation of the gradient of the cost function, iterating backward through the layers of the deep neural network. Computing this gradient is useful in applying gradient descent or derived optimization algorithms.

Backpropagation was then applied to convolutional neural networks. Convolutional neural networks greatly reduce the amount of parameters and turn out to be more efficient than multi-layer perceptrons to extract meaningful features of the input for the classification stage. Such a method was applied to handwritten digits recognition by LeCun in 1989 [11].

At the core of backpropagation is a reverse automatic differentiation which can be computed by a machine. This is the main component of modern deep learning software frameworks, such as PyTorch [12], TensorFlow [13] and JAX [14] among others.

However, performing gradient descent to train deep neural networks is costly and in general requires a large amount of data. For this reason, deep neural network training is typically performed on high-performance computing infrastructures for large models, and at least on a high-end computer for smaller ones. For the past decade, deep neural networks have been trained using graphics processing units. Notably, the deep convolutional neural network winning the ImageNet [15] image classification contest in 2012 participated in launching the modern era of machine learning with results considerably better than previous methods [16]. The authors explain that the size of the network is limited by the amount of video memory and the time required for the training process. They suggest that the results could be improved simply by using higher performance hardware to train bigger networks, which was not available at that time. The authors were already using two of the near top-of-the-line graphics processing units with a training time of five to six days for a model of 60 million parameters.

The following years will see larger and larger neural networks solving problems with increased complexity as illustrated in Figure 2.2. One popular example is GPT-3 released in 2020. This model makes use of 175 billion parameters to solve advanced natural language processing and text generation tasks [17]. GPT-3 has been estimated to require around 1.287 GWh of energy for training, and on the order of 400 Wh to generate 100 pages of text. These figures are far beyond what can be expected in a low-power embedded system. Tasks of such complexity are out of reach for constrained devices for now.

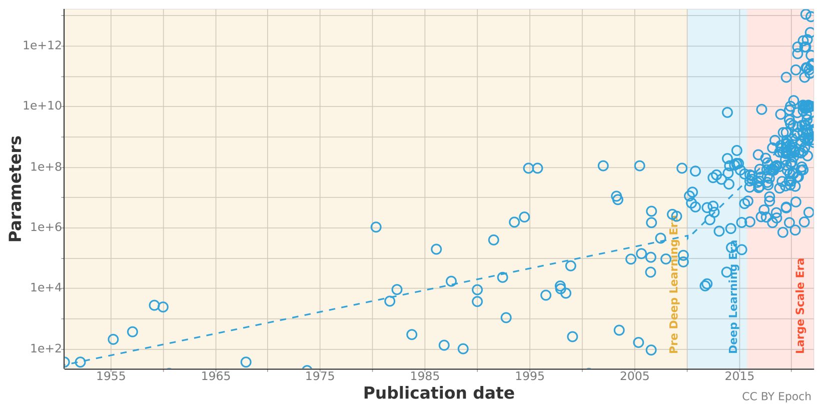


Figure 2.2: Parameter trend in machine learning[18].

In the meantime, a vast amount of deep neural network models and applications have surfaced, varying in complexity, memory and energy requirements [19, 20]. While there is a major focus on training larger and larger models to solve more and more complex tasks of computer vision or natural language processing, new deep neural network architectures as well as training methods can also benefit smaller models for less complex tasks. Therefore, while complex tasks cannot be tackled by embedded systems with strong constraints yet, smaller deep neural networks can now be used for less complex tasks while satisfying memory and power constraints.

2.3 Embedded Systems

Embedded systems are at the heart of many applications such as industrial control, home appliances, automotive or aerospace. Embedded systems are tailored to a specific use-case, and from this use-case arise various constraints. Many situations require an autonomous device without a permanent source of power as well as a small form factor. Mobile phones, wearables, remote sensors and satellites are examples of such devices for which the available energy is heavily constrained. As a result, the power consumption must be reduced as much as possible. This limits both the available processing power and memory. Consumer electronics also bring the cost of the device into the equation.

2.3.1 Single-Board Computers and System on Chips

High-performance embedded systems such as the very popular Raspberry Pi or the more machine learning-focused Nvidia Jetson lineup (Figure 2.3) come with an amount of RAM in the same order of magnitude as a regular personal computer, having at least several gigabytes for the recent models. Furthermore, while slightly less powerful than a personal computer, their central processing unit can still run a desktop operating system and most day-to-day applications. These systems mostly rely on system-on-chips architectured around general purpose ARM Cortex-A cores. An embedded graphics processing unit is also present to accelerate some 3D or compute tasks. As they come in a compact form-factor and embed most of the required components for a computer to run, except peripherals for human-computer interaction, these systems are often described as single-board computers.

Figure 2.3: Nvidia Jetson Nano 2 Development Kit[21].

However, all these characteristics also make them not suitable for very-low power purposes. Indeed, in order to provide this level of performance, these systems need to draw several watts of power [22]. Moreover, these systems introduce a non-negligible cost to the product they are integrated in. Thefore, single-board computers are not suitable for embedded systems found in wearables, wireless sensor networks, satellites, small consumer electronic devices and other type of devices. Instead, much less powerful systems such as microcontrollers must be used.

2.3.2 Microcontrollers

Many families of microcontroller exist. At the very low-end, we can find cost-effective solutions such as the Microchip PIC10 family with only a few hundred bytes of ROM and few dozen bytes of RAM running at up to a few megahertz to use for a single simple task [23]. On the contrary, there are also many high-performance solutions offered by various vendors. The STMicroelectronics STM32H7 family goes up to a megabyte of RAM and two megabytes of ROM, and the core runs at several hundred megahertz [24]. The Ambiq Apollo3 is a slightly less powerful alternative, running at 48 MHz with 384 KiB of RAM and 1 MiB of ROM, but with a focus on very low power consumption. This integrated circuit also has a small form factor and can be seen mounted on a SparkFun Edge development board in Figure 2.4 (green box).

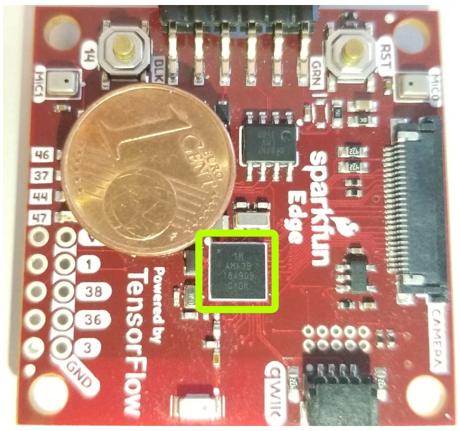


Figure 2.4: SparkFun Edge board with an Ambiq Apollo3 microcontroller.

The memory is many times smaller but the power consumption is also much lower compared to single-board computers. Power consumption can range from a few dozen microwatts to a few hundred milliwatts when processing instructions, depending on the part and the operating conditions, especially the chosen core frequency. Using sleep modes, power consumption can be significantly reduced, under a few microwatts in some instances. Furthermore, thanks to their design being much simpler than high-performance system-on-chips, entering and exiting sleep modes is generally fast and not troublesome.

The significantly lower performance figures and the lack of a memory-management unit means that microcontrollers cannot run desktop operating systems such as Linux. Instead, they can be programmed in a bare-metal fashion: the instructions written by the programmer are direclty directly on the core without intermediate layer, except for an optional but minimal hardware-abstraction layer. Alternatively, a real-time operating system such as Zephyr[25] can be used. The main purpose of a real-time operating system is to provide services for multi-tasking and concurrency management. However, a real-time operating system is still not designed to run high-level applications designed for desktop or servers. Therefore, deep learning frameworks such as TensorFlow or PyTorch cannot be used directly on these platforms.

Performance of the microcontroller for specific tasks can be improved by implenting tiny accelerators inside the core. These accelerators can be called using dedicated instructions in the regular flow of the program. Therefore, communication and synchronization with an off-core or off-chip accelerator are not required. However, extending the instrution set architecture is not possible with an existing product. It rather requires designing a new integrated circuit or implementing the instruction set architecture onto an FPGA.

Most high-performance microcontroller series rely on Cortex-M cores designed and sold by ARM. ARM only recently provided its customers with the ability to extend the Armv8-M instruction set for select Cortex-M cores through its Arm Custom Instructions program [26]. As an alternative, the RISC-V instruction set architecture[27] has been gaining in popularity since the late 2010s. Versatile but simple, RISC-V can be used to design multi-core CPUs for high-performance computing down to ultra-low-power embedded microcontrollers. Being fully open, it allows anyone to implement their own core based on the specifications, and extend it by adding more features as needed. This is the approach chosen by [28] in order to accelerate the processing of deep neural networks.

As the demand for such microcontrollers grows, we could see off-the-shelf products with a specialized instruction set extension become available. In the meantime, dedicated accelerators can be used when the main processor cannot meet the requirements of the application in terms execution time.

2.3.3 FPGAs and ASICs

When the processing power of the main processor is not sufficient, embedded devices can also rely on an external integrated circuit to speed up the computation, and possibly improve the energy efficiency. In such a situation, a custom hardware design can be deployed onto an FPGA (Field-Programmable Gate Array). An FPGA is a type of integrated circuit that can be configured to perform a specific function after its manufacturing. Rather than a microprocessor that would run a software compiled to a specific instruction set and decoded by the core, an FPGA is configured as a set of logic blocks at a much lower level. Logic blocks can be a set of logic gates, flip-flops or memory, enabling the implementation of most digital designs without having to manufacture a dedicated integrated circuit. A hardware architecture deployed onto an FPGA loses in flexibility and capabilities compared to a general-purpose processor programmed with software. However, it can perform a specific task much more efficiently. That said, the cost of FPGAs can be prohibitive in a consumer device. Moreover, their static power consumption is often a drawback in the overall energy efficiency for very low-power devices.

Instead of using an FPGA, it is also possible to use an ASIC (Application-Specific Integrated Circuit). With an ASIC, the design is set in stone since the layout of transistors and interconnect are etched into the silicon at the fab. An ASIC provides the best power efficiency but it also provides the least flexibility. Furthermore, developping and manufacturing an ASIC is tremendously costly especially with the more advanced node processes. The development of an ASIC can be justified for low-cost products only with high volume.

ASICs have already been developped as accelerators for deep neural network applications. The TPU (Tensor Processing Unit) from Google has been initally designed as a high-performance integrated circuit to improve on the energy efficiency of their datacenter when handling deep neural network workloads [29]. Initially put into production in 2015 for internal use, TPUs have been available through Google's cloud service since 2018, with a continuous improvement of their design for better performance and energy efficiency, up to the current TPUv4 [30]. On top of the inference, some of these TPUs were also able to perform training of deep neural networks. However, these accelerators were not initially designed for embedded use and are not available for sale to the general public. Instead, Google released the Edge TPU [31] for use with embedded systems. Power consumption has been heavily reduced compared to the datacenter version. While the inference time can be greatly reduced compared to running on a CPU, the peak power consumption of 2 watts may still not be suitable for very low-power devices. The cost of the Edge TPU module may also not fit a consumer device's price.

Other high-performance accelerators exist such as Habana Lab's Gaudi series [32]. However, most of these accelerators cannot compete in the sub-100mW range [33].

Accelerators can also be integrated into existing system-on-chips like the Neural Engine from

Apple found in all of their new offerings, or the neural processing units of the Samsung Exynos system-on-chips. Once again, these devices are not part of the very low-power and low-cost class of embedded devices. While not a very low-cost solution, the MAX7800 microcontroller from Maxim Integrated embeds a convolutional neural network accelerator alongside a low-power Cortex-M4 core.

2.4 Embedded Artificial Neural Networks

As previously explained, running deep neural networks on embedded devices is a challenging task due to memory and energy constraints as well as unavailability of the popular software frameworks such as TensorFlow and PyTorch. As a result, deep neural networks must be optimized for a reduced memory footprint.

Additionally, a software specifically designed for microcontrollers must be designed to process deep neural networks. This software may not possess all the features provided by popular deep learning frameworks, but its footprint should also be significantly reduced. This software would first focus on implementing the inference phase, while part of the training could also be brought in later on.

2.4.1 Deep Neural Network Compression

In order for deep neural networks to be deployed on embedded systems with hard constraints on memory, the memory usage must be reduced as much as possible while keeping the impact on the prediction performance as low as possible. Various techniques can be used to reach a memory footprint that fits both the target and the application, but there is always a trade-off between memory footprint and prediction performance. Therefore, it is important to consider the memory efficiency by comparing prediction performance to memory footprint. Apart from memory, compression techniques can also impact energy efficiency, either by reducing or by increasing the energy required during inference.

2.4.1.1 Neural Network Architectures

The most straightforward way to fit a given memory limit and power budget with a deep neural network is to choose a small enough neural network that still provides good results on the considered task. Deep neural network architectures are often popularized by achieving the best results for complex computer vision or natural language processing tasks. However, these deep neural networks are very large, with up to millions or even billions of parameters occupying megabytes to gigabytes of memory, which is not available in small embedded devices. VGG-11[34], ResNet-18[35] and ViT-Base[36], to only name a few, have 132.9 millions, 11.7 millions and 86.6 millions parameters respectively when trained for ImageNet-1k, many times larger than what a microcontroller can store.

Low-cost neural network architectures try to solve these problems with a reduced number of parameters, often at the cost of a lower accuracy. This is notably the case of the SqueezeNet [37], MobileNet [38], ShuffleNet[39], and EfficientNet[40] networks. These neural networks often exhibit a lower accuracy, but the number of parameters as well as the number of operations are many times lower than with neural networks providing state-of-the-art accuracy. Still, these neural networks require several hundred kilobytes to a few megabytes of memory, so they are not suitable for microcontrollers. They are rather designed for higher-performance devices such as mobile phones. These networks are often tailored to solve computer vision problems such as ImageNet classification, and are therefore not well suited for general use cases or simpler problems.

Finding a good neural network architecture for a given problem is difficult, and remains an open issue even if hardware constraints are not taken into account. Indeed, the combinatorial explosion of hyperparameters makes the exploration of neural network architectures very expensive. In practice, many applications use a simple generic neural network architecture such as ResNet [35] and optimize it. Optimization can start with scaling down the deep neural network. by making it narrower and shallower as needed. Making it shallower by reducing the number of layers or narrower by reducing the number of neurons can decrease the number of parameters and operations. However, at one point, the drop in accuracy may become an obstacle to solving the given problem. Additionally, reducing the size of the input data and the activations by downsampling using pooling layers decreases the number of operations. Such scaling strategies are studied in [41] to improve ResNet's accuracy and in [42] to improve its execution time.

This is also the approach adopted in this work. The reason is that we want to easily make use of the same deep neural network on different kinds of data (time series, audio spectrum and image) and also simplify the implementation.

Instead of manually building and tweaking a deep neural network architecture, neural architecture search[43] can be used to automate part of this process. Various metrics can be taken into account, including the inference time of the model if real-time constraints need to be met [44]. However, neural architecture search is a costly process. Methods to reduce the time required to achieve good results exist, such as parameters sharing between models [45].

2.4.1.2 Pruning, Weight Sharing and Knowledge Distillation

Another solution is to start with a large deep neural network architecture and reduce its size afterwards.

The number of parameters can be reduced by identifying parts of the network that are not very useful for decision making. These parts can then be removed from the architecture. These pruning techniques can considerably reduce the number of parameters. For example, in [46], the authors managed to remove up to 90% of the parameters. However, unstructured pruning does not allow for an efficient reduction of memory footprint or execution time. Unstructed pruning attempts to remove weights without considering the structure of the neural network. Information about missing weights must be encoded, and the computation also has to take possibly missing weights into account. Recently, several works have structured pruning methods where an entire filter or even a layer is removed [47, 48, 49, 50]. This way, the network can be reshaped with complete removal of the filters or layers, removing all the associated memory, both weights and activations, and all the related computation.

Another method consists in identifying similar parts at various points in the architectures in order to factorize them. For example, in [51] weight sharing is implemented through clustering in order to group similar weights together.

Finally, a smaller neural network can be trained under the supervision of a larger neural network. This technique, called knowledge distillation, can help the smaller neural network to achieve better performance compared to a standalone training. Knowledge distillation has shown promising results, but it is a lengthly and costly process that requires many training epochs for the student model on top of the training of the teacher model itself [52].

2.4.1.3 Quantization

The most popular method to reduce the memory footprint of a deep neural network for embedded systems is quantization, which is the focus of our work. Quantization reduces the precision, i.e., the number of bits used to represent a value in a deep neural network. Quantization can also help changing the way the numbers are represented, e.g. from floating-point numbers to fixed-point numbers. Deep neural networks are often trained using floating-point numbers. However, fixed-point numbers are less costly to process and require a less complex circuit. To reduce the number of bits used to represent a value, quantization maps values from one set to another smaller set. This smaller set can have a constant step between its elements, in which case the quantization scheme is said to be uniform.

In [53] and [54] the authors present the common quantization techniques for deep neural networks. Post-training quantization is performed after training the neural network. Quantization-aware training takes into account quantization error during the training for the computation of the loss function. However, during back-propagation, a straight-through estimator is generally used[55, 56], meaning that the values are not quantized (see Figure 2.5).

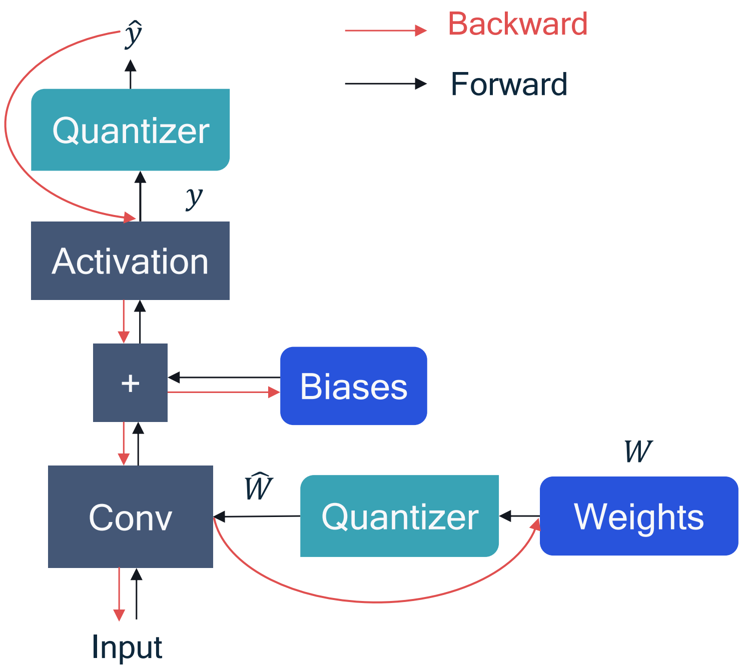


Figure 2.5: Forward and backward computation graph for quantization-aware training with straight-through estimator assumption[53].

When performing quantization-aware training, an alternative to the straight-through estimator is the Differentiable Soft Quantization (DSQ) method presented in [57]. The authors propose to approximate the uniform quantizer by a function that relies on the hyperbolic tangent and stays differentiable over the entire range. Additionally, its slope is adjusted over the training process. It begins as being close to the identity function and progresses more and more towards approximating the step function of the uniform quantizer as shown in Figure 2.6.

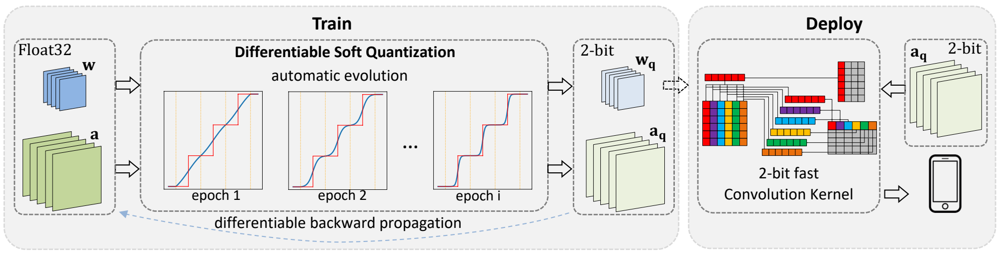


Figure 2.6: Overview of Differentiable Soft Quantization[57]. © 2019 IEEE

The uniform quantization scheme is also described with variants relying on asymmetric ranges for activations. The symmetric range is always centered around 0, while the asymmetric range can be off-center thanks to an offset value. Asymmetric activations range especially helps after ReLU activations, since only the output values cannot be negative. In this case, the ReLU activation has to be fused with the previous layer. Otherwise, the precision would already have been lost before the ReLU activation since negatives values had to be encoded in the range.

Floating-point to fixed-point conversion requires determining a scale factor, so that the floating-point number can be represented as an integer multiplied by a scale factor. In the simplest case, the scale factor is a power of two so that it can be applied using only bit shifts. An alternative consists in finding a scale factor that is not necessarily a power of two, but which scales values within $[-1;+1[$. Using this technique, all the bits (except the sign bit) are used to represent the fractional parts of numbers. As an example, there is the $Q1.15$ format for 16-bit numbers. This allows for a slightly lower quantization error since the quantization step is not necessarily a power of two. The scale factor is also encoded using a fixed-point representation.

The choice of the scale factor (and associated step size and quantization range) can be more fine-grained than a unique choice for the whole network. For example, it can be made different between layers or even between filters of convolutional layers. Although the quantization range can be chosen from the maximum value of the distribution, advanced techniques such as minimizing the Kullback-Leibler divergence metric can reduce the impact of quantization on the accuracy. The authors of [53] present other advanced techniques such as minimizing the quantization error through the mean-squared error metric or the cross entropy of the classification layer. It is also possible to use statistics learnt during training and available in the batch normalization layer to select the quantization range. The impact of the choice of the rounding method in the quantization error is also discussed.

In the case of convolutional neural networks, the authors in [58] show that the weights of convolutional layers typically follow a Gaussian distribution when weight decay is applied. More generally, it has been shown that weights can closely fit Gaussian Mixture Models [59]. Therefore, choosing a non-uniform quantization scheme that better matches the values of the non-uniform distribution of weights would lead to a lower quantization error.

A non-uniform quantization scheme was implemented in [60] on an FPGA device. In this work, instead of coding the value in a fixed-point format, only the nearest power of two is coded. Using this approach, it is possible to obtain a better resolution compared to a fixed-point representation for numbers near 0. This approach also allows large values to be represented, but at the cost of a lower resolution. The quantization step is determined by minimizing the quantization error at the output of the layer, thus balancing the precision and the dynamic range. Furthermore, the computation can be done using bit shifts rather than multiplications since only base 2 exponents are encoded. This solution has some benefits in terms of resource usage and latency for an FPGA target. Additionally, results show that there is a slight degradation of accuracy when using the proposed non-uniform quantization versus a uniform quantization. More recent results show that quantization-aware training can help dampen the deterioration of accuracy [61].

Using lower-precision computation for deep neural networks has been explored in [62]. The authors compare the test error rates on various image datasets for single-precision floating point, half-precision floating point, 20-bit fixed point and their own dynamic fixed-point approach using 10 bits for activations and 12 bits for weights. In their work, it is worth noting that the training is also performed using lower-precision arithmetic. Training with fixed-point arithmetic was presented in [63] with 16-bit weights and 8-bit inputs, causing an accuracy loss of a few percent in the evaluation of text-to-speech, parity bit computation, protein structure prediction and sonar signal classification problems. In [64], the authors have shown that on an Intel® E5640 microprocessor with an x86 architecture, using 8-bit integer instructions rather than floating-point instructions provides an execution speedup of more than 2 without a loss of accuracy for a speech recognition problem. In this case the training was performed using single-precision floating-point arithmetic, and the evaluation was done after the network parameters quantization.

More recently, numerous works propose to use even less bits (i.e., 2 or 3 bits to quantize both weights and activations) and mitigate the accuracy loss with various techniques. PArameterized Clipping acTivation (PACT) is a technique where a clipping parameter for the activation is learnt, instead of being fixed to 6 in the ReLU6 for example[65]. Combined with Statistics-Aware Weight Binning (SAWB) where the quantization range of the weights is chosen using the statistics of the distribution, this can lead to a 2-bit quantized network with only a few percent accuracy loss compared to the baseline [66]. The step size can also be learnt thanks to the Learned Step Size Quantization (LSQ) [67] method, as well as the number of bits with BitPruning[68], or the step size and the dynamic range[69].

In the extreme case, this amounts to binarizing the network parameters[70], leading to drastic optimizations as only binary operations are used in XNOR-Net[71]. For example, up to 32× memory savings and 58× faster computation have been observed. However, binarization also comes with substantial accuracy drop. In [71], when the XNOR-Net is applied to a ResNet-18, a 18% accuracy drop compared to the baseline using single-precision floating-point numbers has been observed with the ImageNet dataset. More recent works have shown that by carefully choosing which part of the residual network to binarize, wisely selecting the quantization scheme and performing other adjustments, a ResNet-50 can be modified into a PokeBNN with most operations being binary and only a few being performed in fixed-point with 4- or 8-bit integers [72]. Such a quantized deep neural network exhibits a much higher accuracy than other binarized neural networks (close to the baseline), while saving a considerable amount of computation and memory.

However, these prior works about quantization and neural network compression in general were mostly not concerned with embedded computing on microcontrollers. Running deep neural networks on microcontrollers began to be popular in the last few years thanks to the rise of the Internet of Things as well as the improved efficiency of deep neural networks.

2.4.2 Deep Neural Network on Microcontrollers

Deep neural networks have already been deployed on 8-bit microcontrollers several years ago. One of the first solution was proposed in [73]. Although interesting, this method requires a lot of work to implement pseudo-floating-point coding, a custom multiplication algorithm over 16 bits, as well as a hyperbolic tangent approximation for the activation function, all in assembly language. Over the last few years, implementations have relied on 32-bit microcontrollers with either a hardware floating-point unit or fixed-point computations. In addition, the Rectified Linear Unit (ReLU) [74] has become widely used as an activation function. ReLU has the benefit of being easily computed as a max between 0 and the layer's output, thus being much less complex than a hyperbolic tangent. In the meantime, neural network architectures and training methods have continued to evolve, providing more and more efficient models. As a result, applications such as spoken keyword spotting [75] can now be performed in real time on IoT devices relying on low-power microcontrollers.

In [28], the authors emphasize that the instruction set architecture (ISA) of available microcontrollers can be a great limitation to running quantized neural networks. Indeed, most microcontroller architectures do not exhibit any kind of SIMD instructions. On the other hand, most microcontrollers rely on 32-bit registers. Thus, even if the neural network parameters and the input data use a lower precision representation, operations have to be computed one by one using 32-bit registers. Some more advanced microcontroller architectures offer instructions able to handle 4 × 8-bit or 2 × 16-bit data packed in 32-bit registers. However, these architectures do not allow working with intermediate or sub-byte precision, and not all arithmetic and logic instructions are covered. Even though it helps further reducing the memory footprint, sub-byte data must be manually packed and unpacked, thus inducing a noticeable computation overhead.

To overcome these limitations, the authors in [28] proposed an extension of the RISC-V instruction set architecture, with instructions to handle sub-byte quantization. Unfortunately, microcontrollers implementing RISC-V are still scarce on the market, and not readily available with the proposed extension. Thefore, this approach cannot be reasonably used to deploy IoT devices since it would require manufacturing a custom microcontroller. Manufacturing a custom microcontroller is hardly feasible when the goal is to release an IoT product on the market, due to large costs, time and the required level of expertise. As a result, only off-the-shelf microcontrollers using 8-, 16- or 32-bit precision are considered in this work.

2.4.3 Existing Embedded Deep Learning Frameworks

Several embedded machine learning frameworks are already available. Among them, the most popular ones are TensorFlow Lite for Microcontrollers[76] and STM32Cube.AI[77]. Other frameworks stemming from research projects also exist and are discussed in the following.

2.4.3.1 TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers (or TFLite Micro) is a project derived from TensorFlow Lite. TensorFlow Lite is originally focused on deep neural network deployment on smartphones. However, the Microcontrollers variant makes it available for microcontrollers. TFLite Micro supports a wide range of operations [78], enabling the deployment of a variety of deep neural networks such as multi-layer perceptrons and convolutional neural networks, including residual neural networks. Deep neural networks are developed and trained using TensorFlow, usually with the Keras interface[79], and can then be semi-automatically deployed onto a microcontroller.

TFLite Micro is intended to be generic enough to be deployed on any kind of 32-bit microcontroller. The inference library is therefore portable; however, there is no integration with specific microcontroller families and vendor tools. While the trained deep neural network (topology and weights) can be automatically converted into a format understandable by the inference library, there are no tools to generate and deploy the application code. Moreover, the test application must be written by hand. Nevertheless, a template source code for a few development boards (e.g., the SparkFun Edge) as well as a few demo applications (e.g., keyword spotting) are available. Finally, TFLite Micro does not come with tools to measure metrics such as the inference time or the RAM and ROM usage.

TFLite Micro supports computation in both single-precision floating-point format and fixed-point numbers on 8-bit integers. The quantization technique uses a non-power-of-two scale factor, a symmetric range for the weights and an asymmetric range for the activations. Biases are quantized on 32-bit integers. Convolution operations can make use of a per-filter scale factor and offset, while other operations use a per-tensor (i.e., per-layer) scale factor and offset [80, 81]. More recently, there has been ongoing work in adding support for 16-bit fixed-point activations.

Inference with fixed-point numbers can be accelerated using low-level optimizations provided by the CMSIS-NN [4] library from ARM. Optimizations include loop unrolling, optimized register allocation, optimized memory access, as well as using specialized instructions. The library also offers acceleration using SIMD (Single Instruction, Multiple Data)-like instructions on some targets. For example, the ARMv7E-M instruction set architecture of Cortex-M4 cores provides an instruction to perform two multiply–accumulate (MACC) operations on 16-bit operands with a single 32-bit accumulator in one cycle.

While being entirely free/open-source, the complexity of the software architecture makes it quite difficult to manipulate and extend. This is a substantial drawback in a research environment, and it also comes with additional overhead. The deep neural network topology is deployed as a sort of microcode that is interpreted at runtime instead of being statically compiled. This process makes it more difficult for the compiler to perform optimizations and causes a larger memory usage.

2.4.3.2 STM32Cube.AI

STM32Cube.AI is a software suite from STMicroelectronics that enables the deployment of deep neural networks onto their STM32 family of microcontrollers. STM32Cube.AI supports deployment of trained deep neural network models from several frameworks including Keras and TensorFlow Lite. A wide range of operations are supported [82], allowing the deployment of several deep neural network architectures, such as multi-layer perceptron and convolutional neural networks, including residual neural networks.

STM32Cube.AI is a software suite fully integrated with other STMicroelectronics development and deployment tools such as STM32CubeMX, STM32CubeIDE and STM32CubeProgrammer. This provides a very straightforward and easy to use flow. Moreover, a test application is included to evaluate the model on target with a real test dataset, without having to write a single line of code This application provides metrics on inference time as well as ROM and RAM usage.

Like TFLite Micro, STM32Cube.AI supports computations in single-precision floating-point format and fixed-point numbers on 8-bit integers. In fact, the quantization on 8-bit integers comes from TFLite. There is no support for fixed-point numbers on 16-bit integers.

STM32Cube.AI also has an optimized inference engine that seems to be partially based on CMSIS-NN. However, as the source code of the inference engine is not freely available, it is not clear what is optimized and how.

The inference library is entirely proprietary and closed-source, therefore it is not possible to manipulate and extend this library. This represents a major drawback in a research environment. It is also not possible to use STM32Cube.AI on microcontrollers which are not part of the STMicroelectronics portfolio. The inference process and optimizations are not detailed, but unlike TFLite Micro, the network topology is compiled into a set of function calls to the closed-source library rather than being interpreted at runtime.

Since 2021, STMicroelectronics provides an alternative proprietary paid solution called NanoEdge AI[83] which also includes automated machine learning aspects.

2.4.3.3 N2D2

N2D2[84] is software framework for end-to-end design and deployment of artificial neural networks. Developped by the CEA-List, N2D2 is aimed at offering a European alternative to overseas frameworks such as TensorFlow.

N2D2 provides post-training quantization[85] and quantization-aware training[86]. The quantization scheme consists in a non-power-of-two scale factor, a symmetric range for weigths and an asymmetric range for the activations, with a per-filter scale factor similar to what TensorFlow Lite offers. Furthermore, N2D2 can optimize the chosen scale factor by minimizing the quantization error using either the mean-squared error or the Kullback–Leibler divergence metric. The quantization-aware training also includes support for the Learned Step size Quantization[87] and the Scale-Adjusted Training[88] methods.

N2D2 supports various targets including CPUs, GPUs, and their own accelerator architecture for FPGA called PNeuro. While STMicroelectronics STM32 microcontrollers family is supported, the support for this target is proprietary and not publicly available. Details about this support are scarse. Nevertheless, there are target-specific optimizations for ARM Cortex-M4 and Cortex-M7 cores.

The fact that N2D2's STM32 target support is not publicly available does not make it suitable for inclusion in our study. However, in the future, thanks to the Deep Green project[89], N2D2 will be extended to cover more targets and to improve interoperability with existing frameworks. It is also planned that N2D2 becomes fully open-source.

2.4.3.4 NNoM

NNoM (Neural Network on Microcontroller)[90] enables the deployment of a trained Keras model to microcontrollers thanks to a portable C library for inference. It is in fact similar to our proposition in Chapter 3, although we designed an end-to-end training and deployment framework around our conversion tool, while NNoM is just a standalone conversion tool. The design approach of the conversion tool is also slightly different. It proposes a more complete and complex C library while the Python tool simply performs the quantization and generates the configuration of the layers as C structures to call in the C library. We instead generate a simple C code from templates of the layers with the configuration being written as constants or literals.

NNoM does not support PyTorch models. However, support for recurrent cells (such as GRU and LSTM) was recently added, something we did not focus on yet. NNoM can optionally make use of the CMSIS-NN library for target-specific optimizations. Furthermore, the included non-optimized implementation of the layers' operations is almost identical to CMSIS-NN non-optimized code.

Post-training quantization can be used to run the inference with fixed-point numbers on 8- or 16-bit integers. Similarly to N2D2, it can use the Kullbak-Leibler divergence metric to find the scale factor for the quantization, or simply use the maximum value of the tensor as we do. There is no support for quantization-aware training since NNoM does not interface with any training capabilities of the deep learning framework. The quantization scheme relies on a power-of-two scale factor and symmetric ranges. The non-optimized implementation can make use of per-filter scale factor.

A short comparison between NNoM, STM32Cube.AI and TensorFlow Lite Micro is provided in [91] and [92].

2.4.3.5 Other Frameworks

Some other frameworks have been developed as part of research projects. These frameworks mainly focus on "classical" machine learning (SVM, Decision Tree, etc.), for example, emlearn [93] and Micro-LM [94], or multi-layer perceptron, for example, Gravity [95] and FANN-on-MCU [96]. These frameworks do not support convolutional neural networks with residual connections.

MicroTVM[97] brings support for microcontrollers to TVM[98], a compiler framework focused on deploying and optimizing deep learning models. At the time of this work, microTVM seems less mature and less popular than TensorFlow Lite for Microcontrollers and STM32Cube.AI. It is, therefore, not studied in this work.

Finally, apart from STM32Cube.AI from STMicroelectronics, other microcontroller vendors have developped their in-house solution for embedded deep neural networks, such as e-AI from Renesas[99] and eIQ from NXP[100] among others. Furthermore, many companies providing solutions for embedded artificial intelligence have emerged in the past few years [101].

2.5 Smart Glasses and Human Activity Recognition

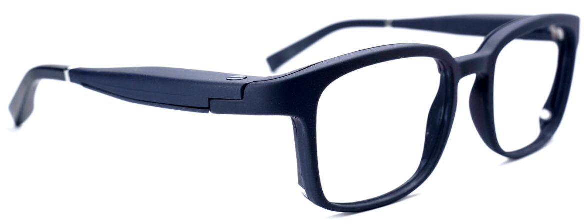
Among existing embedded systems, wearables mainly bring constraints on energy, physical dimensions and cost. Other aspects such reliability and safety are often less critical in these consumer devices. In this work, we are especially interested in smart glasses.

2.5.1 Smart Glasses Products and Applications

In smart glasses, the space used by the electronic circuits must be as small as possible since it must fit in the branches of the glasses, including the battery. As a result, the battery can only contain a very small amount of energy, so the electronic circuits must be as efficient as possible in order to obtain a reasonable battery life.

Smart glasses have many applications in the healthcare, entertainment or industrial fields and their popularity have been steadily increasing in research environments for the past few years [102]. Usually, smart glasses are associated with augmented reality, where a camera captures the scene in front of the user to analyze the environment, and additional information are overlayed on top of the environment through a display on the lens [103]. However, smart glasses used for augmented reality are often heavy and bulky due to the processing power and energy required by image processing and associated computer vision algorithms. Use-cases of smart glasses are not limited to augmented reality. Various sensors such as a microphone, an inertial measurement unit or even a GPS can be used to enhance the experience of the wearer. For example, the SPIDERS+ system provides emotion sensing through an infrared camera, a proximity sensor and an inertial measurement unit, as well as bio-signal acquisition with other sensors [104].

Different smart glasses product are either available on the market or as research prototypes. In this work, we rely on the Smart Connected Eyewear offered by Ellcie Healthy, a local company developping smart glasses and associated applications. These glasses were initially used for driver's drownsiness detection through blink measurements [105]. The blink detection algorithm performance was improved by making use of a convolutional neural network, deployed on the smart glasses' microcontroller using STM32Cube.AI [106]. Compared to products such as the Microsoft Holo Lens 2 or a the Google Glass Enterprise Edition 2, Ellcie Healthy's product embeds a very low-power microcontroller rather than a high-performance system-on-chip. No camera or display are present since it is not designed for augmented reality. Ellcie Healthy's smart glasses (Figure 2.7a) are similar to the JINS MEME smart glasses (Figure 2.7b). Although details about the JINS MEME hardware are scarce, it also includes an inertial measurement unit and can be used for blink measurement in a driver's drowsiness detection application [107].


(a) Ellcie Healthy Smart Connected Eyewear.

(b) JINS MEME[107]. © 2016 SAE International

Figure 2.7: Smart glasses products.

In [108, 109], the inertial measurement unit of Ellcie Healthy's smart glasses has been used for evaluation of sit-to-stand and gait movements, showing that the smart glasses can be used for elderly care applications. Artificial neural networks have not been used for these evaluations yet, and the data is processed on a computer after the recording.

In this thesis, we attempt to integrate artificial intelligence on Ellcie Healthy's smart glasses in order to handle a human activity recognition task.

Before building and deploying a deep neural network for human activity recognition, data must be obtained for training and evaluation purposes.

2.5.2 Human Activity Recognition Datasets

Datasets for human activity recognition using various modalities have been flourishing for the past decade [110]. Two sensor categories are mainly used for human activity recognition:

vision-based and body-worn sensors. Vision-based sensing relies on cameras placed in the environment to capture a video stream of a subject performing activities of daily living [111]. This requires external cameras and a considerable processing power, therefore vision-based approaches are not suitable for embedding into a wearable device. In this work, we mainly focus on body-worn sensors. Body-worn sensors rely on inertial measurement units (IMU), including an accelerometer, a gyroscope and sometimes additional sensors (magnetometer, barometer, etc.) to measure the subject movements.

The most iconic dataset for human activity recognition using an inertial measurement unit is likely the Human Activity Recognition dataset hosted by the University of California Irvine, commonly dubbed UCI-HAR [112]. A total of 30 subjects participated in the experiments, performing 6 activities: WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, and LYING. 21 subjects are used for training while the other 9 are used for testing, representing 7352 and 2947 vectors, respectively. This dataset is built from a 3-dimensional accelerometer and a 3-dimensional gyroscope sampled at 50 Hz, embedded into a smartphone attached to the subject's waist. The acceleration signal is filtered to create an additional signal without gravity. Therefore, there is a total of nine channels of sensor data. The data are windowed over 2.56 s with 50% overlap to create windows of 128 samples. The data are provided in two forms: vectors of 128 samples for each of the nine sensor channels, and vectors of 561 features computed from the 128×9 values. Despite providing a better classification accuracy, pre-computed features are often generated using computationally expensive Fast Fourier Transforms (FFT). On the other hand, feature extraction can be performed automatically through convolutional layers. However, existing work show that in this case the accuracy decreases from 95.8% to 93.3% [113]. When convolutional layers are used on top of pre-computed features, the accuracy can be improved up to 97.5% in [114]. As it will be seen further, some aspects of our dataset are inspired by UCI-HAR such as some classes and the window duration.

The UCI-HAR dataset was extended in [115] to provide the transitions between static activities: STAND_TO_SIT, SIT_TO_STAND, SIT_TO_LIE, LIE_TO_SIT, STAND_TO_LIE, LIE_TO_STAND. This SBHAR dataset was used to evaluate the Transition-Aware Human Activity Recognition [116] system along with two other datasets: PAMAP2 and REALDISP.

Instead of using a single smartphone with an accelerometer and a gyroscope, the PAMAP2 dataset [117] rather uses dedicated IMU devices called Colibri Wireless from Trivisio. One device is placed on the wrist, another one on the chest and a last one on the ankle. Each device contains a 3-dimensional accelerometer, a 3-dimensional gyroscope and a 3-dimensional magnetometer, along with a temperature sensor, all sampled at 100 Hz. Additionally, a heart-rate monitoring device is sampled at 9 Hz. In this dataset, nine subjects perform 12 to 18 activities. This setup is much more intrusive than UCI-HAR as multiple dedicated devices are used at specific location, making this approach harder to use in real conditions for live human activity recognition.

The REALDISP [118] dataset has an even more complex setup, using 9 IMU devices from Xsens sampled at 50 Hz, each with a 3-dimensional accelerometer, a 3-dimensional gyroscope, a 3-dimensional magnetometer. The IMU devices also provide orientation estimates in quaternion format (4D) [119]. This dataset contains more classes and more subjects than PAMAP2: 33 classes and 17 subjects. Its purpose was to study the impact of sensor placement.

Other popular human activity recognition datasets include UniMiB SHAR [120] containing accelerometer samples captured from a smartphone, Real-Life HAR [121] also collected from a smartphone but focusing on real-life situations (for example inactive, active or driving) rather than a laboratory setting, and OPPORTUNITY [122] that uses many sensors of different modalities.

Apart from these datasets using data collected from smartphones or specific devices, there are few other datasets based on wearables available from the market. Let us cite WISDM [123] using a combination of a smartphone and a smartwatch (LG G Watch) to collect data from 51 subjects performing 18 activities. Other datasets for human activity recognition, such as [124] relying on a Microsoft Band 2, have been created from consumer smartwatches. However, these datasets have not been released so far.

More specifically, smart glasses are still not a popular device to use for human activity recognition. Nonetheless, prior works have been done to build a dataset for smart devices including smart glasses in [125]. This dataset makes use of JINS MEME smart glasses as well as a smartphone and a smartwatch to collect data from different sensors. The smart glasses provide data from an embedded IMU. This dataset has however some noticeable drawbacks. First, only one subject participated in the experiment. Moreover, there is no well-defined set of activities or well-defined protocol, which makes it difficult to evaluate or to extend.

Another work used JINS MEME smart glasses for human activity recognition. However the dataset does not appear to be publicly available. Machine learning techniques such as Random Forest were used on the data collected from the JINS MEME smart glasses while artificial neural networks were not evaluated [126].

Some efforts have been made in [127] to develop a system for activity recognition using smart glasses (Google Glass Explorer Edition XE 22). The authors compare the classification performance of a Support Vector Machine (SVM) between data collected either from a smartphone or smart glasses for 4 activities (Biking, Jogging, Movie Watching, and Video Gaming). This system can perform inference on the Android smartphone but not on the smart glasses.

However, and as it has been said in the introduction, each dataset has its own characteristics depending on the device that has been used. The device itself and its position greatly impacts the angle of the acceleration (both gravity and linear acceleration) as well as the signal shape for some movements. Additionally, the sensors themselves can have varying sensitivity and sampling rate. Therefore, using an existing dataset for a different device or application would produce poor classification results. For this reason, we decided to build our own dataset for the Ellcie Healthy's smart glasses.

2.6 Unsupervised On-Device Fine-Tuning

Previous discussions on embedded deep neural network were mostly about inference, i.e., a deep neural network pre-trained on a workstation and then deployed onto an embedded system, without any modification of the deep neural network afterwards. However, performing domain adaptation through online learning directly on the embedded system may help improving the performance of the deep neural network without requiring a new offline training from scratch, and especially a new deployment of the neural network model. Updating the neural network model on the device once it is deployed in the field can indeed be difficult. It requires either physical access with manual intervention from an operator, or over-the-air upgrade through a wireless network, both with an associated downtime.

Instead, the neural network model could be updated continuously or as needed directly on the target. Nevertheless, as previously discussed, training is usually a resource-intensive process in terms of memory and computation. Additionally, once the device is deployed, it may not be possible to provide labelled data for training. Therefore, alterative solutions to train a neural network on-target in an unsupervised online fashion must be explored.

2.6.1 Unsupervised Learning

Unsupervised learning methods are used when no label is available for the input data. It is usually required to label by hand a large amount of data to build an accurate model. Without labels, it is difficult to train a deep neural network using backpropagation. The reason is that the loss function usually measures the difference between the prediction of the deep neural network and the label from the ground truth. Instead, either a different loss function, a different training algorithm, or a different machine learning method altogether must be used.

For example, the deep neural network can be designed as an autoencoder to learn an embedding of the input called the latent space. The latent space is generally of a smaller dimension than the input, but sometimes it can be larger [128]. An auto-encoder is built as an encoder network followed by a decoder part, both typically constructed using convolutional layers, with the encoder input and the decoder output being of the same dimension. The autoencoder can be trained with backpropagation using a loss function minimizing the distance between the input and the output. Therefore, the training only relies on unlabelled data. Autoencoders can be used for data compression[129], denoising[130] and anomaly detection[131] among other applications. Autoencoders cannot be directly used for clustering or classification purposes. However, the encoder part can be used as a feature extractor which then feeds a clustering or classification algorithm [132]. Nonetheless, features learnt by the autoencoder may not be of the highest relevance for a clustering or classification problem since these features were learnt for a good reconstruction of the input rather than a good separability of the clusters or classes. In fact, it is possible to reuse a feature extractor such as a convolutional neural network trained in a supervised manner to feed the features to an unsupervised algorithm. The feature extractor can even be trained on a different dataset for a different but similar task in a transfer learning fashion [133].

In the case of spiking neural networks, the Spike-Timing-Dependent Plasticity (STDP) learning rule[134] is also an unsupervised method which reinforces synapses for neurons with a correlation in time. The learning rule is local and does not take into account the inputs or the outputs of the neural network.

Apart from feedforward neural networks, other bio-inspired neural models can rely on an unsupervised learning rule, such as the Self-Organizing Map[135]. The self-organizing map is made of neurons arranged in a two-dimensional grid with each neuron connected to its four neighbors (other arrangement with higher dimensions or different connectivity are also possible). In a self-organizing map, the learning rule is local and depends on the inputs. Such an algorithm can be described as a vector quantization algorithm. As a self-organizing map cannot be used directly for a clustering or classification problem, another clustering method such as k-means or a labelling method[136] can be applied on the neurons to get clusters or classes.

Outside of artificial neural networks, many clustering methods exist and can be used in an unsupervised manner, such as k-means[137], DBSCAN[138], its hierarchical variant HDBSCAN[139], or OPTICS[140].

In order to solve a classification problem, labelling the clusters or the output neurons of the artificial neural network is still required. This labelling could be performed manually by an expert, or automatically using a small subset of labelled data from the training set. For example, the number of required labels can be reduced down to 20% for human activity recognition [141] or even only 1% for handwritten digit recognition[136]. Overall, this combination of methods can be seen as a semi-supervised approachapproach since some data needs to be labelled for classification purposes, whereas the training itself is unsupervised. Furthermore, active learning[142] can be used to make a better choice of the data to label.

In the human activity recognition literature, unsupervised approaches have already been evaluated several times[143], mostly with non-neural-based machine learning methods such as k-means. The bio-inspired approaches such as the self-organizing map remain mostly unexplored for human activity recognition with body-worn sensor, most of the prior works being about video data[144]. The accuracy of several supervised or semi-supervised learning methods have been already evaluated in the litterature on the UCI-HAR dataset as presented in Table 2.1 with two types of input: raw data (designed by * on the table) or pre-computed features. The vast majority of methods use pre-computed features that require to pre-process the signals.

Finally, unsupervised learning approaches make online learning possible when it is difficult to label data as they come.

Table 2.1: Mean accuracy (%) for various learning methods on UCI-HAR.

Method	**Accuracy**
Neural Networks	
Perceptron[145]	92.4
Recurrent neural network[145]	95.1
Multi-layer perceptron[145]	94.7
Deep convolutional neural network [146]	95.75
Stacked Autoencoder [147]	92.16
Denoising Autoencoder [147]	90.50
3-layers convolutional neural network *[148]	92.71
LSTM (Long Short-Term Memory)[148]	89.1
Bidirectional LSTM[148]	89.40
Multi-layer perceptron[148]	86.83
Convolutional neural network frequency domain[114]	97.50
3 layers LSTM[149]	97.4
Supervised Machine Learning	
Nearest Neighbours[150]	91.0
Decision Tree[150]	87.4
Random Forest[150]	91.4
Support-Vector Machine (SVM)[150]	90.7
AdaBoost[150]	40.8
Naive Bayes[150]	88.9
Quadratic discriminant analysis[150]	90.0
Multi-Class SVM [112]	96
One-vs-one multiclass linear SVM with majority voting [151]	96.4
A sparse kernelized matrix learning vector quantization model [152]	96.23
Confidence-based boosting algorithm Conf-AdaBoost [153]	94.33
Two-Stage Continuous Hidden Learning Markov Models [154]	91.76
Semi-Supervised Methods	
Recurrent Variational Autoencoders with Sparse Labels (1.4% of labels)[155]	63.0
Recurrent Variational Autoencoders with Sparse Labels (14% labels)[155]	83.9
Recurrent Variational Autoencoders with Sparse Labels (100% of labels)[155]	91.7
Transformation Prediction Network (0.4% of labels)[156]	(f-score) 0.88
Transformation Prediction Network (4% of labels)[156]	(f-score) 0.90
Transformation Prediction Network (100% of labels)[156]	(f-score) 0.91
Learning to Predict Cross-Dimensional Motion (1% of labels)[157]	(f-score) 0.80
Learning to Predict Cross-Dimensional Motion (100% of labels)[157]	(f-score) 0.90

* Raw data.

2.6.2 Online and Continual Learning

There are various aspects of machine learning associated with online learning such as incremental learning, continual learning and the associated problem of catastrophic forgetting. First and foremost, online learning is about a model learning from the data as they come, in a streaming fashion. In contrast, the more traditional offline or batch learning techniques has access to the entire dataset and can generally fetch data at random and multiple times [158]. A key characteristic of online learning is that all the data cannot be stored in memory, as the stream of data can be infinite. Therefore, the model should be able to learn input vectors one by one. As mentioned, memory is the main limit to the implementation of a batch learning algorithm. It is indeed required to store a large dataset. Online learning does not have this limitation and makes it possible to implement deep neural network training on an embedded system.

Furthermore, online learning leads to another class of learning methods: continual, incremental or lifelong learning. There is no clear consensus on the exact definition of these various methods. In this work, we assume that continual, incremental and lifelong learning all refer to the ability of a model to learn from new information on-the-fly, after the initial training is over, without requiring a new training from the ground up. Additionally, there are mainly two different learning scenarios: new instances, where the model learns new information for existing classes, and new classes, where the model learns new information for previously unknown classes [159].

One of the main problem caused by continual learning which is studied extensively in the litterature is catastrophic forgetting [160]. Catastrophic forgetting happens when a model learn new information from the incoming data, but forgets the information previously learnt from data that is not available anymore. For example, in a classification task, if the new data only contains information about a few classes, the classes previously learnt may be forgotten and the model may not be able to classify data into these classes correctly anymore. Similary, if a distribution shift happened, the model may not be able to handle the original distribution anymore.

A common mitigation of catastrophic forgetting is to use replay or rehearsal methods: while learning new information, a small part of old information is also made available to the training algorithm [161]. This way, the model can learn the old information again, thus preventing it from being forgotten. Rehearsal methods typically require additional memory to store part of the information learnt previously. While this memory is many times smaller than a full offline dataset, it can be substantial for an embedded system. Furthermore, if the new information also has to be remembered, then it becomes difficult to dimension the memory and to select which information needs to be recorded.

In [162], authors propose to use quantized latent replays to reduce the memory usage on embedded systems. Latent replays use feature maps of a lower dimension than the input taken at an intermediate layer and only train the remaining layers as shown in Figure 2.8. Another approach is to generate data on-the-fly instead of using pre-recorded data, a kind of pseudo-rehearsal. For example, in [163], the mean and standard deviation of the latent space for the old tasks are recorded. Then during the learning of new tasks, features from the old tasks are generated using these statistics as Gaussian distributions parameters. In [164], the authors propose to use reverberating networks to fight against catastrophic forgetting with the process illustrated in Figure 2.9. One network learns information from the new data and rehearses the old information from data generated by the second network. This second network only requires random input to generate data. The role of the networks can be exchanged for the second network to also learn the new information.

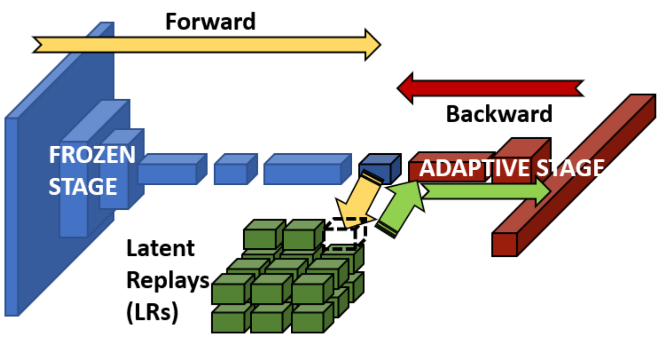


Figure 2.8: Continual Learning with Latent Replays[162]. © 2021 IEEE

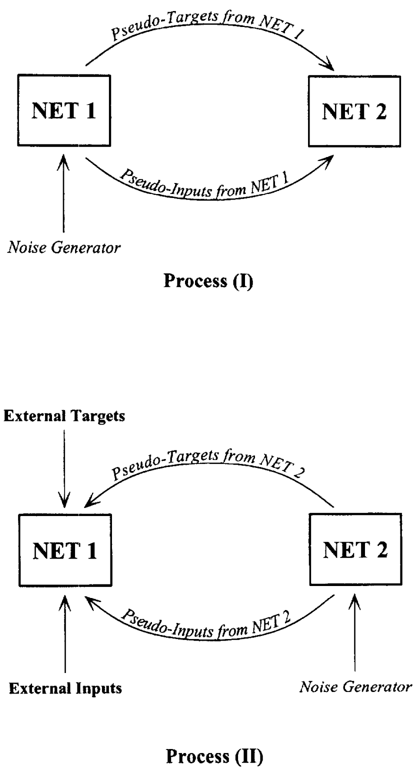


Figure 2.9: Reverberating Networks[164]. © 1997 Elsevier Masson SAS

Despite training with an online learning method being less efficient than with an offline mini-batch method, continual learning can make use of backpropagation. Backpropagation, however, generally requires a supervised training. Some unsupervised learning methods can also be adapted to work in an online learning setting. Self-organizing maps are initially not suitable for online learning due to the convergence process relying on a finite time constant. The Dynamic Self-Organizing Map (DSOM)[165] removes this time dependency to enable online learning, where there is no end to the training process.

In order to evaluate the performance of continual learning and the associated catastrophic forgetting, a dedicated dataset is required. However, not many are available and most of them target computer vision. Additionally, they do not reflect a real-world use-case of continual learning [166]. For example, the permutated MNIST dataset used in [167] is an artificially modified MNIST dataset with different permutations of pixels. The CORe50 dataset[159] attempts to propose an evaluation method for continual learning, by providing different sessions with diverse settings for the same objects that must be learnt sequentially. Nonetheless, it is still targeted towards computer vision and not really suitable for low-power embedded system evaluation.

In fact, our goal is slightly different in the human activity recognition scenario, since we aim to specialize the model for a new subject, after an initial training performed on multiple other subjects. Therefore, our experiments mostly rely on datasets divided by subjects, where one subject from the testing set is selected for continual learning. This is often the case for human activity recognition datasets such as UCI-HAR, as well as our own dataset, but can sometimes be found in other domains such as keyword spotting with the Heidelberg Digits[168] dataset. Despite the Heidelberg spiking datasets being originally designed for spiking neural networks evaluation, we use the raw audio data of the Heidelberg Digits dataset in non-spiking neural networks.

2.6.3 On-Device Learning

One of the objective of this thesis is also to be able to perform the training directly on the device, rather than on a remote server. As training requires even more resources, this is a bigger challenge than inference.

In [169], the authors propose a system to perform on-device online learning on microcontrollers. The TinyOL system relies on an additional layer that can be updated on-the-fly, while the rest of the pre-trained neural network is frozen. The layer gets updated using stochastic gradient descent when new labelled data are presented to the neural network. Additionally, the authors show that the system can also be used in an unsupervised learning setting, with a use-case of anomaly detection using an autoencoder. In my opinion, the use case (anomaly detection on a fan using an accelerometer) may be too simplistic to properly show the usefulness of such a solution.

Similarly, the authors of [170] present the same use case, but with a slightly different method. The additional layer is trained using OS-ELM (Online Sequential Extreme Learning Machine)[171], and the data is pre-processed with a fast Fourier transform.

A more complex task to be solved by embedded deep neural networks is presented in [162]. Authors propose to perform continual learning for object recognition on-device thanks to a high-performance multi-core processor. The software implementation is specifically optimized for this custom processor, and leads to energy figures multiple times better than what can be achieved on a conventional low-power STM32L476RG microcontroller.

The Tiny Training Engine presented in [172] enables transfer learning for a visual recognition task in less than 256 KiB of RAM. This way, continual learning can be performed on an embedded system. The authors achieve near state-of-the-art accuracy (89.1%) on the Visual Wake Words dataset with a model pre-trained on ImageNet, the transfer learning being performed in 206 KiB of RAM. The Tiny Training Engine provides a code generation tool with compile-time differentiation in order to support backpropagation through multiple layers on the device in the lightest possible way. Additionally, the backpropagation can take into account a partially pruned backward graph in order to skip the processing of frozen layers or weights, thus significantly reducing the amount of memory to perform the weights update.

Other works focus on making use of on-device learning for federated learning. Federated learning is a technique to train a model in a decentralized manner, where multiple nodes contribute to the training without requiring to centralize or even exchange the input data. At the edge, it means that the data does not leave the device, while multiple devices connected together can share their processing power to learn new information. In [173], a modification of the stochastic gradient descent, called Lightweight Stochastic Gradient Descent (L-SGD) is proposed in order to support deep neural network training on microcontrollers. The results show that L-SGD is able to perform a faster training with less memory using the MNIST or the CogDist datasets with only a small difference in accuracy. Despite this work discussing federated learning, it does not provide an actual use-case. Additionally, the federated learning approach would still rely on a remote server for knowledge aggregation. [174] presents a more concrete use-case of keyword spotting using 3 microcontroller devices, and a remote server to share the knowledge.

Train++[175] is a simple perceptron that performs binary classification with a supervised online learning rule to update its weights. While the learning rule is original as it does not rely on gradient computation and it is executed on a microcontroller, the model is too simplistic to perform realistic tasks requiring non-linear multiclass classification.

2.7 Conclusion

In this chapter, we presented existing works related to the three main topics that this thesis focuses on: embedded artificial neural networks, human activity recognition with smart glasses, and unsupervised on-device fine-tuning.

The first topic of embedded artificial neural networks covered the compression and the deployment of deep neural networks on embedded devices. As of 2022, the scientific literature covers many aspects of deep neural network compression with various methods that can be applied to reduce the memory footprint or the execution time of deep neural networks. While we do not contribute a new method to this field, we implement and evaluate quantization in several use cases to provide an overview of the optimization of the resources for several applications running on microcontrollers (human activity recognition, keyword spotting, and traffic-sign recognition). However, at the beginning of the work on this thesis (late 2019), we identified several shortcomings to the existing software frameworks for deep neural network deployment on microcontrollers. This led us to the development of our own software framework, which we released under an open-source license.

As for human activity recognition with smart glasses, existing works do not cover this application well, and no usable dataset is available. Therefore, we decided to build our own dataset and release it under an open access policy. We developed a prototype of live human activity recognition on smart glasses using this data.

Finally, unsupervised on-device fine-tuning is for the most part still unexplored. Apart from the use-case of anomaly detection, most of the existing work focused on supervised learning. This topic combines the aspects of unsupervised learning, online learning and on-device learning which are all challenging. Furthermore, instead of exploring these aspects only on synthetic data or simple datasets, we wanted to know if these approaches could work in a real-world situation. Therefore, we propose an unsupervised fine-tuning method and evaluate it on human activity recognition and keyword spotting datasets. Then, we implement it on a microcontroller to provide an on-device learning method.

Chapter 3

Quantization and Deployment of Deep Neural Networks on Microcontrollers

Contents

3.1 Introduction

Microcontrollers have only a very small amount of Flash memory, often less than 1 MiB. They also run deep neural network algorithms several orders of magnitude more slowly than GPUs or even CPUs. The reason is that microcontrollers generally rely on a general-purpose processing core that does not implement parallelization techniques such as thread-level parallelism or advanced vectorization. Moreover, microcontrollers generally run at a much lower frequency than GPUs (typically 8 MHz to 80 MHz compared to 1 GHz to 2 GHz). Their simpler design and lower performance also allow them to run with a reduced power consumptions compared to general purpose CPUs found in personal computers or GPUs for instance.

Table 3.1 gives examples of a microcontroller, a CPU and a GPU. The time taken to compute one prediction is then compared for each of these devices in Table 3.2. These figures outline a power consumption approximately 3000 times lower for the microcontroller compared to the CPU or the GPU. However the microcontroller is also 2000 to 7000 times slower to compute a prediction than the CPU, and up to 27 000 times slower than the GPU for the considered deep neural networks.

Table 3.1: Microcontroller (STM32L452RE), CPU (Intel Core i7-8850H) and GPU (NVidia Quadro P2000M) platforms. Power consumption figures for the GPU and the CPU are the TDP (Thermal Design Power) values from the manufacturer and do not reflect the exact power consumption of the device.

Platform	**Model**	**Framework**	**Power Consumption**
MCU	STM32L452RE	STM32Cube.AI	0.016 W
CPU	Intel Core i7-8850H	TensorFlow	45 W
GPU	NVidia Quadro P2000M	TensorFlow	50 W

Table 3.2: Comparison of 32-bit floating-point inference time for a single input on the microcontroller, the CPU and the GPU of Table 3.1. The neural network architecture is described in Section 3.4 with the number of filters per convolution layer varying from 16 to 80. The dataset is described in Section 3.4.1.1. For the CPU and the GPU, the inference batch size is set to 512 to exploit parallelism and the dataset is repeated 104 times to try to compensate for the large startup overhead compared to the total inference time. Measurements are averaged over at least 5 runs.

Platform	**Inference Time (ms)**						
	16 Filters	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
MCU	85	174	271	404	544	921	1387
CPU	0.0396	0.0552	0.0720	0.0937	0.1134	0.1538	0.2046
GPU	0.0227	0.0197	0.0223	0.0284	0.0317	0.0395	0.0515

Thanks to their low power consumption, microcontrollers can be powered from tiny battery cells. Yet the power consumption must still be kept as low as possible in order for the batteries to discharge as slow as possible. In some cases, when data are collected in remote areas for instance, they cannot even be recharged in the field. Therefore, performing inference at the edge faces major issues in terms of real-time constraints, power consumption and memory footprint. To meet these constraints, the deployment of a deep neural network must respect an upper bound for one inference response time as well as for the number of parameters of the network.

As a result, a deep neural network must be limited in width and depth to be deployed on a microcontroller.

As it has been observed, deeper and/or wider networks are often able to solve more complex tasks with better accuracy [40]. As such, there is always a trade-off between memory footprint, response time, power consumption and accuracy of the model.

Additionally, compression techniques can be applied to a deep neural network in order to decrease the memory footprint, and, in some cases, the energy required by the computation.

A technique that provides a significant decrease in the memory footprint is network quantization. Quantization consists in reducing the number of bits used to encode each weight of the model, so that the total memory footprint is reduced by the same factor. Quantization also enables the use of fixed-point arithmetic rather than floating-point arithmetic. In other words, operations can be performed using integer rather than floating-point data types.

This is interesting because integer operations require fewer computations on most processor cores, including microcontrollers. Without a floating-point unit, floating-point instructions must be emulated in software, creating a large overhead, as illustrated in [176]. In that study, a comparison between software, hardware and custom hybrid floating-point unit implementations is provided. Even if a hardware floating-point unit is available, operations can often be performed slightly faster on the integer arithmetic and logic unit rather than the floating-point unit. For example, the ARM Cortex-M4F core needs two clock cycles to perform an integer multiply-accumulate instruction[177], or even just one cycle with the appropriate DSP (Digital Signal Processing) instruction. In the meantime, the hardware floating-point unit needs three cycles to perform a single-precision floating-point multiply-accumulate instruction[178]. Multiply-accumulate instructions are at the heart of the deep neural network inference since the computation model relies on multiplying the synaptic weights with the inputs and accumulating the result into the neuron's potential.

In this chapter, we present an open-source [179] framework, called MicroAI, to perform end-to-end training, quantization and deployment of deep neural networks on microcontrollers. The training phase relies on the well-known TensorFlow and PyTorch deep learning frameworks. Our objective is to provide a framework that is easy to adapt and extend, while maintaining a good compromise between accuracy, energy efficiency and memory footprint.

Results using two different microcontrollers (STMicroelectronics STM32L452RE and Ambiq Apollo3) and three different inference engines (STM32Cube.AI, TensorFlow Lite for Microcontrollers, and our own, MicroAI) are provided. Results are compared in terms of memory footprint, inference time and power efficiency.

Finally, we propose to apply 8- and 16-bit quantization methods on three datasets dealing with different modalities: acceleration and angular velocity from body-worn sensors for UCI-HAR, speech for Spoken MNIST and images for GTSRB. These datasets are light enough to be handled by a deep neural network running on a microcontroller, but still relevant for applications relying on embedded artificial intelligence.

Section 3.2 presents the methodology implemented in our MicroAI framework for deep neural network quantization. Section 3.3 details our MicroAI framework and compares it to existing solutions. In Section 3.4, some comparative results between our framework MicroAI and two popular embedded neural network frameworks (TensorFlow Lite for Microcontrollers and STM32Cube.AI) are provided for two microcontroller platforms (Nucleo-L452RE-P and SparkFun Edge) in terms of memory footprint, inference time, and power efficiency. The impact of our 8- and 16-bit quantization methods for three different datasets (UCI-HAR, Spoken MNIST and GTSRB) is also presented. In Section 3.5 the results obtained are discussed. Finally, Section 3.6 concludes this chapter.

3.2 Deep Neural Network Quantization

3.2.1 Representation of Real Numbers

3.2.1.1 Floating-Point

In modern computation systems, real numbers typically use a floating-point representation. Floating-point representation relies on the encoding of three different pieces of information: the sign, the significand and the exponent. Coding the significand and the exponent separately makes it possible to represent values with a very large dynamic range, while at the same time providing increasing precision as the numbers approach 0.

Most floating-point implementations follow the IEEE754 [180] standard which defines how the sign, significand and exponent are coded in a binary format. Floating-point numbers can be coded in half, single or double precision requiring 16, 32 or 64 bits, respectively. Obviously, the more bits allocated to code a value, the more precise it is. A higher number of bits allocated to the exponent also allows for a larger dynamic range. In deep neural networks, single precision is more than enough for training and inference. Double precision requires more computing resources, so it is generally not used. Recently, it has been shown that half-precision can further accelerate the training and inference without a significant drop in the accuracy of deep neural networks [181].

However, the choice is much more restricted for low-power microcontrollers. When present, the hardware floating-point unit usually supports single-precision computation only. Double-precision computations must be performed in software and are therefore significantly slower than single precision. Half-precision data are converted to single precision before the computation. In 2019, ARM released the ARMv8.1-M instruction set architecture which includes instructions for half-precision support. Even though the Cortex-M55 core is planned to implement these instructions, there is so far no microcontroller with this core available on the market. As a result, when floating point is used on a microcontroller, only single precision is considered.

The binary representation of single-precision floating-point numbers is called binary32 and is represented with 1 bit for the sign, 8 bits for the exponent and 23 bits for the significand (see Table 3.3). It allows a dynamic range of roughly $[-10^{38}, 10^{38}]$, far beyond the values typically seen in a deep neural network, while increasing the resolution for numbers close to 0. The closest possible numbers to 0 are approximately $\pm 1.4 \times 10^{-45}$.

Table 3.3: Single-precision floating-point binary32 representation.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sign	exponent								significand																						

3.2.1.2 Fixed-Point

Fixed-point is another way to represent real numbers. In this representation, the integer part and the fractional part have a fixed length. As a result, the dynamic range and the resolution are directly limited by the length of the integer part and of the fractional part, respectively. The resolution is constant across the whole dynamic range. In binary, the Q notation is often used to specify the number of bits associated with each part. $Qm.n$ is a number where m bits are allocated to the integer part and n bits to the fractional part [182]. It is important to note that we consider signed numbers in two's complement representation. The number of bits for the integer part can be increased to obtain a larger dynamic range, but it will conversely reduce the number of bits allocated to the fractional part, thus reducing the precision.

Given a $Qm.n$ signed number, its dynamic range is $[-2^{m-1}, 2^{m-1} - 2^{-n}]$ and its resolution is 2^{-n}.

As an example, in Table 3.4, a signed $Q16.16$ number stored in a 32-bit register has 16 bits for the integer part and 16 bits for the fractional part. This translates to a dynamic range of [−32 768, 32 767.9999847], much smaller than the equivalent floating-point representation, and a constant resolution of 1.5259×10^{-5} across the whole range, less precise than the floating-point representation near 0.

Table 3.4: Fixed-point Q16.16 on 32-bit representation.

31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
integer part																fractional part															

3.2.2 Floating-Point to Fixed-Point Quantization of a Deep Neural Network

In this work, the training is always performed using single-precision floating-point computation, that is, in the binary32 format. As training is done offline on a workstation, there is no need to perform it in fixed point. Despite this being feasible[62], it comes with additional challenges regarding gradient computation and it is not supported by the major deep learning frameworks. Since the training relies on floating-point computation, a conversion from a floating-point to a fixed-point representation must be performed before the deep neural network is deployed on the target. As the set of possible values is different between floating-point and fixed-point representations, this requires quantizing the weights of the deep neural network.

3.2.2.1 Uniform and Non-Uniform

Similarly to the works presented in Section 2.4, we also observed in our experiments that convolutional layer weights are close to Gaussian distributions, with a mean close to 0 when inputs are normalized. Such a distribution of weights for a convolutional layer kernel is shown in Figure 3.1. As a result, convolutional layer weights can be better represented using a non-uniform distribution of numbers. This is what floating-point numbers originally do to get a better precision around 0.

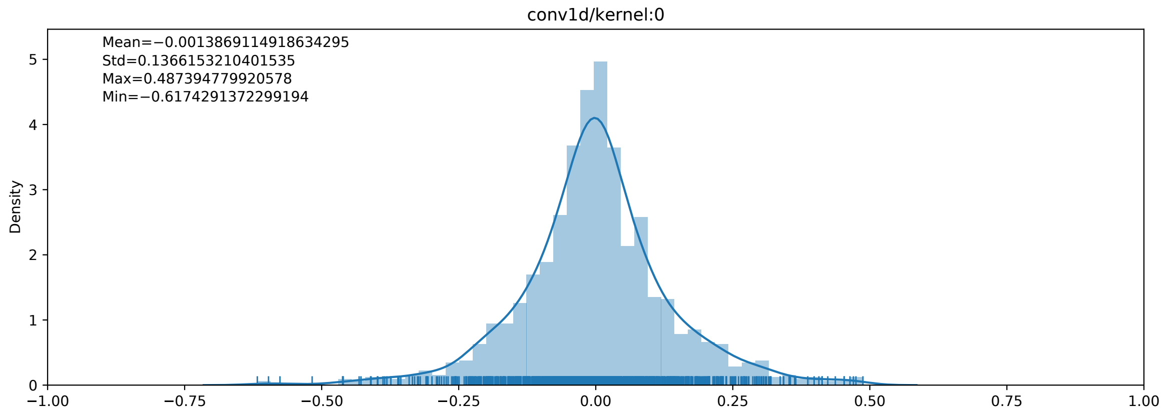


Figure 3.1: Example of the distribution of weights for a convolutional layer kernel.

However, as the goal is to perform fast computations, a uniform quantization is preferred. Non-uniform quantization would require performing additional transformations before using the microcontroller's instructions. This overhead can be non-negligible and lead to quantization performance lower than floating-point computations. To obtain a nonconstant quantization step, a nonlinear function must be computed, either online, or offline to generate a lookup table where operands for each operation are stored. In contrast, uniform quantization is based on a constant quantization step. Furthermore, the work previously presented in Section 2.4.1.3 does not bring an improvement on Cortex-M4-based microcontrollers. Indeed, this non-uniform quantization scheme encodes only the base 2 exponent of the weights [60]. Therefore, multiplications can be replaced by bit shifts. However, on a Cortex-M4 core, both multiplications and bit shifts take a single cycle. For these reasons, we rely on uniform quantization in this work.

3.2.2.2 Scale Factor

For simplicity's sake, the scale factor is a positive or negative power of two so that it can be computed using only left or right shifts. The scale factor is either fixed at a constant value (when it provides a sufficiently large range and enough precision), or chosen to represent the whole range of values (while avoiding data overflow), but at the cost of a lower precision for small numbers.

To reach the best quantization, the scale factor should in theory be chosen for each weight value. However, storing a scale factor for each value is obviously not a reasonable approach since it leads to an important memory overhead, defeating the purpose of implementing quantization to reduce memory usage. On the other hand, using a scale factor for the whole network can be too coarse to achieve a good overall quantization error. Instead, the scale factor can be made different for each layer. Another solution consists in using a scale factor for each filter of each layer. Although more complex to implement and introducing some overhead (scale factors of the layers must be stored in memory), this approach can slightly decrease the quantization error. So far, our implementation only allows a per-network and a per-layer scale factor. Inside a layer, the scale factor for the parameters and the scale factor for the activations can be different, however the weights and biases must use the same scale factor.

3.2.2.3 Conversion Method

To convert from floating-point to fixed-point, the method starts with finding the required number of bits m to represent the unsigned integer part:

$$m = 1 + \left\lfloor \log_2(\max_{1 \leq i \leq N} |x_i|) \right\rfloor \tag{3.1}$$

where x_i is an element of the floating-point vector x of length N (e.g. a weights or activations matrix). A positive value of m means that m bits are required to represent the absolute value of the integer part, while a negative value of m means there is no integer part and that the fractional part has m leading unused bits. This enables a greater precision to be obtained for vectors with numbers smaller than 2^{-1}, since the leading unused bits can be replaced instead by more trailing bits for precision.

From this we can compute the number of remaining bits n for the fractional part:

$$n = w - m - 1 \tag{3.2}$$

where w is the data type width (e.g., 16 bits). In this equation, 1 is subtracted to take into account the additional bit required to represent signed numbers. A positive value of n means that n bits are available to represent the fractional part. A negative value of n means that the fractional part cannot be represented, and the integer part cannot be represented to its full precision.

An element $xfixed_i$ of the fixed-point vector $xfixed$ is computed from the element x_i of the floating-point vector x as:

$$xfixed_i = \lfloor x_i \times 2^n \rfloor \tag{3.3}$$

And the scale factor s is defined as:

$$s = 2^{-n} \tag{3.4}$$

Quantization can be performed either after or during the training phase of a deep neural network. These methods are detailed in the following.

3.2.2.4 Post-Training Quantization

In post-training quantization, the neural network is entirely trained using floating-point computation (a single-precision format is assumed here). Once the training is over, the neural network is frozen, and the parameters are then quantized. The quantized neural network is then used to perform the inference, without any adjustments of the parameters.

The quantization phase introduces a error on each parameter as well as on the input, thus leading to a error on the activations. The accumulation of quantization errors at the output of the neural network can cause an incorrect class prediction of the input data, creating an accuracy drop compared to the non-quantized version. As the bit width of the values decreases, the quantization error increases, and the resulting accuracy typically decreases as well. In some situations, a slight increase in the quantization error can help the network generalize better over new data, inducing a slight increase in the accuracy over test data.

3.2.2.5 Quantization-Aware Training

The objective of the quantization-aware training (QAT) is to compensate for the quantization error by training a deep neural network using the quantized version during the forward pass. This could help to mitigate the accuracy drop to some extent.

In this work we decided to perform all the computations using a floating-point representation during the training phase. As shown in Figure 3.2, the inputs, weights and biases of each layer are quantized and clamped before actually performing the layer's computation. The layer's output is quantized and clamped after the computation, before reaching the next layer. The quantization operation is done following the method presented above. During the training phase, the range of values is reassessed each time to adjust the scale factor before performing the layer's computation. When doing inference only, the scale factor is frozen. However, due to limitations of the deep learning frameworks, the values are kept in floating-point representation, but discretized to emulate a fixed-point representation.

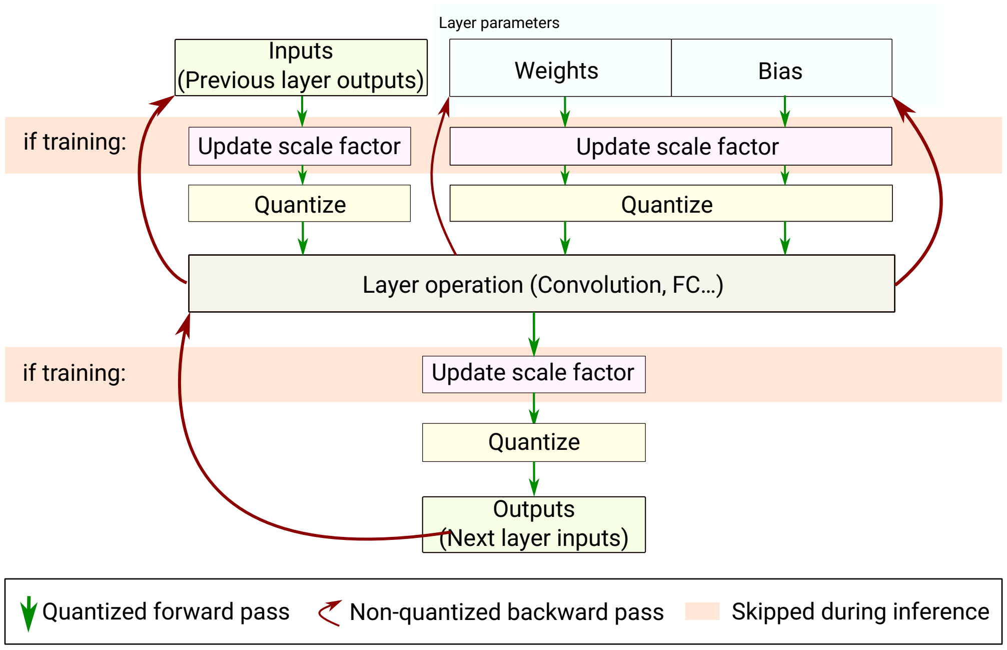


Figure 3.2: Quantization-aware training

The backpropagation uses a straight-through estimator, which means that it still relies on non-quantized values. However, the clamping to the maximum range that would be allowed by the fixed-point representation stays effective.

To stabilize the learning phase with the quantized version, and thus obtain better results on average, the deep neural network can be pre-trained without quantization in order to initialize the parameters to sensible values.

In the case of a convolutional neural network, the convolutional and fully connected layers require a quantization-aware training for the weights. Please note that batch normalization layers would also require quantization-aware training. However, quantization of batch normalization layers has not been implemented yet. As an alternative, the batch normalization layer can be fused with the previous convolutional layer[183], but then quantization-aware training will not be able to take advantage of the statistics of the current batch.

Layers that do not contain weights but that may change the range of the outputs must have their outputs quantized. For example, average pooling layers may reduce the range while increasing the precision. In this case, one bit of precision can be added to the fractional part while being removed from the integer part. For consistency's sake, the max pooling layers also have their outputs quantized, even though there is no need for additional precision on the fractional part. Conversely, the element-wise addition layers requires quantization. The range of their outputs can increase after adding two large numbers.

It is similar for the ReLU activation when it is considered as a separate layer. However, it can also be fused with the previous layer, this is the case for our inference program deployed on the target. While we do not use asymmetric quantization for activations and therefore we cannot reclaim the sign bit for more precision after a ReLU activations, fusing the ReLU activation can still reduce the dynamic range in case of negative values that are larger in absolute value than the positive ones.

3.3 Proposed Framework for the Deployment of Deep Neural Networks

After the network has been trained and optionally quantized, the objective is to deploy it onto a microcontroller to perform the inference on the target platform. The deep neural network deployment involves the following phases:

- exporting the weights of the deep neural network and encoding them into a format suitable for on-target inference,
- generating the C inference program according to the topology of the deep neural network,
- compiling the inference program for the target processor, and
- uploading the inference program with the weights onto the microcontroller's ROM.

As mentioned in Section 2.4.3, existing tools for quantized neural networks had some drawbacks that motivated the development of our own framework at the beginning of the work on this thesis. This framework addresses the following issues:

- most open-source frameworks do not support non-sequential topologies,
- frameworks that support convolutional neural networks are proprietary or too complex to be easily modified and extended,
- other frameworks do not provide 16-bit quantization (at the time of development, mainly in 2020, TensorFlow Lite for Microcontrollers did not support 16-bit activations),
- some frameworks are dedicated to a limited family of hardware targets,
- most frameworks do no support PyTorch models.

In this work, we aim to provide a framework, called MicroAI, that is easy to extend and modify, and that allows for a complete pipeline from the neural network training to the deployment and evaluation on a microcontroller. Additionally, this framework must provide a lightweight runtime on the microcontroller to reduce the memory and computation overhead. Finally, our objective is to provide a level of a performance close to existing solutions. MicroAI is built in two parts:

1. a neural network training code that relies on Keras or PyTorch, and
2. a conversion tool that takes a trained Keras or PyTorch model and produces a portable C code for the inference.

Both parts are written in Python since it is the most popular programming language to build deep neural networks and it easily interfaces with existing frameworks, libraries and tools.
The initial public version (v1.0) of the open-source MicroAI software framework[179, 184] was released in September 2021 and is available online at https://bitbucket.org/edge-team-leat/microai_public and archived in Zenodo at https://doi.org/10.5281/zenodo.5507396. The description of the software framework along with the quantization and embedded execution results were published in the MDPI Sensors 2021 Volume 21(9) journal[185].

3.3.1 General Flow

As seen in Figure 3.3, MicroAI provides an interface to automatically train, deploy and evaluate an artificial neural network model.

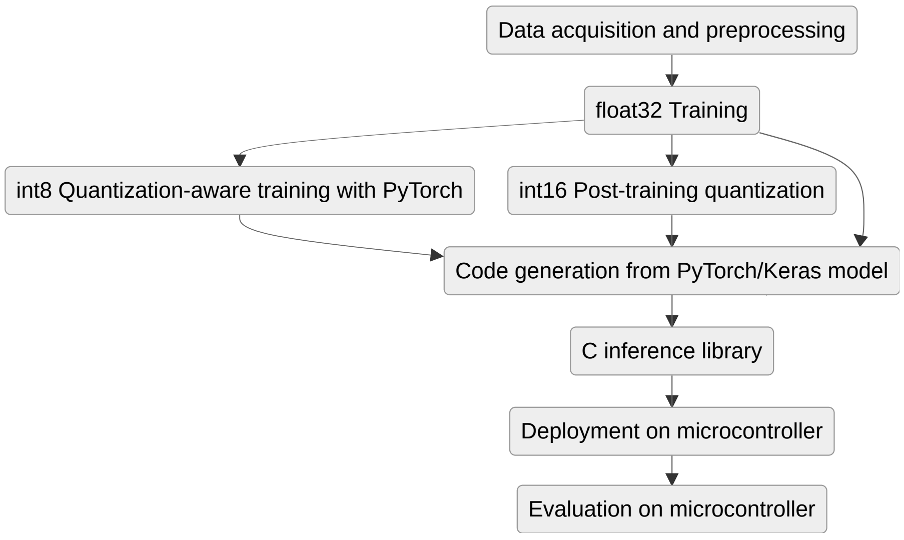

Data acquisition and preprocessing
float32 Training
int8 Quantization-aware training with PyTorch
int16 Post-training quantization
Code generation from PyTorch/Keras model
C inference library
Deployment on microcontroller
Evaluation on microcontroller

Figure 3.3: MicroAI general flow for neural network training, quantization and evaluation on embedded target.

A configuration file written in TOML [186] is used to describe the whole flow of an experiment:

- the number of iterations for the experiment (for statistical purposes),
- the dataset to use for training and evaluation,
- the pre-processing steps to apply to the dataset,
- the framework used for training,
- the data augmentation transformations to apply during training,
- the configuration of the various neural network models to train, deploy and evaluate,

- the configuration of the optimizer,
- the post-processing steps to apply to the trained model (including quantization),
- the target configuration for deployment and evaluation.

An example of a TOML configuration file is provided in Section A.1 of Appendix A along with explanations about various aspects of the configuration in Section A.2.

The three main steps, training, deployment and evaluation, are described in the following. The commands used to trigger them are available in Section A.3 of Appendix A

3.3.2 Training

For the training phase, MicroAI is simply a wrapper around Keras or PyTorch.

A dataset requires an importation module to be loaded into an appropriate data model. Dataset importation modules for UCI-HAR, SMNIST and GTSRB (described in Section 3.4, among others) are included and can easily be extended to new datasets.

To make use of a deep neural network architecture in the model configuration, it must be first described according to the training framework API in use. The description of the model is a template where parameters can be set by the user in the TOML configuration file.

MicroAI provides the following neural network architectures for both Keras and PyTorch:

- MLP: a multi-layer perceptron with a configurable number of layers and neurons,
- CNN: a 1D or 2D convolutional neural network with configurable number of layers, filters, kernel and pool size, and number of neurons per fully connected layer for the classifier,
- ResNet: a 1D or 2D residual neural network (v1) with convolutional layers. The number of blocks and filters per layer, stride, kernel size, and optional BatchNorm can be configured,
- DSOM: a single dynamic self-organizing map[165],
- DLSOM: a CNN or ResNet feature extractor followed by a dynamic self-organizing map, described in Section 5.2.4 of Chapter 5.

The activation function used is the Rectified Linear Unit (ReLU) as it is one of the least computationally intensive function and still provides the required non-linearity for a deep neural network to approximate any function.

3.3.3 Deployment

MicroAI can deploy a trained model to perform inference on a target using either STM32Cube.AI, TensorFlow Lite for Microcontrollers or our own code generation tool.

STM32Cube.AI can be used for all STM32 platforms, and support for the Nucleo-L452RE-P with an STM32L452RE microcontroller is included. Support for other platforms using a STM32 microcontroller can be added by providing a sample STM32CubeIDE project including the X-CUBE-AI package. STM32Cube.AI does not support microcontrollers outside the STM32 family.

TFLite Micro is a portable library that can be included in any project. Therefore, it could be used for any 32-bit microcontroller. However, only integration with the SparkFun Edge platform with an Ambiq Apollo3 microcontroller is included in our framework so far.

Similarly, our C code generation tool produces a portable library that can be included in any project. So far, only support for the Nucleo-L452RE-P and the SparkFun Edge boards is included. Other platforms can be added by providing project files that call the inference code and a module that interfaces with the build and deployment tools for that platform.

Please note that STM32Cube.AI and TFLite Micro cannot take a trained PyTorch model as an input to deploy onto a microcontroller. The trained PyTorch model must therefore be converted to a Keras model prior to the deployment. Our C code generation tool can take either a trained Keras or a trained PyTorch model, thus this conversion is not required.

3.3.4 C Code Generation Tool from Trained Model

As part of MicroAI, we developed a C code generation tool. The tool can automatically generate a portable C library for inference from a trained Keras or PyTorch model. It can also be used independently of the training part of the MicroAI framework.

Both 1D and 2D models are supported, with the following layers:

- Add
- AveragePooling1d/2d
- BatchNormalization1d/2d
- Conv1d/2d
- FullyConnected: Dense (Keras)/Linear (PyTorch)
- Flatten
- MaxPooling1d/2d
- ReLU
- Softmax
- ZeroPadding1d/2d (Keras)
- DSOM

Layers can have multiple inputs such as the *Add* layer, thus allowing residual neural networks (ResNet) to be built. Sequential convolutional neural networks or multi-layer perceptron models are also supported.

3.3.5 C Code Generation Process

The process for generating the inference code from a trained model is shown in Figure 3.4. The "..." illustrates the possibility to add support for a different deep learning framework or a different target by adding a graph translation module or code generation module, respectively.

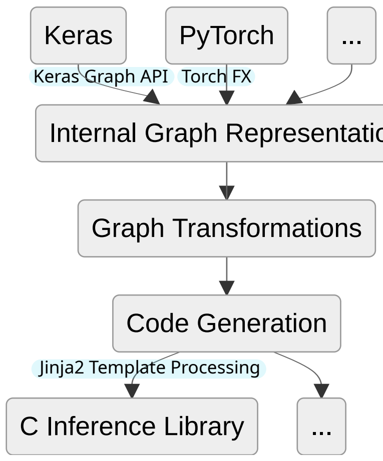


Figure 3.4: Code generation process.

This tool first parses the Keras model using Keras 2.6.0 API thanks to Keras' graph representation of the model, or the PyTorch model using the FX module[187] to trace the model and build a graph of the layers. In both cases, an internal representation is generated as a graph with generic layers, independent from the original training framework.

Then, a series of transformations is performed to produce a graph better suited for deployment on a microcontroller:

- remove Dropout layers, unused during inference,
- combine ReLU activation layers with the previous Conv1D, MaxPooling1D, Dense/Linear or Add layer,
- Keras only: combine ZeroPadding1D layers with the next Conv1D layer,
- Keras only: convert BatchNorm[188] weights from the mean μ, the variance V, the scale γ, the offsets β and ϵ to a multiplicand w and an addend b using the following formula:

$$w = \frac{\gamma}{\sigma} \tag{3.5}$$

$$\sigma = \sqrt{V + \epsilon} \tag{3.6}$$

$$b = \beta - \frac{\gamma \times \mu}{\sigma} \tag{3.7}$$

  so that the output of the batch normalization layer can be computed as $y = w \times x + b$.

Then, for each node in the graph, the weights of the layer go through the quantization and conversion module if the conversion to fixed-point representation is enabled. The C inference function is generated from a Jinja2 [189] template file using the layer's configuration. Similarly, the layer's weights are converted into a C array from a Jinja2 template file. An example of a Jinja2 C template file for a fully-connected layer is provided in Section A.5 of Appendix A. Code generation is used to avoid runtime overhead of an interpreter such as the one used in TensorFlow Lite for Microcontrollers. Additionally, it allows the compiler to perform better optimizations. In fact, the layer's configuration is generated as constants or literals in the code, allowing the compiler to perform appropriate optimizations such as loop unrolling, using immediates when relevant and doing better register allocation. By default, GCC's `-Ofast` optimization level is enabled. So far, no special effort has been made to further optimize the source code for faster execution, except for the ability to make use of ARM's CMSIS-NN library[4] which optimizes the execution of some operations on some microcontrollers (see Section 3.3.7).

The allocator module aims to reduce RAM usage. To do so, it allocates the layer's output buffers in the smallest number of pools without conflicts (i.e. memory overlay). For each layer of the model, its output buffer is allocated to the first pool that satisfies two conditions: it must neither overwrite its input, nor the output of a layer that has not already been consumed. If there is no such available pool, a new one is created. It is worth noting that the allocator module does not yet try to optimize the allocation to minimize the size of each pool (this is a harder problem to solve). In consequence, the total RAM usage is not optimized.

Finally, the main function `cnn(...)` is generated. This function only contains the allocation of the buffers done by the allocator module and a sequence of calls to each of the layers' inference functions. The correct input and output buffers are passed to each layer according to the graph of the model. More details about the interface of the library are provided in Section A.6 of Appendix A.

3.3.6 Quantization and Fixed-Point Computation

The post-training quantization is performed by the quantization module itself. The scale factor is found for each layer according to the method described in Section 3.2.2.3, but it can also be specified manually for the whole network. The fixed-point coding used for all the weights is computed according to this method as well. The data type is converted from a floating-point (`float32`) to an integer data type, either `int8_t` for 8-bit quantization or `int16_t` for 16-bit quantization.

When performing quantization-aware training or post-training quantization using MicroAI's `QuantizationAwareTraining` module, the scale factors are determined during the training phase, according to the method presented in Section 3.2.2.3 as well. Therefore, the quantization module reuses them during the code generation. However, the weights are still in floating-point representation since the training phase only relies on floating-point computation. In consequence, the quantization module must still perform data type conversion.

Once the model is deployed and running on the target, the fixed-point computation can be done using an integer arithmetic and logic unit. The data type for the input and output of a layer is the same as the one used to store the weights. To avoid overflows, computation is done using a data type larger than the operands' data type. For example, if the data type of the weights and inputs is `int16_t`, then the intermediate results are computed and stored using an `int32_t` data type. The result is then scaled back to the correct output scale factor before saturating and converting it back to the original operands' data type.

In case of an addition or a subtraction, operands must be represented with the same number of fractional bits. This is not required for multiplication, but the number of bits allocated for the fractional part of the result is the sum of the number of bits for the fractional part of the two operands. Therefore, after a multiplication, the result must be scaled to the required output format by shifting the result to the right by the appropriate number of bits.

In Table 3.5, the required number of operations for the main layers of a residual neural network in our implementation are provided, along with the number of cycles taken for these operations. Enabling compiler optimizations generates some ARMv7E-M instructions, namely SMLABB that performs a multiply–accumulate operation in one cycle (instead of two cycles). However, the compiler does not make use of the SSAT operation that could allow saturating in one cycle. Instead, it uses the same instructions as a regular max operation, that is, a compare instruction and a conditional move instruction, thus requiring two cycles[177].

Table 3.5: Number of arithmetic and logic operations for the main layers of a residual neural network required to perform inference usnig fixed-point integers. f is the number of filters (output channels), s is the number of input samples, c is the number of input channels, k is the kernel size, n is the number of neurons and i is the number of input layers to the residual Add layer. Conv1D is assumed to be without padding and with a stride of 1.

	MACC (1 Cycle)	**Add (1 Cycle)**	**Shift (1 Cycle)**	**Max/Saturate (2 Cycles)**
Conv1D	$f \times s \times c \times k$	N/A	$2 \times f \times s$	$f \times s$
ReLU	N/A	N/A	N/A	$c \times s$
Maxpool	N/A	N/A	N/A	$c \times s \times k$
Add	N/A	$s \times c \times (i - 1)$	$s \times c \times i$	$c \times s$
FullyConnected	$n \times s$	N/A	$2 \times n$	n

3.3.7 Optimizations using CMSIS-NN

The ARM CMSIS-NN library[4] can further optimize the execution of deep neural network operations by:

- using specialized instructions,
- optimizing memory accesses,
- optimizing register allocation,
- statically unrolling important loops.

For example, when used on the Cortex-M4 core, the CMSIS-NN library makes use of the ARMv7E-M instruction set extension, also called DSP (Digital Signal Processing) instructions. These instructions provide optimized execution for some combined operations, and also provide SIMD (Single Instruction, Multiple Data)-like execution by processing two 16-bit or four 8-bit values from a 32-bit register at a time. In particular, the following single cycle instructions can be leveraged to speed up the execution[177]:

- `SMLAD`: performs two 16-bit multiply-accumulate operations in a 32-bit register,
- `SXTB16`: extracts two 8-bit values and extends them to signed 16 bits,
- `QSUB16`: performs two 16-bit subtractions and saturates to signed 16 bits,
- `QSUB8`: performs four 8-bit subtractions and saturates to signed 8 bits,
- `SSAT`: signed saturation to any bit position (also present in the regular ARMv7-M instruction set).

Section A.7 of Appendix A details the CMSIS-NN functions used in our framework and the optimizations they provide when CMSIS-NN is enabled.

3.4 Results

All the results presented in this section rely on the same model architecture, a ResNetv1-6 network with the layers shown in Figure 3.5. While the number of filters per layer f is the same for all layers, it is modified to adjust the number of parameters of the model. The convolutional and pooling layers are one-dimensional except for the GTSRB dataset. Image processing indeed requires two-dimensional layers.

For each experiment, the residual neural network is initially trained using 32-bit floating-point numbers (i.e., without quantization), and then evaluated over the testing set. This baseline version is depicted as *float32* in the figures shown in the following.

The *float32* neural network is quantized with fixed-point on 16-bit integers and is then evaluated without additional training. This version is depicted as *int16* in the figures shown hereafter. Quantization is performed using the Q7.9 format for the whole network, meaning the number of bits n for the fractional part is fixed to 9.

The *float32* neural network is also trained and evaluated with 8-bit fixed-point integers using quantization-aware training. This version is indicated as *int8* in the figures. In this case the choice of the fixed-point format can vary from layer to layer and is determined using the method introduced in Section 3.2.2.3.

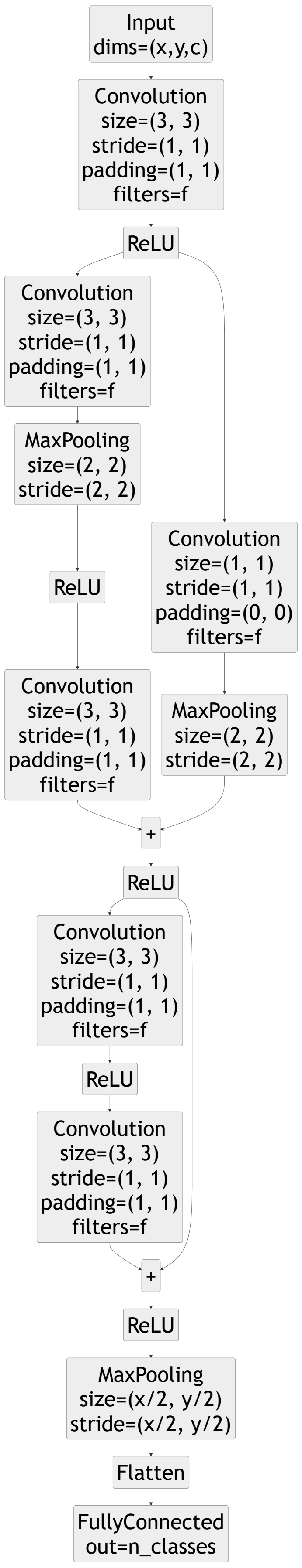


Figure 3.5: ResNet model architecture.

The SGD optimizer is used for all experiments. This choice has been motivated by the stability of the SGD optimizer, especially for the quantization-aware training. Training parameters are described below for each dataset. Additionally, training and testing sets are normalized using the z-score of the training set. It is worth noting that the Mixup [190] data augmentation method with $\alpha = 1.0$ is also used during training.

As it would require a large amount of time, the accuracy is not evaluated directly on the target. Only the inference time for the UCI-HAR dataset is measured on the target.

In the figures, each point represents an average over 15 runs. The error bars represent the the standard deviation for these runs.

3.4.1 Evaluation of the MicroAI Quantization Method

3.4.1.1 Human Activity Recognition dataset (UCI-HAR)

The University of California Irvine's hosted Human Activity Recognition dataset (UCI-HAR) [112] is a dataset of activities of daily living recorded using the accelerometer and gyroscope sensors of a smartphone. In this experiment, we use the raw data from the sensors divided into fixed time windows, rather than the precomputed features. The reason is that we want to perform real-time embedded activity recognition. To do so, it is preferable to avoid the features computation overhead for each inference before entering the deep neural network. Instead, the features are extracted by the convolutional neural network itself.

The dataset is divided into a training set and a testing set of 7352 and 2947 vectors, respectively. Each vector is a one-dimensional time series of 2.56 s composed of 128 samples sampled at 50 Hz, with 50% overlap between vectors. Each sample has 9 channels: 3 axes of total acceleration, 3 axes of angular velocity and 3 axes of body acceleration. Noise was filtered out with a median filter and a 3rd order Butterworth filter at 20Hz. Body acceleration was seperated using a Butterworth filter at 0.3Hz. Six different classes are available in the dataset: walking, walking upstairs, walking downstairs, sitting, standing and lying.

The initial training without quantization is performed using a batch size of 64 over 300 epochs. The initial learning rate is set to 0.05, the momentum is set to 0.9 and the weight decay is set to 5×10^{-4}. In order to stabilize the training, the learning rate is multiplied by 0.13 at epochs 100, 200 and 250. The quantization-aware training for fixed-point on 8-bit integers uses the same parameters.

As can be seen in Figure 3.6, for the UCI-HAR dataset, the same accuracy is obtained using a 16-bit quantization (*UCI-HAR int16*) or 32-bit floating-point (i.e., the baseline *UCI-HAR float32*), whatever the number of filters per convolution.

On the other hand, we observe that the 8-bit quantization causes a drop in accuracy that increases in magnitude up to 0.81% when the number of filters per convolution grows, even though quantization-aware training is used to mitigate this issue.

In Figure 3.7, we observe that the accuracy obtained using 8-bit and 16-bit quantization is similar only for deep neural networks that contains a reduced number of parameters, in other words a low memory footprint. As an example, for 16 filters per convolution, an 8-bit quantization leads to an accuracy of 92.41% while requiring 3958 memory bytes to store the parameters. When a 16-bit quantization is used, an accuracy of 92.46% can be achieved, but at the cost of an increase in the required memory for storing the parameters (7916 bytes).

As can be seen, when more than 24 filters per convolution are used, the 16-bit quantization clearly exhibits the best accuracy vs. memory ratio. For more than 48 filters per convolution, the 8-bit quantization provides an even worse ratio than the baseline.

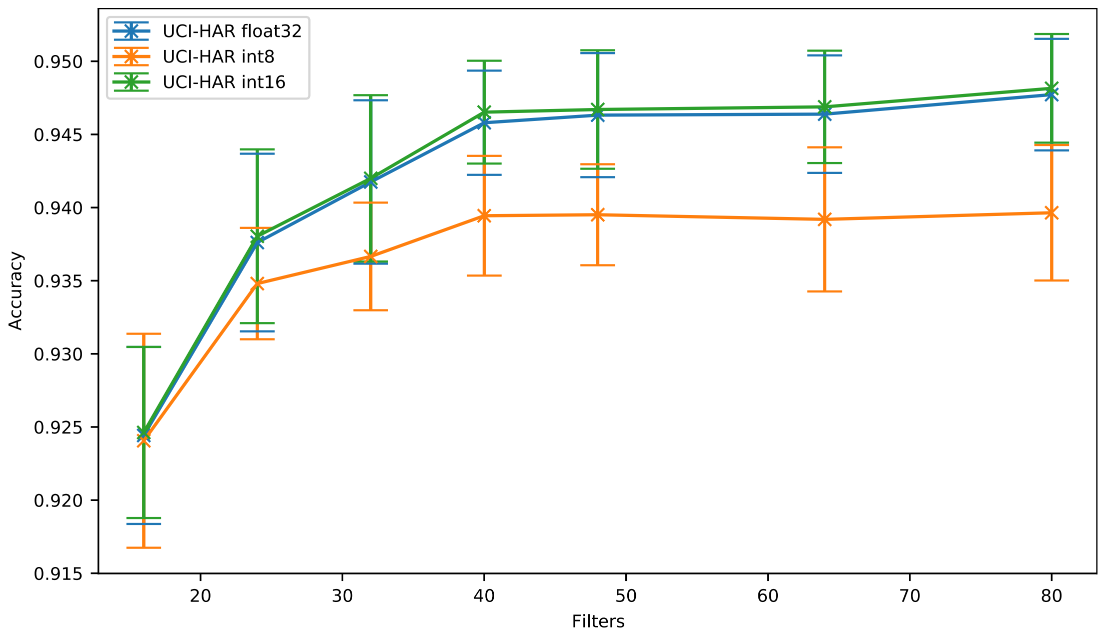


Figure 3.6: Human Activity Recognition dataset (UCI-HAR): accuracy vs. filters.

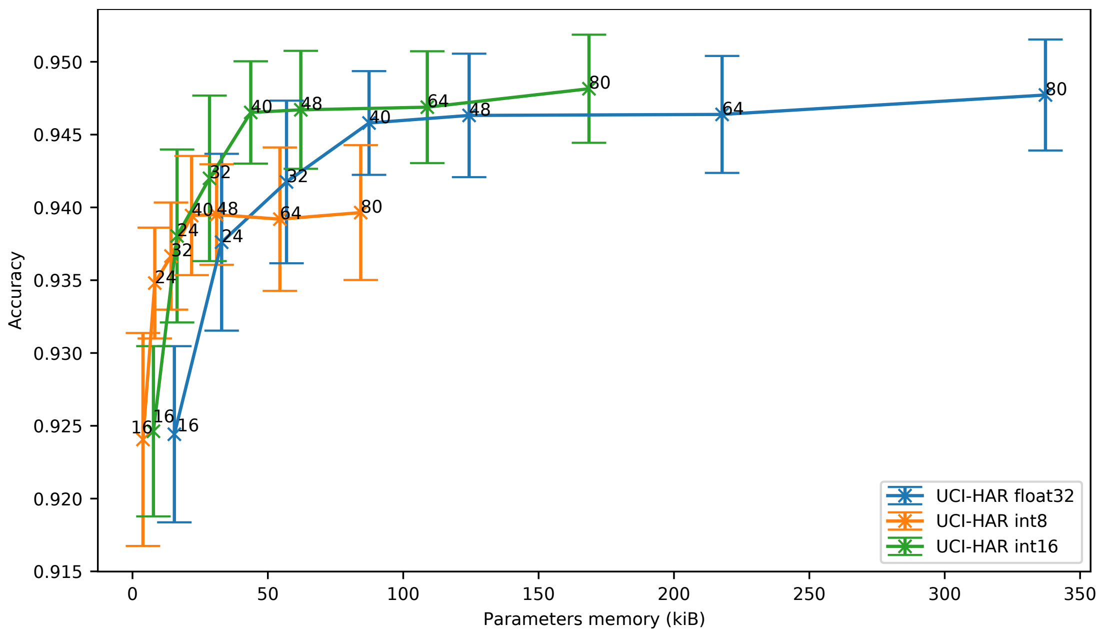


Figure 3.7: Human Activity Recognition dataset (UCI-HAR): accuracy vs. parameter memory.

During our experiments, it has also been observed that the 8-bit post-training quantization of TensorFlow Lite achieved better results compared to the 8-bit quantization-aware training provided by our framework. This is likely due to the combination of per-filter quantization, asymmetric range and non-power-of-two scale factor, as well as optimizations of TensorFlow Lite to avoid unnecessary truncation and thus loss of precision. We also observed that using 9 bits instead of 8 bits during the post-training quantization allows us to outperform the 8-bit TensorFlow Lite quantization performance. Some results showing this improvement can be seen in Figure 3.8. From these results, we can conclude that the slight additional precision brought by the combination of per-filter quantization, asymmetric range and non-power of two scale factor does in fact matter. In the future, implementing these methods in our framework seems therefore required to further reduce the accuracy loss of our 8-bit quantization.

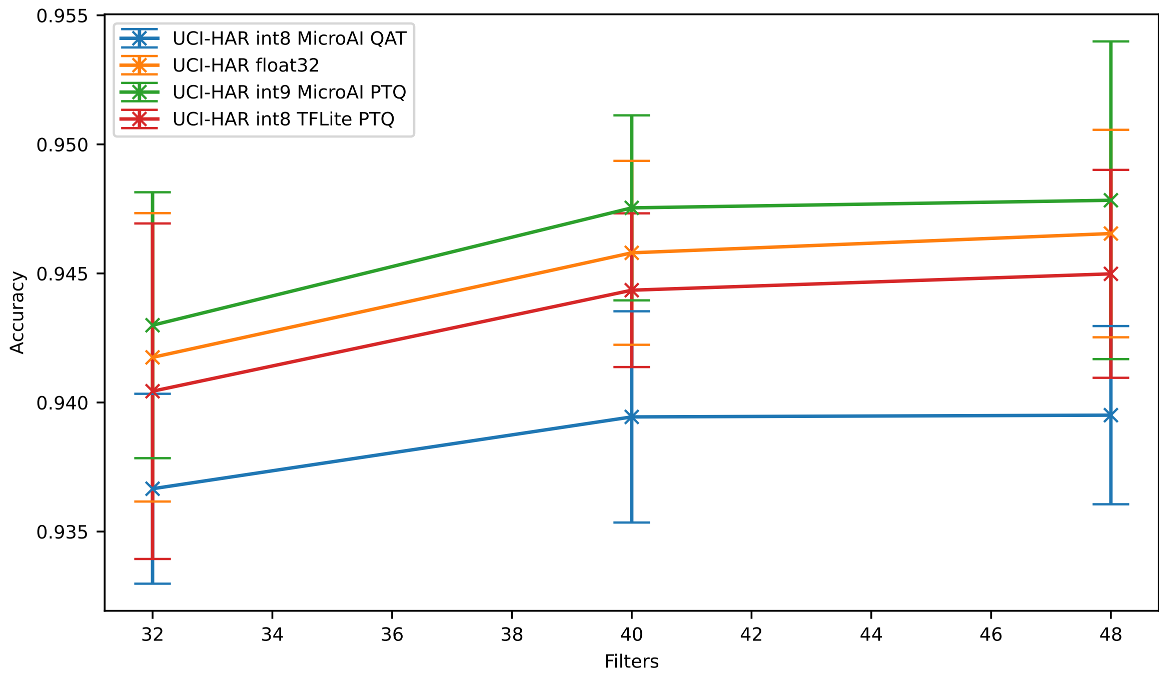


Figure 3.8: Accuracy vs. filters for baseline (*float32*), 8-bit post-training quantization from TensorFlow Lite (*int8 TFLite PTQ*), 8-bit quantization-aware training from our framework (*int8 MicroAI QAT*), and 9-bit post-training quantization from our framework (*int9 MicroAI PTQ*).

3.4.1.2 Spoken Digits Dataset (SMNIST)

Spoken MNIST is the spoken digits part of the written and spoken digits database for multi-modal learning [191].

This dataset is made of spoken digits extracted from the Google Speech Commands [192] dataset. The audio signal sampled at 16 kHz has been preprocessed to obtain 12 MFCC (Mel-Frequency Cepstral Coefficients) plus an energy coefficient using a window of 50 ms with 50% overlap over the audio files of approximately 1 s each, generating one-dimensional series of 39 samples with 13 channels. The dataset is divided into training and testing sets of 34,801 and 4107 vectors, respectively. Some samples are duplicated to obtain 60,000 training vectors and 10,000 testing vectors. There are 10 different classes for each digit, from 0 to 9.

The initial training, without quantization, uses a batch size of 256 over 120 epochs. The initial learning rate is set to 0.05, the momentum is set to 0.9 and the weight decay is set to 5×10^{-4}. The learning rate is multiplied by 0.1 at epochs 40, 80 and 100.

The quantization-aware training for fixed-point on 8-bit integers uses a batch size of 1024 over 140 epochs. Initial learning rate, momentum and weight decay are the same as for the initial training. Learning rate is multiplied by 0.1 at epochs 40, 80, 100 and 120.

As can be observed in Figure 3.9 and regardless of the number of filters, the 16-bit quantization (*SMNIST int16*) provides overall a similar accuracy compared to the floating-point baseline (*SMNIST float32*). On the other hand, the accuracy drops by up to 1.07% when the 8-bit quantization is used. However, the accuracy drop slightly decreases when 48 filters per convolution are used, and then stays around 0.5% or 0.6% for a higher number of filters.

In Figure 3.10, we can see that the 16-bit quantization is still the best solution in terms of memory footprint. Despite the fact that the 8-bit quantization stays closer to 16-bit quantization on SMNIST than on UCI-HAR, the 8-bit quantization does not provide any benefit over 16-bit quantization in terms of accuracy vs. memory ratio, even for small neural networks.

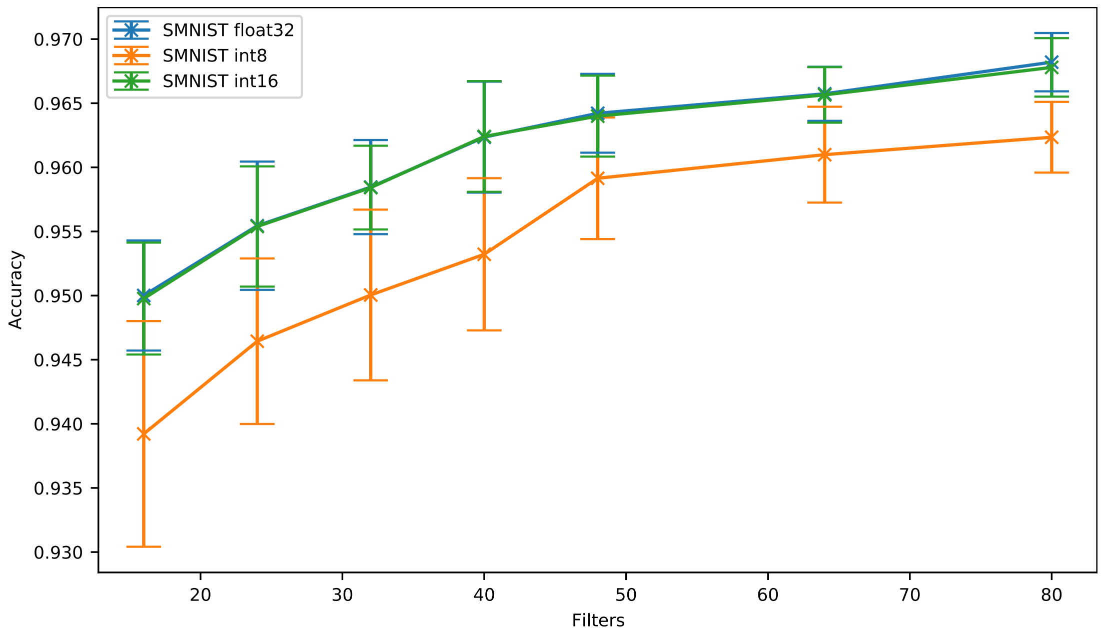


Figure 3.9: Spoken digits dataset (SMNIST): accuracy vs. filters.

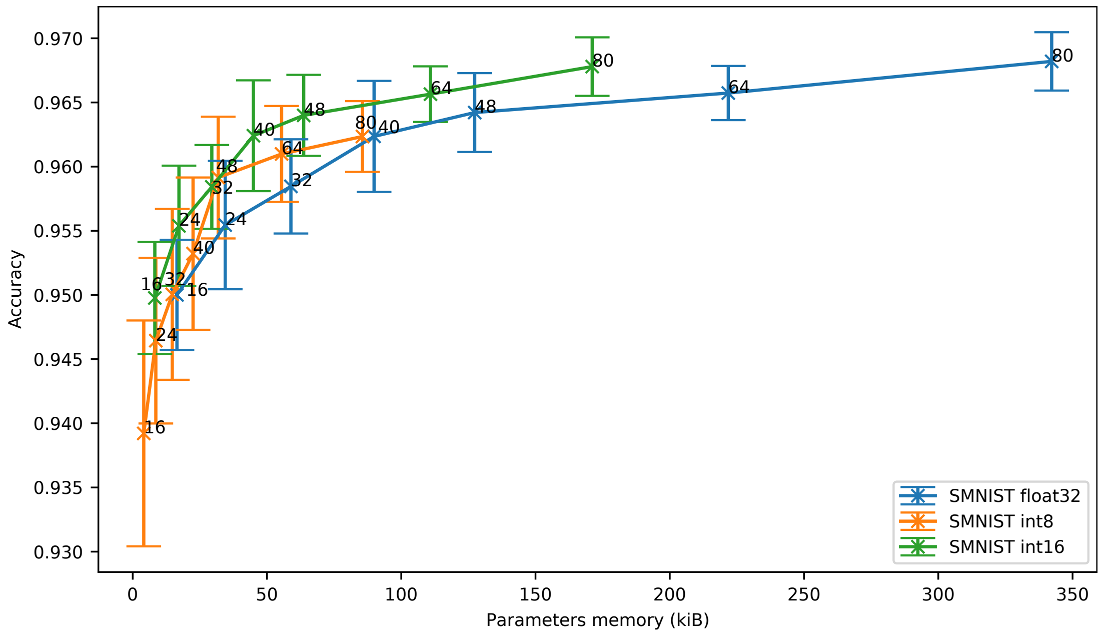


Figure 3.10: Spoken digits dataset (SMNIST): accuracy vs. parameter memory.

3.4.1.3 The German Traffic Sign Recognition Benchmark (GTSRB)

The German Traffic Sign Recognition Benchmark (GTSRB [193]) is a dataset containing various colour pictures of road signs. Image sizes vary between 15 × 15 to 250 × 250 pixels. In this experiment, the two-dimensional images were scaled to 32 × 32 pixels using bilinear interpolation and anti-aliasing, while keeping the 3 colour channels (red, green, blue). The dataset is divided into training and testing sets of 39,209 and 12,630 vectors, respectively. There are 43 different classes, one for each type of road sign in the dataset.

The initial training without quantization uses a batch size of 128 over 120 epochs. The initial learning rate is set to 0.01, the momentum is set to 0.9 and the weight decay is set to 5×10^{-4}. The learning rate is multiplied by 0.1 at epochs 40, 80 and 100.

The quantization-aware training for fixed-point on 8-bit integers uses a batch size of 512 over 120 epochs. The initial learning rate, momentum and weight decay are the same as for the initial training. The learning rate is multiplied by 0.1 at epochs 20, 60, 80 and 100.

The accuracy results obtained for 8- and 16-bit quantization and the 32-bit floating-point versions are shown in Figure 3.11 for different numbers of filters. As can be seen, the 16-bit quantization (*GTSRB int16*) provides an accuracy similar to the one obtained with the baseline (*GTSRB float32*). In the meantime, a drop in accuracy of up to 1.1% can be observed when the 8-bit quantization is used with this GTSRB dataset. However, as it has been observed with the SMNIST dataset, the accuracy drop is less important when the network has more filters (a drop of only 0.33% for 64 filters).

Moreover, even though the 8-bit quantization does not outperform the results obtained with the 16-bit quantization, Figure 3.12 shows that the 8-bit quantization can represent an interesting solution when a two-dimensional network is used on an image dataset.

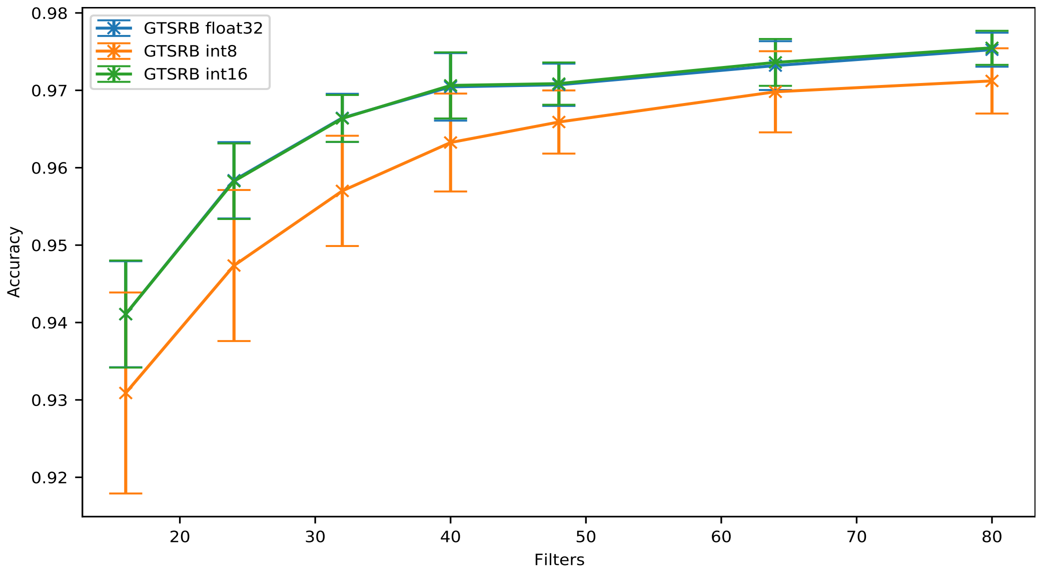


Figure 3.11: German Traffic Sign Recognition Benchmark: accuracy vs. filters.

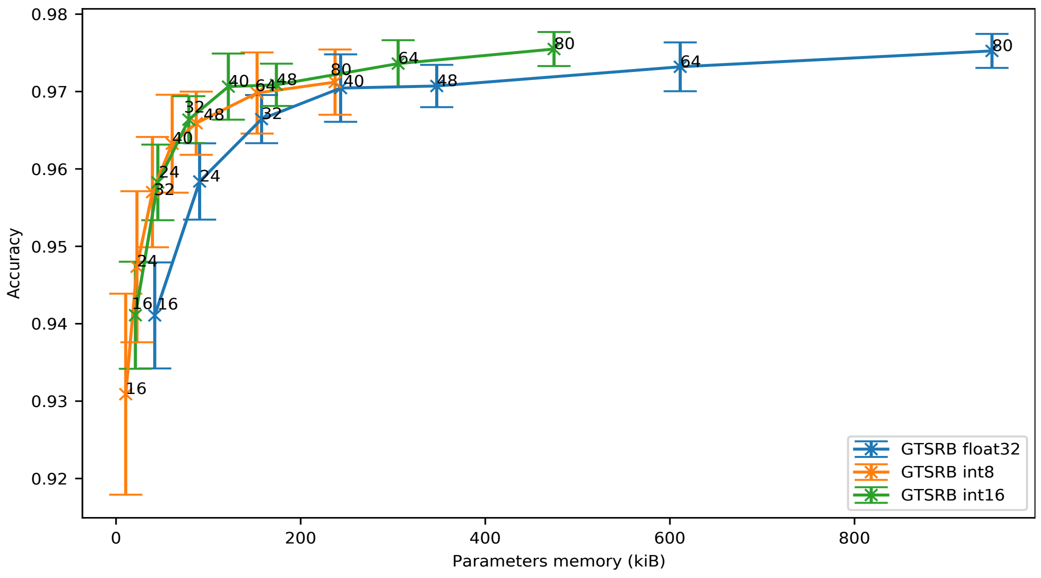


Figure 3.12: German Traffic Sign Recognition Benchmark: accuracy vs. parameter memory.

3.4.2 Evaluation of Frameworks and Embedded Platforms

In our experiments, two different targets have been used to deploy a deep neural network on a microcontroller: the SparkFun Edge and the Nucleo-L452RE-P. Both platforms are set to run at 48 MHz on a 3.3 V supply and their main specifications are summarized in Table 3.6.

Table 3.6: Embedded platforms main specifications.

Board	**Nucleo-L452RE-P**	**SparkFun Edge**
MCU	STM32L452RE	Ambiq Apollo3
Core	Cortex-M4F	Cortex-M4F
Max Clock	80 MHz	48 MHz (96 MHz “Burst Mode”)
RAM	128 kiB	384 kiB
Flash	512 kiB	1024 kiB
CoreMark/MHz	3.42	2.479
Run current @3.3 V, 48 MHz	4.80 mA	0.82 mA *

* After removing peripherals (Mic1&2, accelerometer …)

V_{DD_MCU} is set to 1.8 V for the Nucleo-L452RE-P platform and current measurement is taken from the I_{DD} jumper (where the voltage is 3.3V regardless). It does not have any on-board peripherals. On the SparkFun Edge board, the measure of the current is done using the power input pin of the board (after the programmer). The built-in peripherals were unsoldered from the board to eliminate their power consumption. The current consumption was measured using a Brymen BM857s auto-ranging digital multimeter configured in max mode. The energy results are based on this maximum observed current consumption and the supply voltage of 3.3 V.

As can be seen in Table 3.6, and even though both platforms are built around a Cortex-M4F core running at the same frequency, the SparkFun Edge board consumes considerably less power than the Nucleo-L452RE-P, while also having more Flash and RAM memory. Thi is due to the subthreshold operation of the Ambiq Apollo3 microcontroller. However, results obtained with the CoreMark benchmark show that the Ambiq Apollo3 microcontroller is slower than the STM32L452RE. It is worth noting that the CoreMark results have been measured on the Ambiq Apollo3 microcontroller, while they have been taken from the datasheet for the STM32L452RE microcontroller.

The deep neural network used in our experiments is the ResNetv1-6 described in Section 3.4. This network has been trained on the UCI-HAR dataset presented in Section 3.4.1.1. The inference time is measured from 50 test vectors from the testing set of UCI-HAR on both microcontrollers. TensorFlow Lite for Microcontrollers version 2.4.1 has been used to deploy the deep neural network on the SparkFun Edge board, while STM32Cube.AI version 5.2.0 has been used to deploy it on the Nucleo-L452RE-P board, both for the 32-bit floating-point and fixed-point on 8-bit integers inference. Our MicroAI framework is used to deploy the deep neural network on both platforms for 32-bit floating-point, fixed-point on 16-bit integers and fixed-point on 8-bit integer inference. It is worth noting that optimizations for Cortex-M4F provided by CMSIS-NN are used by all the frameworks for 8- or 16-bit inference. Our framework uses CMSIS-NN version 3.1.0 from the CMSIS 5.9.0 package. A comparison between enabling and disabling CMSIS-NN with our framework is provided in Section 3.4.3. The main characteristics of the frameworks are summarized in Table 3.7.

Table 3.7: Embedded AI frameworks.

Framework	**STM32Cube.AI**	**TFLite Micro**	**MicroAI**
Source	Keras, TFLite, …	Keras, TFLite	Keras, PyTorch
Validation	Integrated tools	None	Integrated tools
Metrics	ROM/RAM footprint, inference time, MACC	None	ROM/RAM footprint, inference time
Portability	STM32 only	Any 32-bit MCU	Any 32-bit MCU
Built-in platform support	STM32 boards (Nucleo, …)	32F746GDiscovery, SparkFun Edge, …	SparkFun Edge, Nucleo-L452RE-P
Sources	Private	Public	Public
Data type	`float`, `int8_t`	`float`, `int8_t`	`float`, `int8_t`, `int16_t`
Quantized data	Weights, activations	Weights, activations	Weights, activations
Quantizer	Uniform (from TFlite)	Uniform	Uniform
Quantized coding	Offset and scale	Offset and scale	Fixed-point $Qm.n$

To compare software and hardware platforms, only the results with 80 filters per convolution are analyzed below. Nevertheless, detailed results with less than 80 filters are presented in the tables of Appendix B to highlight how fast and efficient a small deep neural network can be when deployed on a constrained embedded target.

In Figure 3.13, we can observe that TFLite Micro has a higher memory overhead than STM32Cube.AI, while MicroAI exhibits a slightly lower memory overhead than STM32Cube.AI.

The inference time obtained for both platforms and the different deployment tools is illustrated in Figure 3.14. As can be seen, STM32Cube.AI with the 8-bit inference running on the Nucleo-L452RE-P board provides the best solution as it requires only 352 ms for one inference. Our framework comes very close at 368 ms and 356 ms when running on the SparkFun Edge and the Nucleo-L452RE-P, respectively. In this configuration, TensorFlow Lite for Microcontrollers requires 592 ms for one inference on the SparkFun Edge board.

When using fixed-point on 16-bit integers, our framework provides a slightly higher inference execution time than with 8 bits. 16-bit integers indeed require more memory reads to fetch the values. Moreover, the ReLU activation can only process two elements at a time as explained in Section A.7. On the Nucleo-L452RE-P, we can observe that the inference time for one input is 403 ms and 404 ms when running on the SparkFun Edge and the Nucleo-L452RE-P, respectively. As the other frameworks do not provide a full int16 execution mode, it is not possible to compare them.

Figure 3.14 also shows that, whatever the tool and target, the 32-bit floating-point inference is slower than with 16- or 8-bit quantization. Our framework requires 1561 ms and 1512 ms for one inference on the SparkFun Edge and the Nucleo-L452RE-P boards, respectively. STM32Cube.AI requires 1387 ms for one inference on the Nucleo-L452RE-P board. Our framework therefore exhibits a comparable performance to STM32Cube.AI. Finally, we can see that TensorFlow Lite for microcontrollers on the SparkFun Edge board provides lower performance, requiring 2087 ms to perform one inference. It is important to remember that there is no optimization possible with CMSIS-NN for floating-point operations.

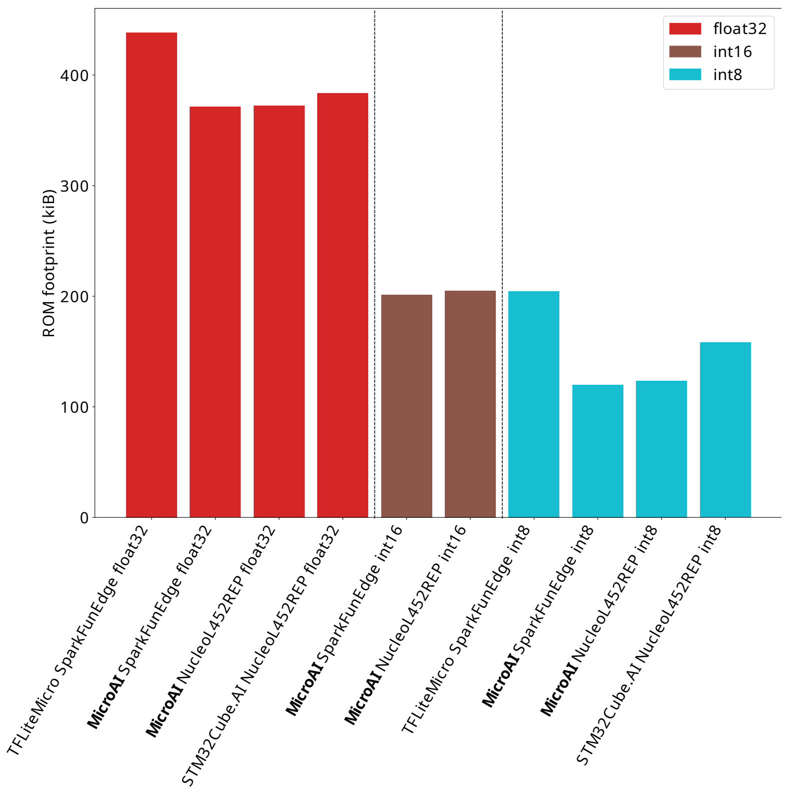


Figure 3.13: ROM footprint for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

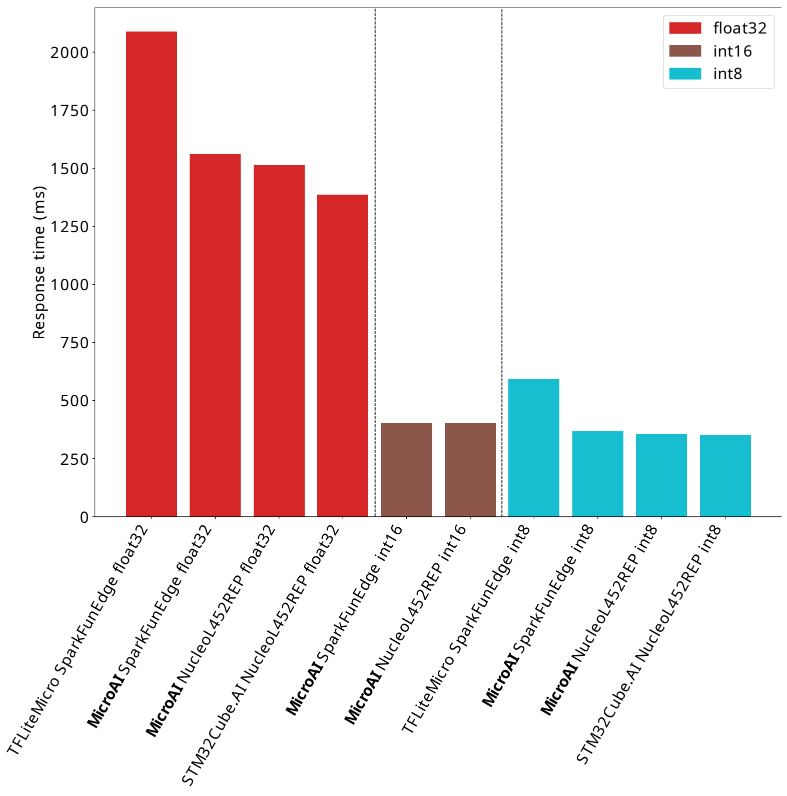


Figure 3.14: Inference time for 1 input for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

To conclude, and as outlined in Figure 3.15, we can say the SparkFun Edge board provides a better power efficiency than the Nucleo-L452RE-P platform whatever the framework and the data type. The reason is that the SparkFun Edge board power consumption is approximately 6 times lower than the Nucleo-L452RE-P board. Our framework requires 0.276 µWh and 0.303 µWh on the SparkFun Edge board for inference with fixed-point on 8-bit and 16-bit integers, respectively. In contrast, using the SparkFun Edge board and TensorFlow Lite for Microcontroller with fixed-point on 8-bit integers, one inference requires 0.445 µWh of energy consumption. When 32-bit floating-point is used for inference on the SparkFun Edge board, our framework provides a better energy efficiency than TensorFlow Lite for Microcontrollers as it requires 1.174 µWh instead of 1.569 µWh.

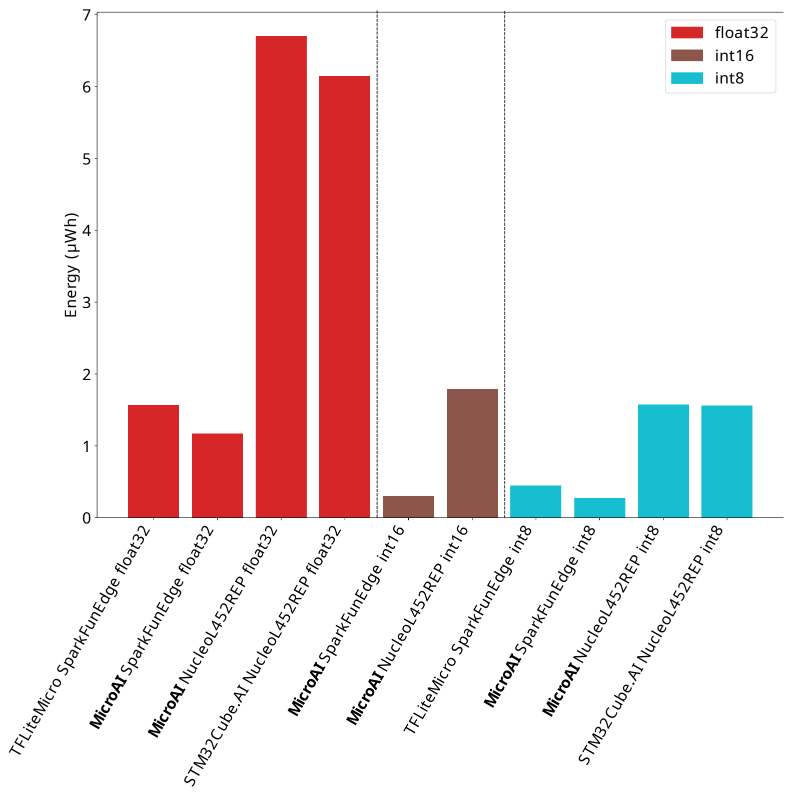


Figure 3.15: Energy consumption for 1 input for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

Concerning the energy consumed by the Nucleo-L452RE-P board, our framework requires 1.578 µWh, 1.794 µWh and 6.700 µWh for one inference using 8-bit and 16-bit fixed-point integers, and 32-bit floating-point, respectively. In comparison, 6.146 µWh are required for one inference when the STM32Cube.AI framework is used with 32-bit floating-point. Finally, we can see that the required energy for one inference when using STM32Cube.AI with fixed-point on 8-bit integers is 1.560 µWh on the Nucleo-L452RE-P. This amount of energy is similar to the one obtained with TensorFlow Lite for Microcontrollers on the SparkFun Edge board when performing floating-point inference.

3.4.3 Comparison Between Enabling and Disabling CMSIS-NN Optimizations with MicroAI

Figure 3.16 shows that CMSIS-NN does not require a significant amount of additional ROM to store the inference code. For example, on the SparkFun Edge board, the ROM occupation increases from 202.699 kiB to 206.012 kiB for the 16-bit quantized network, only a 1.6% increase. However, while it is not detailed here, the RAM usage grows when using CMSIS-NN since it requires additional temporary buffers.

However, the inference time significantly decreases after enabling CMSIS-NN optimizations as illustrated in Figure 3.17. As can be observed, the inference time decreases by 61.3% from 1042 ms to 403 ms on the SparkFun Edge board with the 16-bit quantized network. The same improvement is also observed for the 8-bit quantized network on the SparkFun Edge board, dropping from 1003 ms to 368 ms, a reduction of 63.3%. The results are similar with the Nucleo-L452RE-P board, going from 1225 ms to 405 ms and 1034 ms to 356 ms for the 16- and 8-bit quantized networks, respectively.

As a result, and as it can be seen in Figure 3.18, the energy decreases by the same factor. The SparkFun Edge board still requires significantly less energy thanks to its much lower power consumption.

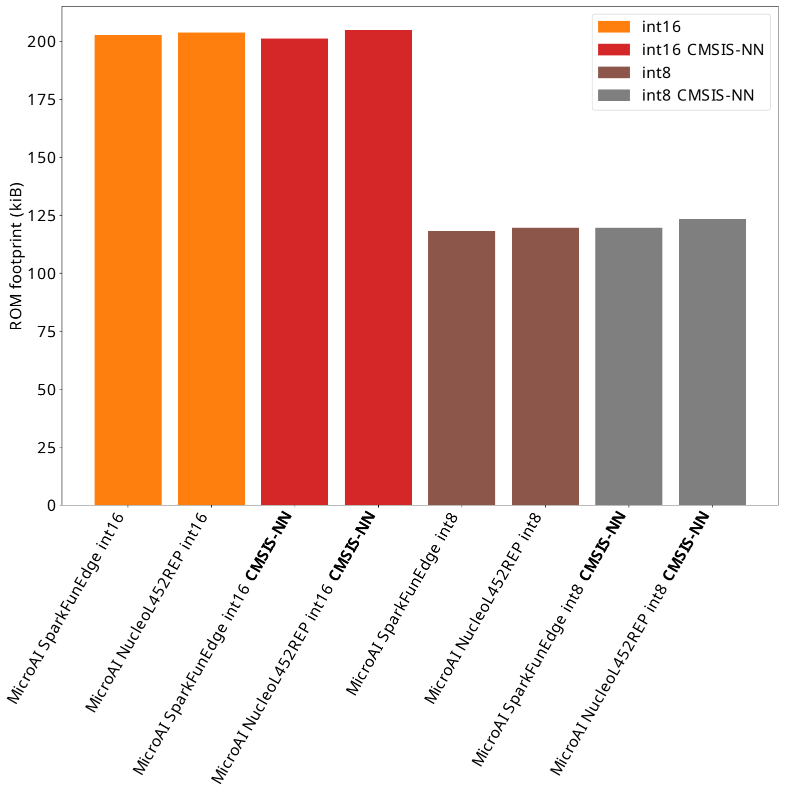


Figure 3.16: ROM footprint for MicroAI with and without CMSIS-NN with 80 filters per convolution.

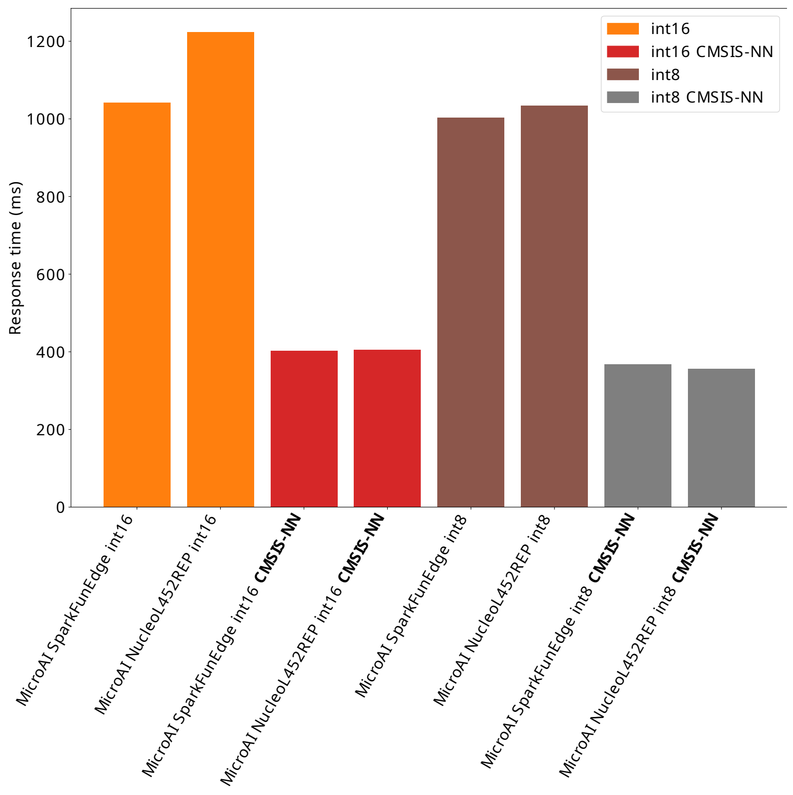


Figure 3.17: Inference time for 1 input for MicroAI with and without CMSIS-NN with 80 filters per convolution.

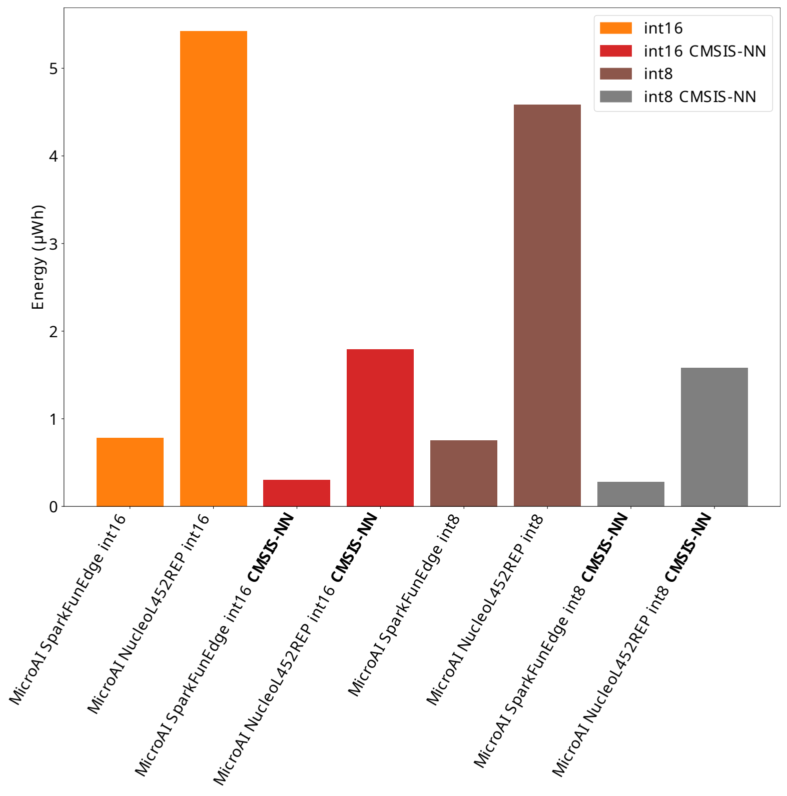


Figure 3.18: Energy consumption for 1 input for MicroAI with and without CMSIS-NN with 80 filters per convolution.

3.5 Discussion

First, a high variance is observable when we compare the accuracy results obtained on the three datasets versus the model size. This variability makes it difficult to draw any definitive conclusions. However, there is a trend in our results that provides some insights into the performance for each experiment.

As it has been shown, execution using fixed-point on 8-bit and 16-bit integers provides a significant decrease in the inference time, thus reducing the average power consumption as well. As power consumption is key in embedded systems, shorter inference times are interesting as they make reducing the microcontroller's operating frequency or putting the microcontroller in sleep mode for a longer period between two inferences possible. In addition, execution using 8-bit and 16-bit integers also provides a significant reduction in memory footprint. The memory required for the model parameters is divided by 4 and 2 for for 8-bit and 16-bit quantization, respectively. It is worth noting that the RAM usage, which is not illustrated here, is also reduced since the activation buffers are also quantized on 8 or 16 bits.

Our results also show that performing inference using quantization with fixed-point on 16-bit integers does not lead to an accuracy drop, whatever the considered test case. Moreover, inference using 16 bits does not require quantization-aware training to achieve such results. As both the power consumption and the memory footprint can be decreased, fixed-point quantization on 16-bit integers is therefore always preferable to 32-bit floating-point inference.

Conversely, 8-bit quantization does not provide a substantial improvement over 16-bit quantization. Moreover, 8-bit quantization requires performing quantization-aware training. It is worth noting that 8-bit quantization-aware training introduces more variance in the results over the baseline, and is also more sensitive to a change of training parameters. To reduce the variance, it is preferable to use an optimizer such as SGD with conservative parameters, instead of optimizers such as Adam or RAdam, to reduce the variance of the results, even though it may reach a lower maximum accuracy.

Another benefit of 16- or 8-bit quantization is that SIMD instructions can be used with some classes of microcontrollers. These instructions enable an improvement of the inference time and thus further reduce the power consumption. For example, such instructions allow performing 2 multiply–accumulate operations with 16-bit operands and a common accumulator (*SMLAD*) in a single cycle. Furthermore, a 16-bit quantization scheme can be used with our framework. As shown in the results, 16-bit quantization provides a good compromise between accuracy, inference time and memory footprint, without requiring additional work on quantization-aware training.

The results obtained on inference time clearly show that both the software and hardware platforms have a substantial impact on energy efficiency. STM32Cube.AI offers a well-optimized inference engine in terms of execution time, both in floating-point and fixed-point on integers. Our results show that TensorFlow Lite for Microcontrollers is slower than STM32Cube.AI in both cases. For the floating-point inference, our framework is in between these two software platforms and is only slightly slower than STM32Cube.AI. For fixed-point inference, our framework offers a performance level similar to STM32Cube.AI.

The Ambiq Apollo3 microcontroller of the SparkFun Edge board is much more energy efficient than the STM32L452RE microcontroller of the Nucleo-L452RE-P board. Running TensorFlow Lite for Microcontrollers or our framework on the SparkFun Edge board provides better energy efficiency figures than running STM32Cube.AI or our framework on the Nucleo-L452RE-P board. In fact, our MicroAI framework running on the SparkFun Edge board achieves the best energy figures.

3.6 Conclusion

In this chapter, we presented MicroAI, a framework to perform quantization and then deployment of deep neural networks on microcontrollers. This framework represents an alternative to the STM32Cube.AI proprietary solution and TensorFlow Lite for Microcontrollers, an open-source but complex environment. Inference time and energy efficiency measured on two different embedded platforms demonstrated that our framework is a viable alternative to the aforementioned solutions to perform deep neural network inference. Our framework also introduces a fixed-point on 16-bit integer post-training quantization which is not available with the two other frameworks. We have shown that 16-bit fixed-point quantization provides an improvement over a 32-bit floating-point inference, while being competitive with fixed-point on 8-bit integer quantization-aware training. 16-bit fixed-point numbers provide a reduced inference time compared to floating-point inference. Moreover, the memory footprint is divided by two while keeping the same accuracy. The 8-bit quantization provides further improvements in terms of inference time and memory footprint but at the cost of a slight decrease in accuracy and a more complex implementation.

Work is still in progress to implement some optimization techniques for fixed-point on 8-bit integer inference. Three optimizations are especially targeted: per-filter quantization, asymmetric range and non-power-of-two scale factor. These optimizations would make our framework more competitive in terms of accuracy compared to TensorFlow Lite for Microcontrollers for 8-bit quantization. Another possible improvement for fixed-point inference consists in using 8-bit quantization for the weights and 16-bit quantization for the activations. TensorFlow Lite for Microcontrollers is currently in the process of implementing this technique. Mixed precision can also provide a way to reduce the memory footprint of layers that do not need a high-precision representation (using 8 bits for weights and activations), while keeping a higher precision (16-bit representation) for layers that need it. The CMix-NN [194] library already provides an implementation of convolution functions for various data type configurations (in 2, 4 and 8 bits). To further improve power consumption and memory footprint, binary neural networks can also be considered. However, to run them efficiently on microcontrollers, binary neural networks would need to be implemented using bit-wise operations on 32-bit registers. This way, as many as 32 computations could be performed in parallel.

Apart from quantization, other techniques can be used to improve the execution of deep neural networks on embedded targets. One of these techniques is the big/LITTLE DNN approach [195] where the inference is first done on a very small deep neural network. Then, if the confidence of the prediction is too low, inference is done using a larger deep neural network to reduce the confusion of the classification task. This technique allows a fast inference response time for most inputs, thus lowering the power consumption. In fact, it has been shown that the set of inputs that are difficult to classify, so requiring running the bigger deep neural network, is small. However, this approach does not decrease the memory footprint. Other techniques such as pruning can also be used to obtain a smaller deep neural network while keeping the same accuracy. When structured pruning [196] is used, for instance, entire filters are completely removed from the convolutional neural network model. This technique reduces both the memory footprint and the power consumption. Finally, other optimization techniques also consider new neural network architectures. One can cite for example the recently published MCUNet [3] framework with its TinyNAS tool that aims to identify the neural network model that provides the best trade-off between prediction performance and embedded constraints on a given target.

In the next chapter, the MicroAI framework will be used to train, quantize and deploy a deep neural network for human activity recognition on the smart glasses. This requires building a dataset from the smart glasses in order to train and evaluate the deep neural network.

Chapter 4

Human Activity Recognition on Smart Glasses

Contents

4.1 Introduction

In this work, human activity recognition is solved as a machine learning problem that predicts activities of daily living performed by a subject using sensors data. Various devices such as smartphones[112], wearables[123] or application-specific devices[117] can be used to collect data, some being more invasive than others. Our approach is based on an inertial measurement unit embedded in smart glasses. Smart glasses are less invasive than some other devices such as dedicated IMU devices or even smartphones, especially for elderly who often wear glasses. However, and to the best of our knowledge, there is no available and usable dataset for human activity recognition based on smart glasses. Moreover, data would vary from one device to another due to sensors having different orientations, ranges, accuracy, and sampling rates.

In this chapter, we present a new dataset[197] called UCA-EHAR with data collected from Ellcie Healthy's smart glasses [105]. Our dataset provides raw data collected from an accelerometer, a gyroscope and a barometer for 8 classes of activity performed by 20 subjects.

For privacy, connectivity and latency reasons, all the data processing related to human activity recognition is performed directly on the smart glasses. Therefore, the machine learning algorithm performing the classification task is executed on the smart glasses' microcontroller. In Chapter 3, we presented our open-source MicroAI framework for end-to-end training, quantization and deployment of deep neural networks on microcontrollers[185, 179]. In this work, MicroAI is used to deploy a deep neural network model performing human activity recognition on the smart glasses. Quantization with 8- and 16-bit fixed-point representations is used to optimize the memory footprint and the inference time, thus reducing the power consumption as well.

Section 4.2 presents the smart glasses used for collecting data and performing live inference. Section 4.3 details the dataset and the protocol used to collect and label the data. Section 4.4 describes the deep neural network architecture used to classify activities from our dataset as well as the training phase. In Section 4.5, classification results using our dataset are given and power consumption on the smart glasses is analysed. Section 4.6 showcases our human activity recognition application running on the smart glasses. Finally, Section 4.7 concludes this work.

4.2 Ellcie Healthy Smart Glasses

Ellcie Healthy smart connected glasses are a multiple-purpose wearable device designed for e-health and road safety applications such as driver drowsiness detection, fall detection for elderly people or human activity recognition to prevent a fall. The Ellcie Healthy smart connected glasses shown in Figure 4.1 contain infrared proximity sensors embedded inside the rims for oculography purposes.

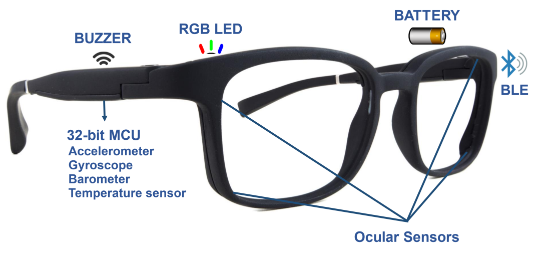


Figure 4.1: Ellcie Healthy Smart Glasses.

Other sensors such as a barometer, a thermometer, a triaxial accelerometer and a gyroscope are integrated within the frame temples. The accelerometer and the gyroscope are located inside the same inertial measurement unit component. The barometric sensor and the temperature sensor are located in another component. The accelerometer provides a tree-dimensional acceleration vector along the orthogonal coordinate system shown in Figure 4.2. When the glasses are placed onto a table for example, most of the acceleration vector modulus (i.e., the gravity) is projected onto the Z axis, approximately giving 9.81 $m{\cdot}s^{-2}$. Depending on how the subject is wearing the glasses, the shape of the nose and other physiological factors, the gravity may not be perfectly projected onto the Z axis.

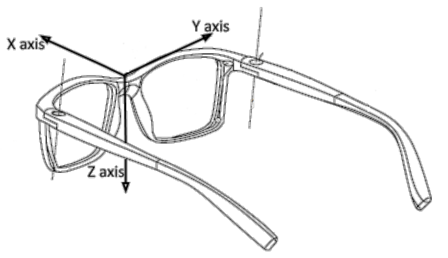


Figure 4.2: Accelerometer axes on Ellcie Healthy Smart Glasses.

The frame also includes a 32-bit microcontroller. The STM32L451RE microcontroller from STMicroelectronics has been chosen for its low power consumption while still being versatile. This microcontroller relies on a Cortex-M4F core running at 40 MHz in active mode and alongside 512 KiB of Flash memory and 160 KiB of SRAM. The microcontroller runs the Zephyr real-time operating system to handle the various concurrent tasks. When no active application is running, the microcontroller enters STOP2 mode after a timeout of a few seconds. This does not happen while the human activity recognition application is running. However, if no higher priority task is running, the Zephyr's idle task runs and calls the `WFI` instruction, which causes the microcontroller to enter a shallow sleep mode by turning off the core clock until an interrupt occurs, while other clocks and peripherals stay active. A Bluetooth Low Energy (BLE) transceiver is integrated inside the frame to enable wireless communication with a gateway (typically a smartphone). Finally, a 350 mWh lithium polymer battery placed on the left temple of the frame provides the energy to the whole system using a flat flexible cable. This cable allows energy and data to flow back and forth through the bridge, the rims and the temples. Embedded algorithms, signal processing and data collection can therefore be directly performed on the smart glasses to provide health information to users. Alerts can be triggered when a risk event (e.g., driver drowsiness) is detected.

4.3 Dataset for Human Activity Recognition on Smart Glasses

To address the lack of usable data for human activity recognition using smart glasses, we have built a dedicated dataset called UCA-EHAR. To build the UCA-EHAR dataset, we have enrolled 20 adult subjects, 8 women and 12 men (30.6 y.o average; 12 y.o standard deviation). Adults or children below 1.60 m of height as well as people with disabilities such as limping or backache were not included in our dataset.

UCA-EHAR contains 8 distinct classes:
STANDING, SITTING, WALKING, LYING, WALKING_DOWNSTAIRS, WALKING_UPSTAIRS, RUNNING, and DRINKING. The choice of activities has been inspired by the UCI-HAR dataset presented in Chapter 2. Additionally, these activities are simple to perform, common and relevant for elderly activity monitoring.

STANDING, SITTING, and LYING are static activities where the subject stays in the same position for a given duration. However, the subject does not need to stay completely still, but rather be natural as long as they keep either a standing, sitting or lying position, respectively.

WALKING, WALKING_DOWNSTAIRS, WALKING_UPSTAIRS and RUNNING are dynamic activities associated to mobility. The RUNNING activity is closer to walking fast than a sprint.

DRINKING is an activity that has been specifically added since we believe dehydration can be a risk for the elderly. The DRINKING activity is performed by drinking from a glass or a bottle, sip by sip, from a sitting position.

The composition of the dataset is detailed in Table 4.2.

The data presented in this study are openly available in Zenodo at `Htps://doi.org/10.5281/zenodo.5659336`. All the published data are fully anonymized.

Informed consent was obtained from all subjects involved in the study. The study was conducted in accordance with the Declaration of Helsinki, and approved by the Ethics Committee of Université Côte d'Azur CER (Comité d'Ethique de la Recherche, approval identifier: 2022-033). The data was handled according to the GDPR (General Data Protection Regulation) and registered to Université Côte d'Azur DPO (Data Protection Officer, registration identifier: UCA-C22-091). The description of the dataset along with the classification and embedded execution results were published in the MDPI Applied Sciences 2022 Volume 12(8) journal[198].

4.3.1 Data Collection Protocol

Each subject was given a table stating the guidelines of the recording. One voice recording per session was acquired in order to help in the labelling process. The entire signal recorded during a session can contain multiple status and transition classes as shown in Table 4.1.

Each data recording corresponds to one session as described in the table. Each session is described with 2 lines that must be read from left to right. The first line indicates the activity, while the second line gives the expected activity duration. Each session is a succession of activities. In order to provide a compact representation of sessions, an activity can be replaced by "repeat x times". In that case, no duration is indicated, it is rather replaced by the activity number to start again from. Subjects did not necessarily repeat the activities as many times as recommended due to time constraints or physical conditions.

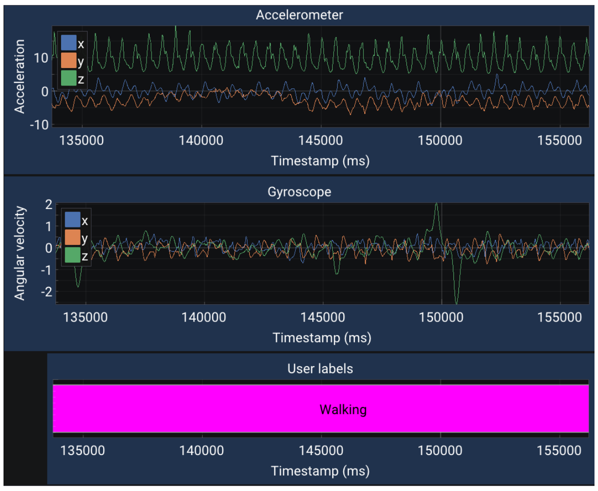
It is well known that homogeneous classes can be of premium importance to reach a good accuracy for some neural network family. As a transition is by nature shorter in time compared to a status class, the number of transition signal samples is very small compared to the status classes' samples. Even tough transitions are labelled in the dataset (SIT_TO_STAND, STAND_TO_SIT, SIT_TO_LIE, LIE_TO_SIT), they are not considered meaningful for classification in this work and are therefore filtered out for classification results. Additional data for transitions were recorded after the initial dataset was constructed thanks to a new TRANSITION session. However these samples were not used in a classification experiment as the quantiy was deemed still not to be enough. A DRIVING session has also been recorded where the subject drives a car in an urban area for several minutes. However, most of the subjects were unable to perform it. Therefore, the DRIVING session has not been used in the classification experiments. Both the TRANSITION and the DRIVING sessions were not included as part of the published dataset. However, they are used internally by Ellcie Healthy to develop future safety-related applications on the smart glasses.

The recording process is performed by a smartphone, collecting the accelerometer, gyroscope and barometer data sent by the smart glasses through a Bluetooth Low Energy connection. At the same time, audio is recorded from the phone's microphone in order to provide auditory cues for labelling. The subject or the test assistant must pronounce the keyword corresponding to the activity that the subject is currently performing.

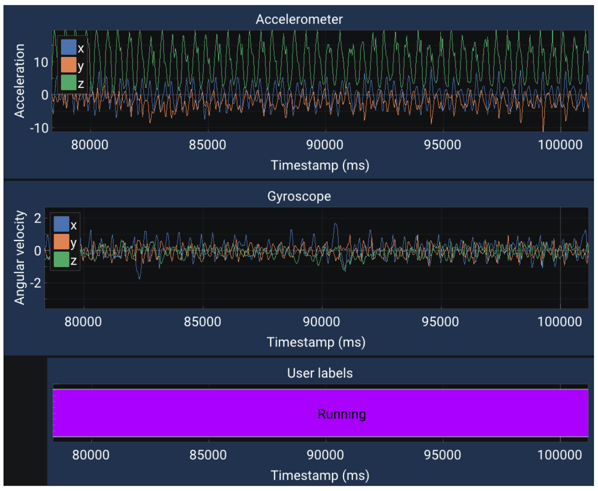
Examples of recordings of approximately 20 s for each session are shown thereafter in Figure 4.3a to Figure 4.3g. In each figure, the topmost graph shows the 3-axis data from the accelerometer. The graph in the middle shows the 3-axis data from the gyroscope. The graph on the bottom shows the corresponding activity.

Table 4.1: Instructions for each session of activity recording.

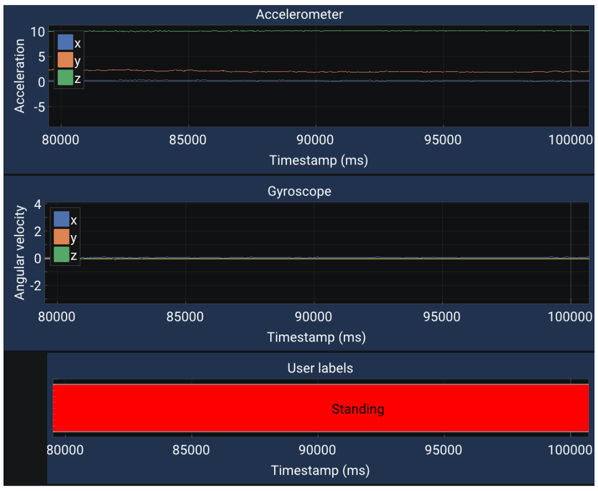
Session	**Activity 1**	**Activity 2**	**Activity 3**	**Activity 4**	**Activity 5**	**Activity 6**	**Activity 7**	**Activity 8**	**Activity 9**	**Activity 10**
WALKING	STANDING 5 s	WALKING 240 s	STANDING 5 s							
RUNNING	STANDING 5 s	RUNNING 180 s	STANDING 5 s							
STANDING	STANDING 5 s	WALKING 6 s	STANDING 180 s	WALKING 6 s	STANDING 5 s					
SITTING	STANDING 5 s	STAND_TO_SIT (no rush)	SITTING 90 s	SIT_TO_STAND (no rush)	Repeat once from Activity 1	STANDING 5 s				
LYING	STANDING 5 s	STAND_TO_SIT (no rush)	SITTING 7 s	SIT_TO_LIE (no rush)	LYING 90 s	LIE_TO_SIT (no rush)	SITTING 7 s	SIT_TO_STAND (no rush)	Repeat once from Activity 1	STANDING 5 s
STAIRS	STANDING 5 s	WALKING (5 to 6 steps)	WALKING_ UPSTAIRS (15 to 25 stairs)	WALKING (5 to 6 steps)	WALKING_ DOWNSTAIRS (15 to 25 stairs)	Repeat 7 times from Activity 2	WALKING (5 to 6 steps)	STANDING 5 s		
DRINKING	SITTING 5 s	DRINKING 1 sip/10 mL	Repeat 29 times from Activity 1	SITTING 5 s						
DRIVING	STANDING 5 s	STAND_TO_SIT (no rush)	SITTING 10 s	DRIVING 7 x 60 s	SITTING 10 s	SIT_TO_STAND (no rush)	STANDING 5 s			
TRANSITION	STANDING 5 s	STAND_TO_SIT (no rush)	SITTING 5 s	SIT_TO_LIE (no rush)	LYING 5 s	LIE_TO_SIT (no rush)	SITTING 5 s	SIT_TO_STAND (no rush)	Repeat 24 times from Activity 1	STANDING 5 s



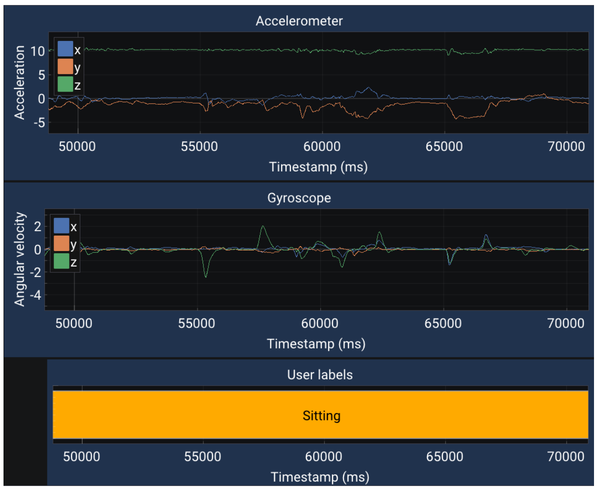
(a) WALKING



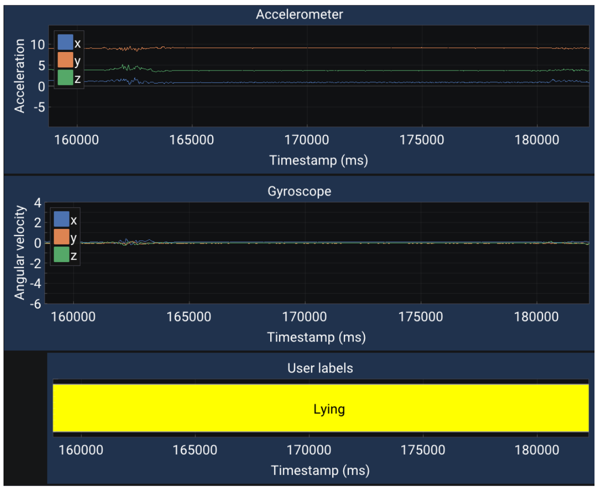
(b) RUNNING



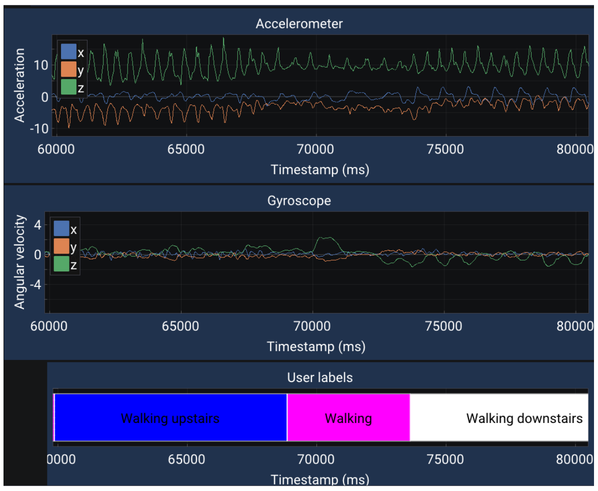
(c) STANDING



(d) SITTING

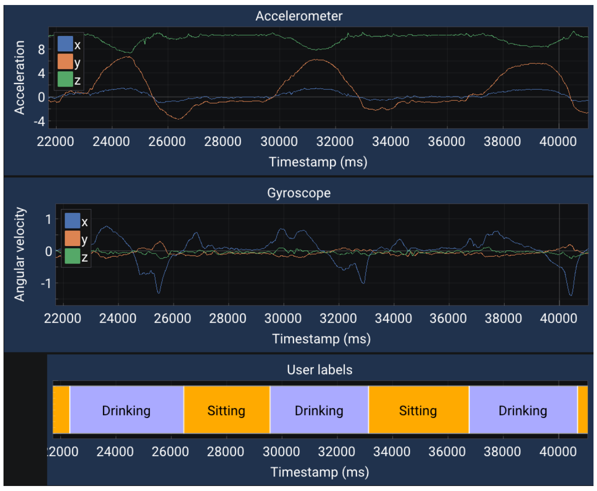


(e) LYING

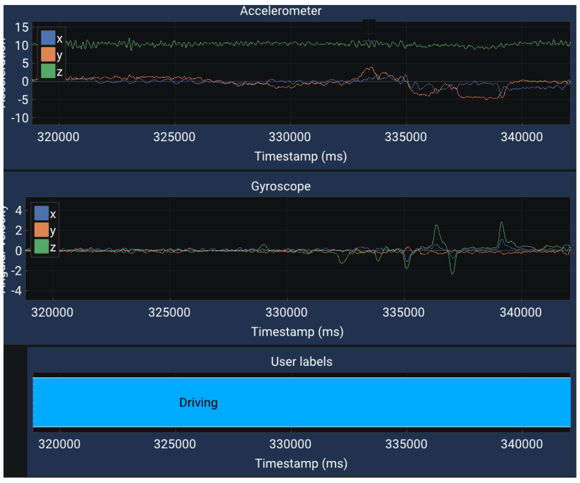


(f) STAIRS

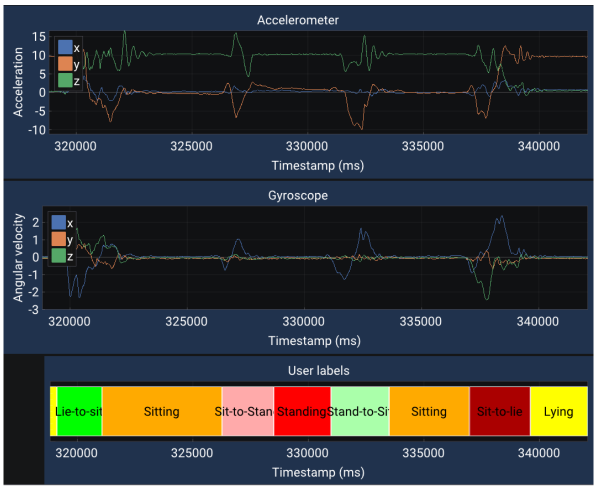
Figure 4.3: 20 s extracted from each session of subject T1.



(g) DRINKING



(h) DRIVING



(i) TRANSITION

Figure 4.3: 20 s extracted from each session of subject T1. (continued)

4.3.2 Data Visualisation and Labelling

The dataset was initially manually labelled after capturing all the data with the help of the audio recordings and a change point detection Matlab routine. However, this process was time-consuming and prone to introducing errors.

To imrpove the data labelling process, we developped a custom graphical user interface, called MicroAI-GUI, firstly to visualize time series data from the smart glasses sensors. A screenshot of this software can be seen in Figure 4.9. Thanks to this visualization, it was discovered that some activities were mislabelled, and in some cases the start time or the end time were not accurate. For example, the DRINKING session contains a fast switching of a few seconds between the DRINKING and SITTING activities. Therefore, a slight alignment issue can cause one activity to contain a significant amount of data from the other.

The MicroAI-GUI tool was then extended to allow the modification of the existing labels, and the labels were readjusted in case of inaccuracy. Additionally, in order to optimize the labelling process, MicroAI-GUI can communicate with the smart glasses through Bluetooth Low Energy, so that the collected data can be labelled in real time. This removes the need for a manual labelling after the session is captured. However, MicroAI-GUI still captures the audio at the same time in case an error is suspected in the labelling so that is can be corrected afterwards.

MicroAI-GUI is originally a desktop application. However, in order to run on a tablet, MicroAI-GUI has been ported to the Android operating system, thanks to the use of a portable graphics toolkit (ImGui[199]) and portable C++ code. This way, it is possible to collect and label data on the move. The subject can also perform the operation themselves, without the need of an assistant.

4.3.3 Data Format

The accelerometer, gyroscope, and barometer have three values for acceleration, three values for the angular velocity, and one atmospheric pressure value, respectively.

The full sensitivity range is ±2g (g = 9.81 m·s^{-2}) for the accelerometer and ±2000 dps (degrees per second) for the gyroscope. The Ellcie Healthy glasses used in this experiment sample the 6 signals from the accelerometer and the gyroscope at a rate of 26 Hz, whereas the barometer is sampled at 6.66 Hz.

Before the labelling process, an interpolation routine has been executed in order to provide the atmospheric pressure values interpolated for each accelerometer timestamp. The resulting file contains a vector with seven elements for each timestamp, with no empty value. It is worth noticing that the barometer, the gyroscope and the accelerometer share the same sampling time origin. The values are provided in m·s^{-2} , rad·s^{-1} and hPa.

Files are provided in CSV format with a semicolon as the column delimiter. The files contain one line every 40 ms approximately, with nine columns labelled "T" for the timestamp, "Ax", "Ay" and "Az" for the accelerometer, "Gx", "Gy" and "Gz" for the gyroscope, "P" for the atmospheric pressure and "CLASS" for the activity label. All numeric values are provided with 2 decimals. Finally, the name of the file is a combination of the identifier of the subject and the session name. The identifier of the subjects is numbered T1 to T21. However, T11 has been removed from the dataset due to not having performed enough activities. Some recordings have been performed in two sessions, in such a case "_1" or "_2" is appended to the filename.

An extract of the file for the DRINKING activity of subject T21 from timestamp 29 000 to timestamp 30 000 is provided in Appendix C.

4.3.4 Train/Test Split

The dataset is split in two parts: one for training a machine learning algorithm and one for testing the ability of the machine learning algorithm to generalize over previously unseen data. There are 14 subjects in the training set and 6 subjects in the testing set, representing approximately 77% and 23% of the total number of samples, respectively. Subjects number 5, 15, 17, 18, 19, and 20 have been chosen for the testing set since they have completed all activities. Moreover, these subjects have the lowest standard deviation on the percentage of samples for each class in the testing set. Therefore, and as can be observed at the bottom of Table 4.2, activities are balanced as much as possible between the training and testing sets.

The total number of time samples in the training and the testing sets are 563,469 and 170,150, respectively. After applying the windowing process described in Section 4.4.1, the total number of vectors in the training and the testing sets are 35,213 and 10,631, respectively. The distribution of time samples before windowing by subjects and activities for both the training set and the testing set can be seen in Table 4.2.

Table 4.2: Distribution of time samples across subjects and activities for training and testing sets.

	Activities								
Subject	**STANDING**	**SITTING**	**WALKING**	**LYING**	**WALKING_DOWNSTAIRS**	**WALKING_UPSTAIRS**	**RUNNING**	**DRINKING**	**TOTAL**
	Training set								
T1	8620	12 021	9955	5712	1588	1701	4310	4543	48 450
T2	6198	15 617	11 245	2626	3368	3298	4555	4069	50 976
T3	4973	17 904	17 029	3539	3887	3917	5300	5287	61 836
T4	7568	8822	10 871	5578	3132	3496	4002	3754	47 223
T6	6152	16 560	10 144	3199	2420	2305	5464	5093	51 337
T7	2162	16 436	9120	1984	2701	3333	4465	1383	41 584
T8	5151	4024	9378	4289	2145	2156	4064	0	31 207
T9	5113	6074	9578	3276	2596	3399	4015	0	34 051
T10	5899	4954	12 354	4226	1893	1943	4793	0	36 062
T12	4614	8509	10 559	1681	2368	2469	4641	1314	36 155
T13	7444	9957	13 449	12 224	2789	3373	6064	0	55 300
T14	4474	3611	7160	3025	1128	1384	4122	0	24 904
T16	5501	3489	8542	2250	1880	1940	3162	0	26 764
T21	1558	6524	2870	1937	1139	1148	1563	881	17 620
Total	75 427	134 502	142 254	55 546	33 034	35 862	60 520	26 324	563 469
	Testing set								
T5	5587	8662	16410	3390	2276	2583	6016	1954	46 878
T15	1513	6295	3581	2388	1746	1490	1626	463	19 102
T17	4394	7749	7404	3227	1940	2611	3005	1683	32 013
T18	4684	7784	7110	2412	1299	1590	3210	1288	29 377
T19	1566	4780	3435	2401	1204	1564	1884	1011	17 845
T20	5495	7755	4150	2099	973	1177	1734	1552	24 935
Total	23 239	43 025	42 090	15 917	9438	11 015	17 475	7951	170 150
Set	Distribution between sets								
Training	76%	75%	77%	77%	77%	76%	77%	76%	76%
Testing	23%	24%	22%	22%	22%	23%	22%	23%	23%

4.4 Machine Learning for Classification of Activities of Daily Living

In this section, a machine learning method to perform classification on the UCA-EHAR dataset is presented. Our aim is to provide a baseline for classification performance, so that these results can be used by other works for comparison. It is also the model used later on to perform inference for live human activity recognition on the smart glasses.

4.4.1 Data Pre-Processing

As the objective is to perform live inference directly on the smart glasses, the amount of computation done before entering the artificial neural network must be minimized. In consequence, only a windowing pre-processing task is performed. The neural network indeed requires time series, in other words a context around each data point. The windowing process uses windows of 64 time samples, each time sample containing a value for the three accelerometer and gyroscope axes. Each window overlaps by 25% over the previous one. Since data are sampled at 26 Hz, each window has a duration of approximately 2.46 s. This is close to the choice made by the authors of the UCI-HAR dataset [112]. The raw data from the dataset have one label per time sample. Time samples in a window may therefore have different labels. During windowing, the labels are reduced to one per window by selecting the label with the highest number of occurrences in the window. Despite the barometer data being provided in the dataset, they are not used in the embedded experiments since the barometer is not sampled at the same rate as the accelerometer and gyroscope. To use the barometer data during live inference, resampling the data coming from the sensor would have to be performed on the smart glasses. Furthermore, a band-pass filter would need to be applied to remove the mean of the signal and the high-frequency noise. This has not been implemented on the smartglasses yet and could cause a computational overhead.

4.4.2 Data Augmentation

In order to mitigate overfitting and improve generalization, three different data augmentation techniques have been used during training: time shifting, time warping and 3D rotations. Time shifting performs a uniformly distributed random rotation over the time axis in order to shift the centre of the window. Time warping performs a dilation over the time axis in order to speed up or slow down the movement. The dilation scale factor is chosen randomly from a normal distribution with a mean $\mu = 0$ and a standard deviation $\sigma = 0.15$. 3D rotation performs a three-dimensional rotation over the three accelerometer and gyroscope axes. The three rotation angles are randomly chosen from a normal distribution with a mean $\mu = 0$ and a standard deviation $\sigma = 0.15$.

4.4.3 Artificial Neural Network Architecture

A deep neural network is used as the machine learning algorithm. More specifically, the residual neural network presented in Section 3.4 has been used here as well as it performed well on the UCI-HAR dataset. Moreover, this type of network is easy to scale down for embedded hardware by changing the number of filters per convolutional layer. All convolutional layers have the same number of filters f.

The neural network is trained over 750 epochs using stochastic gradient descent (SGD) with momentum set to 0.9 and weight decay set to 5×10^{-4} . The batch size is set to 768. The initial learning rate is set to 0.025 and divided by 10 at epochs 200, 400, 600 and 675.

4.5 Results

4.5.1 Training and Prediction Results

The residual neural network is trained for 8, 16, 24, 32, 40, 48, 64, and 80 filters per convolution. It is then quantized using the post-training quantization method described in Section 3.2.2. Results are averaged over 15 runs for each number of filters. The detailed results for each number of filters per convolution are reported in Table 4.3.

Table 4.3: Accuracy and parameters memory for each configuration of residual neural networks.

Filters Per Convolution	Data Type	Parameters	Parameters Memory (B)	Accuracy
8	float32	1096	4384	78.08%
16	float32	3848	15 392	78.99%
24	float32	8264	33 056	79.14%
32	float32	14 344	57 376	79.28%
40	float32	22 088	88 352	79.48%
48	float32	31 496	125 984	79.87%
64	float32	55 304	221 216	80.24%
80	float32	85 768	343 072	80.20%
8	int16	1096	2192	78.08%
16	int16	3848	7696	79.06%
24	int16	8264	16 528	79.28%
32	int16	14 344	28 688	79.21%
40	int16	22 088	44 176	79.50%
48	int16	31 496	62 992	79.79%
64	int16	55 304	110 608	79.97%
80	int16	85 768	171 536	80.16%
8	int8	1096	1096	75.83%
16	int8	3848	3848	77.69%
24	int8	8264	8264	78.58%
32	int8	14 344	14 344	77.90%
40	int8	22 088	22 088	77.78%
48	int8	31 496	31 496	77.94%
64	int8	55 304	55 304	77.71%
80	int8	85 768	85 768	78.27%

The results for the original 32-bit floating-point model (UCA-EHAR float32), the 16-bit fixed-point quantized model with post-training quantization (UCA-EHAR 16-bit PTQ) and the 8-bit fixed-point quantized model with post-training quantization (UCA-EHAR 8-bit PTQ) are shown in Figure 4.4. As can be seen, 16-bit fixed-point quantization does not cause any accuracy loss while 8-bit fixed-point quantization causes up to a 2.4% drop in accuracy.

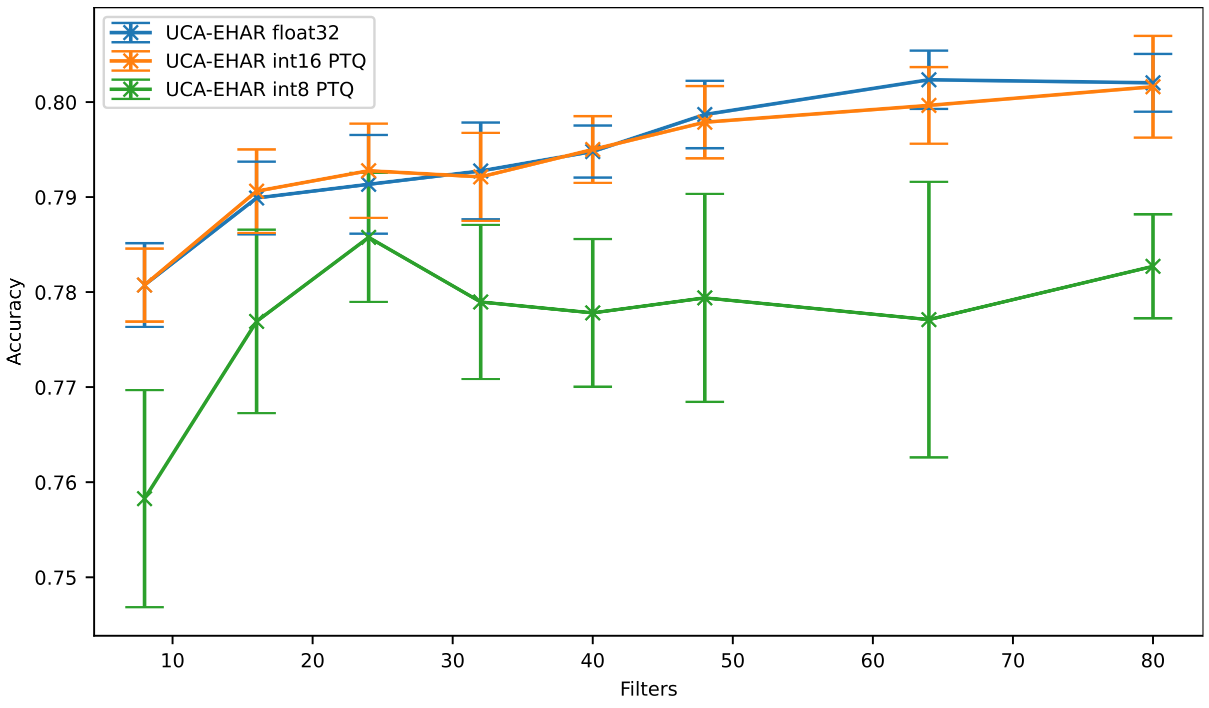


Figure 4.4: Accuracy vs. filters.

Concerning the memory used by the parameters, Figure 4.5 shows that the 16-bit fixed-point model is the most efficient, as it uses half the memory of the 32-bit floating-point model without any accuracy loss. On the other hand, the 8-bit fixed-point model is less efficient than the 32-bit floating-point model since a noticeable loss of accuracy can be observed.

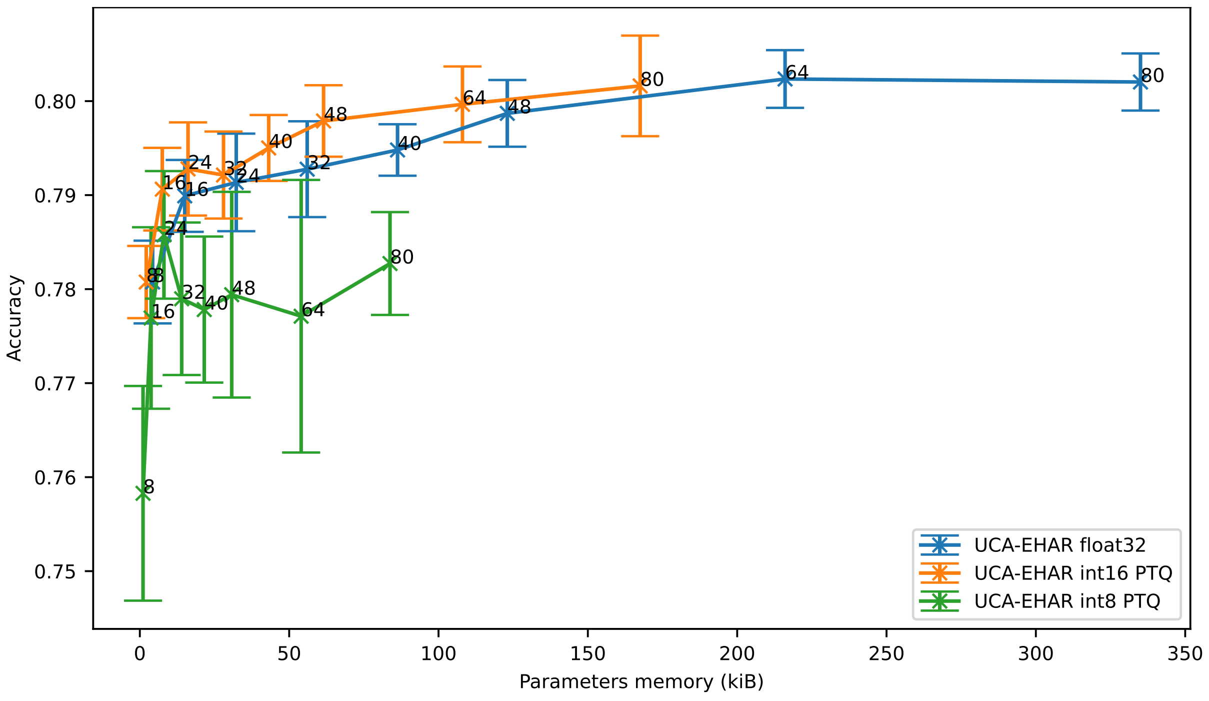


Figure 4.5: Accuracy vs. parameters memory.

The confusion matrix, shown in Figure 4.6 and extracted from one training for 80 filters per convolution, highlights the difficulty for an artificial neural network to differentiate the SITTING and STANDING activities from the collected data. The reason is that the orientation of the smart glasses remains the same for both classes, and the signals mostly stay constant for both of these motionless activities as observed in Figures 4.3c and 4.3d. It can be noted that a similar confusion, albeit to a lesser extent, was already observed on the UCI-HAR dataset.

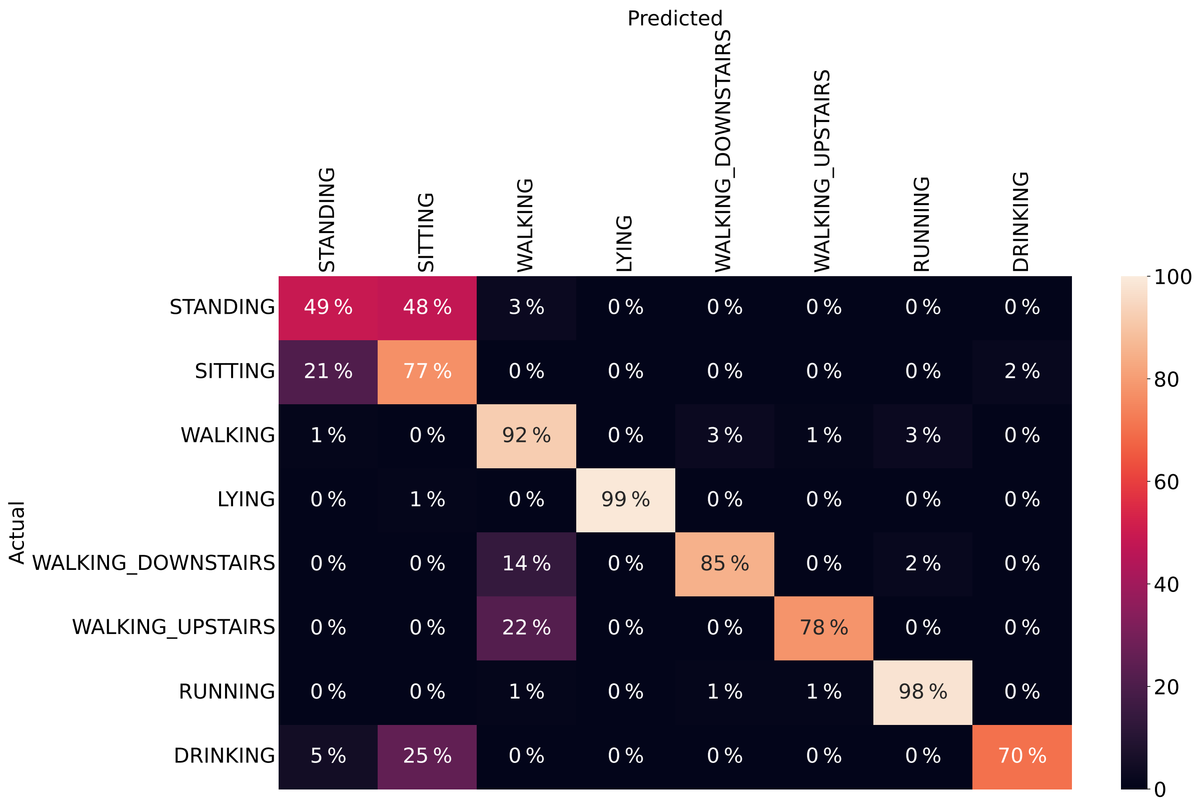


Figure 4.6: Confusion matrix for 80 filters per convolution.

An evaluation per subject has also been performed and is reported in Figure 4.7. The training set and the parameters are the same as the one used for the previous confusion matrix. However, inference is evaluated using each subject of the testing set one by one.

It is important to note that since the classes are unbalanced, the accuracy in the “TOTAL” column does not represent the average of each class’s accuracy. Instead, it is the accuracy over all the test vectors of a given subject. Classes with more test vectors have a greater influence on the resulting percentage of correct predictions. For example, for subject T20 the “TOTAL” of 75% is the most influenced by the “STANDING” activity, having much more samples than other activities and bringing the accuracy down.

The same applies for the “TOTAL” line, since subjects do not all have the same number of test vectors per class. The “TOTAL” line of Figure 4.7 therefore contains the same values as the diagonal of the confusion matrix in Figure 4.6. The bottom right cell, at the intersection of the “TOTAL” line and the “TOTAL” column, represents the accuracy over the entire testing set.

Results show a discrepancy between subjects for some activities such as WALKING_DOWNSTAIRS, WALKING_UPSTAIRS and DRINKING, while other activities are more homogeneous. However, the STANDING activity is hard to classify for all subjects. The reason is a large confusion with the SITTING activity, as previously shown in the confusion matrix.

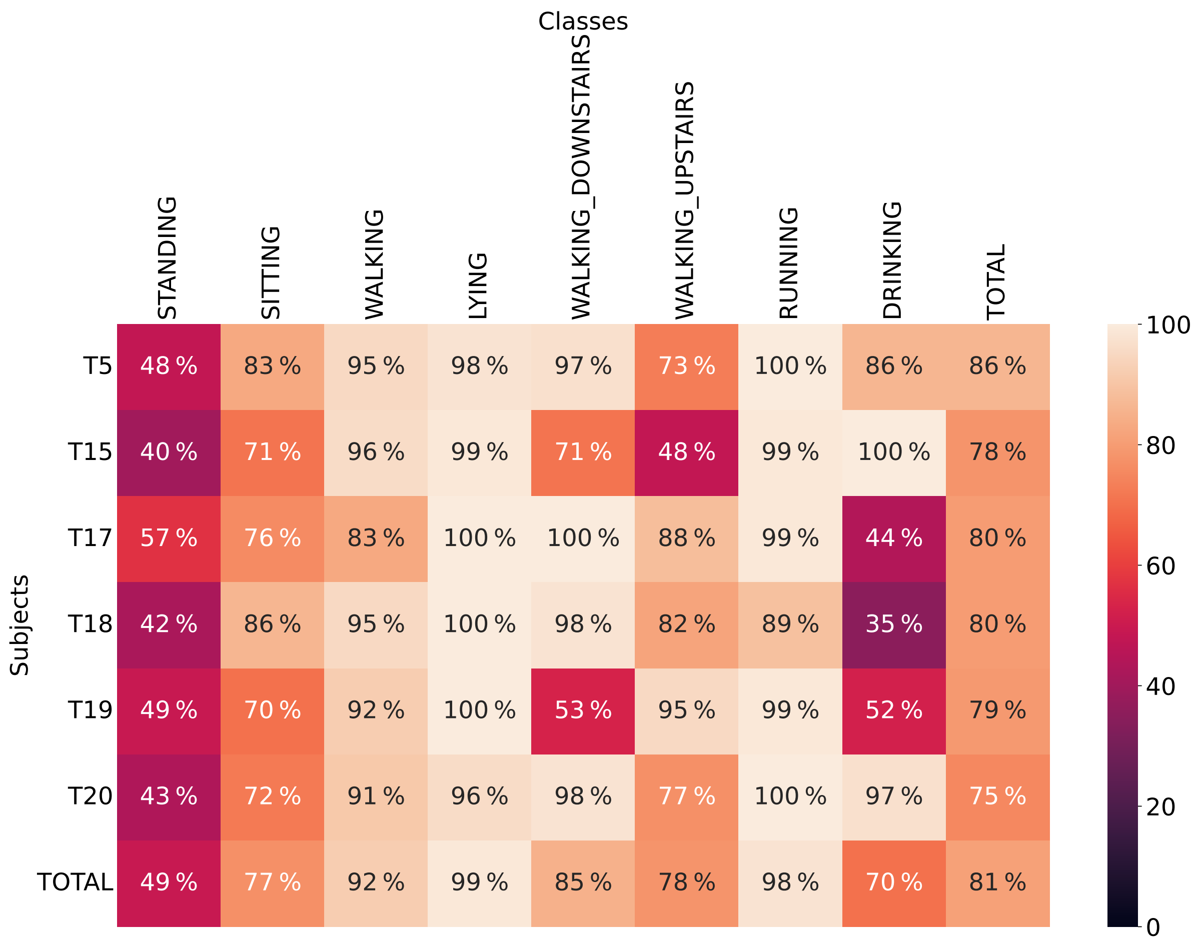


Figure 4.7: Accuracy per class and per subject for 80 filters per convolution.

4.5.2 Deployment on Smart Glasses

The ResNetv1-6 neural network is integrated into Ellcie Healthy's smart glasses firmware version 6.1.2 using the C inference library generated by our framework. A comparison is led between the non-quantized (float32) network, the 16-bit quantized network and the 8-bit quantized network, optionally with CMSIS-NN optimizations, both from a memory footprint and a power consumption perspective.

4.5.2.1 Memory

In this firmware version, only 77 604 B of Flash (for the inference code and the weights) and 40 572 B of RAM (for the intermediate computation and the layers' output buffers of the deep neural network) can be used. Therefore, these memory limitations constrain the neural network that can be executed on the microcontroller. For the 32-bit floating-point inference, the largest ResNetv1-6 that can be deployed only contains 32 filters per convolution. Since the 16-bit fixed-point quantization provides the best memory efficiency, we also deployed a 16-bit ResNetv1-6 with 48 filters per convolution to get the best possible accuracy on the smart glasses. It is worth noting that the same deep neural network without quantization (i.e., using 32-bit floating point) does not fit in Flash memory.

The memory footprint in Flash and the statically allocated RAM for each configuration is summarized in Table 4.4.

Table 4.4: Flash usage and static RAM allocation of the deep neural network (code and data).

Data Type	Optimizations	Flash (Available: 77,604 B)	RAM (Available: 40,572 B)	Accuracy
		32 filters		
int8	CMSIS-NN	17 776 B	20 680 B	77.90%
int8	None	17 216 B	6 664 B	77.90%
int16	CMSIS-NN	31 440 B	26 192 B	79.21%
int16	None	32 720 B	13 328 B	79.21%
float32	N/A	60 336 B	23 200 B	79.28%
		48 filters		
int16	CMSIS-NN	65 736 B	38 512 B	79.79%
float32	N/A	128 952 B *	33 440 B	79.87%

* Memory overflow.

As expected, 8-bit and 16-bit quantizations allow reducing both the Flash and RAM usage. Therefore, models with more parameters can be deployed compared to the original 32-bit floating-point network. Using a 16-bit quantization, a network with 48 filters per convolution can indeed be deployed on the smart glasses. For this network, almost all the available memory is used: 94.43% of Flash and 98.43% of statically allocated RAM. On the other hand, a maximum of 32 filters per convolution can be used for the 32-bit floating-point network. For this network, the available memory is used as follows: 91.89% of Flash and 86.75% of statically allocated RAM.

4.5.2.2 Power consumption

The inference is performed each time a series of 64 samples is collected by the inertial measurement unit (IMU) whose sampling rate is 26 Hz. As the barometer sampling rate is 6.66 Hz, this sensor is not used in these experiments since resampling the signal would be required.

The power consumption of the smart glasses is measured using a Qoitech Otii Arc laboratory power supply, supplying 3.75 V in place of the lithium polymer battery. Energy values are computed by the Otii software from the current and voltage over a one minute window starting from the beginning of an inference. Obtained measurements over one inference period is shown in Figure 4.8 for 16-bit fixed-point inference with 48 filters per convolution and CMSIS-NN optimizations. The graph on the top shows the current consumption in mA while the graph at the bottom shows the voltage in V. The Δ time indicates the duration of the selection, and the computed energy *E* over the selection is shown in the top right corner. It is worth noting that periodic spikes of current can be observed on the figure. Spikes at 20 Hz are related to the BLE transmission, while the spikes at 26 Hz are caused by the IMU sampling.

In the Figure 4.8, the inference task starts at the very beginning of the measurement. After 173 ms of inference, 64 new samples are collected from the IMU. This figure clearly shows that the inference task requires much less time than collecting 64 samples. Additionally, the shallow sleep mode between inferences does not enable a significant reduction of the power consumption. The average power consumption is 21.7 mW during inference, and 13.5 mW between inferences, only a 37.8% reduction Therefore, in this configuration the inference time does not have a significant impact on the overall energy consumption. Over one inference period (i.e., approximately 2.6 s), 10 200 nWh represents the sum of the energy for the inference (1120 nWh) and the energy to collect the samples (9100 nWh).

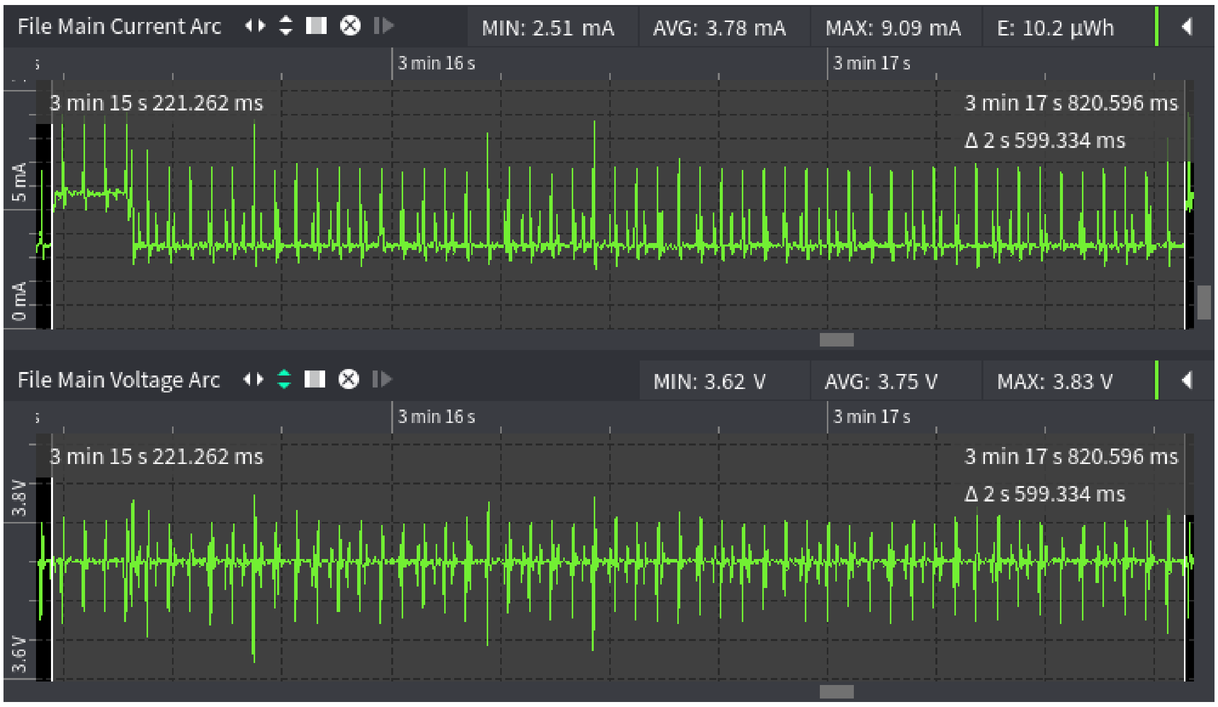


Figure 4.8: Current and voltage captures over one inference period by Qoitech Otii software for int16 model with 48 filters per convolution and CMSIS-NN optimizations

Inference time and energy measurements have been collected for various configurations and are shown in Table 4.5.

Table 4.5: Inference time and energy measurements on the smart glasses

Data Type	Optimizations	Inference Time	Energy for 1 Inference	Energy over 1 Minute
		32 filters		
int8	CMSIS-NN	53 ms	387 nWh	220 µWh
int8	None	115 ms	722 nWh	231 µWh
int16	CMSIS-NN	88 ms	605 nWh	232 µWh
int16	None	130 ms	853 nWh	234 µWh
float32	N/A	140 ms	919 nWh	235 µWh
		48 filters		
int16	CMSIS-NN	173 ms	1120 nWh	237 µWh

Results show that quantization also helps to reduce inference time and therefore energy consumption for one inference, especially when making use of CMSIS-NN optimizations. The original 32-bit floating-point network requires 140 ms on average for one inference, while the 16-bit quantized version only takes 88 ms for the same accuracy. Furthermore, the 8-bit quantized version only requires 53 ms, but, as seen previously, with a noticeable degradation of accuracy. However, the overall energy consumption over one minute does not significantly change with quantization. The overall energy is reduced by at most 7% between the 32-bit floating-point network and its 8-bit quantized version. As it has been observed in Figure 4.8, the inference time is indeed small compared to the time required to collect data. For that reason, the impact of inference over the overall energy consumption is small. Therefore, even if the largest network that fits in memory (48 filters per convolution with 16-bit quantization) is used, the autonomy of the smart glasses would not be impacted as long as the inference execution time remains small compared to the inference period. Hence, the energy consumption over one minute only grows by 2% when a 16-bit quantized network is used with 48 filters per convolution rather than 32 filters per convolution.

Ellcie Healthy's smart glasses embed a 350 mWh battery. Therefore, when the 16-bit quantized network with 48 filters per convolution is used (this network consumes 237 µWh per minute), the autonomy can reach 1476 min, i.e., 24.6 h. This estimated lifetime does not take into account additional applications that could run concurrently as well as battery ageing.

The larger the neural network, the higher the memory footprint and the higher the energy consumption. In our case study, the memory footprint is a more restrictive constraint than energy consumption, primarily making artificial intelligence in the smart glasses a memory bound problem.

4.6 Live Human Activity Recognition on Smart Glasses

The ResNetv1-6 model with 48 filters per convolution, 16-bit fixed-point quantization, and CMSIS-NN optimizations has been integrated onto the smart glasses firmware to perform live human activity recognition. Data are collected from the accelerometer and the gyroscope of the smart glasses when worn by a subject. The smart glasses' microcontroller performs the classification and sends the label of the recognized activity to a computer for visualization through a Bluetooth Low Energy communication. Additionally, the accelerometer and gyroscope data are also sent for visualization, even though the classification is not performed on the computer. MicroAI-GUI is used to provide a graphical visualization of the data as well as the predicted class. A 30-second sample of such a live recognition has been extracted and can be seen in Figure 4.9. In this extract, the following sequence of activities has been performed by the subject: walking downstairs, walking upstairs, walking, stopping in a standing position and finally drinking a sip of water.

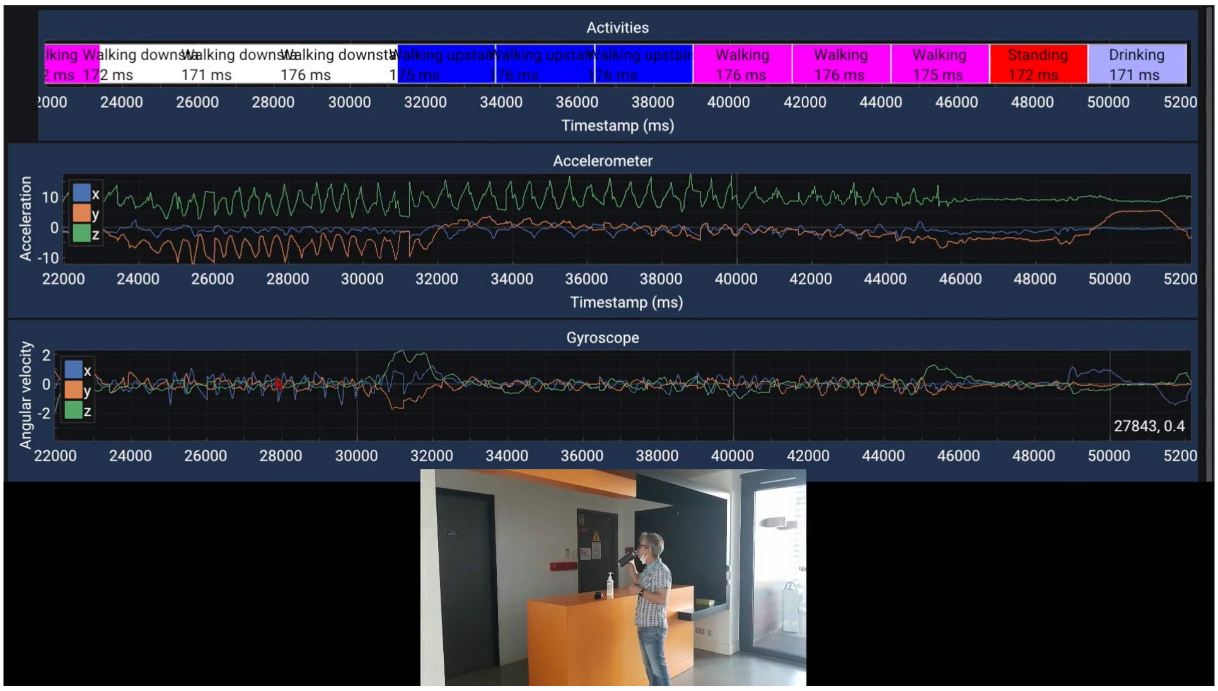


Figure 4.9: Live human activity recognition on smartglasses.

A video of a short human activity recognition session with various activities is available at http://3ia-demos.inria.fr/.

No quantitative evaluation of the live recognition performance has been done so far. However, it can be said that qualitatively the performance follows the results presented in the confusion matrix. Activities such as WALKING, WALKING_DOWNSTAIRS, WALKING_UPSTAIRS and DRINKING are generally recognized properly, while the STANDING and SITTING activities cannot be distinguished properly.

4.7 Conclusion

In this chapter, a novel dataset for human activity recognition called UCA-EHAR has been presented. This dataset gathers data collected from the accelerometer, the gyroscope and the barometer of smart glasses. UCA-EHAR is the first publicly available dataset dedicated to human activity recognition on activities of daily living using smart glasses. To provide a comparison baseline for a classification task, we evaluated the performance of a residual neural network on our dataset and we provided accuracy results as well as a confusion matrix. Using a floating-point ResNetv1-6 with 80 filters per convolution, the accuracy for this dataset is 80.2%. However, this floating-point implementation does not respect the smart glasses' embedded constraints. Therefore, the neural network has been quantized using 8-bit and 16-bit fixed-point numbers to optimize the memory footprint and the inference time, thus the energy consumption. Obtained results show that the 16-bit quantization provides the best accuracy vs. memory efficiency. We deployed different configuations of deep neural networks onto the smart glasses using our MicroAI framework. We then measured the current and voltage during a human activity recognition task running on the smart glasses. Using the 16-bit quantized network with 48 filters per convolution and CMSIS-NN optimizations, we have shown that human activity recognition can be performed for up to 24 h on the smart glasses. Finally, the human activity recognition task can run on the smart glasses' microcontroller with live data collected from the smart glasses' inertial measurement unit. The predicted activity is sent through Bluetooth Low Energy to a computer or a tablet.

Chapter 5

Semi-Supervised Learning and Unsupervised Fine-Tuning

Contents

5.1 Introduction

In Chapter 4, we presented the dataset we built for human activity recognition on smart glasses. The effort of collecting, filtering and labelling the data is a tedious and time-consuming task. More generally, labelling by hand the continuous flow of data coming from embedded sensors is not always possible, unless systematically recording the data on remote servers to label it manually afterwards. Supervised learning techniques can therefore turn out to be limited. Moreover, unpredictable variations of the input distribution (such as a subject with a slightly different behaviour) limit the use of representative pre-built datasets.

Unsupervised learning can therefore be used to learn from unlabelled data. As seen in Table 2.1, unsupervised learning methods generally lead to worse classification performance than supervised learning methods. Without labels, it is indeed difficult to guide the training towards the expected outcome. For example, supervised learning of deep neural networks often rely on a loss function to minimize (such as the cross-entropy loss), comparing the prediction of the model to the ground truth. In absence of labels, unsupervised learning must use a different learning rule. Some of the existing unsupervised learning methods were presented in Section 2.6.1. Unsupervised and supervised learning can be combined to make use of labels when they are available, but still learn from unlabelled data in a semi-supervised fashion. Such a method is to use the convolutional layers of a convolutional neural network trained in a supervised manner as a feature extractor to feed another model trained in an unsupervised manner.

Futhermore, unsupervised learning paves the way to online learning in the environment. For example, when human activity recognition is performed by smartphones or wearables, the device is often worn by only a single person throughout its lifetime. Due to a large variation across users, being able to adapt the trained model to the person's behaviour thanks to online learning could improve the recognition performance. However, it is difficult to obtain labelled data as it would require manual intervention, either by prompting the user or transmitting the data to an operator to label it. Unsupervised online learning could try to adapt the model without needing any label.

However, online learning methods suffer from a major issue: catastrophic forgetting. Catastrophic forgetting refers to the inability of a model to retain previous information when the model is trained on new data without the old data being available. That said, catastrophic forgetting may not be as catastrophic in our case, since it is not necessarily relevant to retain the information from the original training which was used to generalize on many subjects.

To overcome the limitations of supervised approaches, we propose to explore unsupervised and online learning methods and compare them to supervised learning methods, with the goal of optimizing the predictions of human activity recognition. In this work, unsupervised learning relies on Self-Organizing Maps (SOM)[135], while unsupervised online learning uses the Dynamic Self-Organizing Maps (DSOM)[165]. As a self-organizing map cannot provide labels it has never seen by itself, the supervised labelling method using few labels presented in [136] is applied to be able to handle a classification task. A hybrid approach combining the supervised and unsupervised methods is also presented: in order to improve the classification accuracy, features learnt by a convolutional neural network in a supervised manner are used to train a self-organizing map in an unsupervised manner.

The rest of this chapter is organized as follows. Section 5.2 describes the supervised, unsupervised, and online learning methods as well as the hybrid approach. Section 5.3 details our implementation of Self-Organizing map using the PyTorch framework for efficient training on GPUs. Section 5.4 presents experiments and results for the different learning approaches on several datasets: UCI-HAR, Heidelberg Digits, CORe50, and our own, UCA-EHAR. Finally section 5.5 concludes this chapter.

5.2 Supervised, Unsupervised, Semi-Supervised and Online Learning

5.2.1 Supervised Learning with Convolutional Neural Networks

A deep neural network consists of an input and an output layer, connected through one or several hidden layers. Artificial neural networks making use only of successive fully-connected layers are called multi-layer perceptrons, whereas networks making use of convolutional layers are called convolutional neural networks. A convolutional layer performs a convolution between the output of the previous layer and a kernel (or filter). Convolutional layers are typically used for automatic features extraction. Pooling layers usually follow convolutional layers to reduce feature maps dimensions. The pooling process consists in computing the average or the maximum of a set of values from the output of the previous layer. In a fully-connected layer, each neuron is connected to all neurons from the previous layer, thus performing a matrix-vector product between the synapses weights and the output of the previous layer. The final layer is the output layer which is usually a fully-connected layer for a classification problem, with its output being the activity of each class.

During the learning phase, the backpropagation process adjusts the trainable parameters (weights and biases) of each layer. The goal of the backpropagation is to reduce the loss computed between the ground truth (i.e., the labels) and the predictions, such as the cross-entropy loss.

5.2.2 Unsupervised Learning with Self-Organizing Maps

Unsupervised learning is a useful method to detect potential hidden patterns within a dataset. It also enables exploratory data analysis and clustering to understand relationships between patterns.

There are different methods taking advantage of unsupervised learning with artificial neural networks such as autoencoders, generative models and self-organizing maps. These methods can be used for various applications with times series classification or data prediction [5]. However, these methods are usually not able to achieve a level of performance equivalent to supervised methods. Nevertheless, having labels is increasingly difficult to comply with in realistic applications, our objective is to to assess in which extent non-supervised approaches can be used in the context of these applications.

Self-organizing maps allow a reduction of a multidimensional space into a lower-dimensional space. In general, the target space is a 2-dimensional grid, but variants exist with more dimensions or other types of lattices. As shown in Figure 5.1, the grid consists of several neurons connected to each other through a synpase by a relative neighbourhood relationship. Neurons have a weights vector of the same dimension as the observation space. Neurons weights evolve progressively with time according to the learning algorithm. When learning is stabilized, the organization of neurons reflects the vector organization of the input space as well as the probability density of the input data. Indeed, two similar data in the observation space correspond to nearby regions on the map.

Using self-organizing maps, it is possible to use unlabelled samples for training, as opposed to supervised learning. The learning rule does not attempt to optimize the set of parameters in order for the predictions to match the ground truth. Instead, the learning rule tries to match the neurons weights to the input distribution, with no knowledge of a possible expected output. The self-organizing map can be thus seen as a vector quantization algorithm, as it will encode any input vector to the position of a neuron on the grid. The codebook is the set of neurons position on the grid associated with their weights vector.

Initially, all neuron weights are randomly initialized. An input vector is presented to the map.

The Euclidean distance between each neuron and the input vector is computed as:

$$d_2(v, w_i) = \sqrt{\sum_{j=0}^{n} (w_{i_j} - v_j)^2} \tag{5.1}$$

with v the input vector, w_i the weights of the neuron i, and n the dimension of the input vector.

The neuron the closest to the input vector (i.e., the smallest distance) is called the best matching unit (BMU). It corresponds to the most representative neuron of the input data. The neighbourhood of the best matching unit is defined by the Manhattan distance on the map between each neuron and the best matching unit.

For a 2D map M of neurons, the Manhattan distance between two points p_i and p_j, with respective coordinates (X_{p_i}, Y_{p_i}) and (X_{p_j}, Y_{p_j}) is defined by:

$$d_1(p_i, p_j) = |X_{p_j} - X_{p_i}| + |Y_{p_j} - Y_{p_i}| \tag{5.2}$$

All the neurons present in this neighbourhood have their weights changed in order to get closer to the input vector. The weights w_i of each neuron i are moved toward the input vector v according to the following learning rule[165] :

$$\Delta w_i = \epsilon(t) h(t, i, s)(v - w_i) \tag{5.3}$$

where s is the best matching unit and h is a neighbourhood function of the form:

$$h(t, i, s) = e^{-\frac{d_1(p_i, p_s)^2}{2\theta(t)^2}} \tag{5.4}$$

with $\epsilon(t)$ being the learning rate and $\theta(t)$ the size of the neighbourhood defined as:

$$\theta(t) = \theta_i * \left(\frac{\theta_f}{\theta_i}\right)^{t/t_f} \tag{5.5}$$

$$\epsilon(t) = \epsilon_i * \left(\frac{\epsilon_f}{\epsilon_i}\right)^{t/t_f} \tag{5.6}$$

with θ_i and θ_f being the size of the initial and final neighbourhood, ϵ_i and ϵ_f being the initial and final learning rate. The learning function iterates between time $t = 0$ and time $t = t_f$.

A single learning step is illustrated in Figure 5.1. The best-matching unit (blue circle) is elected as the closest neuron to the input data (green triangle). The largest weights update (longest red arrow) is applied to the best-matching unit, since $d_1(p_s, p_s) = 0$ and therefore $h(t, s, s) = 1$. As the Manhattan distance between a neuron and the best-matching unit increases from 0 to 3 in this example, the magnitude of the weights update continues to decrease (shorted red arrows).

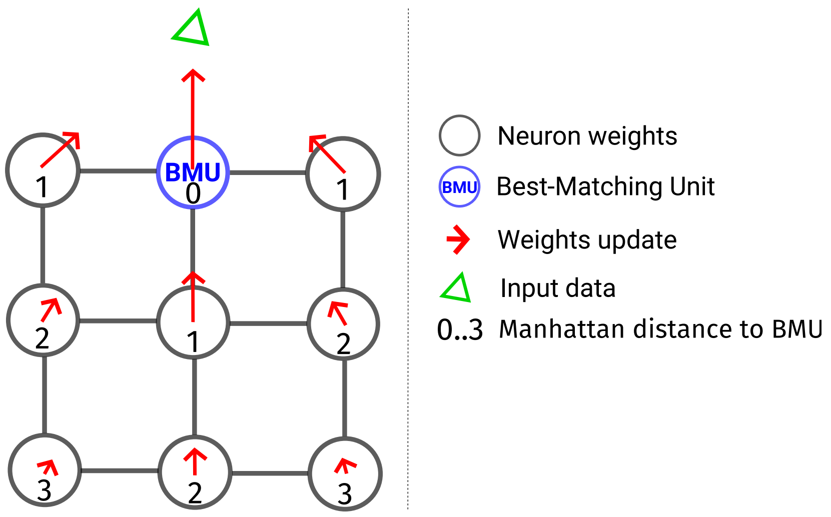


Figure 5.1: A single step of the self-organizing map training.

In order to stabilize the self-organizing map, the learning rate and the neighbourhood width are decreased during the training process.

The convergence of the training can be measured with the mean squared error (MSE) between data sampled from the input distribution and the best-matching unit weights among other metrics. This metric should decrease as the training progresses if the input distribution does not change.

The state of a 2-dimensional self-organizing map of 8×8 neurons (in black) at various learning stages of 3 clusters of data (in orange) is illustrated in Figure 5.2. The self-organizing map neurons are initialized following a uniform distribution, hence the regular grid at $t = 0$. At $t = 1$ (Figure 5.2a), some neurons on the edges already started to move towards the input data. At $t = 15$ (Figure 5.2b), more and more neurons are attracted towards the input data by the effect of the neighbourhood. At $t = 55$ (Figure 5.2c), most neurons have converged towards the input distribution.

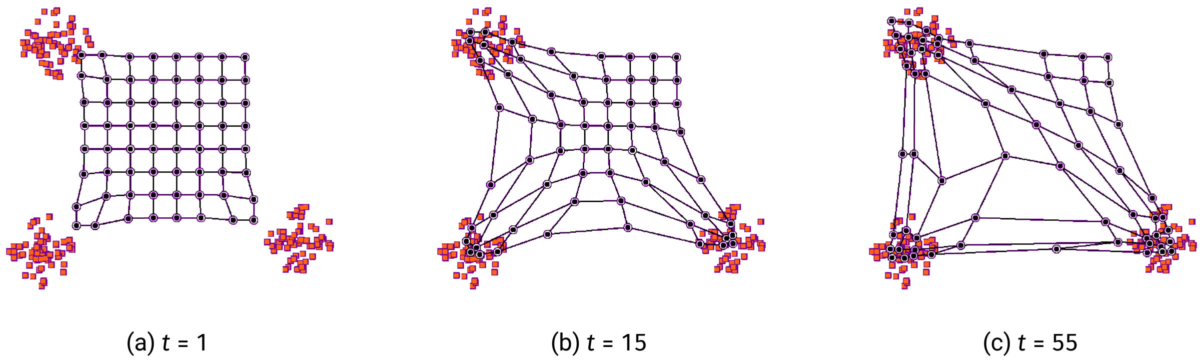


Figure 5.2: 8×8 self-organizing map learning 3 clusters [200].

The previous examples presented the learning of 2-dimensional data by a self-organizing map for illustration purposes. However, self-organizing maps are not limited to learning 2-dimensional data. They can learn from any kind of finite multi-dimensional vectors, for example time series or feature vectors.

5.2.3 Supervised Neuron Labelling With Few Labels

Self-organizing maps are trained in an unsupervised fashion, so without labels. In a classification problem, self-organizing maps have no knowledge of the classes to recognize. Therefore, a neuron labelling algorithm, initially proposed in [136], is used to extract meaningful classes for each neuron.

In order to associate a class to a neuron, inputs are injected in the self-organizing map, then for each neuron, its activity is accumulated separately for each class over all inputs. This creates a two-dimensional table of neurons and classes, with the value in each cell being the activity for a given class on a given neuron. After having injected enough inputs, each neuron is affected to the class having the highest activity for this neuron. The activity can be measured using various metrics, such as the dot product of the input and the neuron weights, the Euclidean distance between the input and the neuron weights, or a Gaussian method. The Gaussian method seems to give the best results. It applies an exponential function to the opposite of the Euclidean distance d_2 divided by a σ hyperparameter as shown in Equation 5.7:

$$gaussian\_activity(v, w_i) = e^{\frac{-d_2(v, w_i)}{\sigma}} \tag{5.7}$$

with v the input vector and w_i the weights vector of the neuron i.

A method is also proposed in [136] to select the σ hyperparameter. While it can give a starting point, it does not seem to provide a very good approximation of the optimal value in some cases. σ can instead be adjusted manually or found by hyperparameter research.

It is worth noticing that the labelling method is necessarily supervised as labels are required to accumulate the activity for each class. However, the labelling method can be performed using only a subset of the training set, in which case the training set does not need to be fully labelled. Of course, all classes must be present in the labelled samples. The amount of required labelled data depends on the complexity of the data. With the MNIST dataset, which is one of the easiest multi-class classification dataset to tackle, it has been shown in [136] that using 1% of the training set is enough.

5.2.4 Self-Organizing Maps Learning From Feature Maps

As convolutional layers are quite effective at improving the classification accuracy, we evaluated the use of additional convolutional layers as the input of self-organizing map. The resulting hybrid network is built by replacing the fully-connected layers of the convolutional neural network with a self-organizing map. As the self-organizing map hyperparameters depend on the range of the input, a min-max feature scaling layer is inserted between the last convolutional layer and the self-organizing map. Moreover, as the range of the input is normalized between 0 and 1, the self-organizing map can be initialized with a uniform distribution between 0 and 1. An example of such a hybrid network is shown in Figure 5.3.

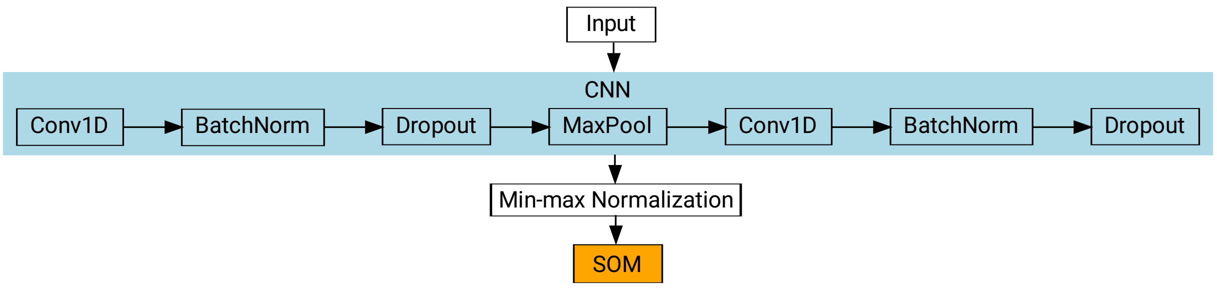


Figure 5.3: Self-organizing map learning from convolutional neural network feature maps (CNN+SOM).

This example is built around 2 convolutional layers. However, many other convolutional neural network variants are possible. In particular, residual neural networks can also be used as illustrated in Figure 5.4 with a simple ResNetv1 with one block. Residual neural networks may be useful to extract features of higher quality with better separability.

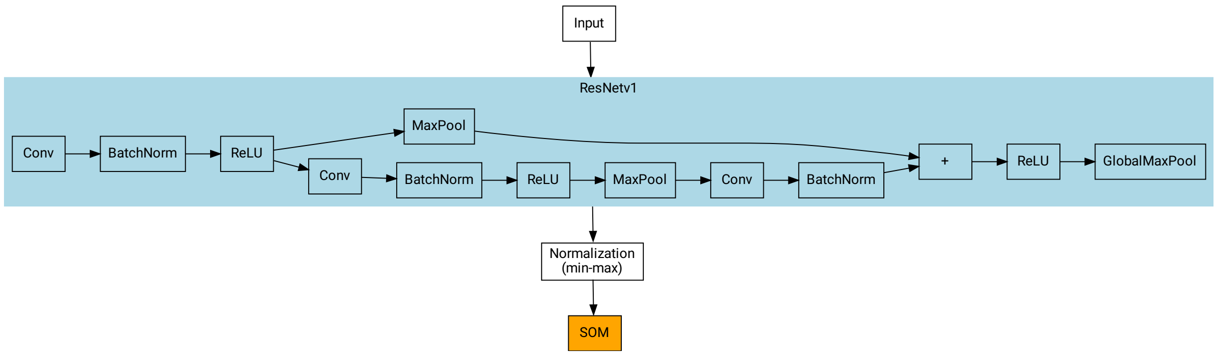


Figure 5.4: Self-organizing map learning from residual neural network feature maps (ResNet+SOM).

The training process of this hybrid network is as follows:

1. train the convolutional neural network in a supervised manner,
2. freeze the trainable parameters of the convolutional neural network,
3. replace the final fully-connected layer(s) by a self-organizing map,
4. train the hybrid network in an unsupervised manner,
5. label the self-organizing map neurons as described in Section 5.2.3.

Another advantage of feature extraction is that the dimension of the data at the input of the self-organizing map can be greatly reduced. For example, a ResNet-18 applied on ImageNet with an input resolution of 224×224 reduces the dimensions from 150 528 ($224 \times 224 \times 3$) to 512 after the final global pooling layer. Global pooling reduces the spatial dimensions to 1, therefore the dimension of the output matches the number of filters of the previous convolutional layer. This greatly reduces the memory required for the self-organizing map, since each neuron has to store a vector of dimension 512 instead of 150 528. As an alternative to reducing the number of neurons, the output dimension of the feature extractor can be adjusted in order to reduce the memory footprint of the self-organizing map.

5.2.5 Adapting Unsupervised Learning to Online Learning

Self-organizing maps described in Section 5.2.2 are not able to perform online learning since the learning hyperparameters decrease over time. However, self-organizing map can still learn in an unsupervised manner. Therefore they can be fine-tuned without requiring labels. In order to combine unsupervised and online learning, the original self-organizing map algorithm has to be slightly modified to remove time dependency from the learning rule.

This modification was proposed in [165] as the Dynamic Self-Organizing Map (DSOM). In this variant, the learning rate is a constant, and the neighbourhood width is replaced with an elasticity parameter which is also a constant. In other words, there is no final learning boundary. For a better convergence and stability of the algorithm, the neihbourhood function and the weights update function are slightly modified.

The learning function is thus modified as:

$$\Delta w_i = \epsilon d_2(v, w_i) h(i, s, v)(v - w_i) \tag{5.8}$$

where ϵ is the constant learning rate. In this Equation 5.8, a new term appears compared to Equation 5.3. This multiplication by the distance between the neuron and the input vector enables the neurons to get close to the input data faster when they are far away from the input distribution.

The neighbourhood function is also modified as follows:

$$h(i, s, v) = e^{-\frac{1}{\eta^2}\frac{d_1(p_i, p_j)^2}{d_2(v, w_s)^2}} \tag{5.9}$$

where η is the elasticity constant. If $v = w_s$, then $h(i, s, v) = 0$.

In this Equation 5.9, modulating the neighbourhood by the distance between the best-matching unit and the input vector means that other neurons learn less if the best-matching unit is already close to the input data.

Using these learning and neighbourhood functions, the network can learn at any time from new unlabelled input samples. The DSOM therefore enables online learning on top of unsupervised learning.

A downside of the dynamic self-organizing map is that it is more sensitive to hyperparameters. It has indeed a much higher chance to diverge than a self-organizing map with hyperparameters decreasing over time. The dynamic range and the dimensions of the input as well as the size of the map affect the choice of hyperparameter. Thus, it can be difficult to find an appropriate range of hyperparameters to even avoid the training from diverging. Therefore, hyperparameter research, for example with grid search, can help to narrow down a working range for the hyperparameters for a given problem. That said, a rough range still has to be provided for the hyperparameter research.

5.3 Custom PyTorch Layers for Self-Organizing Maps

Our first experiments were performed with TensorFlow using an extended version of the implementation of the self-organizing map from [201]. The algorithm has been encapsulated into Keras layers for ease of use. We then decided to switch to the PyTorch training environment for its better flexibility and popularity in the research community.

Therefore, the dynamic self-organizing map has been implemented in PyTorch to leverage GPU acceleration, similarly to the work done in [201]. Making it a layer (a `nn.Module` in PyTorch's terms) enables both a standalone usage and the ability to combine it with other layers in a deep neural network. In order to use the high parallelism of the GPU as much as possible, the computation relies on the vectorized mathematical functions of PyTorch as much as possible. However, the self-organizing map learning rule does not allow the processing of input vectors in parallel. Since the modification of the weights can affect the choice of the best matching unit for the next vector, input vectors must be processed sequentially and the weights must be updated after receiving each vector.

In order to speed up the training process, batches of multiple input vectors have been used. Batches can indeed be loaded more efficiently on the GPU's memory. Processing performed before the self-organizing map algorithm also benefits from this parallelism. In the self-organizing map, however, the batch dimension is processed sequentially. Despite our efforts to parallelize the processing of vectors inside a batch by using a map-reduce pattern, the deterioration of the results was too important.

Futhermore, it is important to keep in mind that self-organizing maps do not rely on back-propagation to update the weights, but rather on a local learning rule. Therefore, automatic differentiation for gradient computation is disabled for the entire processing of the self-organizing map.

The labelling phase used to solve a classification problem is implemented as a separate layer and based on the Gaussian labelling method proposed in [136]. Since labelling may not be required for some tasks, this layer is optional. Without this layer, the position or the weights vector of the best matching unit position can be used as the output of the self-organizing map.

5.4 Experiments and Results

As mentioned in the introduction, we are interested in a use case where the device is worn by a single person throughout its lifetime. In our approach, we still rely on a general training phase using a dataset representing various subjects. That way, the device is able to operate out of the box in most situations. However, the general training will be less relevant than a specific training for that person. In this work, we aim to leverage online learning properties to adjust the weights of the model to a specific subject wearing the device. This fine-tuning is done after the initial general training phase.

In this section, four different datasets are investigated, with different goals for each one of them. Therefore, despite using the methods presented in Section 5.2, the experiments will differ between the datasets.

The UCI-HAR dataset is used to perform an initial validation of unsupervised learning. The supervised and unsupervised methods are briefly compared, and a preliminary experiment with unsupervised online fine-tuning is provided. This dataset is split by subject, thus enabling fine-tuning experiments for a specific subject.

Our own dataset, UCA-EHAR, is then used to validate whether unsupervised online learning is feasible or not in the case of human activity recognition using smart glasses. Later on, the unsupervised online learning system will be deployed onto the smart glasses. This dataset is representative of the data that will be collected in the environment. Additionally, since we have control over the collection, filtering and labelling process, more data for some subjects can be provided. It is worth noting that the lack of data from the UCI-HAR dataset may be a problem for fine-tuning. Results mainly focus on comparing the supervised and the unsupervised online fine-tuning approaches with some specific subjects from the dataset.

Both the UCI-HAR and the UCA-EHAR datasets are designed for human activity recognition, with data collected from body-worn sensors.

The Heidelberg Digits[168] brings a different kind of data since it is made of spoken digits audio recording, split by subjects. Our objective is to reduce the bias that could come from applying our methods on our own dataset. Additionally, this could be a starting point in order to implement keyword spotting in the smart glasses in the future.

Results about catastrophic forgetting using the CORe50 dataset are provided in Appendix G, since the problem is of a different nature.

5.4.1 UCI-HAR

The UCI-HAR dataset[112], described in Section 3.4.1.1, is used for our initial validation. As a reminder, UCI-HAR is a human activity recognition dataset with accelerometer and gyroscope data captured from smartphones, with 30 subjects performing 6 activities. The pre-computed features are not used in these experiments.

In this section, we set up a measurement platform to compare the supervised and unsupervised learning methods. The supervised method is evaluated with various multi-layer perceptrons and convolutional neural networks. Self-organizing maps and dynamic self-organizing maps of various sizes are used for the unsupervised method comparison. Furthermore, the convolutional layers of a convolutional neural network trained in a supervised manner are used to feed a self-organizing map trained in an unsupervised manner as a hybrid network (CNN+SOM).

A summary of these results in terms of accuracy and number of parameters can be seen in Figure 5.5, while detailed results are available in the following sections. Overall, the convolutional neural network (CNN) provides the best accuracy. The multi-layer perceptron (MLP) does not scale as good as the convolutional neural network, showing the importance of convolutional layers. Both these supervised methods outperfom the self-organizing maps (SOM and DSOM) trained in an unsupervised manner. The accuracy of the dynamic variant (DSOM) is slightly lower than the regular self-organizing map (SOM) for the same number of parameters. The self-organizing map learning from feature maps (CNN+SOM) provides a better accuracy than the standalone self-organizing map but worse than the convolutional neural network. Moreover, the standard deviation of the results can be quite large. However, this hybrid network can still be useful in an online context when combined with a dynamic self-organizing map.

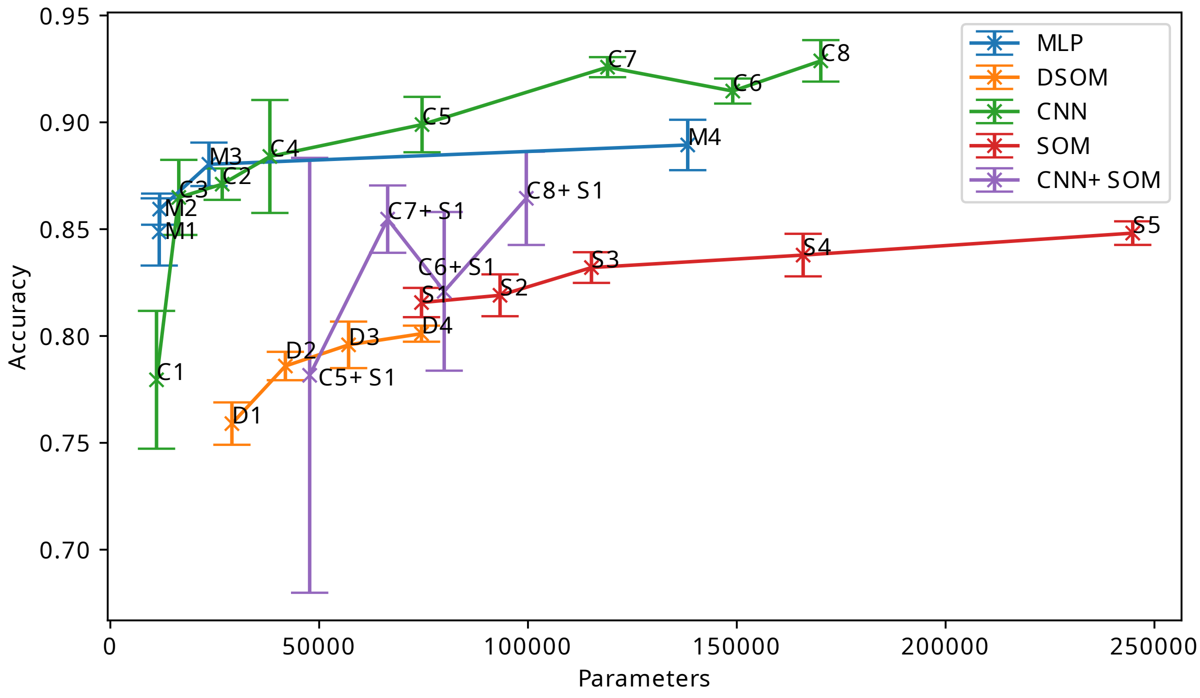


Figure 5.5: Test accuracy vs. number of parameters for each model.

Then, we present an experiment of fine-tuning a dynamic self-organizing map in an unsupervised manner on some data of a test subject.

While this chapter focuses on comparing the classification performance of the models rather than embeddability, the models were chosen with memory footprint limitations in mind. Therefore, the memory footprint of the models' weights is kept below 1 MiB to fit in the ROM of higher-end low-power microcontrollers. However, for online learning, the weights have to be stored in RAM in order to be modified. As RAM limitations are more restrictive than ROM, we chose to target an upper limit of 384 KiB. Both these limits match the characteristics of the Ambiq Apollo3 microcontroller presented in Section 3.4.2.

5.4.1.1 Results with Supervised Learning

The different neural networks architectures used for this experiment are described in Table 5.1. Supervised models are trained with the Adam optimizer with a learning rate of 1×10^{-3} and a batch size of 32 for 120 epochs.

Table 5.1: Deep neural network architectures. Each layer can be 1D Conv(olutions) with f the number of filters and ks the size of the kernel, MaxPool(ing) or FC (Fully-Connected) layer.

Name	1st Layer	2nd Layer	3rd Layer	4th Layer	5th Layer	6th Layer	7th Layer	8th Layer
				Multi-layer perceptrons				
M1	FC(10)	ReLU	FC(10)	ReLU	FC(6)			
M2	FC(10)	ReLU	FC(10)	ReLU	FC(10)	ReLU	FC(6)	
M3	FC(20)	ReLU	FC(20)	ReLU	FC(6)			
M4	FC(100)	ReLU	FC(100)	ReLU	FC(120)	ReLU	FC(6)	
				Convolutional neural networks				
C1	Conv(f=2, ks=2)	ReLU	MaxPool(2)	Conv(f=2, ks=2)	ReLU	FC(6)		
C2	Conv(f=5, ks=2)	ReLU	MaxPool(2)	Conv(f=5, ks=2)	ReLU	FC(6)		
C3	Conv(f=5, ks=3)	ReLU	MaxPool(2)	Conv(f=3, ks=2)	ReLU	FC(6)		
C4	Conv(f=5, ks=2)	ReLU	MaxPool(2)	Conv(f=5, ks=2)	ReLU	FC(120)	ReLU	FC(6)
C5	Conv(f=10, ks=3)	ReLU	MaxPool(2)	Conv(f=10, ks=3)	ReLU	FC(120)	ReLU	FC(6)
C6	Conv(f=20, ks=3)	ReLU	MaxPool(2)	Conv(f=20, ks=3)	ReLU	FC(120)	ReLU	FC(6)
C7	Conv(f=48, ks=5)	ReLU	MaxPool(4)	Conv(f=32, ks=3)	ReLU	FC(120)	ReLU	FC(6)
C8	Conv(f=64, ks=7)	ReLU	MaxPool(4)	Conv(f=48, ks=5)	ReLU	FC(120)	ReLU	FC(6)

The results in Table 5.2 show that multi-layer perceptrons quickly reach an accuracy ceiling and become much less efficient than convolutional neural networks in terms of number of parameters for a given accuracy. The multi-layer perceptron M4 reaches an accuracy similar to the convolutional neural network C4, but with 3.6× as many parameters. The feature extractor formed by convolutional layers helps the classification stage to separate classes. Different convolutional neural network architectures, inspired from [148], have been explored according to various parameters (number of layers, filters per layers and filters size). Accuracy reaches up to 92.88% with the largest convolutional neural network C8 presented here. However, with 170 110 parameters the memory footprint of the weights starts to be significant: around 664 kiB of memory are needed with single-precision floating-point numbers.

Table 5.2: Results using supervised learning.

Name	Parameters	Accuracy (%)
Multi-layer perceptron		
M1	11 706	84.87
M2	11 816	85.94
M3	23 606	88.03
M4	138 246	88.94
Convolutional neural network		
C1	11 058	77.95
C2	26 784	87.11
C3	16 391	86.49
C4	38 196	88.40
C5	74 636	89.89
C6	149 026	91.46
C7	119 054	92.58
C8	170 110	92.88

5.4.1.2 Results with Unsupervised Learning of Self-Organizing Maps

The same measurements have been realized with our self-organizing maps implementation. Training parameters of the self-organizing maps are listed in Table 5.3, and the labelling process uses the Gaussian method with $\sigma = 0.25$.

Table 5.3: Self-organizing map architectures.

Name	Grid size	ϵ_i	ϵ_f	θ_i	θ_f	Epochs
S1	8 × 8	0.4	0.001	5.00	0.01	25
S2	9 × 9	0.4	0.001	5.00	0.01	80
S3	10 × 10	0.4	0.001	5.00	0.01	100
S4	12 × 12	0.28	0.0005	5.00	0.01	80
S5	13 × 13	0.28	0.0005	5.00	0.01	82
S6	16 × 16	0.98	0.0005	5.00	0.01	65

Table 5.4: Dynamic Self-organizing map architectures.

Name	Grid Size	Learning Rate	Elasticity	Epochs
D1	5 × 5	0.0034	0.050	30
D2	6 × 6	0.0034	0.055	30
D3	7 × 7	0.0034	0.057	30
D4	8 × 8	0.0034	0.058	30

It is commonly expected that the prediction accuracy of unsupervised learning methods is lower than networks trained with supervised learning. Unsupervised learning method are not guided to reach the best separability between classes as in supervised training of a deep neural network. Using this dataset, and for a similar number of parameters, a 10% drop in accuracy has been observed for a self-organizing map (Table 5.5) compared to a convolutional neural network (Table 5.2).

As can be seen in Table 5.5, the self-organizing map can reach an accuracy up to 86.60% with a 16 × 16 network. However, this model requires 1152 kiB of memory in single-precision floating point to store its 294 912 parameters. In consequence, this model would not fit in microcontrollers since most of them are limited to less than 1 MiB of ROM. Instead, a 13 × 13 self-organizing map reaching 84.82% of accuracy requires 760 kiB of memory (inference code excluded) which is available on some low-power microcontrollers.

The accuracy of the dynamic self-organizing map is slightly worse than the one of the self-organizing map. Using the same size of 8 × 8, the accuracy drops from 81.57% (S1) to 80.11% (D4) when using the dynamic variant.

Table 5.5: Results using self-organizing maps.

Name	Parameters	Accuracy (%)
Self-organizing map		
S1	73 728	81.57
S2	93 312	81.90
S3	115 200	83.20
S4	165 888	83.79
S5	194 688	84.82
S6	294 912	86.60
Dynamic self-organizing map		
D1	28 800	75.90
D2	41 472	78.60
D3	56 448	79.59
D4	73 728	80.11

5.4.1.3 Results with Supervised Labelling after Unsupervised Learning

It is important to note that the labelling process is still supervised. The results presented in Table 5.5 were obtained with 100% of labels from the training set. However, the labelling method does not need a fully labelled dataset to achieve almost a top accuracy. As it can be seen in Figure 5.6, only 10% of the labelled dataset is needed to almost reach the upper accuracy bound for the self-organizing map S3 with a size of 10 × 10. A k-means clustering of the data with 100 clusters was also performed. The clusters were labelled using the same method. The k-means method can reach a slightly higher accuracy than the self-organizing map with 10% of labels or more, but falls below with less than 5% of labels. As a comparison, the convolutional neural network C7 (similar in terms of number of parameters) has been trained with the same ratio of data. The results are shown in the same Figure 5.6. As it can be observed, the convolutional neural network exhibits a lower accuracy than the self-organizing map when less than 2.5% of the labelled dataset is used.

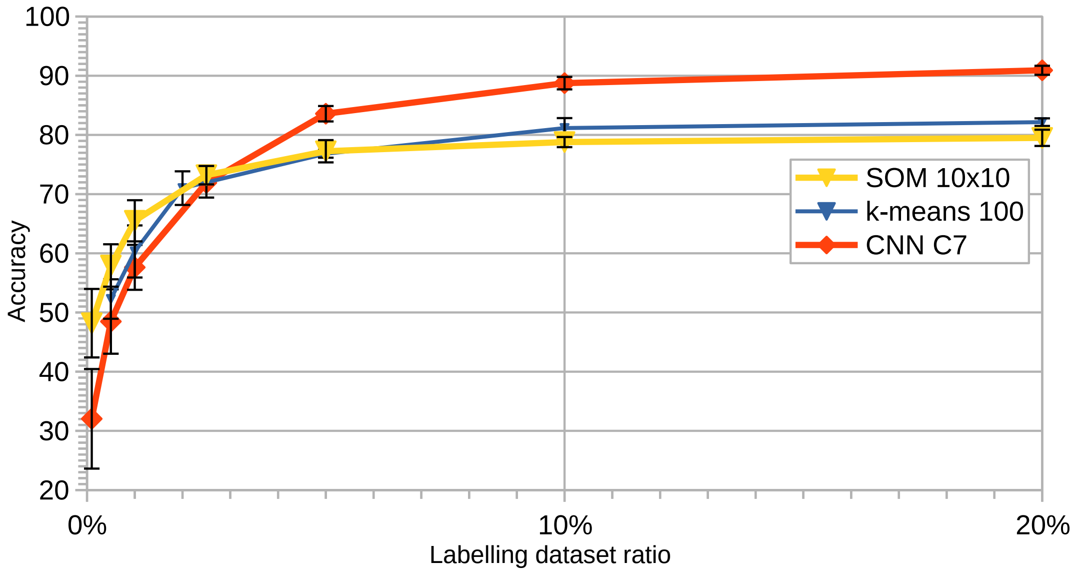


Figure 5.6: Accuracy vs. percentage of the training set used for labelling (SOM, k-means) or learning (CNN)

5.4.1.4 Results with Self-Organizing Maps Learning from Feature Maps

In this experiment, the fully-connected layers of C5 to C8 from Table 5.1 are replaced with the self-organizing map S1 from Table 5.3. To achieve stable learning and better results, it was necessary to add batch normalization and dropout of 50% after each convolutional layer. The accuracy of the original convolutional neural networks is slightly improved (+0.9% on average for the considered models), but it especially seems to provide better features for the self-organizing map to learn. This can lead to an accuracy increase of several percents compared to the self-organizing map as shown in Table 5.6 compared to Table 5.5. In this instance, using larger self-organizing maps does not improve accuracy.

A side effect of using feature maps as an input to a self-organizing map is that the dimension of the weights vector can be smaller. Therefore, the number of parameters for the self-organizing map is lowered as well. For example, the dimensions of the feature maps produced by the feature extractor of C5 are $(61, 10)$, while the dimensions of the raw data are $(128, 8)$ (both channels last). Therefore, each neuron of the self-organizing map has to store a vector of $61 \times 10 = 610$ dimensions, nearly half of the $128 \times 9 = 1152$ required for raw data. With a specially-crafted feature extractor, this number could be further reduced. The number of parameters required for the feature extractor is small compared to the number of parameters for the self-organizing map. The feature extractor of C5 only requires 630 parameters, while C8's requires 19 728 parameters.

Table 5.6: Results using self-organizing maps learning from feature maps.

Name	Parameters	Accuracy (%)
C5+S1	39 670	78.16
C6+S1	79 940	82.09
C7+S1	66 400	85.47
C8+S1	99 600	86.44

5.4.1.5 Results with Unsupervised Fine-Tuning

In this section we evaluate how a dynamic self-organizing map can learn new patterns after a first common learning phase. The largest dynamic self-organizing map from Table 5.4 that fits in the RAM limit of 384 KiB, D4, is chosen for these experiments. D4 is of size 8 × 8 with a total of 73 728 parameters, and therefore a memory footprint of 288 kiB with single-precision floating-point numbers.

Following the experiment using feature maps as inputs of a self-organizing map in Section 5.2.4), the same method is also applied here. The dynamic self-organizing map D4 if fed with feature maps from the convolutional neural network C6 of Table 5.1. Since the dynamic self-organizing map input dimension and dynamic range change, learning rate and elasticity have been set to 0.078 and 0.316 manually, respectively. As a reminder, the weights of the feature extractor from the convolutional neural network are frozen. Therefore, they can be stored in ROM.

The performance is compared to the convolutional neural network C6. We measured the accuracy out of two different cases. This process is illustrated in Figure 5.7.

Both cases start with a general training using the 21 subjects of the original training set. The 9 testing subject are excluded. After the training phase, the neurons of the dynamic self-organizing map are labelled using the method described in section 5.2.3. Then, a test subject is chosen from the 9 test subjects, and its data are split in two to create subject-specific training and testing sets. Half of the vectors are randomly chosen while keeping the classes balanced for the subject-specific training set, and the other half is assigned to the subject-specific testing set. Case n°1 stops there and the accuracy is evaluated on the subject-specific testing set.

In case n°2, an additional unsupervised learning phase with the unlabelled subject-specific training set is performed. The accuracy is then evaluated on the subject-specific testing set.

The vertical lines in Figure 5.8 show the full original test set accuracies for the dynamic self-organizing map D4 and the convolutional neural network C6.

Subject n°4 and subject n°10 have been chosen for their different behaviour. As it can be seen in Figure 5.8 for case n°1, subject n°4 has overall a good accuracy and exhibits a behavior similar to the subjects from the original training set. On the other hand, subject n°10 has one of the worst accuracy and is thereby not well represented within the learning dataset. Subject n°4 has 154 training vectors and 148 testing vectors, while subject n°10 has 147 training vectors and 141 testing vectors.

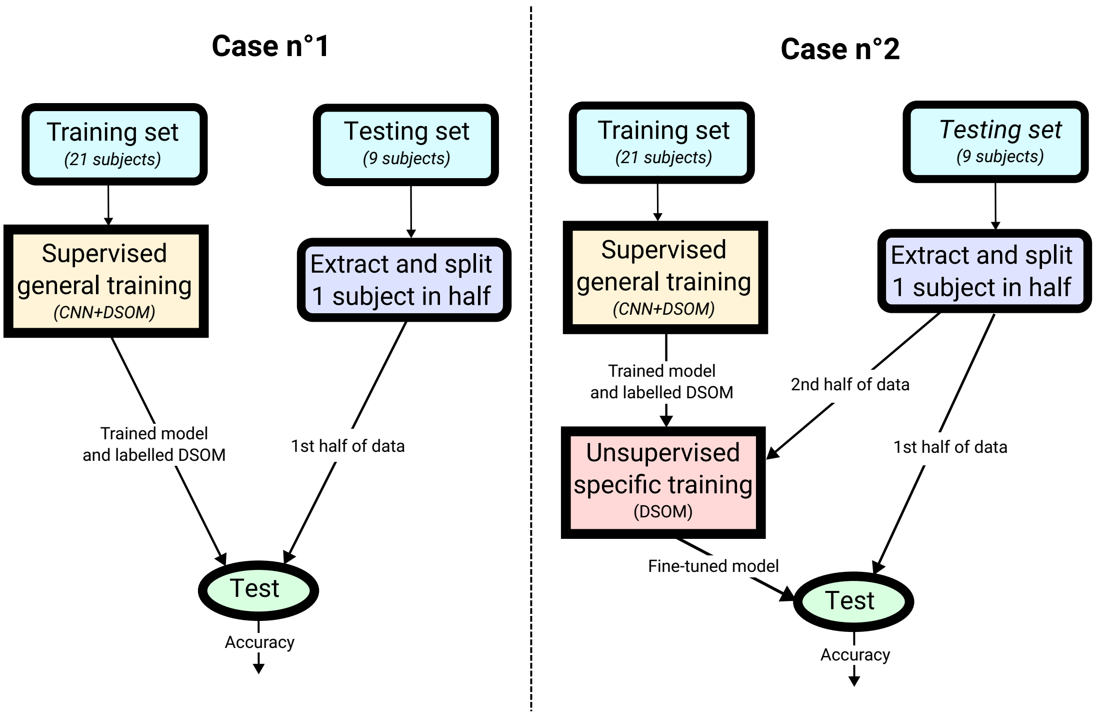


Figure 5.7: Method for unsupervised fine-tuning on UCI-HAR.

Leveraging unsupervised fine-tuning, the dynamic self-organizing map reaches during case n°2 an accuracy close to the original CNN for subject 4 (DSOM: 90.13%, CNN: 90.27%). Using the feature maps as an input to the dynamic self-organizing map further improves the accuracy (CNN+DSOM: 96.87%). However, fine-tuning is not as efficient for subject 10. The reason may be that its behaviour is too far from the general training. Moreover, the reduced amount of available data may not allow the model from adapting well to the different behaviour. The unsupervised nature of the method also prevents adjusting the labels of the neuron if they do not match with the original training.

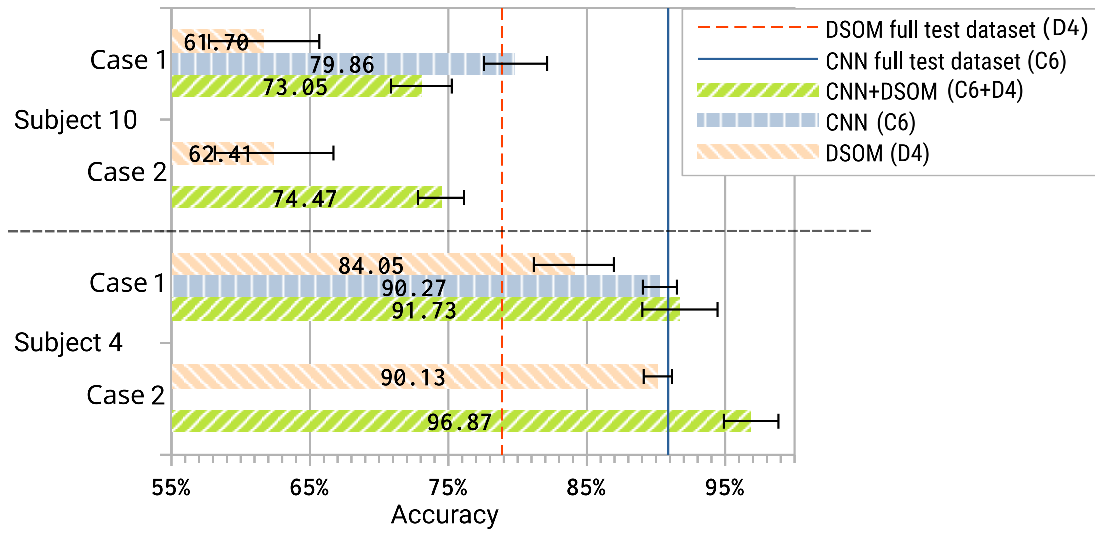


Figure 5.8: Accuracy of test subjects n°4 and n°10 of UCI-HAR with unsupervised fine-tuning.

5.4.2 UCA-EHAR

In this section, our own UCA-EHAR dataset presented in Section 4.3 is used. As a reminder, it is a human activity recognition dataset with data captured from smart glasses from 20 subjects performing 8 activities.

For the following experiments, some subjects have performed additional recording sessions to obtain more data. These subjects, T2, T3, T5 and T20, are used as the testing set while the other subjects are used as the original training set. T2, T3, T5 and T20 were selected because they had performed all of the activities. It was also difficult to obtain new data from the other subjects, after the initial round of recording sessions was over.

Futhermore, in these experiments, the STANDING class was removed. Using only data from an inertial measurement unit in the smart glasses, the confusion betweend the STANDING and the SITTING class is difficult to solve and it would make the results look more confusing. In consequence, 7 activities remain: SITTING, WALKING, LYING, WALKING_DOWNSTAIRS, WALKING_UPSTAIRS, RUNNING, and DRINKING. The resulting distribution of samples for this extended dataset can be seen in Table 5.7.

Figure 5.9 provides an overview of the experiments performed in this section. First, at step n°1, a general supervised training of a ResNet is performed on the original training set. Then, at step n°2, the method presented in Section 5.2.4 to train a dynamic self-organizing map from feature maps is used. The feature maps are generated by a ResNet, still with the original training set. The dynamic self-organizing map is labelled in a supervised manner as presented in 5.2.3.

Step n°3 is dedicated to the fine-tuning on a specific subject from the test set. The dynamic self-organizing map is trained again in an unsupervised manner, but this time using 50% of the one subject's data. At step n°4, the neurons are relabelled using the original training set, in case the fine-tuning caused the class of a neuron to change. Testing is performed on the remaining 50% of the subject's data at the various steps. Labels from the subject's data are not used for training or labelling since it is assumed that they are not available.

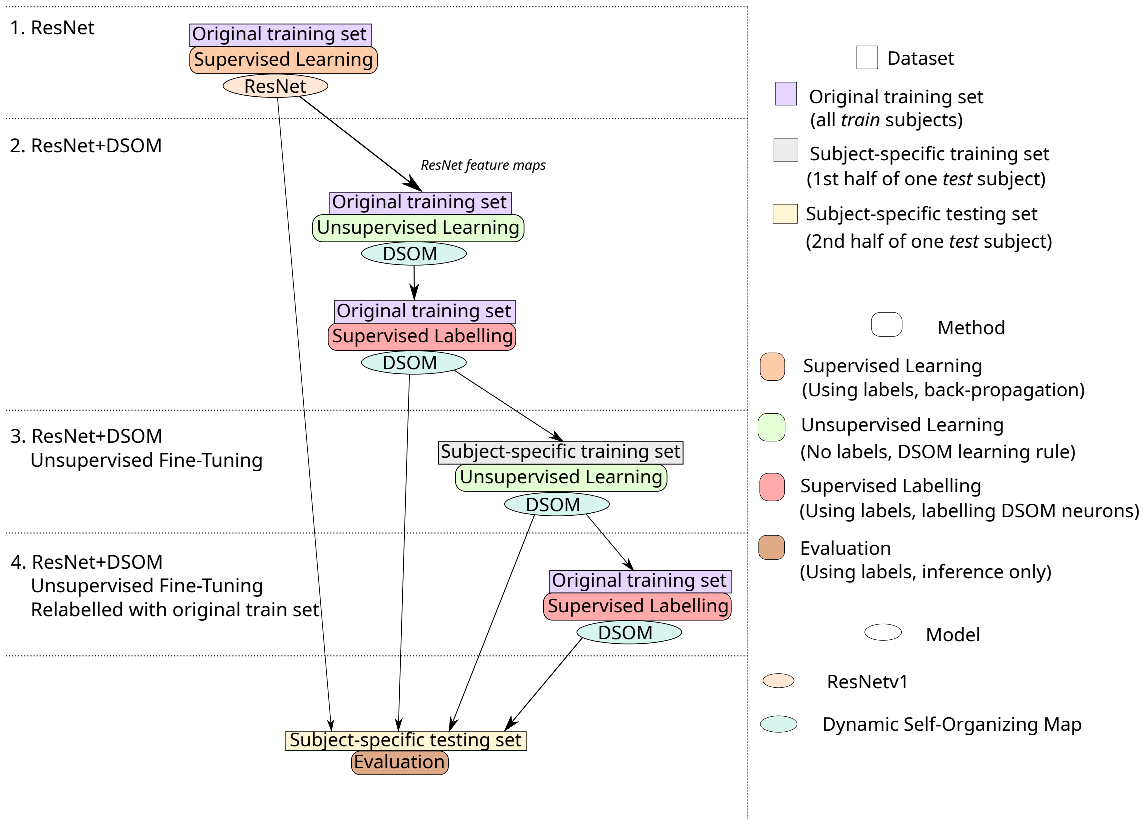

1. ResNet
Original training set
Supervised Learning
ResNet
2. ResNet+DSOM
ResNet feature maps
Original training set
Unsupervised Learning
DSOM
Original training set
Supervised Labelling
DSOM
3. ResNet+DSOM
Unsupervised Fine-Tuning
Subject-specific training set
Unsupervised Learning
DSOM
4. ResNet+DSOM
Unsupervised Fine-Tuning
Relabelled with original train set
Original training set
Supervised Labelling
DSOM
Subject-specific testing set
Evaluation
Dataset
Original training set (all *train* subjects)
Subject-specific training set (1st half of one *test* subject)
Subject-specific testing set (2nd half of one *test* subject)
Method
Supervised Learning (Using labels, back-propagation)
Unsupervised Learning (No labels, DSOM learning rule)
Supervised Labelling (Using labels, labelling DSOM neurons)
Evaluation (Using labels, inference only)
Model
ResNetv1
Dynamic Self-Organizing Map

Figure 5.9: Method for unsupervised fine-tuning on UCA-EHAR.

Table 5.7: Distribution of time samples across subjects and activities for training and testing sets, extended dataset for the fine-tuning scenario.

	Activities							
Subject	**0. SITTING**	**1. WALKING**	**2. LYING**	**3. WALKING_DOWNSTAIRS**	**4. WALKING_UPSTAIRS**	**5. RUNNING**	**6. DRINKING**	**TOTAL**
	Training set							
T1	12 021	9955	5712	1588	1701	4310	4543	39 830
T4	8822	10 871	5578	3132	3496	4002	3754	39 655
T6	16 560	10 144	3199	2420	2305	5464	5093	45 185
T7	16 436	9120	1984	2701	3333	4465	1383	39 422
T8	4024	9378	4289	2145	2156	4064	0	26 056
T9	6074	9578	3276	2596	3399	4015	0	28 938
T10	4954	12 354	4226	1893	1943	4793	0	30 163
T12	8509	10 559	1681	2368	2469	4641	1314	31 541
T13	9957	13 449	12 224	2789	3373	6064	0	47 856
T14	3611	7160	3025	1128	1384	4122	0	20 430
T15	6295	3581	2388	1746	1490	1626	463	17 589
T16	3489	8542	2250	1880	1940	3162	0	21 263
T17	7749	7404	3227	1940	2611	3005	1683	27 619
T18	7784	7110	2412	1299	1590	3210	1288	24 693
T19	4780	3435	2401	1204	1564	1884	1011	16 279
T21	6524	2870	1937	1139	1148	1563	881	16 062
Total	127 589	135 510	59 809	31 968	35 902	60 390	21 413	472 581
	Testing set							
T2	29 719	19 519	11 833	4588	4394	9653	5801	85 507
T3	30 989	27 715	12 457	6295	6490	9106	7513	100 565
T5	22 320	24 458	12 267	4736	5260	11173	4155	84 369
T20	28157	30651	14696	3549	4404	11631	5592	98 680
Total	111 185	102 343	51 253	19 168	20 548	41 563	23 061	369 121
Set	Distribution between sets							
Training	53%	56%	53%	62%	63%	59%	48%	75%
Testing	46%	43%	46%	37%	36%	40%	51%	43%

The deep neural network used for the supervised learning is the ResNetv1-6 already presented in Section 3.4, with 32 filters per convolutional layer. Additionally, a batch normalization layer is introducted after each convolutional layer. The number of epochs for training is set to 350 and the batch size is 768. The Adam optimizer is used with a learning rate of 5×10^{-3} and a weight decay of 5×10^{-4}. The learning rate is divided by 10 at epochs 100, 200, 250, 300, 325, and 335

The hyperparameters of the dynamic self-organizing map used for the unsupervised learning were selected using grid search. The size of the map, the learning rate ϵ, the elasticity η, the σ parameter for the labelling phase, and the number of epochs were all part of the parameter research. The hyperparameter research ended up with the best results using the following parameters: map size of 22×22, $\epsilon = 0.52$, $\eta = 0.71$, $\sigma = 4.88$, and 15 epochs. However, as it will be explained in Section 5.4.2.1, 22×22 is much larger than necessary, with most neurons being unused. Thus, a 8×8 self-organizing map is selected instead, with $\epsilon = 0.52$, $\eta = 1.43$, $\sigma = 4.99$, and 15 epochs. The higher elasticity η also helps to have more neurons from the map being used.

A short analysis for each subject is provided in Appendix D to illustrate the possible outcomes of the method. Statistical results for each subject are provided below in Section 5.4.2.3.

5.4.2.1 Results with Supervised Labelling after Unsupervised Learning

Figure 5.10 shows the ratio of labels required from the training set to achieve the maximum level of performance for the ResNet+DSOM approach, with the 22×22 self-organizing map. As can be seen, very few labels are required in this case. In fact, using more than 5% of labels does not change the result. With 484 neurons in the self-organizing map, it seems unlikely to obtain a correct labelling using only 41 labels (0.14% of the training set). However the accuracy reaches almost 80%.

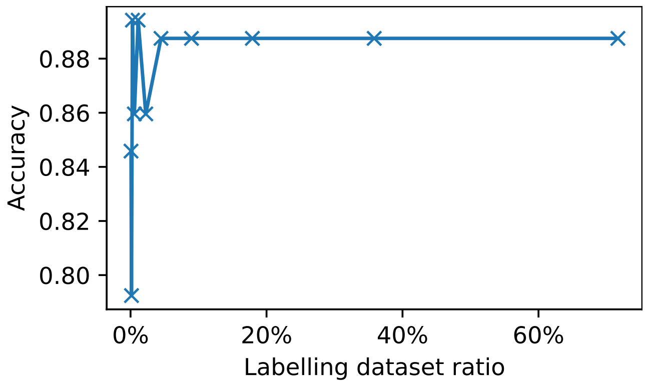


Figure 5.10: Accuracy vs. percentage of the original training set (step n°2) used for labelling of the 22×22 DSOM trained from ResNet feature maps.

This can be explained by looking at a t-SNE (t-distributed Stochastic Neighbor Embedding)[202] visualization of the neurons weights in Figure 5.11. t-SNE is a dimensionality reduction method often used for visualization purposes. The neurons of the self-organizing map have 32 dimensions since the ResNet outputs 32 feature maps of size 1 each. t-SNE reduces the 32 dimensions to 2 dimensions, enabling them to be plotted on a graph. Each point correponds to a neuron, and the number next to it is the class it was assigned during the labelling phase (matching with the order in Table 5.7). The colors correspond to an HDBSCAN clustering method, but it is not relevant here. The distribution of the points looks as if no training was performed on the self-organizing map, and the neurons are simply in their initial state after the uniform initialization. The class n°2 (LYING) has been assigned to most neurons after labelling. However, a few neurons towards the right (circled in magenta) stand out, and were assigned a different class. In fact, the training phase only had an effect on this small group of neurons.

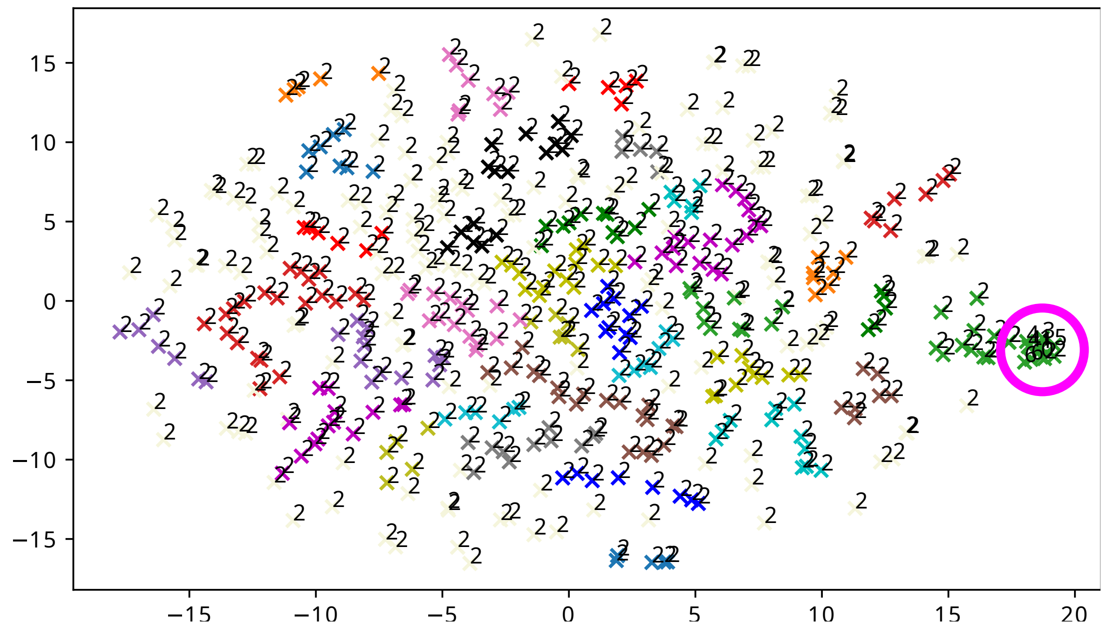


Figure 5.11: t-SNE visualization of the 22 × 22 DSOM neurons prototype vector on UCA-EHAR with labels after step n°2.

Looking at the 2-dimensional neurons grid in Figure 5.12, the neurons that were assigned to a class other than class n°2 are all grouped in the center. We can deduce that the elasticity of this self-organizing map is too small for the neighbourhood to have a significant effect on the training. The neurons far away from the best-matching unit do not learn anything. As a reminder, these parameters were found through parameter research, therefore the self-organizing map performs well even though the size of the map is innapropriate.

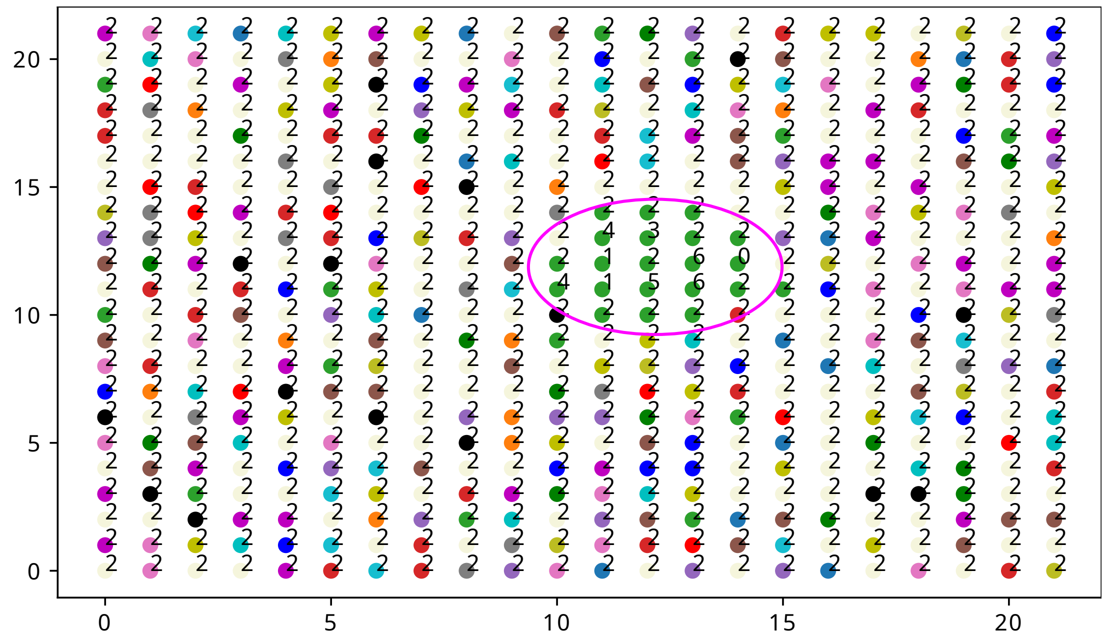


Figure 5.12: Labelled 22 × 22 DSOM 2-dimensional neurons grid on UCA-EHAR after step n°2.

Furthermore, a self-organizing map of size 4 × 4 can also perform well. In this case, even with only 20 labels (0.07%), the self-organizing map could be labelled properly and using more labels does not improve the performance as seen in Figure 5.13. A possible explanation is that the feature extractor already provides a very good separability of the classes. Thus, the self-organizing map only requires very few neurons to represent these classes, as well as very few labels to label them properly.

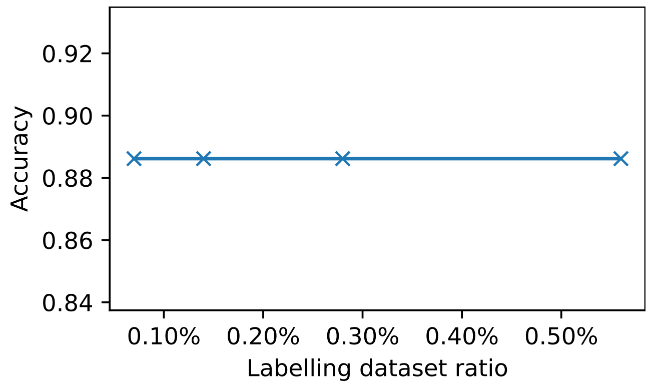


Figure 5.13: Accuracy vs. percentage of the training set used for labelling of the 4 × 4 DSOM trained from ResNet feature maps after step n°2.

It is important to remember that the feature extractor itself was trained with all the labels, therefore this method is overall supervised. However, these results could mean that the dynamic self-organizing map does not have a lot of leeway for fine-tuning, since the bulk of the data separation has already been performed by the feature extractor. As a middle ground and in an attempt to provide more capabilities for the dynamic self-organizing map to readjust itself, a size of 8 × 8 was chosen and the elasticity was increased. The T-SNE visualization for this map shows in Figure 5.14 that this time, all neurons learnt the input data and were labelled accordingly. In this configuration, only 41 labels (0.14%) were required to label the neurons. Nevertheless, all labels were used for the initial labelling (step n°2) and the relabelling processes (step n°4) in the following experiments.

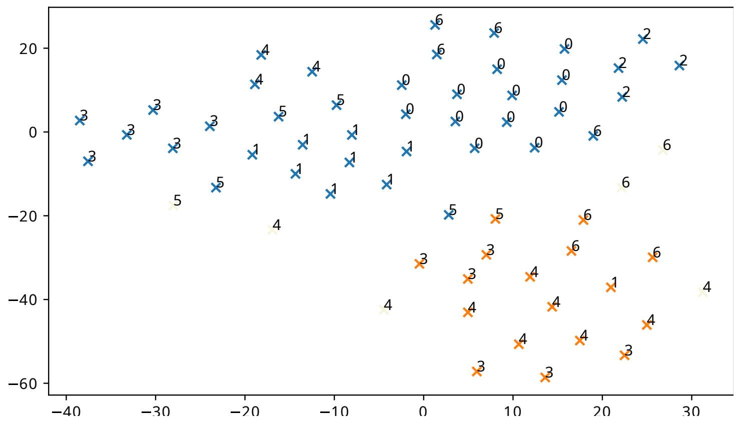


Figure 5.14: t-SNE visualization of the 8 × 8 DSOM neurons prototype vector on UCA-EHAR with labels after step n°2.

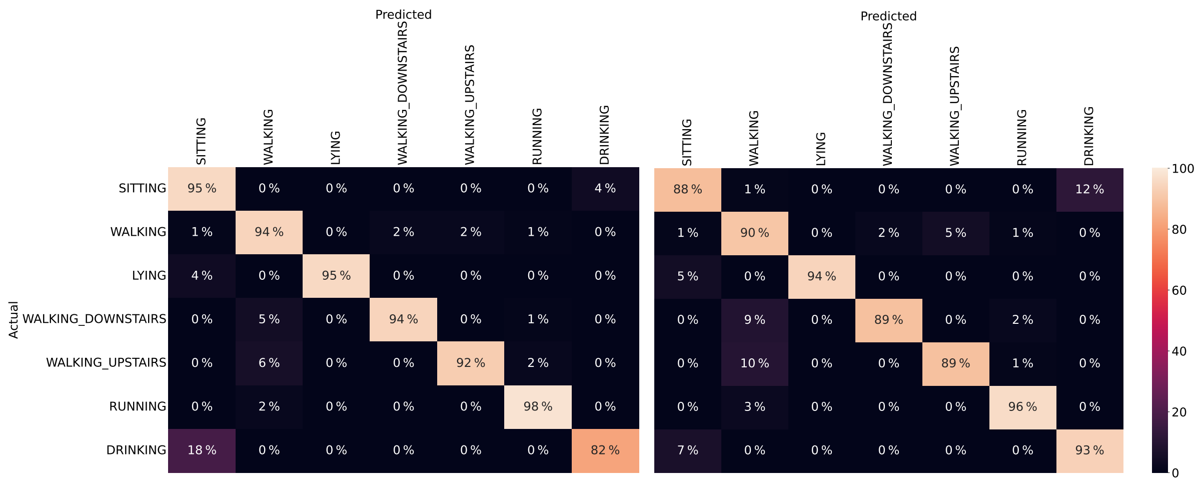
5.4.2.2 Results with Supervised Learning and Self-Organizing Map Learning from Feature Maps

First, we evaluate the supervised training of the ResNet with the new training set. Figure 5.15a shows a confusion matrix similar to Figure 4.6 but without the STANDING class and using subjects T2, T3, T5, and T20 for testing, while the other subjects are used for training. The confusion matrices show the results for a single training only for illustration purposes.

The accuracy figures averaged over 15 runs are presented in Table 5.8. In these results, both the micro accuracy and the macro accuracy are provided since the heavily unbalanced classes can affect both of these metrics in different way. The micro accuracy computes the accuracy across all samples and classes, thus corresponding to the number of correct predictions divided by the total number of predictions across the entire test dataset. The macro accuracy first computes the accuracy for each class separately, then averages the per-class accuracies. The difference is that a high error rate in a class with many samples has the same impact on the macro accuracy as if it was in a class with few samples. However, the impact on the micro accuracy is much more noticeable for a high error rate in a class with many samples than in a class with few samples.

Most of the confusion is between the DRINKING and the SITTING activities. 18% of DRINKING activities are predicted as SITTING and 4% of SITTING activities are predicted as DRINKING. Other classes have much less mispredictions. Some confusion can be noted with the WALKING class as 6% of the WALKING_UPSTAIRS and 5% of the WALKING_DOWNSTAIRS activities are predicted as WALKING. There is also 4% of the LYING activities predicted as SITTING.

When using a DSOM as a classifier after the previous ResNet used as a feature extractor, the confusion is mostly reversed between SITTING and DRINKING as seen in Figure 5.15b. The other mispredictions are slightly worse than with the ResNet.



(a) Step n°1 ResNet (b) Step n°2 ResNet+DSOM

Figure 5.15: Confusion matrices on UCA-EHAR without STANDING and using T2, T3, T5, T20 as testing set.

Table 5.8: Micro and macro accuracy for subjects T2, T3, T5, and T20 of the extended UCA-EHAR dataset with a ResNet and a ResNet+DSOM model.

Name	Micro Accuracy Mean (%)	Micro Accuracy Standard Deviation	Macro Accuracy Mean (%)	Macro Accuracy Standard Deviation
ResNet	94.16	1.28	92.44	0.70
ResNet+DSOM	89.90	1.88	90.80	1.25

5.4.2.3 Summary of Fine-Tuning on Subjects T2, T3, T5, and T20

Results provided in Appendix D presented a set of confusion matrices from one training, and demonstrated the possible outcomes of the various steps of the method presented in Figure 5.9. However, the variability of the trainings with different random initialization creates models that do not all perform the same. Statistical results are difficult to provide with confusion matrices. In this section, a compound metric averaged over 15 different trainings is used instead.

Figure 5.16 presents the micro accuracy obtained for the four subjects at the various steps of the method. Due to the high number of occurences of the SITTING and WALKING activities in the dataset (as previously presented in Table 5.7), the other classes are underrepresented in the micro accuracy. The ResNet+DSOM approach gives an accuracy generally worse than the ResNet itself, and fine-tuning alone does not make up for the loss. It can even deteriorate the result. Relabelling with the original dataset brings the accuracy back up, meaning that some neurons indeed changed classes with the fine-tuning. However the accuracy after fine-tuning and relabelling never exceeds the original ResNet.

Nonetheless, the macro accuracy results shown in Figure 5.17 are more encouraging. However, the standard deviation of the results is high. Therefore, depending on the training, the outcome may not be as good as expected. Additionally, the relabelling requires the original dataset to be available even after the original training and after fine-tuning on the subject. Since the original labelling does not require all the labels, the relabelling step may not require all labels either, but no experiment was done to confirm this so far.

Finally, the effect of catastrophic forgetting mentioned in Section 2.6.2 has not been evaluated. Forgetting the original information after fine-tuning on the subject is not a major issue, since the model is only evaluated on this subject afterwards. On the other hand, if the classes in the new data are not shuffled but rather presented sequentially, then catastrophic forgetting would come into play. As a result, the accuracy of a class not seen for a while may be impacted.

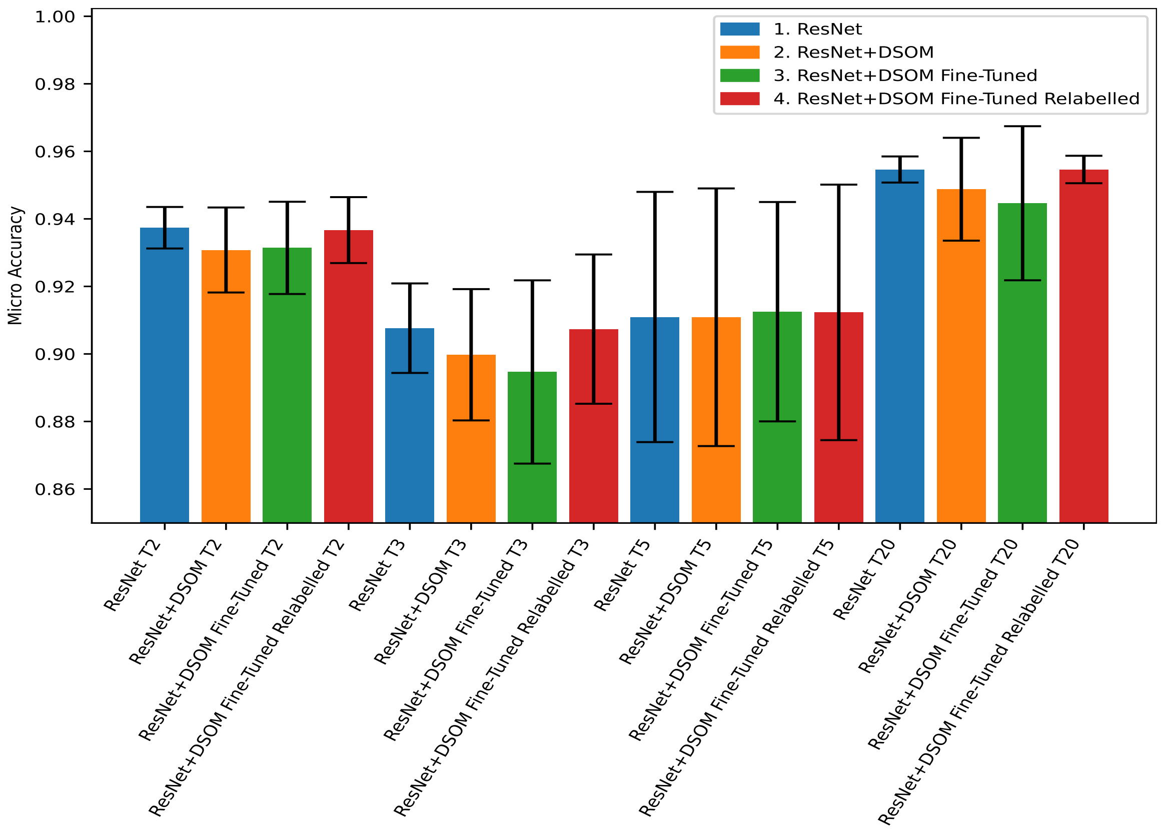


Figure 5.16: Micro accuracy for the ResNet, ResNet+DSOM, Fine-Tuned ResNet+DSOM, and Fine-Tuned and Relabelled ResNet+DSOM for subjects T2, T3, T5, and T20 of UCA-EHAR

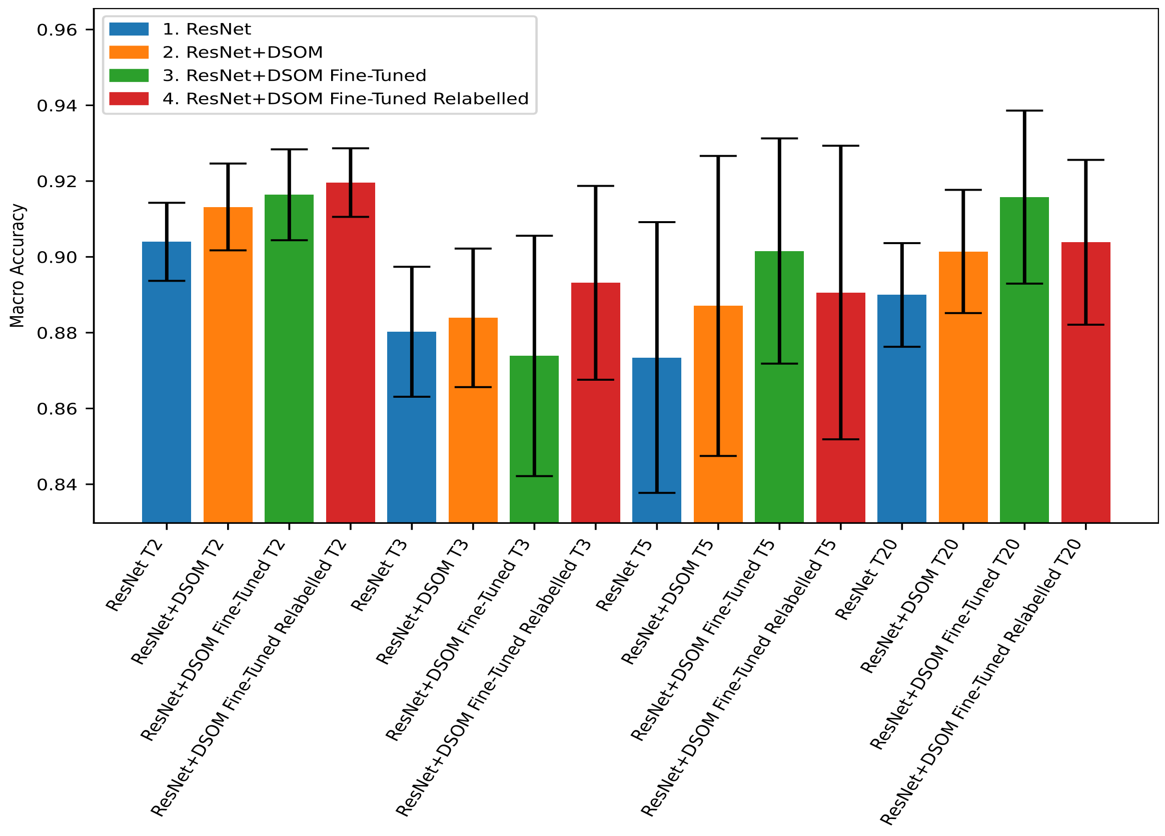


Figure 5.17: Macro accuracy for the ResNet, ResNet+DSOM, Fine-Tuned ResNet+DSOM, and Fine-Tuned and Relabelled ResNet+DSOM for subjects T2, T3, T5, and T20 of UCA-EHAR

5.4.3 Heidelberg Digits

The Heidelberg Digits dataset is originally part of the Spiking Heidelberg Datasets[168], designed for evaluation of spiking neural networks. However, the data before conversion to spikes are also provided. The Heidelberg Digits dataset provides 48 kHz 16-bit mono PCM audio recordings for English and German digits from 0 to 10. 12 subjects participated in the construction of the dataset, and the subject number is present for each recording. It is therefore possible to study each subject separately. Each subject recorded between 35 and 64 recordings for each digit in each language. In our experiments, English and German languages are grouped under the same class, so there are 10 classes in total, one for each digit. This is done in order to reduce the number of classes for easier analysis.

The experiments presented in this section are similar to the ones presented in the previous Section 5.4.2, albeit using a different dataset. A ResNetv1-8 with one more residual block than the ResNetv1-6 presented in Section 3.4 is used. Other changes have been made to accomodate the processing of raw audio data as illustrated in Figure 5.18. In particular, the first convolutional layer of the model needs to have a large kernel size[203]. Input data is normalized with min-max normalization and zero-padded to the largest recording (65 872 samples). The network is trained for 175 epochs with a batch size of 192 using the Adam optimizer with an initial learning rate of 0.005 and weight decay set to 5×10^4. The learning rate is divided by 10 at epochs 50, 100, 125, 150, 165 and 170.

The dynamic self-organizing map hyperparameters are found through hyperparameter research with grid search. The parameters used here are: grid side of 5×16, learning rate $\epsilon = 0.622$, elasticity $\eta = 1.29$, $\sigma = 0.127$ for labelling, and 27 epochs.

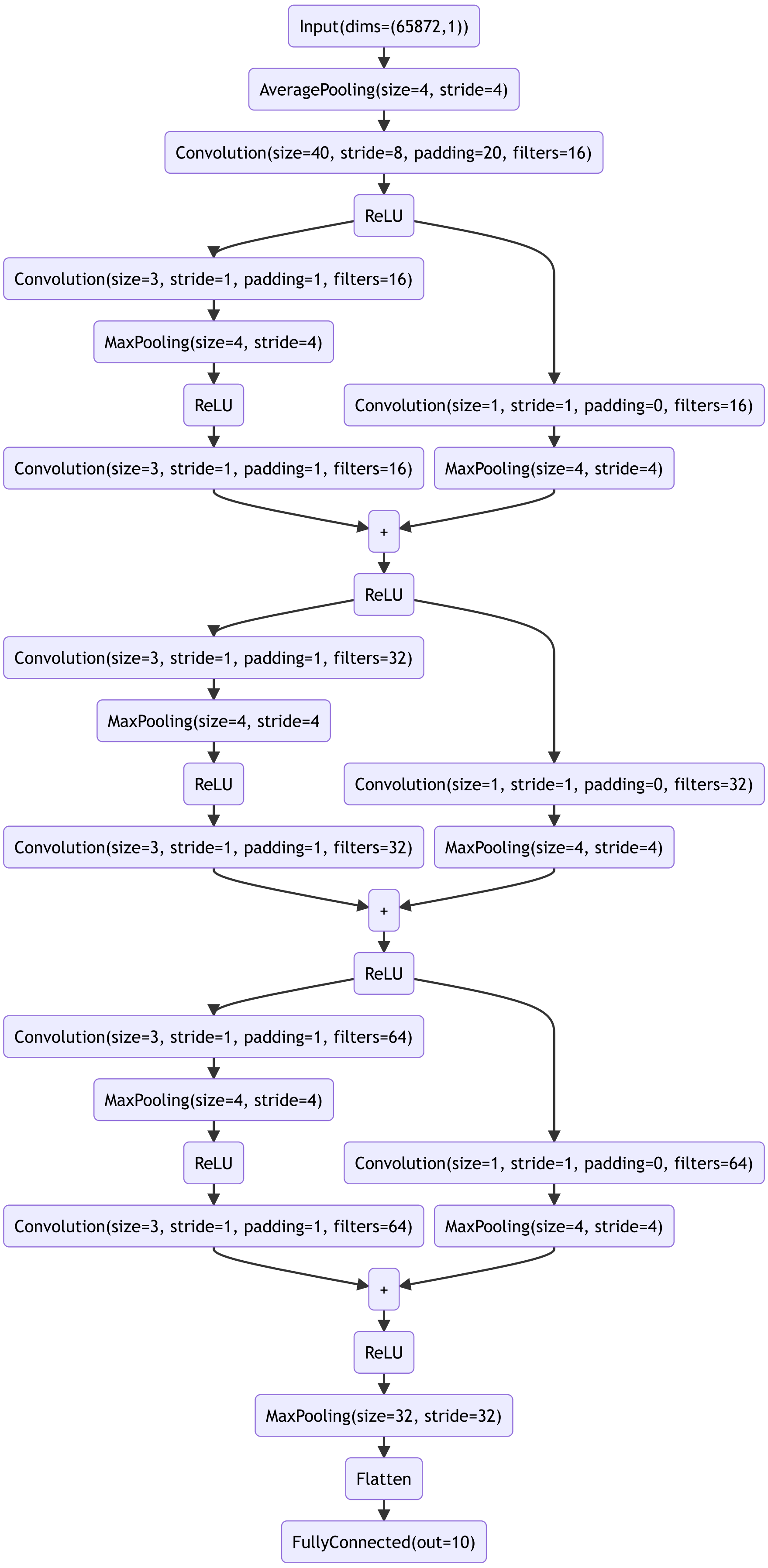
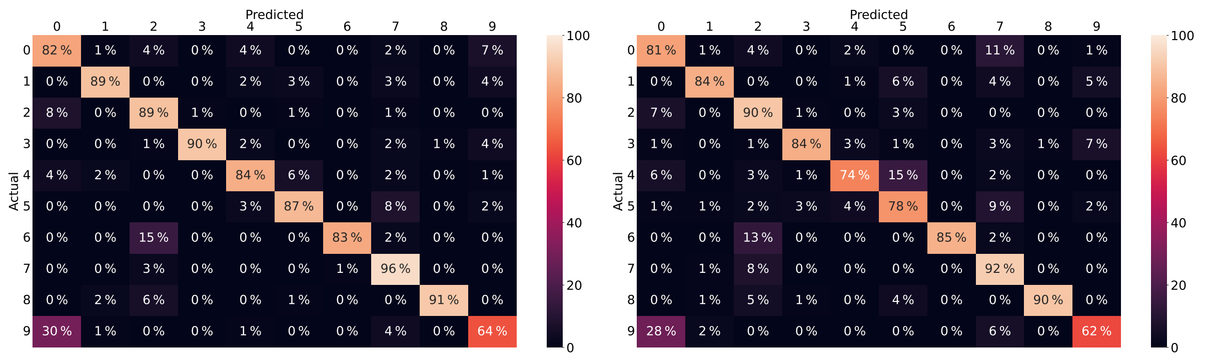

Input(dims=(65872,1))
AveragePooling(size=4, stride=4)
Convolution(size=40, stride=8, padding=20, filters=16)
ReLU
Convolution(size=3, stride=1, padding=1, filters=16)
MaxPooling(size=4, stride=4)
ReLU
Convolution(size=1, stride=1, padding=0, filters=16)
Convolution(size=3, stride=1, padding=1, filters=16)
MaxPooling(size=4, stride=4)
+
ReLU
Convolution(size=3, stride=1, padding=1, filters=32)
MaxPooling(size=4, stride=4
ReLU
Convolution(size=1, stride=1, padding=0, filters=32)
Convolution(size=3, stride=1, padding=1, filters=32)
MaxPooling(size=4, stride=4)
+
ReLU
Convolution(size=3, stride=1, padding=1, filters=64)
MaxPooling(size=4, stride=4)
ReLU
Convolution(size=1, stride=1, padding=0, filters=64)
Convolution(size=3, stride=1, padding=1, filters=64)
MaxPooling(size=4, stride=4)
+
ReLU
MaxPooling(size=32, stride=32)
Flatten
FullyConnected(out=10)

Figure 5.18: ResNetv1-8 for Heidelberg Digits.

In order to select a relevant train and test split to see the effect of fine-tuning, each subject is extracted from the dataset as a test subject, and the model is trained on the other subjects. The resulting confusion matrix is then analyzed to select subjects with the highest confusion. The confusion matrices for all of the 12 subjects are provided in Appendix E.

5.4.3.1 Results with Supervised and Self-Organizing Map Learning from Feature Maps

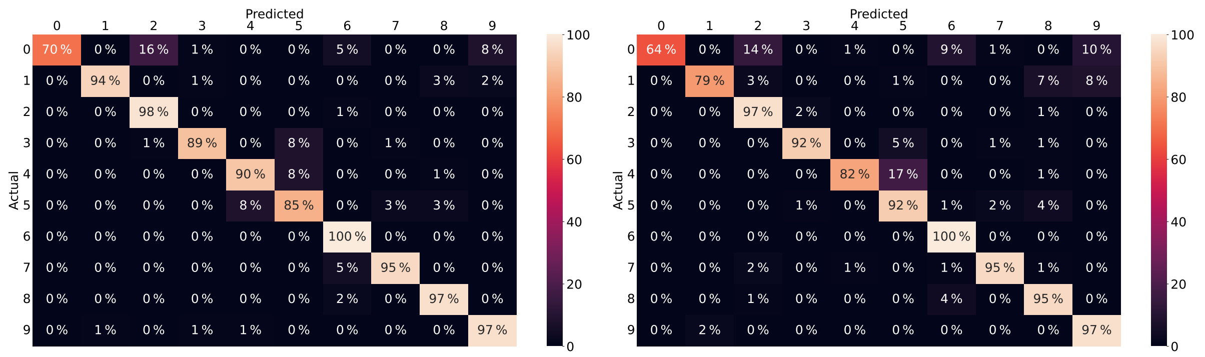
The first set of test subjects is made of subjects number 0, 2, and 11. The other subjects are used as the training set. These subjects have the highest confusion among all subjects. A confusion matrix for training of the ResNet for this set of three subjects is provided in Figure 5.19a. The accuracy for this training is of 85.39%. Then, features extracted from the Resnet are used as inputs of a DSOM. Once the training phase is over, the DSOM neurons are labelled with all the available labels. As can be seen in Figure 5.19b, the confusion increases for most classes, with a resulting accuracy of 81.99%.



(a) ResNet (b) ResNet+DSOM

Figure 5.19: Confusion matrices of Heidelberg Digits with subjects 0, 2, 11 as test set

Similarly, confusion matrices are provided in Figure 5.20 when selecting subjects 3, 7, and 8 as a testing set, with the other subjects as the training set Subjects 3, 7, and 8 have more accurate predictions than the first set with subjects 0, 2, and 11 overall. The accuracy for this training of a ResNet is 91.54%, and decreases to 89.25% with the ResNet+DSOM model.



(a) ResNet (b) ResNet+DSOM

Figure 5.20: Confusion matrices of Heidelberg Digits with subjects 3, 7, 8 as test set

5.4.3.2 Summary of Fine-Tuning

A short analysis of the fine-tuning method for subjects 0, 2 and 3 is provided in Appendix F to illustrate the possible outcomes of the method on the Heidelberg Digits dataset.

Figure 5.21 presents the average accuracy over 15 trainings for the various steps of the method on the 6 subjects (the first set of subjects 0, 2, and 11 and the second set of subjects 3, 7, and 8). Only the micro accuracy is presented since the classes are balanced. Unfortunately, for these subjects of the Heidelberg Digits dataset, unsupervised fine-tuning using the subject's data does not provide an improvement over the original ResNet. Classification performance is often even degraded. Unlike the UCA-EHAR dataset, the studied subjects from Heidelberg Digits did not have additional data captured for this fine-tuning process. Only 380 to 525 vectors for each subject of the Heidelberg Digits dataset have been used for fine-tuning. In comparison, between 2635 and 3141 vectors for each subject of the UCA-EHAR dataset have been used for fine-tuning. The lack of data could be a reason for the lower observed performance. Moreover, the analysis of the behaviour of the training self-organizing map was less thorough with the Heidelberg Digits dataset than with the UCA-EHAR dataset. A more in-depth analysis may reveal a subobtimal configuration of the training as it was seen initially for the UCA-EHAR dataset.

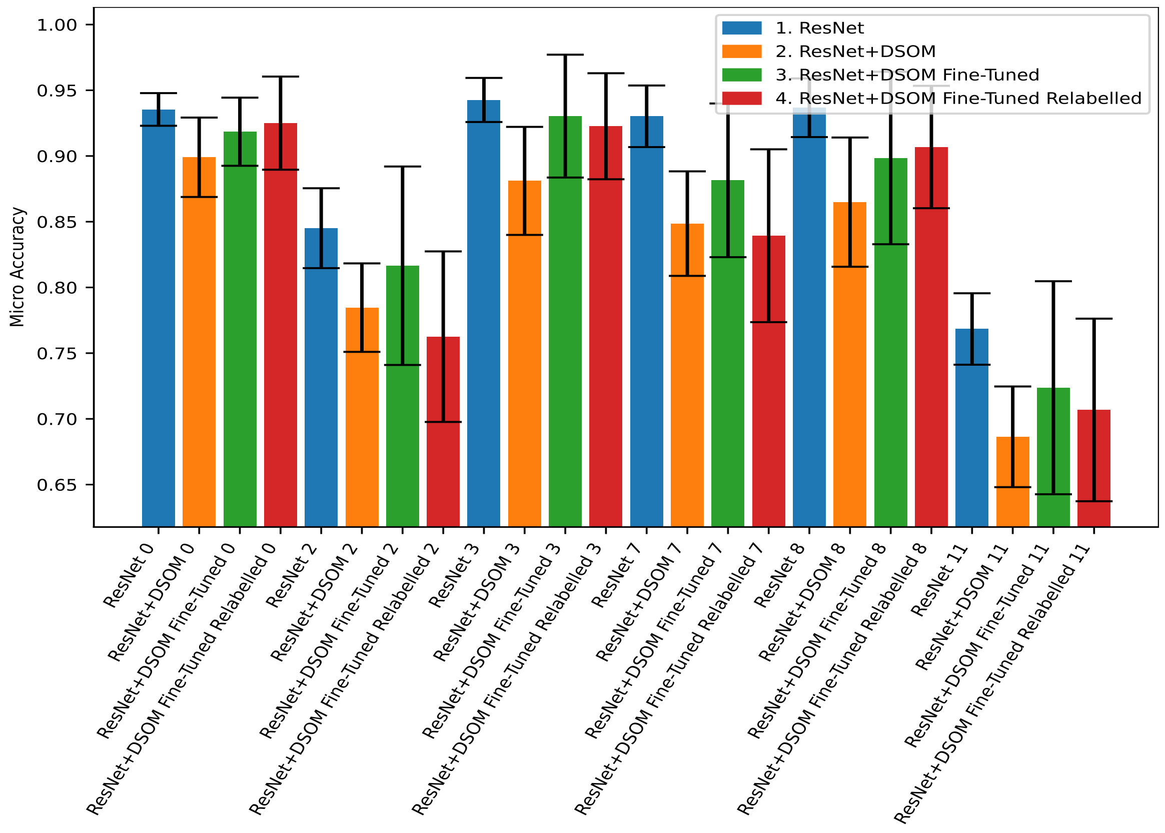


Figure 5.21: Micro accuracy for the ResNet, ResNet+DSOM, Fine-Tuned ResNet+DSOM, and Fine-Tuned and Relabelled ResNet+DSOM for subjects 0, 2, 3, 7, 8, and 11 of Heidelberg Digits

5.5 Conclusion

In this chapter, we presented a method leveraging dynamic self-organizing maps that can be used to perform online fine-tuning in an unsupervised manner. In order to solve a classification problem, some labels are still required to identify which class a neuron of the self-organizing map corresponds to. In order to improve the classification results, the self-organizing map can be preceded by a feature extractor previously trained in a supervised manner.

An initial comparison between supervised training of convolutional neural networks, unsupervised training of self-organizing maps, and the hybrid approach combining a feature extractor and a self-organizing map was led on the UCI-HAR dataset. Despite this method still requiring some labels, less than 20% can be used in this case without compromising on the accuracy. We also showed that using a feature extractor to provide feature maps for learning to the self-organizing map can increase the accuracy by 10%. While this is not useful in an offline context with the original dataset since the feature extractor has to be trained in a supervised manner, it can be used later with unsupervised online learning of a self-organizing map. For online learning, the self-organizing map is replaced with a dynamic self-organizing map to remove the time dependency. In the context of human activity recognition, unsupervised online learning is useful to perform fine-tuning of the self-organizing map on a specific test subject. Results showed that for one of the chosen subject, exhibiting a poor accuracy with a fully-supervised convolutional neural network, unsupervised fine-tuning did not improve the performence. However, another subject saw its accuracy improved by a few percents after fine-tuning, even exceeding the original convolutional neural network.

The unsupervised fine-tuning method was then applied to our own dataset, UCA-EHAR, using a residual neural network as a feature extractor. We found that this type of convolutional neural network can provide a better accuracy and feature maps of smaller dimensions. Moreover, the UCA-EHAR dataset was extended to get more data for some subjects. The results over four subjects were also mixed, and the method may require relabelling at least with part of the original dataset after fine-tuning. While the micro accuracy did not improve, the macro accuracy still slightly increased on average, showing that the fine-tuning can at least improve the accuracy of the classes with less occurences.

Then, we also experimented unsupervised fine-tuning on a different modality, using a keyword spotting dataset, Heidelberg Digits. The deep neural network was also a residual neural network, processing the raw audio data. With this dataset, the fine-tuning results were generally not satisfactory. However, we observed in one case an improvement, and in the other cases the accuracy was not far off from the supervised residual neural network. Therefore, while the method still needs to be improved, it can be applied to various datasets.

In the case of keyword spotting, it could be possible to use a feature extractor trained on a large dataset, e.g. Google Speech Commands[192], and then reuse it with a different set of keywords in a transfer learning fashion. The training of the self-organizing map would be performed in an unsupervised manner with these new keywords, and only a reduced number of labels could be used to label the neurons. Later on, the model could be fine-tuned online without requiring labels to a specific speaker.

In the next chapter, an implementation of this method in an embedded context will be presented, along with memory, energy, and latency metrics on a microcontroller.

Chapter 6

Embedded Self-Organizing Maps and On-Device Fine-Tuning

Contents

6.1 Introduction

As seen in Chapter 3, it is possible to perform deep neural network inference on microcontrollers. However, the deep neural network model still has to be trained on a workstation or in the cloud. Indeed, the resources required for training a deep neural network largely exceed what a microcontroller can provide, both in terms of memory and processing power.

As illustrated in Figure 6.1, the data have been collected from the device to build the UCA-EHAR dataset as well as to perform live activity recognition in Chapter 4. In Chapter 5, we presented the semi-supervised learning approaches along with the unsupervised fine-tuning process, but performed on a workstation.

Nonetheless, fine-tuning a small part of a neural network in an online manner could be performed on a resource-constrained device. Therefore, in this chapter, we study the feasibility of implementing the unsupervised online fine-tuning process on a microcontroller as an on-device learning (ODL) method.

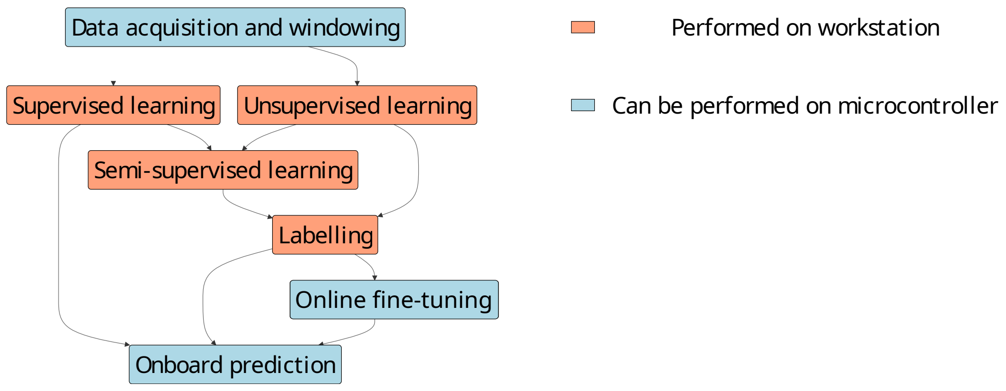


Figure 6.1: On-Device Steps for Embedded Neural Networks.

As a reminder, fine-tuning is applied onto a dynamic self-organizing map learning from the feature maps of a convolutional neural network. To optimize the inference time and the memory footprint, convolutional neural networks were already quantized and implemented with fixed-point computation in Chapter 3. The dynamic self-organizing map also needs to be quantized in order to benefit from a reduced memory footprint and faster computation. Unlike convolutional neural networks, both the inference and the training of the dynamic self-organizing map have to be implemented using fixed-point numbers. This requires implementing a fixed-point exponential function in order to compute the neighbourhood function of Equation 5.9, copied below:

$$h(i,s,v) = e^{-\frac{1}{\eta^2}\frac{d_1(p_i,p_j)^2}{d_2(v,w_s)^2}} \tag{5.9}$$

Finally, the dynamic self-organizing map is integrated in our MicroAI tool to enable seemless integration in a deep neural network and automatic deployment on microcontrollers. The impact on the memory footprint and energy consumption on the target can then be evaluated.

This chapter is organized as follows. First, Section 6.2 describes the integration of the dynamic self-organizing map layer in our framework for quantization and deployment on microcontrollers. Then, Section 6.3 details the method chosen for the exponential function computation in the learning rule of the dynamic self-organizing map with fixed-point numbers. Section 6.4 presents the memory footprint and latency results using this method. Finally, Section 6.5 concludes this chapter.

6.2 Quantization and Deployment of Self-Organizing Maps

In order to automatically perform the training, quantization and deployment of the self-organizing map and the hybrid network (a convolutional feature extractor and a self-organizing map classifier), self-organizing maps support has been added to our MicroAI framework. As presented in Section 5.3, the self-organizing map model has been implemented as a custom PyTorch layer for the training phase performed on the workstation.

To reduce the memory footprint and support fixed-point computation on the target, self-organizing maps can be quantized. The quantization is performed in the same way as typical deep neural network layers such as convolutional layers. As a result, both post-training quantization and quantization-aware training are supported. Inputs and weights are quantized as described in Section 3.2.2. In order to support on-target online learning with fixed-point numbers, the hyperparameters must also be quantized. Therefore, the learning rate and the elasticiy are quantized alongside the weights. For the sake of simplicity, the same scale factor is used for the hyperparameters and the weights.

Then, in order to perform the inference and the fine-tuning on a microcontroller, the dynamic self-organizing map is also implemented as a portable C code. For inference, the processing requires computing the Euclidean distances between the input vector and each neuron, and finding the neuron with the smallest distance to the input vector. The computation of the Euclidean distances (Equation 5.1) relies on accumulating the element-wise squared difference between the input and the neuron vectors, and taking the square root of this accumulator. As we only need to know the smallest distance, we can compare the squared distances directly, without needing the square root computation:

$$(d_2(v, w_i))^2 = \sum_{j=0}^{n} (w_{i_j} - v_j)^2 \tag{6.1}$$

Furthermore, in case of fixed-point computation, the accumulation of the squared differences could be optimized with the use of the `SMLAD` instruction as described in Section 3.3.7. The `SMLAD` instruction can perform two multiply-accumulate operations in a single cycle. However this optimization is not yet implemented.

Fine-tuning also requires the learning rule of the dynamic self-organizing map presented in Section 5.2.5 to be implemented. Despite this being straightforward with floating-point numbers, the exponential function computation is challenging with fixed-point numbers.

6.3 Fixed-Point Computation of the Exponential Function

On-target fine-tuning requires implementing the learning rule of the self-organizing map (see Equation 5.8). The learning rule relies on the computation of an exponential function in the neighbourhood function (see Equation 5.9). Using floating-point computation, the standard C math library can be used since it implements the `float expf(float)` function. However, there is no standard implementation of a fixed-point exponential function. In fact, it is difficult to find a portable implementation of such a function in C. When using 8-bit integers, it is possible to pre-compute a lookup table, which would only requires $2^8 = 256$ bytes of memory. However, a 16-bit lookup table would require $2^{16} \times 2 = 131$ kiB of memory, which is not reasonable to embed on a microcontroller. Therefore, it is necessary to find an algorithm to compute this function with fixed-point numbers, without requiring a large lookup table or a conversion to floating point.

6.3.1 Method

Our solution is based on a library providing the implementation of various fixed-point mathematical functions in the Solidity programming language [204].

This method computes an exponential function with a real-valued exponent without using a full lookup table and without using a polynomial approximation. All the computations are performed in fixed-point so a common intermediate scale factor has to be used. The number of bits for the fractional part is chosen as the maximum between the number of bits for the fractional part of the weights and the input of the layer. Futhermore, all intermediate computation use a `long_number_t` type as defined in Section 3.3.4, larger than the data type used to quantize the weights and activations in order to reduce the possibility of overflows.

There are two main ideas:

- compute a power-of-two with a real-valued exponent instead of the exponential function by changing the exponentiation base,
- compute the power-of-two of the fractional part of the exponent by multiplying by n-th roots of 2.

Changing the exponentiation base can be done since:

$$e^x = 2^{x \log_2 e} \tag{6.2}$$

$\log_2 e$ is a constant and can be pre-computed then converted to a fixed-point number.

The integer part of the exponent of the power-of-two is extracted by shifting to the right by the number of bits allocated to the fractional part. The integer part is computed first then the fractional part, since:

$$2^{a+b} = 2^a \times 2^b \tag{6.3}$$

The computation of the integer part of the power-of-two's exponent is trivial since it simply requires shifting 1 by the integer part of the exponent, to the left if the exponent is positive or to the right if the exponent is negative. In the case of the DSOM neighbourhood function (Equation 5.9), the exponent is always negative.

The computation of the fractional part is an iterative process, successively multiplying n-th roots of two since:

$$2^{2^{-n}} = \sqrt[2^n]{2} \tag{6.4}$$

In a fixed-point number, the most significant bit of the fractional part has a weight of 2^{-1}. Subsequent bits have decreasing weights of $2^{-(1+m)}$, with m the distance to the most significant bit. Therefore, for every bit set to 1 in the fractional part, the temporary value of the power-of-two computation is multiplied by the 2^{1+m}-th root of 2. After each multiplication, the fixed-point result is scaled back to the same number of bits for the fractional part as the operands in order to avoid overflow. If a bit is set to 0, the value stays unmodified.

An example of the fixed-point exponential function computation is provided in Appendix H.

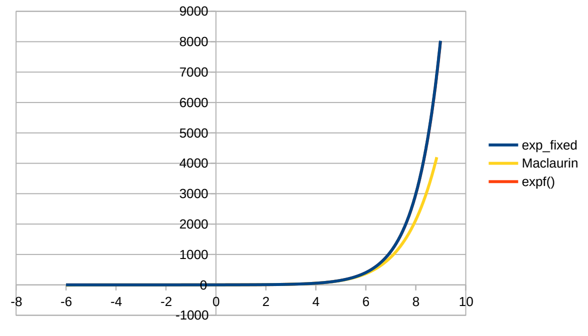
The n-th roots of 2 are pre-computed and converted to fixed-point numbers. For a more accurate computation, the fixed-point conversion of these constants rounds to the nearest instead of rounding down. The number of n-th roots of 2 to pre-compute and store is equal to the number of bits allocated to the fractional part. It also corresponds to the number of conditional operations in this iterative process.

If the function that computes the power of two receives a number that is too small and could cause an underflow, 0 is returned early. Similarly, if the number is too large and could cause an overflow, a very large number is returned.

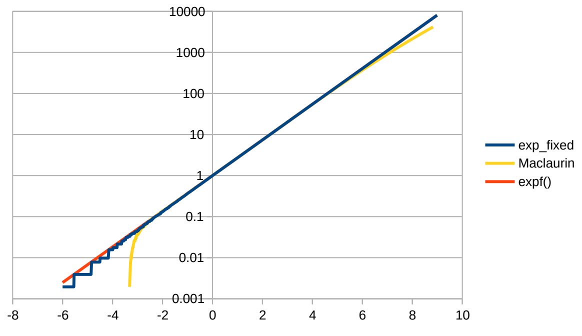
6.3.2 Precision of the Fixed-Point Exponential Function

To analyze the error, the exponential function computed with 32-bit fixed-point numbers is compared to an exponential function computed with single-precision floating-point numbers to analyze the error. The fixed-point implementation is the one described previously with a Q23.9

format, while the floating-point implementation is the `expf()` function of the GNU C library. Results with a fixed-point polynomial approximation using the first 10 terms of the Maclaurin series for the exponential function are also provided.



(a) Linear scale



(b) Logarithmic scale

Figure 6.2: Fixed-point and floating-point exponential functions for $x \in [-6, 9[$.

Figure 6.2a shows that our implementation of the exponential function with fixed-point numbers matches well with the floating-point reference, while the Maclaurin series becomes less accurate for larger numbers above $x = 6$ with only 10 terms. The computation of the Maclaurin series overflows above $x = 8.85$. The logarithmic scales in Figure 6.2b reveals that our fixed-point implementation starts to significantly deviate below $x = -3$. The reason is that the precision given by a Q23.9 format leads to a quantization step of approximately 0.002. The Maclaurin series with 10 terms significantly drops below both our fixed-point implementation and the floating-point reference also at around $x = -3$. Furthermore, for our implementation in Q23.9 format, below $x = -6.24$ the result drops to 0 since $e^{-6.24} < 2^{-9}$, and above $x = 9.02$ the computation would overflow so the result is capped at 8192 instead. This is the reason for the choice of the $[-6, 9[$ range to plot the functions.

Our approach can require more computation than a Maclaurin series with few terms if many bits of the fractional part of the power-of-two exponent are set to 1, however it stays more accurate at the beginning and at the end of the range. In the case of dynamic self-organizing maps, the upper end of the range is not relevant, but high precision at the lower end of the range is important. The exponential function is indeed computed only with negative exponents in Equation 5.9.

To compute small and large exponents with the Maclaurin series, more terms could be used to obtain a better precision. This is the choice made by libfixmath[205] where up to 30 terms are computed for a 16-bit fixed point number. In this case, early exit is used to stop the computation if computing a new term does not significantly change the precision. Alternatively, lookup tables can be used to assist in the computation of some parts of the exponent. This is for example the approach chosen by the neuromorphic multiprocessor system SpiNNaker[206].

6.4 Results

The following results focus first on evaluating the possible negative effect of self-organizing map quantization on the classification performances. Then, the memory footprint and the inference time after the deployment of self-organizing maps on a microcontroller are analyzed.

Some of the datasets and neural network configurations presented in Section 5.4 are reused here. In particular, the evaluation is performed on the UCI-HAR and the UCA-EHAR datasets.

6.4.1 Quantization of Self-Organizing Maps

In this section, the effect of quantization on self-organizing maps is shown. The quantization method of Section 3.2 is applied to self-organizing maps as described in Section 6.2.

6.4.1.1 UCI-HAR dataset

The UCI-HAR dataset is first used to show the effect of quantizing a self-organizing map learning from raw data. The dynamic self-organizing maps of Table 5.4 are quantized on 16 bits and 8 bits. The 16-bit quantization uses Post-Training Quantization (PTQ), while the 8-bit quantization uses either Post-Training Quantization or Quantization-Aware Training (QAT).

As seen in Figure 6.3, the 16-bit quantization (int16 PTQ) keeps an accuracy very close to the baseline (float32). The difference in accuracy for the D4 model is only of 0.2%. The 8-bit post-training quantization (int8 PTQ) has a slightly lower accuracy than the baseline overall. However, the difference is at most of 0.5% for the D2 model. Quantization-aware training with 8 bits (int8 QAT) was also performed to evaluate if the accuracy can be improved compared to post-training quantization. As can be observed, quantization-aware training does not improve the accuracy.

Nevertheless, Figure 6.4 shows that the 8-bit quantization is the most efficient since the accuracy drop is very low. It is worth noting that the accuracy was already poor for the non-quantized model when learning from raw data (float32).

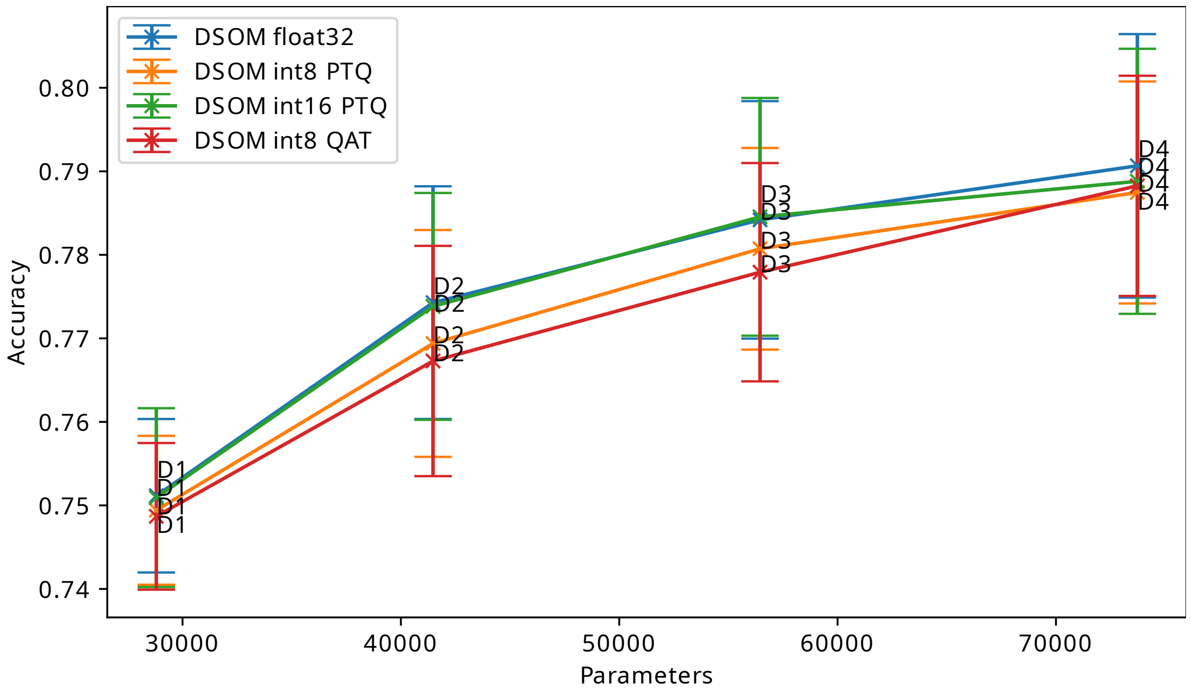


Figure 6.3: Accuracy vs. number of parameters for dynamic self-organizing maps on UCI-HAR.

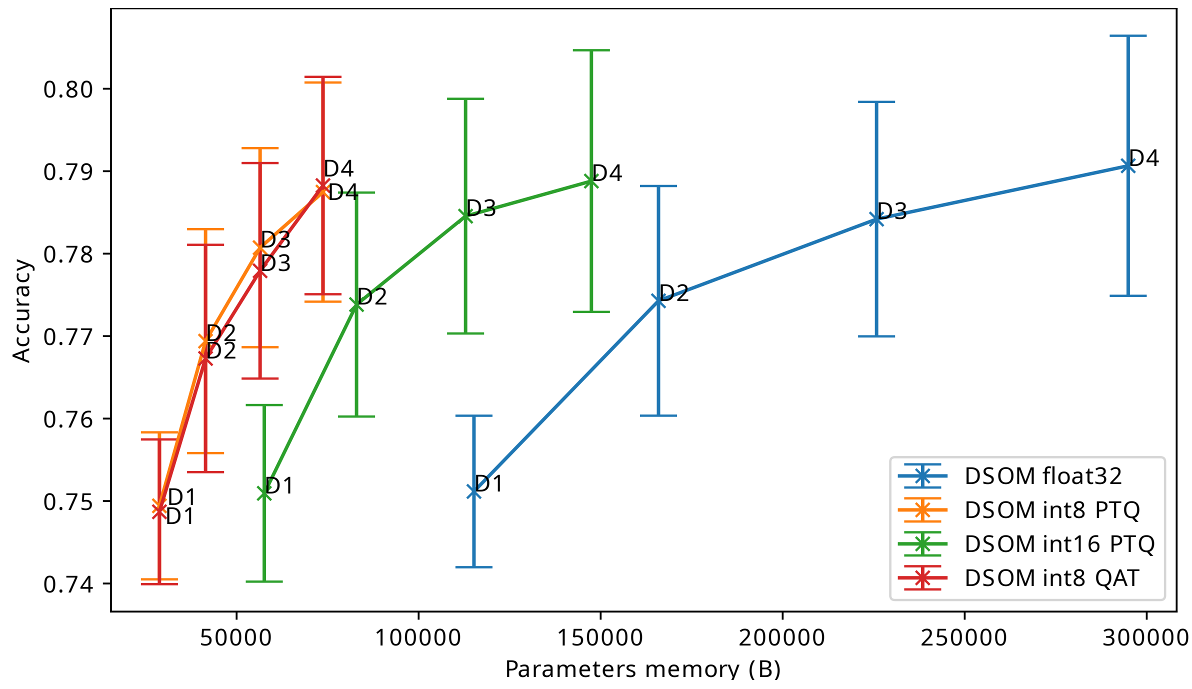


Figure 6.4: Accuracy vs. memory footprint of the parameters for dynamic self-organizing maps on UCI-HAR.

6.4.1.2 UCA-EHAR dataset

The UCA-EHAR dataset is used to evaluate the effect of quantization in a hybrid network, using a residual neural network as a feature extractor and a dynamic self-organizing map as a classifier (ResNet+DSOM). The residual neural network is a ResNetv1-6 with 32 filters per convolution. Batch normalization layers are present after each convolutional layer. However, batch normalization layers are fused to the preceeding convolutional layer after training the feature extractor, before performing an optional quantization-aware training step and before training the dynamic self-organizing map. The dynamic self-organizing map is of size 8 × 8. The extended UCA-EHAR dataset with subjects T2, T3, T5 and T20 as testing set is used here. This configuration was used for the experiments of Section 5.4.2 on unsupervised fine-tuning as well.

The results for the different quantization configurations are presented in Table 6.1. First, the ResNet is quantized with post-training quantization on 16 and 8 bits. While the 16-bit quantization does not degrade the accuracy, the 8-bit quantization decreases the accuracy from 94.16% to 66.88%. This substantial drop can be mitigated using quantization-aware training to obtain an accuracy of 93.96%, only 0.2% away from the baseline.

Similarly, there is no degradation of the accuracy when quantizing the ResNet+DSOM model on 16 bits. However, if 8-bit post-training quantization is applied to both the ResNet and the DSOM parts of the neural network, the accuracy drops to 54.59%. Indeed, the 8-bit post-training quantization already causes a substantial accuracy degradation of the ResNet, so the feature maps are not of great quality. By applying quantization-aware training to the ResNet first, the accuracy only drops to 86.12%. Applying 8-bit quantization-aware training to both the ResNet and the DSOM only increases the accuracy up to 86.80%. This 3.10% difference from the ResNet+DSOM baseline is not negligible, thus justifying 16-bit quantization. As a side note, 8-bit quantization also introduces a higher variability in the accuracy across different trainings as shown by the increase standard deviation.

Table 6.1: Quantization of a ResNetv1-6 with 32 filters per convolution (ResNet), and the same ResNet used as a feature extractor for a 8 × 8 dynamic self-organizing map (ResNet+DSOM) trained on extended UCA-EHAR with subjects T2, T3, T5, and T20 as testing set.

Name	Quantization	Data type	Micro Accuracy (%)	Standard Deviation
ResNet	None	float32	94.16	1.28
ResNet	PTQ	int16	94.18	1.29
ResNet	PTQ	int8	66.88	11.23
ResNet	QAT	int8	93.96	0.65
ResNet+DSOM	None	float32	89.90	1.88
ResNet+DSOM	PTQ	int16	89.94	1.89
ResNet+DSOM	PTQ	int8	54.59	12.53
ResNet+DSOM	QAT (ResNet), PTQ (DSOM)	int8	86.12	3.59
ResNet+DSOM	QAT	int8	86.80	3.00

6.4.2 Embedded Execution of Self-Organizing Maps

The embedded execution of self-organizing maps is evaluated on the SparkFun Edge board. As a reminder (see Table 3.6), the Ambiq Apollo3 microcontroller on this board embeds 1024 KiB of Flash memory (ROM) 384 KiB of RAM. CMSIS-NN optimizations are enabled for the convolutional feature extractor of the hybrid networks. In order to deploy these networks on the SparkFun Edge board, the MicroAI tool presented in Chapter 3 is used to generate the C code both for the feature extractor and the self-organizing map.

6.4.2.1 UCI-HAR dataset

First, the inference time and the ROM footprint are reported in Figure 6.5 for each of the self-organizing maps and dynamic self-organizing maps of Table 5.3 and Table 5.4, respectively. Inference is performed with single-precision floating-point numbers (float32), 16-bit fixed-point numbers (int16), and 8-bit fixed-point numbers (int8). Since the online learning of dynamic self-organizing maps is disabled for this experiment, the inference phase is the same as the self-organizing maps. Therefore, the inference time is also the same between the SOM and the DSOM for a given size of the map. Similarly, the ROM footprint is the same between a self-organizing map and its dynamic variant. This is why S1 and D4 have the same inference time and ROM footprint since both are of size 8 × 8. Although not illustrated here, if the dynamic self-organizing map is used for online learning, the weights would also need to be stored in RAM.

Since the inference phase computes the distance between all neurons and the input, the inference time scales linearly with the number of neurons. The ROM footprint also scales linearly with the number of neurons since they all contain a vector of the same size as the input. However, inference using 16-bit fixed-point numbers is slightly slower than using single-precision floating-point numbers. For 8-bit fixed-point numbers, the inference time is below the implementation using floating-point numbers as expected.

A possible reason for the low performance of the 16-bit fixed-point inference is the number of scaling operations required for the distance computation. In some instances, the scale factor could be different between the input and the weights. In this case, it is necessary to scale one or the other before computing the difference. Furthermore, to avoid overflows, the result of the square is scaled back before accumulating into the distance variable. Both these operations require a shift for each element of the input.

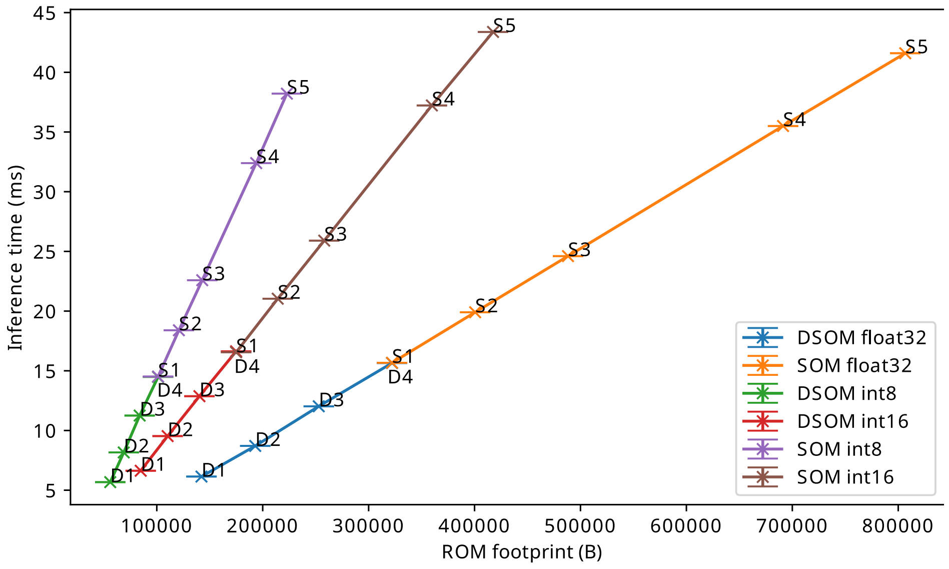


Figure 6.5: Inference time for quantized and non-quantized SOM and DSOM models for UCI-HAR running on the SparkFun Edge board.

In Figure 6.6, the convolutional neural networks (CNN) of Table 5.1 and the hybrid networks (CNN+SOM) using the feature extractor of C5 to C8 and the self-organizing map S1 are compared. Both fully-connected layers of 120 and 6 neurons of the convolutional neural networks C5 to C8 are removed and replaced by a single self-organizing map of size 8 × 8. This explains the lower memory usage of the CNN+SOM models when compared to the CNN models. For the CNN+SOM models, the inference is also slightly faster than the CNN models with floating-point numbers. However, the inference of the CNN+SOM models is slightly slower than the CNN models with fixed-point numbers. This is due to the slower computation of the self-organizing map. Indeed, deep neural networks make use of the optimized CMSIS-NN library while self-organizing maps do not.

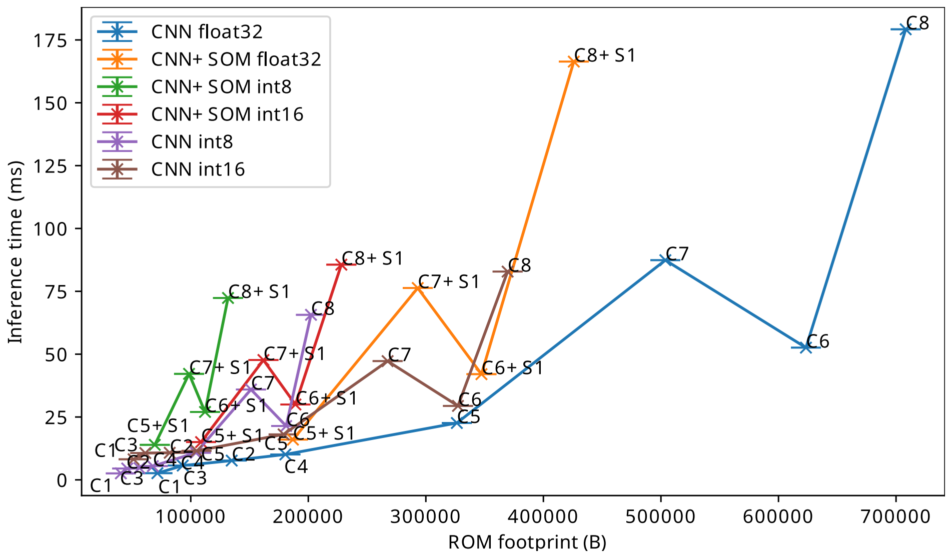


Figure 6.6: Inference time for quantized and non-quantized CNN and CNN+SOM models for UCI-HAR running on the SparkFun Edge board.

Figure 6.7 compares the SOM and the CNN+SOM models. Since the input dimensions of the self-organizing map at the end of the CNN+SOM model are smaller than the raw data processed by a standalone SOM, the memory footprint is also reduced. Furthermore, the convolutional layers use the same set of weights inside a kernel over the entire dimension of the input. However, the SOM models have a shorted processing time than the convolutional layers.

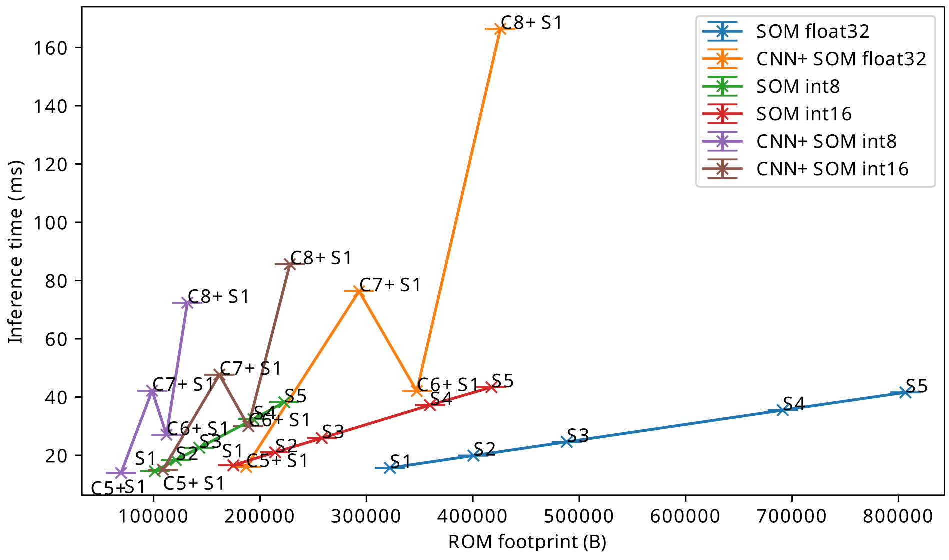


Figure 6.7: Inference time for quantized and non-quantized SOM and CNN+SOM models for UCI-HAR running on the SparkFun Edge board.

6.4.2.2 UCA-EHAR dataset

In this section, the ResNet+DSOM model used for unsupervised fine-tuning on the UCA-EHAR dataset is analyzed when deployed on a microcontroller. Inference time, Flash and RAM memory footprint are provided for the original ResNet, the ResNet+DSOM with on-device learning (ODL) disabled, and the ResNet+DSOM with on-device learning enabled. All these results are largely dominated by the memory and processing of the ResNet feature extractor.

As shown in Figure 6.8, the ResNet+DSOM Flash memory footprint is a few percents higher than the ResNet model. The reason is that the fully-connected classifier of 7 neurons is replaced by a dynamic self-organizing map of 8 × 8 neurons. When on-device learning is enabled, the Flash memory footprint slightly increases as well. Additional memory is required to store the instructions for the learning process. Furthermore, the quantization on 16 or 8 bits reduces the memory footprint compared to the 32-bit floating-point models. However, it does not scale linearly down to 8 bits since this measurement also includes the instructions which are not affected by the quantization of the weights.

As shown in Figure 6.9, the processing of the ResNet+DSOM model does not require more RAM than the ResNet model when on-device learning is disabled. Indeed, the input and output dimensions are the same between the fully-connected layer and the dynamic self-organizing map, so no additional memory is required in the activation buffers. However, and as expected, when on-device learning is enabled, the RAM requirements increase. Indeed, the weights of the dynamic self-organizing map need to be loaded into RAM in order to be modified during the training process. Nonetheless, since the size of the self-organizing map is kept small, only a few percents of additional RAM is required. With 32-bit floating point numbers, the RAM footprint increases from 62 kB for the ResNet model to 71 kB for the ResNet+DSOM ODL model (with on-device learning enabled).

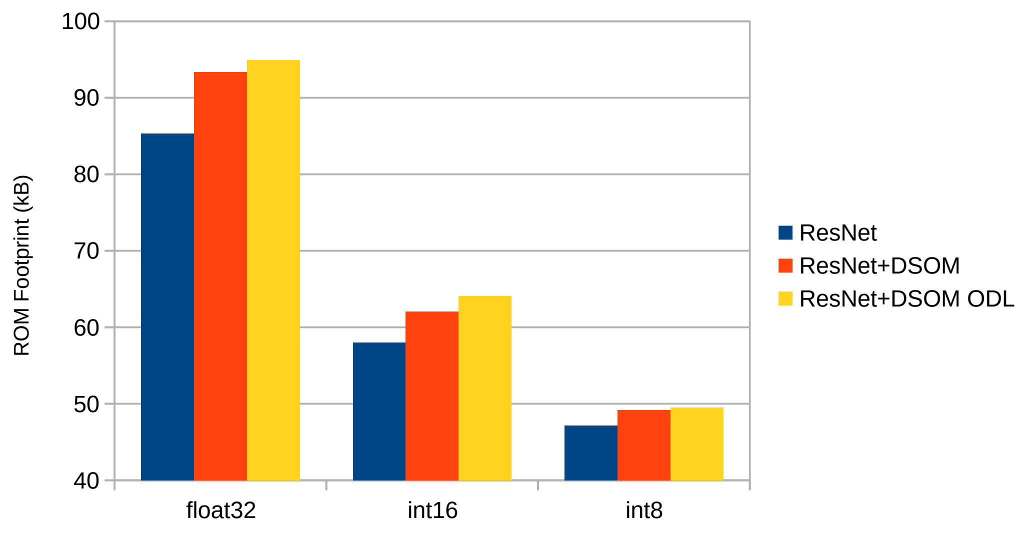


Figure 6.8: Flash memory usage for a ResNet and ResNet+DSOM (with and without on-device learning) models for UCA-EHAR deployed on the SparkFun Edge board with and without quantization.

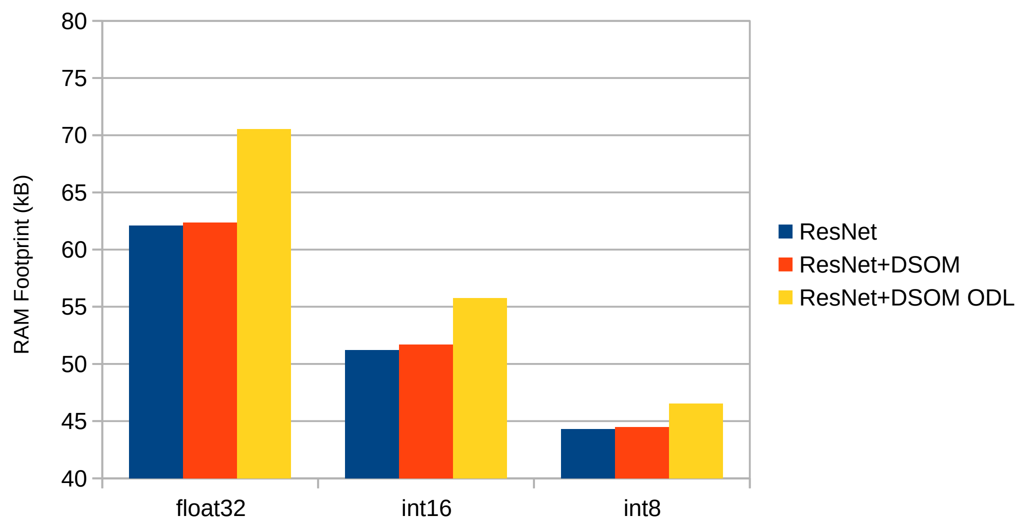


Figure 6.9: RAM usage for a ResNet and ResNet+DSOM (with and without on-device learning) models for UCA-EHAR deployed on the SparkFun Edge board with and without quantization.

As it was the case for the Flash memory, the RAM usage is reduced with 8- or 16-bit quantization. In this case, it is important to note that a large receive buffer for the serial communication (32 kiB) is used to store the input data before the fixed-point conversion. This fixed amount of memory explains why the RAM usage does not scale down well for 8-bit quantization.

Figure 6.11 shows that the ResNet+DSOM model latency overhead with on-device learning disabled is negligible compared to the ResNet model. Enabling on-device learning significantly increases the processing time of the self-organizing map: +4.52ms from ResNet to ResNet+DSOM ODL, compared to +0.53ms from ResNet to ResNet+DSOM. However, the main part of the processing time is spent in the feature extractor, so the overhead of on-device learning remains low (from 118.31 ms to 122.30 ms for the ResNet and ResNet+DSOM ODL models respectively). Using fixed-point numbers with CMSIS-NN optimizations for the feature extractor improves tremendously the inference time down to 41 ms for the 16-bit ResNet+DSOM ODL model. As mentioned previously, the dynamic self-organizing map does not make use of the optimizations provided by the DSP instructions of the Cortex-M4 core. However, since the input dimension of the self-organizing map and its number of neurons are small, the amount of processing required is also small compared to the processing needed for the feature extractor.

As a reminder, for our application of live human activity recognition, an inference is performed with windows of approximately 2.46 s. Therefore, real-time constraints are met since the inference finishes in less than 2.46 s, even when on-device learning is enabled. Furthermore, the difference in processing time between a ResNet and a ResNet+DSOM with on-device learning using 16-bit fixed point numbers is 2.28m s, less than 0.1% of the inference period. As a result, the increase in energy consumption is very small.

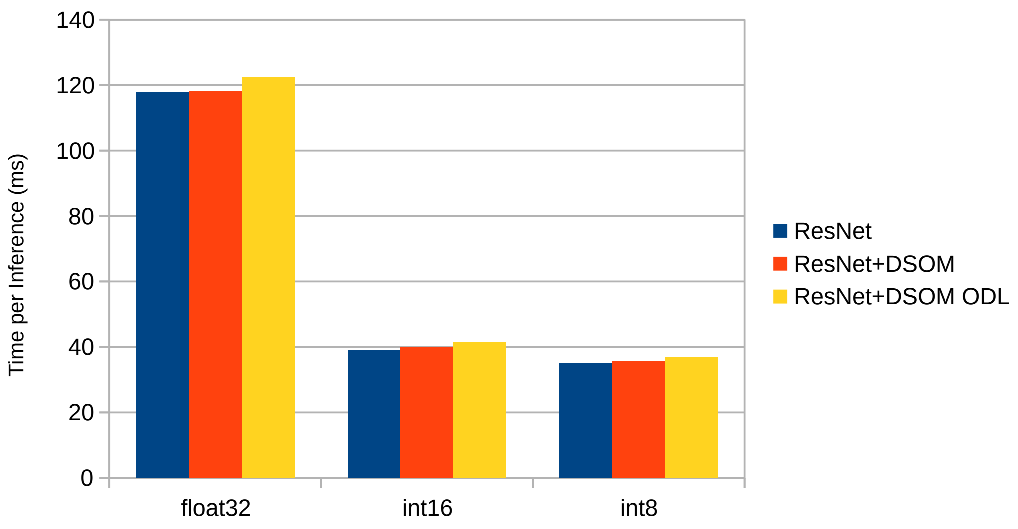


Figure 6.10

Figure 6.11: Inference time a ResNet and ResNet+DSOM (with and without on-device learning) models for UCA-EHAR deployed on the SparkFun Edge board with and without quantization.

Overall, the embedded footprint of the ResNet+DSOM model remains very close to the original ResNet model, even with on-device learning enabled for the DSOM layer. Detailed results are provided in Table 6.2.

Table 6.2: Embedded execution of a ResNetv1-6 with 32 filters per convolution (ResNet), and the same ResNet used as a feature extractor for a 8 × 8 dynamic self-organizing map (ResNet+DSOM) trained on extended UCA-EHAR with subjects T2, T3, T5, and T20 as testing set.

Name	**Data type**	**On-Device Learning**	**Time per Inference (ms)**	**ROM Footprint (kB)**	**RAM Footprint (kB)**
			(2462 ms max)	(Ambiq Apollo3: 1048 kB max)	(Ambiq Apollo3: 393 kB max)
ResNet	float32	N/A	117.78	85.292	62.088
ResNet	int16	N/A	39.13	58.012	51.204
ResNet	int8	N/A	35.03	47.148	44.284
ResNet+DSOM	float32	Disabled	118.31	93.348	62.352
ResNet+DSOM	int16	Disabled	39.85	62.068	51.668
ResNet+DSOM	int8	Disabled	35.57	49.164	44.484
ResNet+DSOM	float32	Enabled	122.30	94.948	70.544
ResNet+DSOM	int16	Enabled	41.41	64.068	55.764
ResNet+DSOM	int8	Enabled	36.83	49.484	46.532

6.5 Conclusion

In the previous chapter (Chapter 5), we proposed an unsupervised fine-tuning method for human activity recognition. In this chapter, we implemented this method on a microcontroller to enable on-device learning. In order to reduce the memory footprint and the inference time, self-organizing maps have also been quantized. Results showed that the 16-bit quantization did not degrade the accuracy of the self-organizing maps, both for a standalone self-organizing map and for a self-organizing map learning from feature maps. 8-bit quantization applied to a standalone self-organizing map only slightly degraded the accuracy, although the accuracy was already low before quantization. However, when 8-bit quantization is appleid to a ResNet+DSOM model, the accuracy dropped by several percents, thus justifying the use of 16-bit quantization.

Then, the memory footprint and the latency for the inference of self-organizing map models were measured on a microcontroller. Measurements performed with a standalone self-organizing map on UCI-HAR revealed that the fixed-point implementation did not provide a significant improvement in terms of inference time compared to a floating-point implementation. The additional scaling operations and the lack of specific optimizations using the Cortex-M4 DSP instructions can explain this limited improvement. Nonetheless, results showed that the impact of a ResNet+DSOM model was small compared to a ResNet model for the UCA-EHAR dataset. This is due to the small dimension of the DSOM layer input (since this ResNet feature extractor produces 32 feature maps of dimension 1) as well as the small number of neurons of the DSOM layer.

Finally, the dynamic self-organizing map learning rule was implemented on-device using both floating-point and fixed-point numbers. For fixed-point numbers, this required implementing a method to compute an exponential function in fixed point, since a lookup table would be too heavy for 16-bit numbers. Enabling on-device learning with the ResNet+DSOM model increased the processing time and the RAM footprint of the self-organizing map. However, the overall impact remained low since most of the processing and memory footprint is due to the feature extractor.

In our approach, only the last layer of a deep neural network is fine-tuned. Fine-tuning an entire network would put a much higher strain on the embedded constraints. As no fine-tuning of the feature extractor can be performed, the quality of the features generated by the frozen feature extractor is highly important.

Chapter 7

Conclusion and Perspectives

Contents

7.1 Conclusion

In this thesis, we studied the integration of artificial intelligence onto low-power embedded devices and its impact on embedded constraints. As a real-world use-case, we focused on human activity recognition using smart glasses. The purpose is to monitor the behaviour of a subject in the context of elderly care. While we have not specifically evaluated the efficiency of our system in terms of preventing fall or detecting a degradation of mobility, which is the focus of other ongoing related works[108, 109], we showed that human activity recognition can be successfully performed for several classes of activities of daily living directly on the smart glasses. Performing the computations on a microcontroller was a requirement for the system to be autonomous and to respect privacy. Indeed, data do not need to be transmitted to another device or possibly to a remote server. This also provides a more deterministic latency to the recognition process.

The field of artificial intelligence on microcontrollers only emerged in the late 2010s as the literature presented in Chapter 2 can attest. Before that, it was believed that the computation and memory requirements would be far beyond the capabilities of such embedded devices. The past few years have shown that with modern design and training of artificial neural networks, it is possible to tackle certain prediction tasks on microcontrollers using deep neural networks. That said, not many applications actually make use of embedded deep neural networks yet. Furthermore, the lack of tooling and comparative studies makes it difficult to evaluate the possibilities of such a feat.

Therefore, in Chapter 3, we presented our open-source end-to-end software framework for training, quantization and deployment of deep neural networks on microcontrollers. This software framework enables the deployment of various kinds of common feedforward neural networks for evaluation on a microcontroller, using a C code generation tool to create the embedded inference library. Additionally, quantization was performed to reduce both the memory footprint and the inference time. Quantization indeed enables computation using fixed-point point numbers which are less expensive than floating-point numbers. The impact of quantization was evaluated on three different datasets for potential embedded artificial intelligence applications: keyword spotting, traffic-sign recognition and human activity recognition. We provided accuracy results for different size of models in order to demonstrate the memory efficiency of each. 8- and 16-bit fixed-point quantizations were compared to the single-precision floating-point baseline. 16-bit quantization divided by two the memory compared to the non-quantized models without compromising on the accuracy. The memory footprint and the inference time were compared to existing inference engines. Our tool achieved similar or better figures compared to STM32Cube.AI and TensorFlow Lite Micro. Finally, the energy consumption of two different microcontrollers, STM32L452RE-P and Ambiq Apollo3, were compared using the three different inference engines. The results showed that using our inference engine in combination with the Ambiq Apollo3 microcontroller, for which STM32Cube.AI is not available, provides the best energy efficiency.

The main takeaway of this result is the importance of hardware-software codesign. This is further confirmed by the work of the authors in [28], showing that a custom design of the hardware with software correctly using it provides substantial improvements in terms of memory efficiency. In the future, sizing the hardware and the software that runs on it hand-in-hand may become a key aspect of energy efficiency for embedded artificial intelligence. Therefore, interdisciplinary curriculum will need to be developed to provide engineers with knowledge and skills to look at both aspects of the problem.

With this powerful tool in hands, we tackled the specific case of human activity recognition on smart glasses in Chapter 4. While many datasets for human activity recognition are available, smart glasses are very rarely used for this kind of application and the few attempts did not provide data usable for our use case. For these reasons, we created our own dataset from different subjects performing various activities of daily living while wearing the smart glasses.

This datased, called UCA-EHAR, has been published under an open-access policy. Using this dataset, we trained deep neural networks to provide an accuracy baseline with an analysis of the confusion matrix. The quantization methods were applied to the deep neural network models to optimize the memory footprint and find the best model that could fit on the smart glasses. To perform live human activity recognition, we then deployed the model onto the smartglasses microcontroller by integrating it into the existing firmware. Energy was analyzed to demonstrate that the impact of the deep neural network inference was low. The latency was also measured to make sure real-time constraints are met for this use-case.

In Chapter 5, we studied the feasibility of unsupervised online learning to fine-tune our model for a specific subject. As the smart glasses will typically be worn by one subject throughout their lifetime, we want to optimize the recognition for this person. Additionally, for the same reason we perform the inference directly on the smart glasses, we want to execute the fine-tuning on the device as well. To support unsupervised learning, we replaced the supervised classification layer of a deep neural network with a self-organizing map. The upstream layers are frozen and act as a feature extractor for the self-organizing map. While the training of the self-organizing map is unsupervised, the training of the feature extractor remains supervised. If a different dataset with labelled input data using the same encoding is available, transfer learning can be leveraged so that the target dataset labels are not required. However, for a classification problem, the neurons of the self-organizing map still need to be labelled. The labelling method can rely on a reduced amount of labels, making the overall process semi-supervised. Self-organizing maps do not allow for online learning since the convergence process relies on a finite time constant. Therefore, dynamic self-organizing maps are used instead when performing online learning.

Finally, both the inference and the training of the dynamic self-organizing map were implemented on a microcontroller in Chapter 6. Our software framework was adapted to support the quantization and deployment of this hybrid network. As it was done for the inference of the deep neural network, memory footprint and energy consumption were analyzed for this on-device learning method, showing that the overhead can be kept under control. While the overall improvements of this unsupervised online fine-tuning process to classification performance are mixed, on-device unsupervised fine-tuning on microcontrollers has been shown to be viable.

7.2 Perspectives

During this thesis, we highlighted several paths to explore to further improve the embedded execution of neural networks as well as refine unsupervised online fine-tuning. Furthermore, the method and tools presented in this thesis can be applied to other use cases.

Some improvements and experimentations can be done in the short term with specific additions to our work, while others require medium-term projects. We also propose new paths and opportunities as longer-term perspectives.

7.2.1 Short-Term Tasks

Use barometer signals for live human activity recognition.

As explained in Section 4.4.1, our UCA-EHAR dataset includes data from the barometer, but the barometer is not used to perform live activity recognition. To use the barometer, an interpolation function has to be used to resample the barometer data to the accelerometer data so that each new sample contains data from both. Linear interpolation or even duplicating the sample could be enough but this has yet to be evaluated. Furthermore, the barometer data contains high-frequency noise and a low-frequency component that may need to be filtered out.

Deploy the unsupervised on-device fine-tuning method on the smart glasses.

The unsupervised fine-tuning has not yet been implemented on smart glasses either, even though the MicroAI software is able to generate the approriate C code. The smart glasses firmware needs to be updated and then the impact of unsupervised online fine-tuning can be evaluated in a live human activity recognition scenario.

Provide a better diversity of subjects (e.g., elderly) in the UCA-EHAR dataset.
To further analyze the possible benefits of this fine-tuning, the dataset should be extended to include more diverse subjects, outside the age range considered originally. Especially, data should be collected from the elderly and from people with disabilities using the existing data collection protocol and tools. Both the generalization capabilities of the deep neural network and the improvements brought by fine-tuning could therefore be tested with data further away from the training distribution than the testing sets used until now.

Evaluate re-labelling with few labels.
When performing unsupervised fine-tuning, and as the results in Section 5.4.2.3 have shown, a relabelling step using the original dataset can improve the accuracy. In those experiments, the entire dataset was still used for relabelling. However, we have shown that only a few labels were necessary for the first labelling step, before fine-tuning. Thus, the relabelling step after fine-tuning may not require many labels either. If this step is also implemented in a low-power embedded device, using only a small amount of labels would significantly reduce the memory usage and the processing time.

Deploy a keyword spotting application with MicroAI.
Outside of human activity recognition, other applications could be augmented with embedded deep neural networks. For example, leveraging the MicroAI software framework as well as the deep neural networks designed for use with Spoken MNIST or Heidelberg Digits, keyword spotting could be implemented on smart glasses or other wearables equipped with a microphone. A first prototype for such a use-case has been developed on a custom board with a low-power microcontroller.

7.2.2 Medium-Term Projects

Experiment with semi-supervised transfer learning and unsupervised fine-tuning for keyword spotting datasets.
Keyword spotting can also be a great use-case for semi-supervised transfer learning and unsupervised fine-tuning. A large dataset such as Google Speech Commands[192] can be used to train a supervised feature extractor. Then, a target dataset such as Heidelberg Digits would be used in a transfer learning fashion. This feature extractor would feed a dynamic self-organizing map for unsupervised learning. Finally, only a reduced number of labels from the target dataset would be needed to apply labels to the dynamic self-organizing map. The overall method would be semi-supervised, and could also perform unsupervised online fine-tuning to adapt to a speaker with a different voice.

Perform other tasks with MicroAI. Apart from classification tasks, MicroAI can also be used to deploy neural networks to solve other tasks. For example, works are ongoing to study the energy consumption reduction in wireless sensor networks using MicroAI to deploy a convolutional neural network for sensor data prediction[207].

Implement and evaluate advanced quantization techniques in MicroAI.
During our study of deep neural networks deployment on microcontrollers, we took a look at compressing the neural network to reduce the memory footprint. However, while we implemented a simple quantization method, we did not investigate the more advanced techniques available in the litterature and presented in Section 2.4.1.3. Furthermore, several other compression techniques exist such as pruning and knowledge distillation, shortly introduced in 2.4.1.2. Therefore,

implementing and evaluating these methods could be a major factor to further optimize deep neural networks deployment on microcontrollers.

Implemented automated machine learning techniques in MicroAI.

Using our software framework, the process of training, quantization and deployment is mostly automated. However, selecting and configuring the deep neural networks to evaluate still needs to be done by hand. Implementing automated machine learning techniques such as neural architecture search could be useful, especially if constraints such as memory and latency are taken into account.

Integrate human activity recognition profiles into an energy analysis tool for the smart glasses.

In Chapter 4 we analyzed the energy consumption during inference on the smart glasses. While the human activity recognition was running on top of the original firmware, we did not take into account other possible applications running concurrently as well as the aging of the battery. The impact of the various hardware and software components on the autonomy of the device still needs to be investigated. In [208], a software tool was proposed in order to perform such an analysis and provide an estimation of the autonomy during different scenarios. The energy consumption of the human activity recognition application could be integrated into this energy analysis tool to provide insights on how to optimize the energy consumption.

Add support for RISC-V platforms to MicroAI.

Another way to reduce the energy consumption of neural networks is to use hardware accelerators. As the authors of [28] have shown, hardware acceleration can be implemented in the form of custom instructions in the processor core. Moving to a more open ecosystem such as RISC-V-based cores enables a more in-depth control of the hardware. We would be able to perform hardware-software codesign to further optimize the execution of deep neural networks.

Extend MicroAI to support spiking neural networks and neuromorphic hardware.

Other computing paradigms such as neuromorphic computing exhibited in spiking neural networks can also provide significant energy savings leveraging mechanisms inspired by the brain. To do so, specific hardware has to be designed and produced. Software must also be adapted to work with both the theoretical models of spiking neurons and the hardware implementation. Software frameworks such as SpikingJelly[209], a deep learning framework for spiking neural network based on PyTorch, provide the necessary tools to design and train spiking neural networks, with good results in simulation[210]. However, existing experiments rarely go all the way to execution on real hardware, both due to a lack of hardware implementation and a lack of automated deployment flows. Some hardware implementation such as SPLEAT[211] support the execution of spiking convolutional neural networks, but the model still has to be ported manually. Our MicroAI framework could therefore be extended to fill this gap: provide a tool that takes a spiking neural network trained with SpikingJelly and generate the appropriate configuration for the SPLEAT architecture, then perform the deployment on the target and evaluate the execution as we did on microcontrollers. Since the hardware architecture performs the computation with fixed-point numbers, quantization would also have to be applied. Thus, the effect of quantization in the case of spiking neural networks needs to be evaluated.

7.2.3 Long-Term Propositions

Evaluate the benefits provided by binary neural networks on microcontrollers.

Quantization can be pushed even further to binarize deep neural networks. Training binary neural networks to obtain a good accuracy is still challenging. However, as presented in Section 2.4.1.3, recent advances have shown that by carefully crafting the neural network architecture, minimal accuracy loss can be obtained after binarizing most of the network. Still, binary neural networks are rarely deployed on embedded devices for real-world applications. A framework to generalize the binarization process needs to be designed, and binarized neural networks have to be evaluated on several different use cases to prove their relevance. The training process also needs to be optimized to be less time-consuming. Binary neural networks could provide a lower memory footprint as well as faster execution on microcontrollers. However, the accumulation operation implemented as a bitwise operation requires counting the number of bits set to 1 in a register. Without a specific instruction, often called *popcount*, this can be a costly process. In most of the available microcontrollers, this instruction is not implemented. The RISC-V Bit Manipulation extensions (including the *cpop* instruction for this purpose) has been ratified in November 2021, but silicon implementing it will take some time to be manufactured.

Design an accelerator for dynamic self-organizing maps.
Apart from deep neural networks hardware accelerators, the unsupervised online learning method presented in Chapter 5 could also benefit from a hardware accelerator to scale to more complex problems. In the SOMA (Self-Organizing Machine Architecture) project[212], an FPGA-based hardware platform for execution of self-organizing models has been developed. Even though a self-organizing map hardware implementation is not yet available, a preliminary design of a neural processing unit for execution of self-organizing map models has been created. The neural processing unit is intended for use in a many-core processor to perform parallel processing of the neurons. Several tools have been developed to simulate the models and the hardware [213, 214]. As a simpler alternative, a custom processing unit could accelerate some of the computation. For example, a custom instruction for the fixed-point exponential function computation for on-device learning would significantly reduce the processing time.

Use out-of-distribution detection to detect subjects and activities far from the training dataset.
For unsupervised online learning, the results showed that the outcome depends on the subject and the class (i.e., the activity for a human activity recognition task). Out-of-distribution detection could be used to find out which class may not be properly recognized for a subject.

Implement reinforcement learning and multi-modal learning to improve fine-tuning.
Furthermore, even though it requires an input from the user or an operator, reinforcement learning could be implemented sparingly to guide the online learning algorithm towards a better prediction. Embedded keyword spotting could be leveraged for the user to provide the label without having to interact with a mechanical input device. Spoken labels are indeed easier to provide while in motion than pressing a button.

Fight against catastrophic forgetting.
Finally, dynamic self-organizing maps do not provide any mitigation against catastrophic forgetting since they are designed to follow a drifting input distribution. However, the impact of catastrophic forgetting depends on the task and the training conditions. In our case, losing part of the initial information that helped the network generalize on multiple subjects is not a problem. However, mechanisms should be implemented to avoid forgetting classes if they do not occur regularly. In other cases, catastrophic forgetting needs to be taken into account in the design of the online learning method. Several promising methods exist as presented in Section 2.6.2. However, the results still depend on the dataset and the use case, therefore the problem remains open.

Appendix A

MicroAI Usage and Implementation Details

Contents

A.1 Example TOML Configuration File

Listing A.1: uci-har_cnn.conf

```
[bench]
name = "UCI-HAR_CNN"
first_run = 1
last_run = 15 # Perform 15 different trainings

[learningframework]
kind = 'PyTorch'

[deploy]
target = 'SparkFunEdge' # Target system for deployment and evaluation
converter = 'KerasCNN2C' # MicroAI code generation tool
quantize = ['int16'] # Fixed-point conversion using 16-bit integers

[dataset]
kind = "UCI_HAR"
params.variant = "raw" # Use raw data rather than pre-computed features
params.path = "data/UCI HAR Dataset/"

[[preprocessing]]
kind = "DatamodelConverter" # Convert subject/activities (HARDataModel) to input data and label matrices (RawDataModel)

[[preprocessing]]
kind = "Class2BinMatrix" # Convert class number to one-hot targets

[[postprocessing]]
kind = "FuseBatchNorm" # Fuse batchnorm layers with preceeding conv layers after training

[[postprocessing]]
kind = "QuantizationAwareTraining" # Quantize model after training
params.width = 16 # 16-bit fixed-point quantization
params.epochs = 0 # Post-Training Quantization: no quantized training performed
params.model.params.batch_norm = false # Disable batchnorm: fused previously and not supported for quantization
export = true # Save the resulting model

[model_template]
kind = "CNN"
epochs = 120
batch_size = 32
params.batch_norm = true # Enable batchnorm layers after each conv layer

[model_template.optimizer]
kind = "Adam"

[[model]]
name = "uci-har_cnn_48_5-32_3_120" # CNN model C7
params.filters = [48, 32] # Number of filters for each conv layer
params.kernel_sizes = [5, 3] # Kernel size for each conv layer
params.fc_units = [120] # Add a fully-connected layer before the final layer
params.pool_sizes = [4, 0] # Pooling size after each conv layer

[[model]]
name = "uci-har_cnn_64_7-48_5_120" # CNN model C8
params.filters = [64, 48]
params.kernel_sizes = [7, 5]
params.fc_units = [120]
params.pool_sizes = [4, 0]
```

A.2 Explanations About Configuration File

The training process expects a `RawDataModel` instance, which gathers the training and test sets. This instance contains *numpy* arrays for the data and the labels. A higher-level data model `HARDataModel` is also available for human activity recognition in order to process subjects and activities more easily. This model is then converted to a `RawDataModel` using the `DatamodelConverter` in the pre-processing phase. The pre-processing phase can also include transformations such as normalization.

In the configuration file, several model settings can be described, each inside their own `[[model]]` block. Each model will be trained sequentially. A common configuration for all the models can be specified in a `[model_template]` block. Model configuration also includes optimizer configuration and other parameters such as the batch size and the number of epochs.

Once the model is trained, some post-processing can be applied. It is for instance possible to remove the SoftMax layer for Keras models with the `RemoveKerasSoftmax` module. This layer is indeed useless for classification when only the inference is performed. In the case of PyTorch models, the batch normalization layers can be fused with the previous convolutional layer using the `FuseBatchNorm` module. PyTorch's FX module[187] is used to automatically detect the Conv/BatchNorm layer pairs and replace them with a single convolutional layer with merged parameters.

The quantization-aware training described in Section 3.2.2.5 is also included for PyTorch as a post-processing step in the `QuantizationAwareTraining` module, even though it also performs model training. The actual training step before post-processing is seen as a general training, before optionally performing post-training quantization or quantization-aware training. The quantization-aware training can be seen as a fine-tuning on top of the more general training (which can also be skipped if necessary). This conversion is instead performed by the C code generation tool.

A.3 MicroAI Commands to Run for Automatic Training and Deployment of Deep Neural Networks

Data can be preprocessed (e.g., to apply normalization) from the source dataset and serialized to an intermediate dataset file with the following command:

```
microai <config.toml> preprocess_data
```

The training phase is started by running the following command:

```
microai <config.toml> train
```

Before being deployed and evaluated, the appropriate code must be generated and built for the targeted platform by running the following command:

```
microai <config.toml> prepare_deploy
```

Once the binaries are generated, they can be deployed, and the model can be evaluated on the target by running the following command:

```
microai <config.toml> deploy_and_evaluate
```

A.4 Example Commands for UCI-HAR on Nucleo-L452RE-P with 16-bit Quantization

The UCI-HAR dataset is first downloaded and extracted in the `data/` directory.

Run the various pre-processing steps in the configuration and generate serialized numpy arrays in `out/data/UCI_HAR_raw/`:

```
microai conf/uci-har/ResNetv1_float32_train.toml preprocess_data
```

Train 7 different networks 15 times in a row according to the configuration:

```
microai conf/uci-har/ResNetv1_float32_train.toml train
```

Perform Post-Training Quantization for int16, generate the source code for the C inference library in `out/kerascnn2c_fixed/`, and build the firmware for Nucleo-L452RE-P in `out/deploy/NucleoL452REP/`:

```
microai conf/uci-har/ResNetv1_KerasCNN2C_NucleoL452REP_int16.toml prepare_deploy
```

Deploy the firmware onto the Nucleo-L452RE-P board connected to the computer and send vectors from the test dataset for evaluation:

```
microai conf/uci-har/ResNetv1_KerasCNN2C_NucleoL452REP_int16.toml deploy_and_evaluate
```

A.5 Fully-Connected Jinja2 C Template File for MicroAI

Listing A.2: fc.cc

```
#define INPUT_SAMPLES {{ node.input_shape[-1] }} // Dimension of the input
#define FC_UNITS {{ node.layer.units }} // Number of neurons
#define ACTIVATION_{{ node.layer.activation.name | upper }}

// For fixed point quantization
#define WEIGHTS_SCALE_FACTOR {{ node.weights_scale_factor }}
#define INPUT_SCALE_FACTOR {{ node.innodes[0].output_scale_factor }} // Output scale factor of previous layer
#define OUTPUT_SCALE_FACTOR {{ node.output_scale_factor }}

typedef number_t {{ node.layer.name }}_output_type[FC_UNITS];

static inline void {{ node.layer.name }}(
  const number_t input[INPUT_SAMPLES],
        const number_t kernel[FC_UNITS][INPUT_SAMPLES],
        const number_t bias[FC_UNITS],
        number_t output[FC_UNITS]) {

  unsigned short k, z;
  long_number_t output_acc;

  for (k = 0; k < FC_UNITS; k++) {
    output_acc = scale_number_t((long_number_t)bias[k], -INPUT_SCALE_FACTOR);
    for (z = 0; z < INPUT_SAMPLES; z++)
      output_acc = output_acc + ((long_number_t)kernel[k][z] * (long_number_t)input[z]);

    // Activation function
#ifdef ACTIVATION_LINEAR // Linear (i.e. none)
    output_acc = scale_number_t(output_acc, INPUT_SCALE_FACTOR + WEIGHTS_SCALE_FACTOR - OUTPUT_SCALE_FACTOR);
    output[k] = clamp_to_number_t(output_acc);
#elif defined(ACTIVATION_RELU) // ReLU
    if (output_acc < 0) {
      output[k] = 0;
    } else {
      output_acc = scale_number_t(output_acc, INPUT_SCALE_FACTOR + WEIGHTS_SCALE_FACTOR - OUTPUT_SCALE_FACTOR);
      output[k] = clamp_to_number_t(output_acc);
    }
#endif
  }
}
```

A.6 MicroAI C Library Interface

The generated library exposes a function in the `model.h` header to run the inference process with the following signature:

```
void cnn(
  const input_t input,
  number_t output[MODEL_OUTPUT_SAMPLES]);
```

where `input_t` is a type defined in the generated `model.h` corresponding to a multi-dimensional array of dimensions ($samples$, $channels$) for 1D input or ($height$, $width$, $channels$) for 2D input with a value type of `number_t`. `number_t` is the data type used during inference defined in the `number.h` header, and `MODEL_OUTPUT_SAMPLES` is the dimension of the output defined in the generated `model.h` header. The input and output arrays must be allocated by the caller.

The model inference function does not proceed to the conversion of the input from floating-point to fixed-point representation when using a fixed-point inference code. The caller must perform the conversion before feeding the buffer to the model inference function (see Section 3.3.6).

To this aim, a floating-point number `x_float` can be converted to a fixed-point number `x_fixed` with the following call:

```
number_t x_fixed = clamp_to_number_t(
  (long_number_t)floor(x_float * (1 << MODEL_INPUT_SCALE_FACTOR)));
```

where `long_number_t` is a type twice the size of `number_t` and `clamp_to_number_t` saturates and converts to `number_t`. Both are defined in the `number.h` header.
`MODEL_INPUT_SCALE_FACTOR` is the scale factor for the first layer, defined in the `model.h` header.

The output array corresponds to the output of the model's last layer, which is typically a fully connected layer when solving a classification problem. If the purpose is to predict a single class, the caller must find the index of the largest element in the output array.

A.7 CMSIS-NN Functions Used in MicroAI

The following instructions from the ARMv7E-M instruction set are used to optimize some aspects of deep neural network inference in CMSIS-NN:

- `SMLAD`: performs two 16-bit multiply-accumulate operations in a 32-bit register,
- `SXTB16`: extracts two 8-bit values and extends them to signed 16 bits,
- `QSUB16`: performs two 16-bit subtractions and saturates to signed 16 bits,
- `QSUB8`: performs four 8-bit subtractions and saturates to signed 8 bits,
- `SSAT`: signed saturation to any bit position (also present in the regular ARMv7-M instruction set).

Our framework can optionally make use of the following functions from the CMSIS-NN library with 8-bit quantization.

- `arm_convolve_HWC_q7_basic_nonsquare(…)`: a generic convolution operation making use of the `SMLAD` instruction for the multiply-accumulate operations. The inner loop is partially unrolled to process up to 16 multiply-accumulate in one iteration. Two or four operands are loaded into registers at a time and expanded from 8 bits to 16 bits two at a time with the `SXTB16` instruction. The intermediate 32-bit results are saturated to 8 bits with the `SSAT` instruction before being converted back to an 8-bit data type.
- `arm_convolve_HWC_q7_fast_nonsquare(…)`: compared to the `basic` variant, this function requires the input channels to be a multiple of 4 and output channels to be a multiple of 2 to further optimize memory accesses.
- `arm_fully_connected_q7(…)`: a fully-connected layer operation that makes use of the `SMLAD` instruction for the multiply-accumulate operations. The inner loop is partially unrolled to process eight elements at a time: four input samples and two neurons. Two or four operands are loaded into registers at a time and expanded from 8 bits to 16 bits two at a time with the `SXTB16` instruction. The intermediate 32-bit results are saturated to 8 bits with the `SSAT` instruction before being converted back to an 8-bit data type.
- `arm_relu_q7(…)`: a rectified linear unit activation function that does not use conditional branches or conditional operations. Instead, a mask is generated by extracting the sign bit from the input and subtracting it from 0 with the `QSUB8` instruction, four elements at a time.

With 16-bit quantization, the `q15` variants are used instead, with the following differences.

- `arm_convolve_HWC_q15_basic_nonsquare(…)`: the inner loop is partially unrolled to process up to 4 multiply-accumulate operations in one iteration. Operands are always loaded two at a time but they do not need to be extended to 16 bits, results are saturated to 16 bits.
- `arm_convolve_HWC_q15_fast_nonsquare(…)`: compared to the `basic` variant, this function requires the input and output channels to be multiples of 2, so that the loop can be further unrolled to perform 8 multiply-accumulate in a single iteration.

- `arm_fully_connected_q15(…)`: operands are always loaded two at time but they do not need to be extended to 16-bit, results are saturated to 16 bits.
- `arm_relu_q15(…)`: the QSUB16 instruction is used instead, processing two elements at a time.

The `nonsquare` variants of the functions allow using width and height dimensions that can be different, both for the input and the kernel. This is required in order to implement one-dimensional convolutions. Further improvements for 2D networks could be achieved by using the more optimized variant for square dimensions when possible, but this has not been implemented in our framework. The `basic` variants (as oppposed to the `fast` variants) allows for odd dimensions of the channels while the `fast` variants have requirements over the input and output channels. Therefore, a single network may use a combination of the `basic` variant for layers with odd channels and `fast` variants when the conditions are fulfilled.

The `arm_convolve_HWC_q15_basic_nonsquare(...)` is not included in the original CMSIS-NN library. We implemented it taking inspiration from the existing functions.

CMSIS-NN also offers s8 and s16 variants (instead of q7 and q15) tailored for TensorFlow Lite quantization. These functions indeed use a per-channel quantization with a non-power-of-two scale factor and an asymmetric range for activations. However, we do not use these variants since MicroAI performs a per-layer quantization with a power-of-two scale factor and a symmetric range for activations.

Furthermore, CMSIS-NN offers other functions for deep neural networks such as pooling functions and element-wise addition (used in residual layers). These functions do not seem to offer a significant execution time improvement on Cortex-M4 and are therefore not used in our framework. As a side note, while this is not directly a feature provided by CMSIS-NN, when CMSIS-NN is enabled, the SSAT instruction is also used to saturate the results of the Add and MaxPooling layers.

CMSIS-NN does not offer optimizations for floating-point computation since they are necessarily performed by the FPU in single precision, i.e. 32-bit operations on 32-bit registers.

Appendix B

Detailed Results of Frameworks and Embedded Platforms Evaluation

The detailed results provided in Table B.1 highlight a higher overhead for very small neural networks especially for TensorFlow Lite for Microcontrollers compared to our framework. For example, for a non-quantized network (float32) with 16 filters per convolution built for the SparkFun Edge board, our framework requires 54.316 kiB of ROM to store both the inference code and the weights, while TensorFlow Lite for Microcontrollers requires more than twice as much at 116.520 kiB. Concerning STM32Cube.AI, the memory overhead is only slightly higher than our framework at 61.965 kiB. The trend is similar for quantized networks, but the memory footprint increases more slowly since the initial overhead of the inference code is more important than the memory used to store the weights. When the number of filters per convolution increases, most of the ROM is used by the model's weights. However, the inference code overhead of the framework can still make a significant difference in the results. The memory footprint for a non-quantized network with 80 filters per convolution built for the SparkFun Edge board is indeed 18% higher using TFLite Micro than our framework.

Table B.1: ROM footprint vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			ROM Footprint (kiB)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	116.520	133.988	157.957	188.426	225.395	318.926	438.363
MicroAI	SparkFunEdge	float32	54.316	67.066	91.035	121.512	158.473	251.863	371.332
MicroAI	Nucleo-L452RE-P	float32	55.770	68.145	92.129	122.582	159.559	253.004	372.434
STM32Cube.AI	Nucleo-L452RE-P	float32	61.965	79.449	103.410	133.898	170.859	264.289	383.742
MicroAI (no CMSIS-NN)	SparkFunEdge	int16	46.952	50.629	62.629	77.832	96.355	142.973	202.699
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int16	48.129	51.629	63.613	78.855	97.340	144.051	203.770
MicroAI (CMSIS-NN)	SparkFunEdge	int16	43.548	50.220	62.492	78.092	97.020	144.852	206.012
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int16	47.364	53.948	66.236	81.836	100.764	148.612	209.764
TFLiteMicro	SparkFunEdge	int8	111.051	117.066	124.691	133.957	144.832	171.473	204.613
MicroAI (no CMSIS-NN)	SparkFunEdge	int8	**43.256**	**42.249**	**48.229**	**55.854**	**65.089**	**88.343**	**118.202**
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int8	45.038	43.474	49.464	57.078	66.322	89.683	119.541
MicroAI (CMSIS-NN)	SparkFunEdge	int8	41.314	44.600	50.724	58.532	67.984	91.900	122.486
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int8	45.212	48.492	54.636	62.436	71.892	95.796	126.380
STM32Cube.AI	Nucleo-L452RE-P	int8	72.742	77.746	84.336	92.582	102.430	126.996	158.098

Similary, the inference times detailled in Table B.2 can also suffer from an important overhead. For a non-quantized network (float32) with 16 filters per convolution, TensorFlow Lite for Microcontroller is more than three times slower (180 ms) than our framework (53 ms), both on the SparkFun Edge board. Interestingly, STM32Cube.AI is also initially slightly slower (85 ms) than our framework (56 ms) on the Nucleo-L452RE-P board. When the number of filters per convolution increases, this overhead is less significant For non-quantized networks, STM32Cube.AI leads in terms of inference time at 80 filters per convolution. However, for 8-bit quantized networks with CMSIS-NN optimizations, the inference times between STM32Cube.AI and our framework are very similar. This is mainly a result of using the CMSIS-NN library.

Table B.2: Inference time for one input vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			Response Time (ms)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	180	294	439	624	861	1407	2087
MicroAI	SparkFunEdge	float32	53	154	259	394	570	1017	1561
MicroAI	Nucleo-L452RE-P	float32	56	152	259	396	559	977	1512
STM32Cube.AI	Nucleo-L452RE-P	float32	85	174	271	404	544	922	1387
MicroAI (no CMSIS-NN)	SparkFunEdge	int16	41	113	191	288	389	668	1042
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int16	45	120	205	318	460	796	1224
MicroAI (CMSIS-NN)	SparkFunEdge	int16	35	57	83	117	156	272	403
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int16	38	64	94	130	173	276	405
TFLiteMicro	SparkFunEdge	int8	93	131	173	225	281	418	592
MicroAI (no CMSIS-NN)	SparkFunEdge	int8	39	102	173	260	376	658	1003
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int8	43	108	181	273	384	660	1034
MicroAI (CMSIS-NN)	SparkFunEdge	int8	25	46	73	105	146	242	368
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int8	**25**	**46**	**73**	**105**	**143**	**238**	356
STM32Cube.AI	Nucleo-L452RE-P	int8	32	54	80	112	146	242	**352**

The energy figures presented in Table B.3 follow a similar pattern, since the energy depends on the inference time. However, the power consumption of the Nucleo-L452RE-P board is much higher than the SparkFun Edge board. Therefore, the energy efficiency is significantly better for the SparkFun Edge board, even though STM32Cube.AI is always faster than TensorFlow Lite Micro, and sometimes faster than our framework.

Table B.3: Energy consumption for 1 input vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			Energy (µWh)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	0.135	0.221	0.330	0.469	0.647	1.058	1.569
MicroAI	SparkFunEdge	float32	0.040	0.116	0.195	0.297	0.428	0.765	1.174
MicroAI	Nucleo-L452RE-P	float32	0.247	0.675	1.148	1.753	2.478	4.327	6.700
STM32Cube.AI	Nucleo-L452RE-P	float32	0.378	0.771	1.202	1.789	2.412	4.083	6.146
MicroAI (no CMSIS-NN)	SparkFunEdge	int16	0.031	0.085	0.144	0.216	0.293	0.502	0.783
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int16	0.199	0.533	0.910	1.410	2.038	3.528	5.421
MicroAI (CMSIS-NN)	SparkFunEdge	int16	0.026	0.043	0.063	0.088	0.118	0.204	0.303
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int16	0.168	0.283	0.416	0.576	0.765	1.223	1.794
TFLiteMicro	SparkFunEdge	int8	0.070	0.098	0.130	0.169	0.211	0.314	0.445
MicroAI (no CMSIS-NN)	SparkFunEdge	int8	0.030	0.076	0.130	0.195	0.283	0.495	0.754
MicroAI (no CMSIS-NN)	Nucleo-L452RE-P	int8	0.191	0.477	0.801	1.209	1.700	2.924	4.581
MicroAI (CMSIS-NN)	SparkFunEdge	int8	**0.019**	**0.035**	**0.055**	**0.079**	**0.110**	**0.182**	**0.276**
MicroAI (CMSIS-NN)	Nucleo-L452RE-P	int8	0.111	0.206	0.322	0.467	0.635	1.055	1.578
STM32Cube.AI	Nucleo-L452RE-P	int8	0.143	0.239	0.356	0.495	0.647	1.072	1.560

Appendix C

Extract of CSV File for DRINKING Activity of UCA-EHAR Subject T1

Listing C.1: DRINKING_T1_1.csv

```
T;Ax;Ay;Az;Gx;Gy;Gz;P;CLASS
29019.00;-0.06;-0.77;10.28;0.04;-0.09;-0.08;1005.65;SITTING
29058.00;-0.10;-0.71;10.30;0.04;-0.08;-0.09;1005.65;SITTING
29101.00;-0.14;-0.70;10.24;0.03;-0.08;-0.08;1005.66;SITTING
29140.00;-0.13;-0.74;10.33;0.02;-0.09;-0.06;1005.66;SITTING
29179.00;-0.06;-0.71;10.41;0.05;-0.10;-0.06;1005.69;SITTING
29218.00;-0.02;-0.70;10.22;0.06;-0.10;-0.06;1005.68;SITTING
29257.00;-0.02;-0.68;10.20;0.07;-0.08;-0.05;1005.66;SITTING
29296.00;-0.09;-0.75;10.31;0.07;-0.07;-0.05;1005.65;SITTING
29339.00;-0.14;-0.84;10.39;0.10;-0.08;-0.05;1005.62;SITTING
29378.00;-0.09;-0.90;10.28;0.15;-0.09;-0.05;1005.62;SITTING
29418.00;-0.17;-0.94;10.27;0.15;-0.09;-0.03;1005.62;SITTING
29457.00;-0.16;-0.76;10.38;0.19;-0.10;-0.03;1005.62;SITTING
29500.00;0.01;-0.77;10.22;0.23;-0.11;-0.04;1005.61;SITTING
29539.00;0.03;-0.59;10.20;0.27;-0.10;-0.06;1005.61;SITTING
29578.00;-0.09;-0.50;10.31;0.23;-0.08;-0.09;1005.62;DRINKING
29617.00;-0.27;-0.46;10.40;0.27;-0.11;-0.09;1005.63;DRINKING
29660.00;-0.21;-0.35;10.41;0.30;-0.13;-0.07;1005.64;DRINKING
29699.00;-0.19;-0.28;10.46;0.32;-0.15;-0.04;1005.64;DRINKING
29738.00;-0.14;-0.15;10.61;0.40;-0.17;0.00;1005.64;DRINKING
29777.00;-0.03;-0.11;10.60;0.51;-0.18;0.04;1005.63;DRINKING
29816.00;0.06;0.09;10.45;0.60;-0.19;0.07;1005.63;DRINKING
29859.00;0.16;0.32;10.29;0.66;-0.19;0.09;1005.63;DRINKING
29898.00;0.24;0.49;10.28;0.69;-0.16;0.09;1005.63;DRINKING
29937.00;0.24;0.94;10.33;0.67;-0.16;0.08;1005.62;DRINKING
29980.00;0.25;1.22;10.18;0.67;-0.17;0.08;1005.63;DRINKING
```

Appendix D

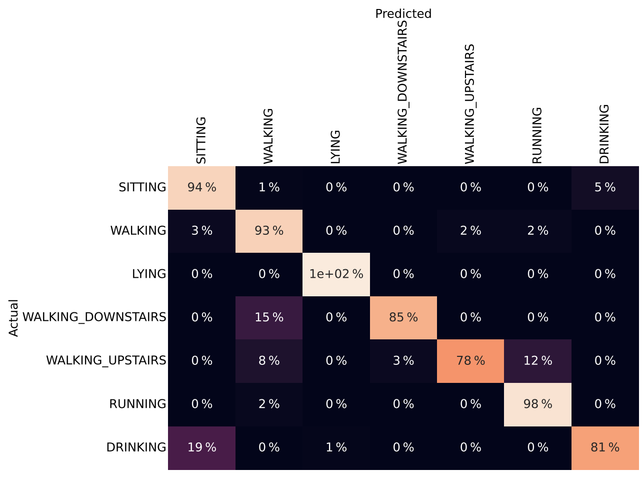
Fine-Tuning for Subjects T2, T3, T5, and T20 of UCA-EHAR

Contents

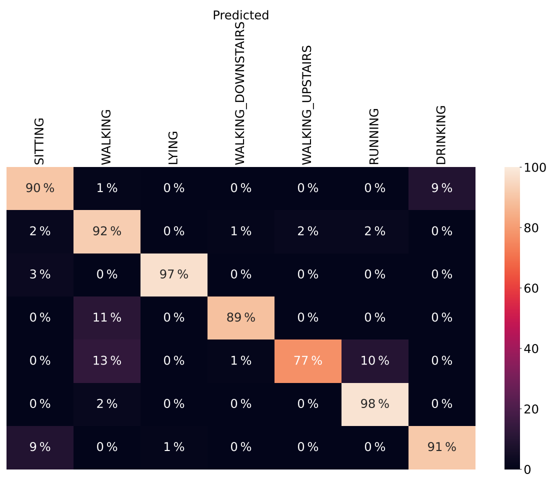
Subjects T2, T3, T5 and T20 of UCA-EHAR are analyzed separately to illustrate the possible outcomes of the method presented in Section 5.4.2. The confusion matrices provided below show the results for a single training only for illustration purposes.

D.1 Results with Fine-Tuning on Subject T2

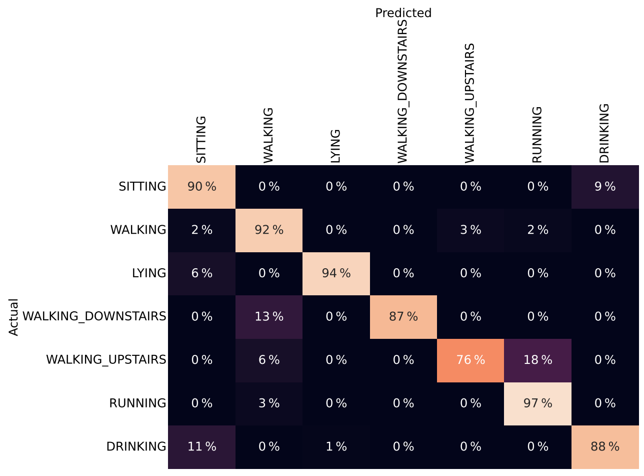
When subject T2 is selected for evaluation, the ResNet shows a high percentage of DRINKING activities predicted as SITTING in Figure D.1a. A high misprediction rate on WALKING_UPSTAIRS and WALKING_DOWNSTAIRS can also be noted. With a dynamic self-organizing map learning from the ResNet feature maps in Figure D.1b, some DRINKING mispredicted as SITTING shift to SITTING mispredicted as DRINKING. There is still a high misprediction rate on WALKING_UPSTAIRS and WALKING_DOWNSTAIRS. The fine-tuning does not seemingly change the results in Figure D.1c. Relabelling the neurons in Figure D.1d helps decreasing the rate of WALKING_UPSTAIRS mispredicted as RUNNING. Overall, the confusion between WALKING_UPSTAIRS and RUNNING has decreased after fine-tuning and relabelling compared to the ResNet.



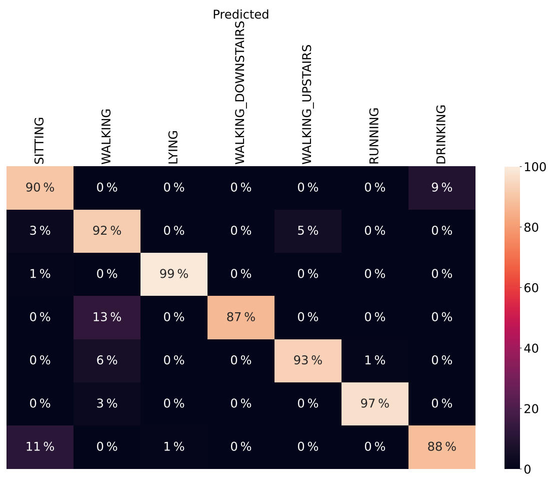
(a) Step n°1 ResNet



(b) Step n°2 ResNet+DSOM



(c) Step n°3 ResNet+DSOM Fine-tuned on subject T2.

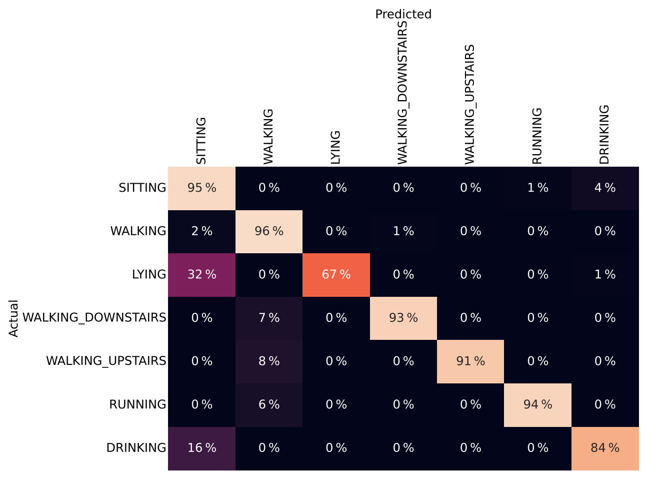


(d) Step n°4 ResNet+DSOM Fine-tuned on subject T2 and relabelled with original training set

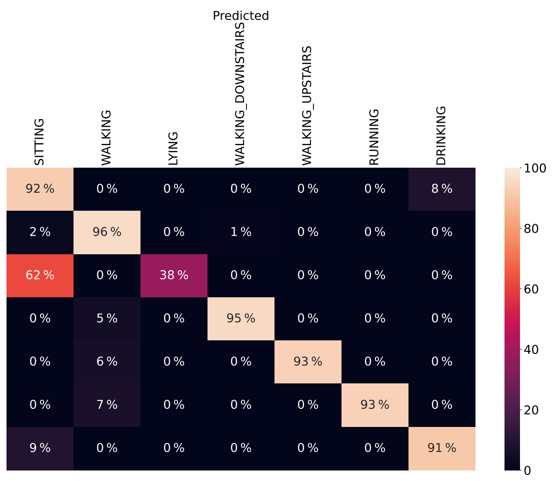
Figure D.1: Confusion matrices for subject T2 of UCA-EHAR without STANDING.

D.2 Results with Fine-Tuning on Subject T3

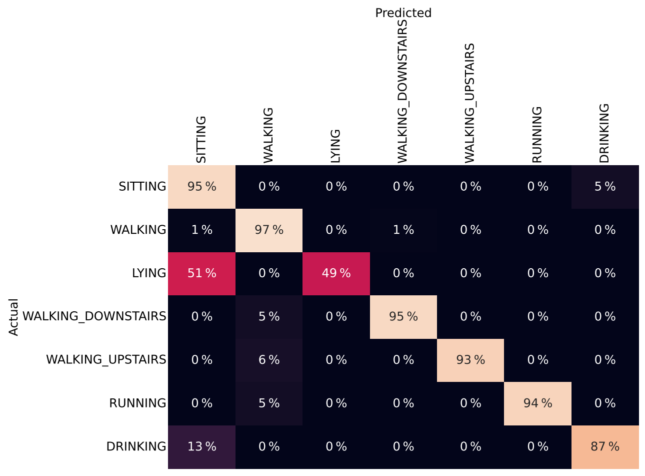
With subject T3, a high number of samples labelled LYING are predicted as SITTING in Figure D.2a for the ResNet at step n°1, with also some confusion between DRINKING and SITTING. At step n°2 (Figure D.2b), the ResNet+DSOM model shows an ever higher confusion between LYING and SITTING. At step n°3 (Figure D.2c), the fine-tuning reduces the misprediction rate of LYING. Relabelling at step n°4 also reduces the misprediction rate of LYING as shown in Figure D.2d. Overall, the confusion is mostly the same after fine-tuning and relabelling compared to the ResNet, although the accuracy for some activities increased ever so slightly.



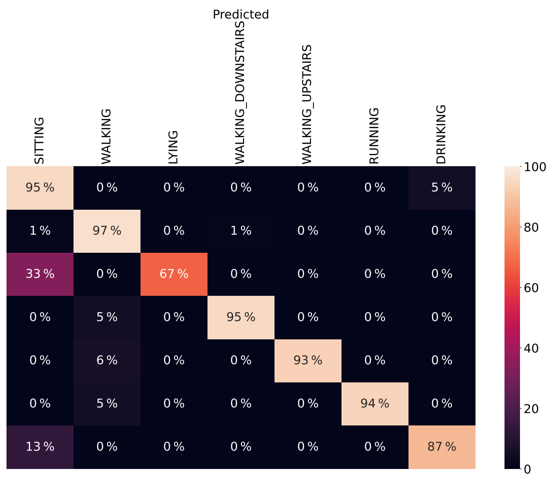
(a) Step n°1 ResNet



(b) Step n°2 ResNet+DSOM



(c) Step n°3 ResNet+DSOM Fine-tuned on subject T3.

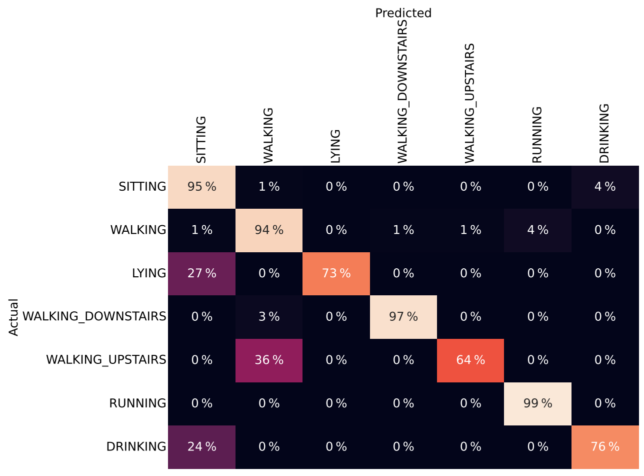


(d) Step n°4 ResNet+DSOM Fine-tuned on subject T3 and relabelled with original training set

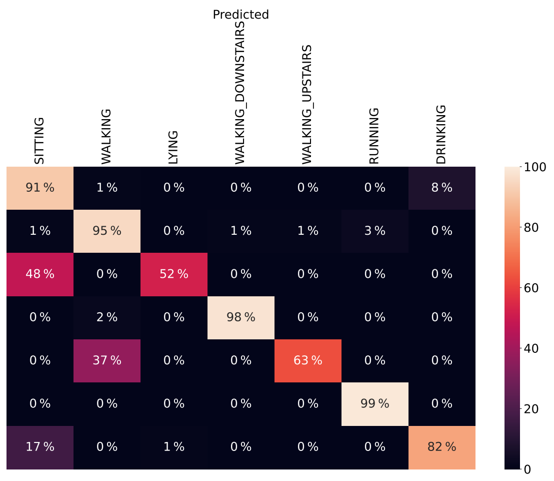
Figure D.2: Confusion matrices for subject T3 of UCA-EHAR without STANDING.

D.3 Results with Fine-Tuning on Subject T5

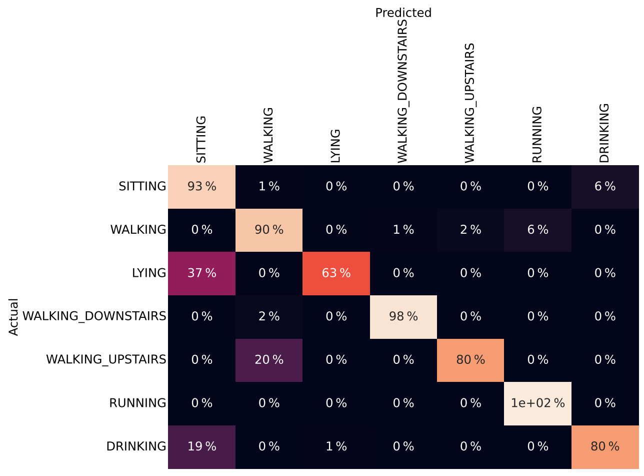
If we take a look at the results of the ResNet on subject T5 in Figure D.3a, several activities have a high misprediction rate: LYING predicted as SITTING, WALKING_UPSTAIRS predicted as WALKING, and DRINKING predicted as SITTING. When using a DSOM classifier after the ResNet feature extractor in Figure D.3b, the confusion on the LYING class gets worse. When learning part of the subject data in an unsupervised manner in Figure D.3c, the confusion on LYING, WALKING_UPSTAIRS, and DRINKING is slightly reduced. Relabelling the neurons afterwards allows solving the confusion between LYING and SITTING as seen in Figure D.3d. Overall, comparing the original ResNet and the fine-tuned and relabelled ResNet+DSOM, the confusion between LYING and SITTING has been completely solved while the confusion on WALKING_UPSTAIRS has decreased.



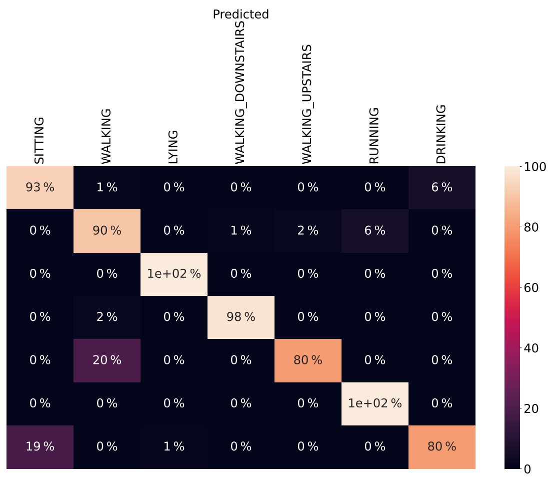
(a) Step n°1 ResNet



(b) Step n°2 ResNet+DSOM



(c) Step n°3 ResNet+DSOM Fine-tuned on subject T5.

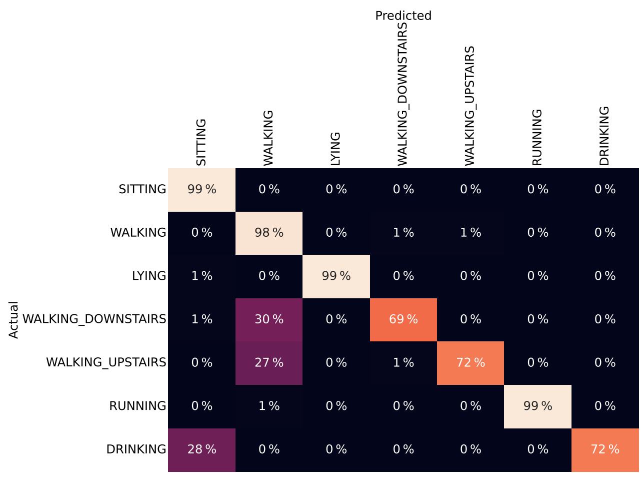


(d) Step n°4 ResNet+DSOM Fine-tuned on subject T5 and relabelled with original training set

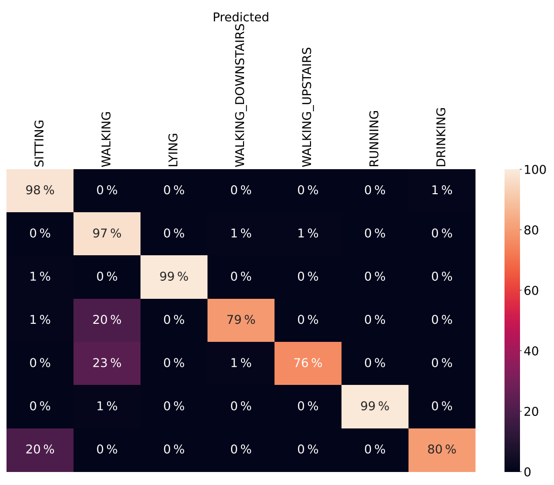
Figure D.3: Confusion matrices for subject T5 of UCA-EHAR without STANDING.

D.4 Results with Fine-Tuning on Subject T20

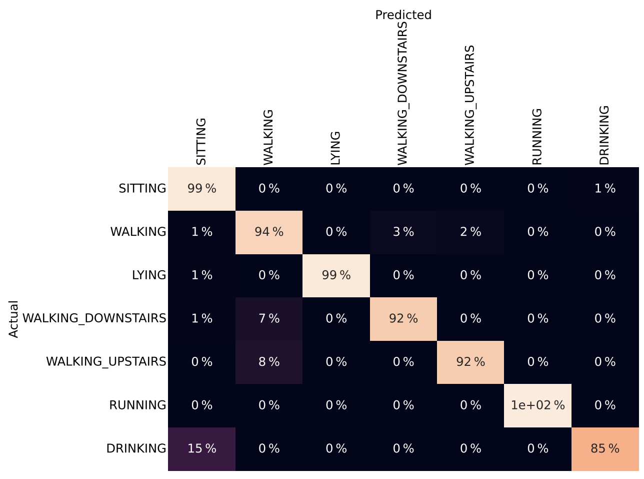
Figure D.4a shows confusion between DRINKING and SITTING similar to the other subjects. There is also some confusion with WALKING_DOWNSTAIRS and WALKING_UPSTAIRS predicted as WALKING by the ResNet. When using the ResNet+DSOM model, the confusion decreases overall in Figure D.4b. Figure D.4c shows that the confusion decreases even further after the fine-tuning process. However, relabelling the neurons (Figure D.4d) brings the confusion on WALKING_DOWNSTAIRS and DRINKING back. Overall, after fine-tuning and relabelling, the confusion betwen WALKING_UPSTAIRS and WALKING decreased, while the confusion on WALKING_DOWNSTAIRS and DRINKING slightly increased. For this subject, if no relabelling is performed, the confusion also decreases for WALKING_DOWNTAIRS and DRINKING.



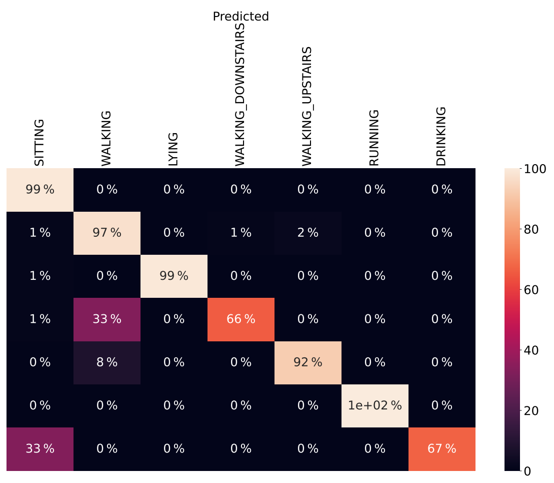
(a) Step n°1 ResNet



(b) Step n°2 ResNet+DSOM



(c) Step n°3 ResNet+DSOM Fine-tuned on subject T20.

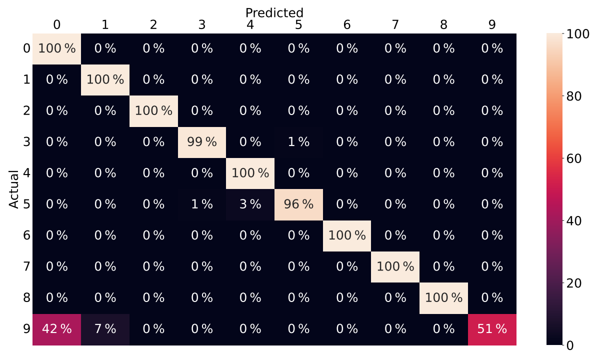


(d) Step n°4 ResNet+DSOM Fine-tuned on subject T20 and relabelled with original training set

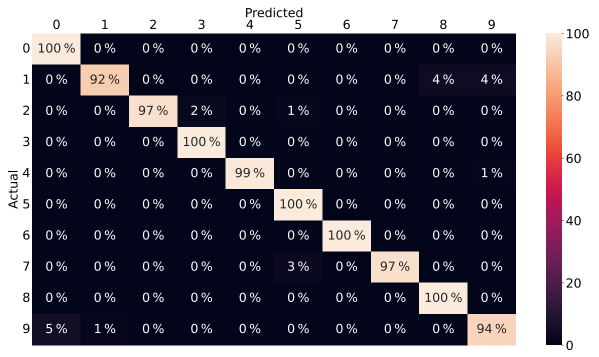
Figure D.4: Confusion matrices for subject T20 of UCA-EHAR without STANDING.

Appendix E

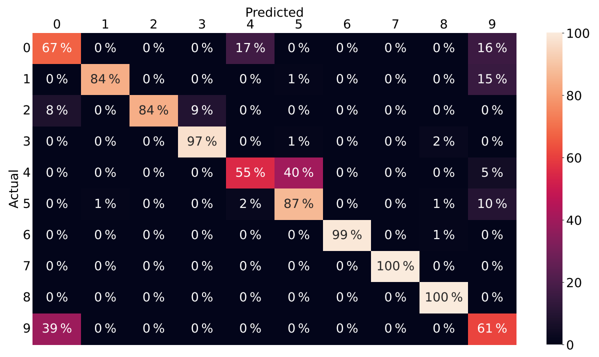
Heidelberg Digits Subjects



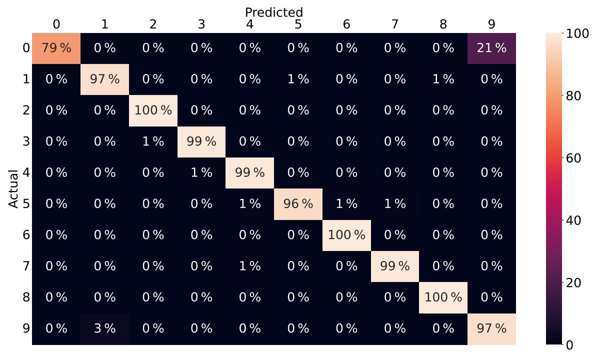
(a) Subject 0



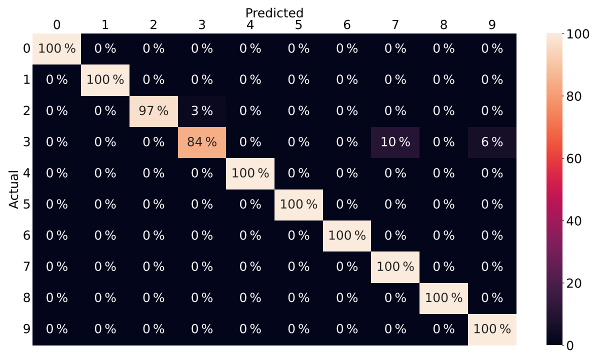
(b) Subject 1



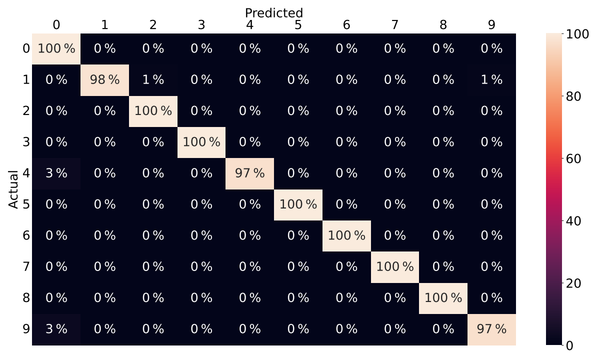
(c) Subject 2



(d) Subject 3

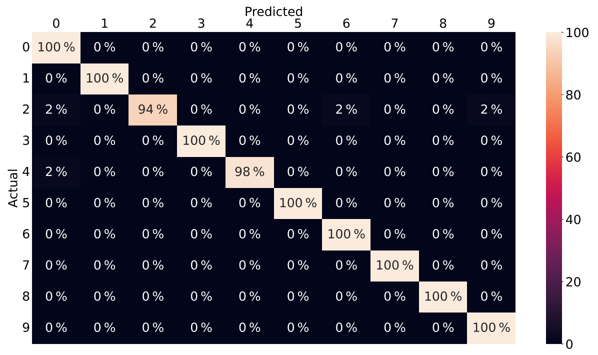


(e) Subject 4

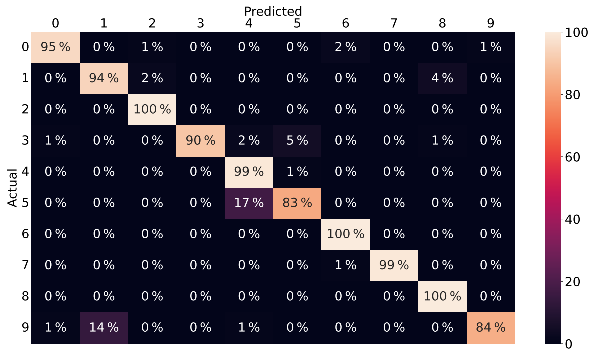


(f) Subject 5

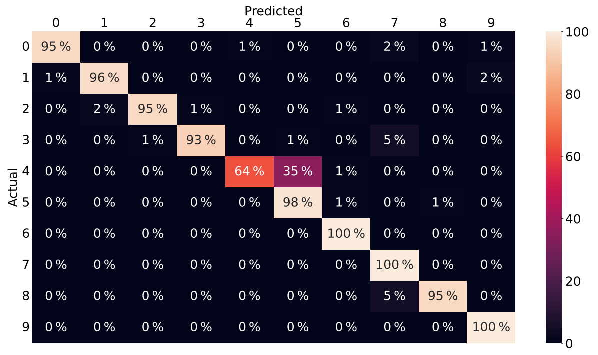
Figure E.1: Confusion matrices for each subject of Heidelberg Digits.



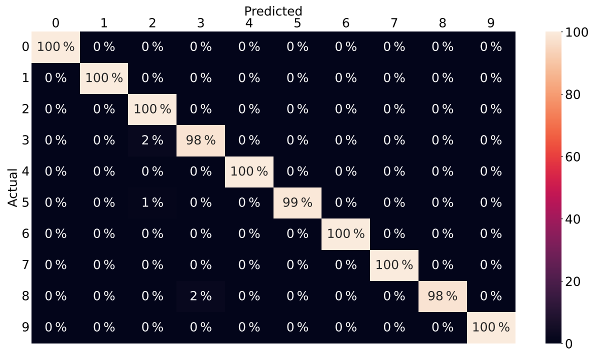
(g) Subject 6



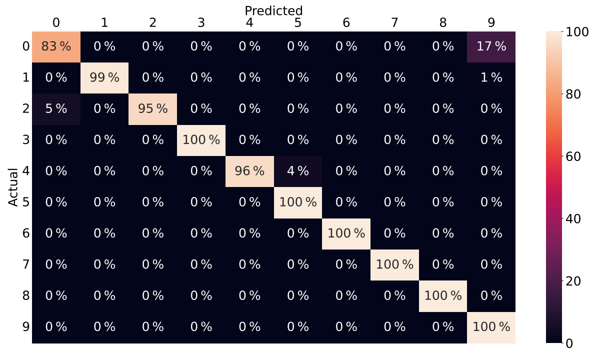
(h) Subject 7



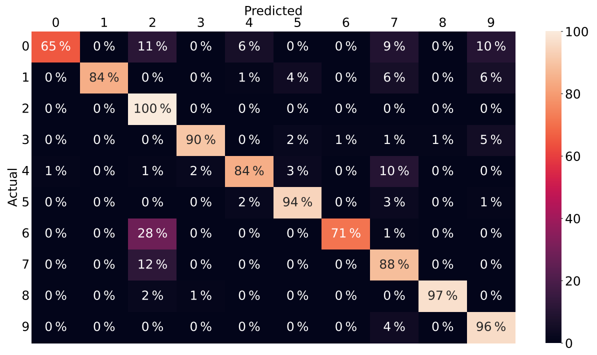
(i) Subject 8



(j) Subject 9



(k) Subject 10



(l) Subject 11

Figure E.1: Confusion matrices for each subject of Heidelberg Digits. (continued)

Appendix F

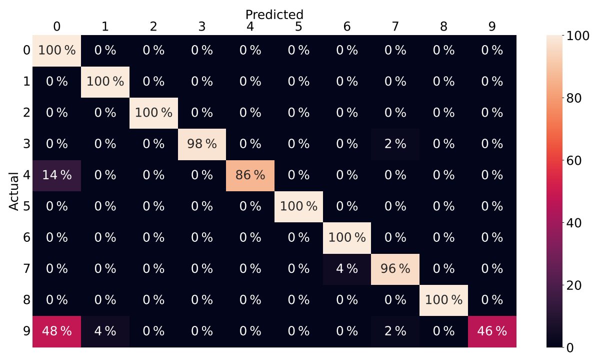
Fine-Tuning for Subjects 0, 2 and 3 of Heidelberg Digits

Contents

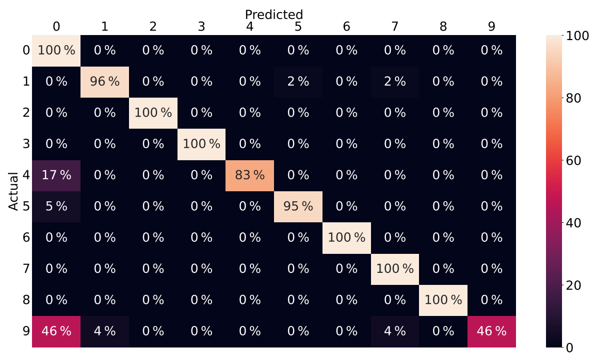
In the following experiments, a single subject of the test set is selected, and the model is tested on 50% of its data chosen randomly. The ResNet is still trained on the training set. The ResNet+DSOM is also originally trained on the training set in an unsupervised manner, with supervised labelling using all of the training set's labels. For subjects 0 and 2, the ResNet and the ResNet+DSOM are trained on all subjects except subjects 0, 2, and 11. For subject 3, the ResNet and the ResNet+DSOM are trained on all subjects except subjects 3, 7, and 8. The fine-tuning is performed with the remaining 50% of the selected subject's data.

F.1 Results with Fine-Tuning on Subject 0

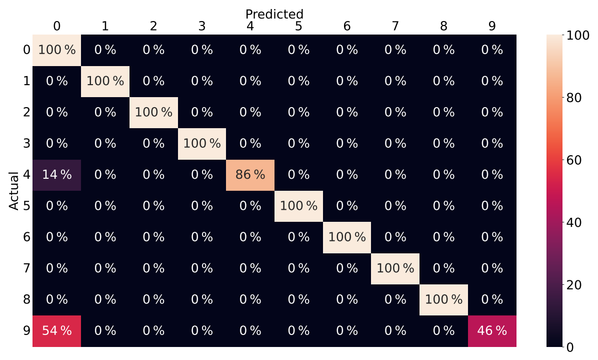
Let us analyze the results obtained for subject 0. For the ResNet, a high rate of misprediction is observed in Figure F.1a for class 9 predicted as class 0 (48%). The ResNet+DSOM model provides similar results in Figure F.1b. Unfortunately, fine-tuning the DSOM in an unsupervised manner on the other 50% of the subject's data does not improve the classification performance as can be seen in Figure F.1c. Labelling the neurons again with the original training set does not help either as shown in Figure F.1d.



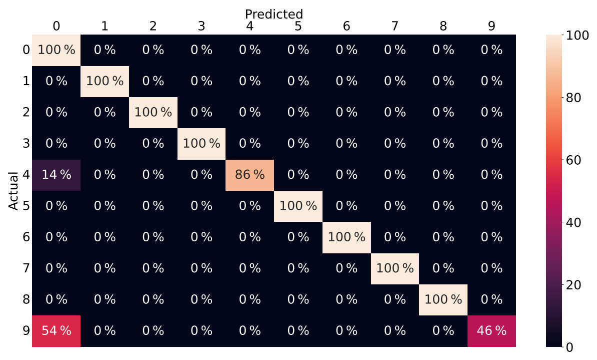
(a) ResNet



(b) ResNet+DSOM



(c) ResNet+DSOM fine-tuned on half of subject 0

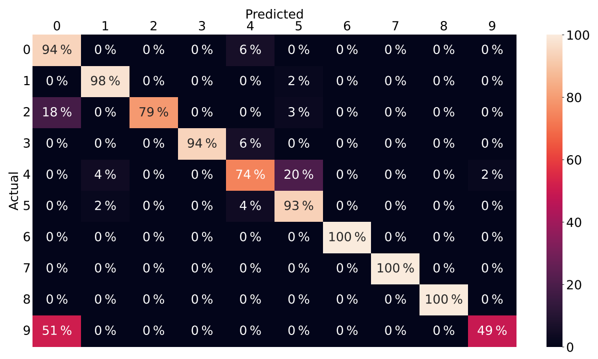


(d) ResNet+DSOM fine-tuned on half of subject 0 and relabelled with original training set

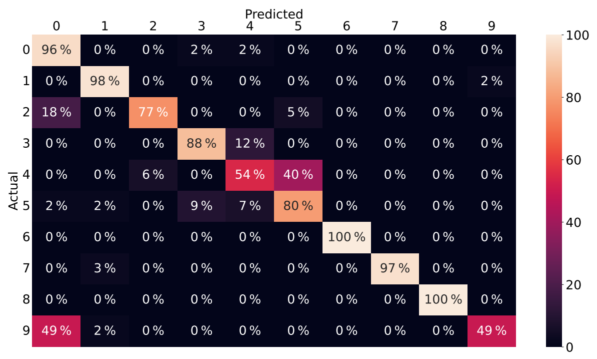
Figure F.1: Confusion matrices for half of subject 0 of Heidelberg Digits trained with subjects 0, 2, 11 excluded from the train set

F.2 Results with Fine-Tuning on Subject 2

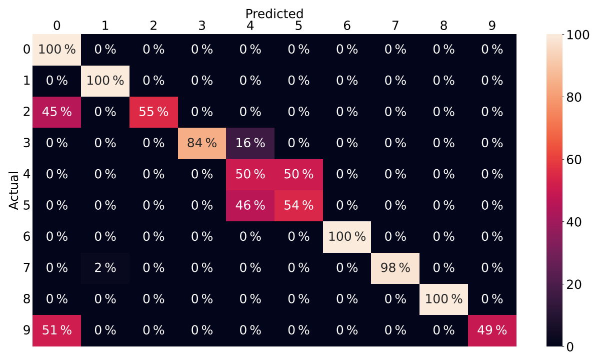
The results are even less encouraging with subject 2. Fine-tuning the DSOM creates even more confusion (Figure F.2c) compared to the ResNet+DSOM model without fine-tuning (Figure F.2b), which was itself worse than the original ResNet (Figure F.2a). Relabelling the DSOM with the original training set slightly recovers accuracy (Figure F.2d). Neverthless, the accuracy after relabelling is still lower than the accuracy of the original ResNet.



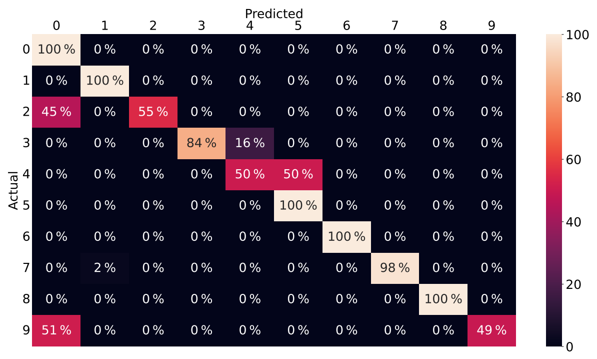
(a) ResNet



(b) ResNet+DSOM



(c) ResNet+DSOM fine-tuned on half of subject 2

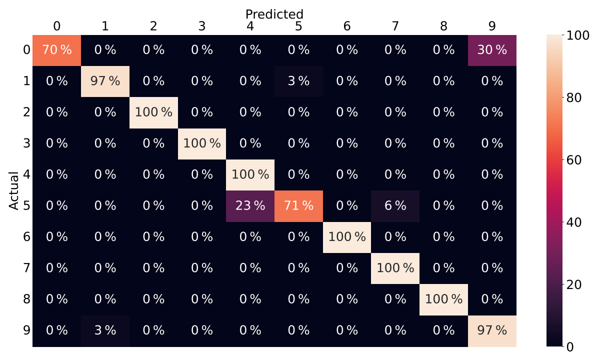


(d) ResNet+DSOM fine-tuned on half of subject 2 and relabelled with original training set

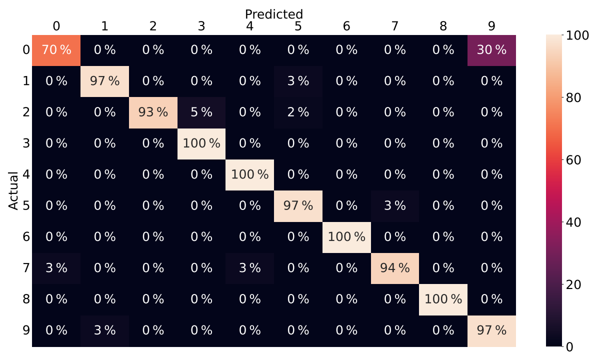
Figure F.2: Confusion matrices for half of subject 2 of Heidelberg Digits with subjects 0, 2, 11 exluded from the train set

F.3 Results with Fine-Tuning on Subject 3

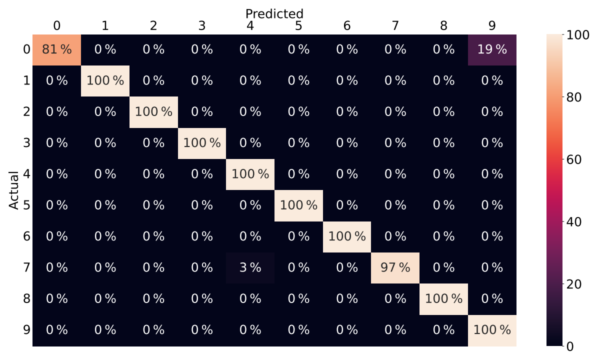
With a different testing set composed of subjects 3, 7, and 8, the predictions were already of better quality than subjects 0, 2 and 11. In this case, the results can be slightly more promising. Subject 3 exhibits slightly less confusion with a ResNet+DSOM (Figure F.3b) than with the ResNet (Figure F.3a) on class 5. The fine-tuning also helps to reduce the confusion between class 0 and 5 (Figure F.3c). However, relabelling after fine-tuning creates confusion between class 5 and 7 that was not present before (Figure F.3d).



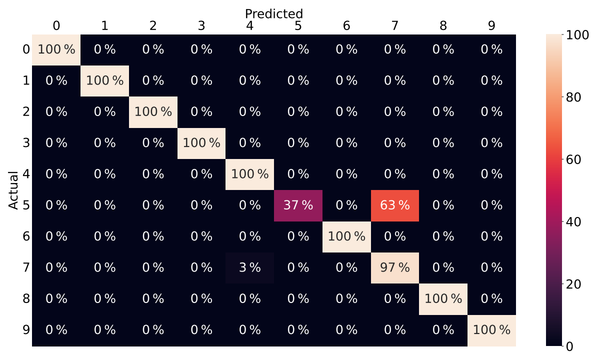
(a) ResNet



(b) ResNet+DSOM



(c) ResNet+DSOM fine-tuned on half of subject 3



(d) ResNet+DSOM fine-tuned on half of subject 3 and relabelled with original training set

Figure F.3: Confusion matrices for half of subject 3 of Heidelberg Digits with subjects 3, 7, 8 exluded from the train set

Appendix G

Catastrophic Forgetting with CORe50

Contents

G.1 Introduction

A different experiment has been performed with CORe50[159], an image dataset for object recognition. Using this dataset, the task to solve is different from the previous datasets' task. The purpose is to challenge a continual learning method on its robustness towards catastrophic forgetting. There is no subject behaviour to study and the goal is different from what we want to achieve with unsupervised online learning. Nonetheless, the CORe50 dataset is popular in the litterature and can provide insights on the characteristics of an online learning method. Since our intent is to learn from new data of the same classes rather than add new classes, we are mainly interested in the New Instances scenario.

CORe50 consists of 50 different objects grouped in 10 categories (5 objects per category) recorded with a camera, 300 frames per object for each session. Both the object and the camera can move during the capture. A given recording session is taken with one background and lighting conditions. There are 11 sessions with different background and lighting conditions. Among these sessions, session 3, 7 and 10 are used for testing.

G.2 Experiments and Results

In these experiments, ResNet-18 model from the torchvision[215] package pre-trained on the ImageNet-1K dataset is used as a feature extractor. A dynamic self-organizing map is placed after the feature extractor, and trained in an unsupervised manner. Grid search is used to optimize the following hyperparameters of the dynamic self-organizing map: dimensions of the grid of neurons, learning rate ϵ, elasticity η, σ for labelling, and the number of epochs. The selected parameters are: 16×19 grid size, $\epsilon = 0.00379$, $\eta = 1.59$, $\sigma = 3.78$, and 32 epochs.

The labels used are those of the 10 object categories rather than the 50 objects, so there are 10 classes in total.

Figure G.1 presents a t-SNE[202] visualization of the prototype vector of each neuron of the self-organizing map, after training with session 1. The numbers next to the point correspond to the label of the neuron, from 0 to 9 for each of the 10 object categories. All the labels from session 1 were used for labelling. Colors correspond to a HDBSCAN[139] clustering method and can be ignored. As can be observed, this t-SNE visualization is significantly different from the one presented for the UCA-EHAR dataset in Section 5.4.2. The hyperparameter research also converged towards a large number of neurons ($16 \times 19 = 304$), but this time the training actually affected all the neurons. Additionally, the neurons for a given class are close to each other, meaning that the features learnt by the neurons for a given class are similar. Some labels have two distinct group of neurons such as the group in the center and the group towards the bottom for the label 9. This can be caused by samples from the same class having distinct sets of features, since different objects belong to the same category in the CORe50 dataset. Some neurons, such as the neuron labelled 0 towards the middle-right of the figure (circled in magenta), have a label that do not match any of their neighbors. This could be a source of misclassifications.

Figure G.2 illustrates the 2-dimensional neurons grid of the same self-organizing map, with numbers and colors having the same meaning as the t-SNE visualization. The labels show that the neurons are mostly grouped relative to their labels, meaning that similar features were indeed learnt by neurons close to each other in the grid, as expected from a self-organizing map. For some classes, two disjoint groups of neurons can be observed such as a group of neurons labelled 9 towards the bottom right-hand corner of the grid and another one near the center of the grid (circled in green). This is however consistant with the observation from the t-SNE visualization. The labelling inconsistancies with the neuron labelled 0 towards the bottom of the grid (circled in magenta) and some others can also be observed on Figure G.2.

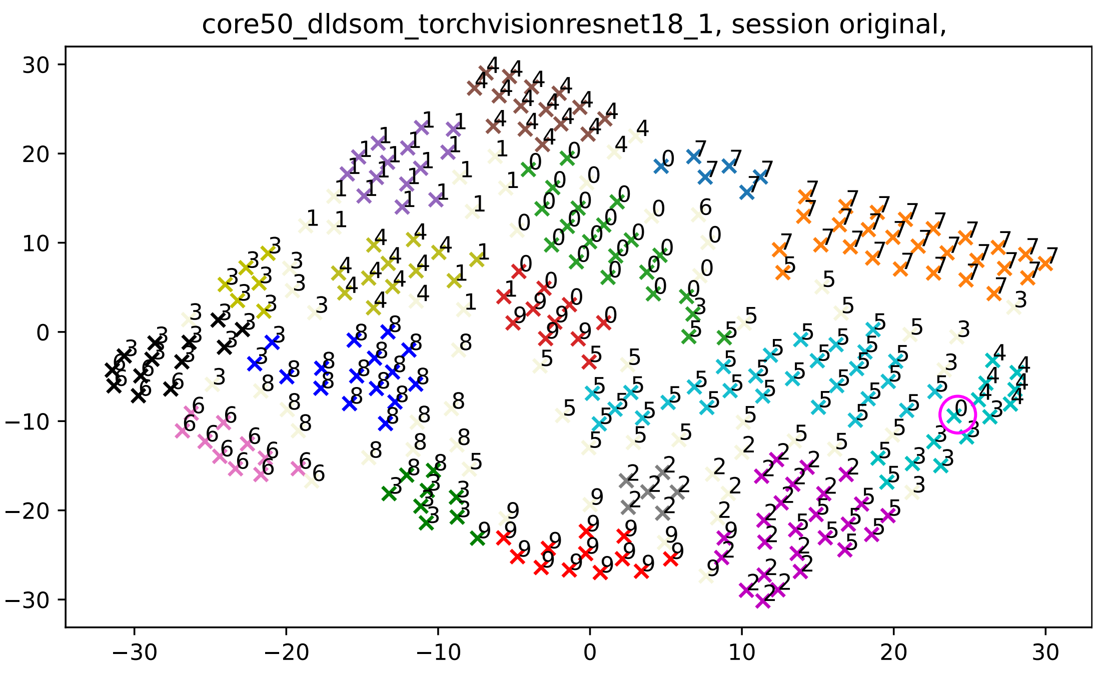


Figure G.1: t-SNE visualization of the DSOM neurons prototype vector on CORe50 with labels.

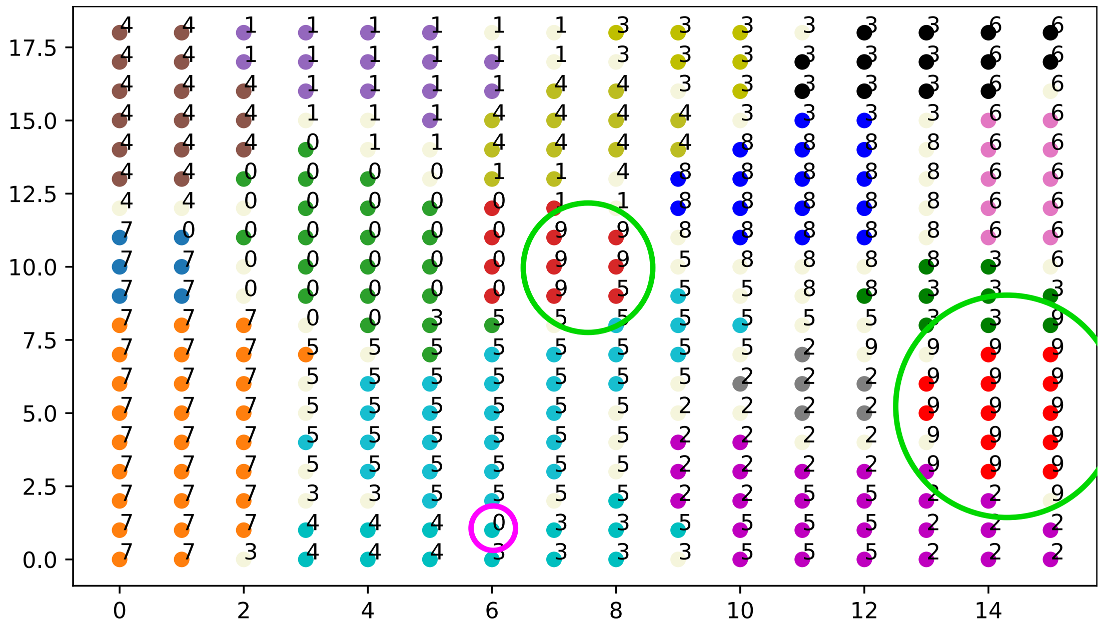


Figure G.2: Labelled DSOM 2-dimensional neurons grid on CORe50.

Figure G.3 illustrates the proportion of data from the training dataset needed for labelling. This result is also significantly different from the one obtained for UCA-EHAR in Section 5.4.2. For CORe50, many labelled data are required. In fact it is not possible to obtain the highest accuracy unless all labelled data are used. This dataset is much more complex and more neurons are being used in the self-organizing map. Additionally, the feature extractor was trained on a different, larger and even more complex dataset.

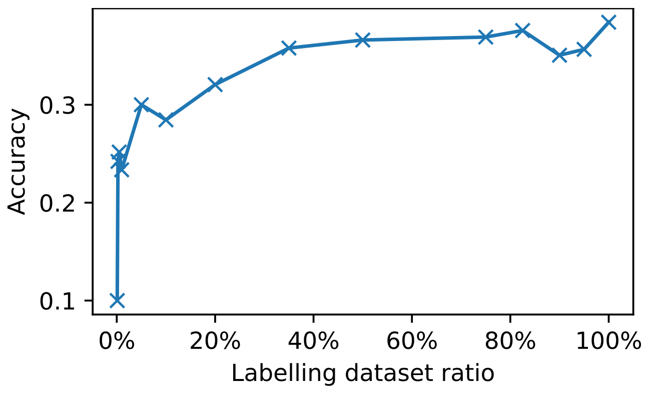


Figure G.3: Accuracy vs. percentage of data used from CORe50 training set for labelling DSOM neurons.

In order to simulate a continual learning situation, session 1 is always used for an initial training, then the other sessions are learnt sequentially in a random order.

The expected result of this sort of learning is that as more and more sessions are learnt, the accuracy on the test set increases. For that, the learning method should be able to consolidate its knowledge using the new data, without forgetting the previously learnt information. Therefore, without a mitigation against catastrophic forgetting, the accuracy would not progessively increase with new sessions being learnt.

This is illustrated in Figure G.4. As it can be seen, the sequential learning of the CORe50 sessions does not provide an increase in accuracy on the test dataset. The reason is that the dynamic self-organizing map is not designed to remember old information on long-term, but rather continuously follow the changes in the input distribution. In this experiment, 32 epochs were used to train the self-organizing map with the first session, and 1 epoch for the subsequent sessions. Using more epochs for the subsequent sessions does not change the interpretation of the results. Colors correspond to a session number in the legend. Neurons were relabelled after learning a new session with all the data from this session. Therefore, while the training of the dynamic self-organizing map is unsupervised, the method is overall supervised.

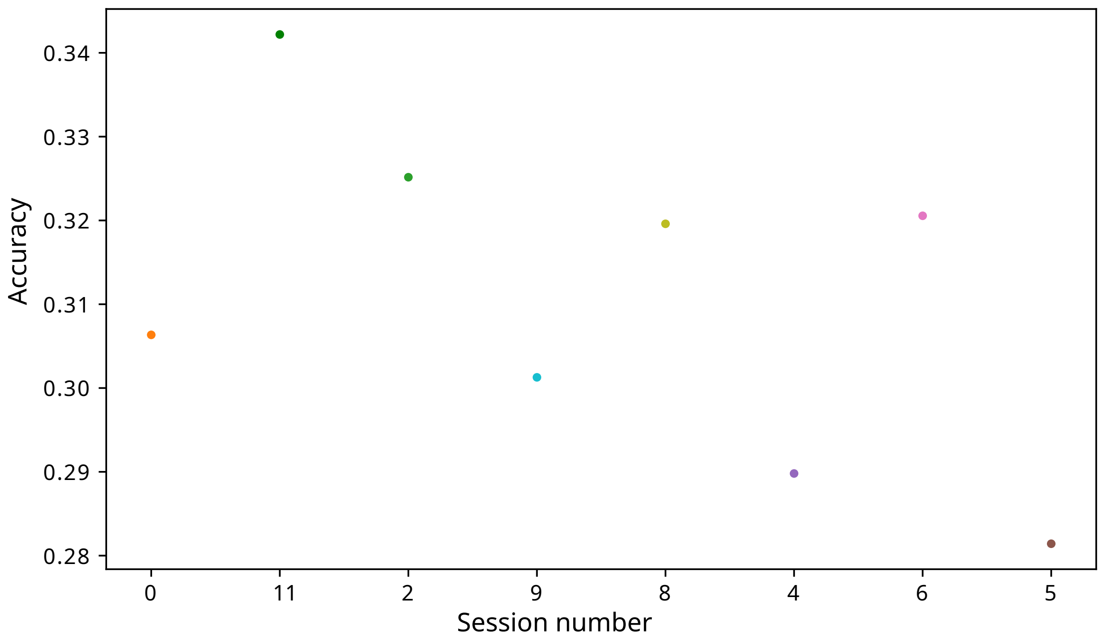


Figure G.4: Accuracy over the test set vs. session learnt.

G.3 Conclusion

We evaluated the problem of catastrophic forgetting using a dedicated dataset from the litterature, CORe50. As expected, results were poor since the dynamic self-organizing map is not designed to prevent this problem. However, the catastrophic forgetting was evaluated in a different manner from the use-case we consider. In the case of human activity recognition, the neural network is trained with several subjects, then fine-tuned and evaluated on another subject. Thus, it does not matter much if the original subjects are forgotten. However, catastrophic forgetting can still be an issue with the new subject if the classes are seen in a sequential order. In this case, there is a risk of forgetting classes that were not seen in a long time. Latent replay[162] could be implemented to mitigate this problem.

The experiment with CORe50 also showed a situation where transfer learning was used: the feature extractor was trained in a supervised manner on the ImageNet dataset, rather than on the CORe50 dataset. It was then used to feed a self-organizing map trained in an unsupervised manner. However, with only a few percents of accuracy loss, less than 25% of labels could be used, making this approach a true semi-supervised method.

Appendix H

Example Computation of Fixed-Point Exponential Function

For example, the value 1.875 encoded as fixed-point number on 8 bits in Q4.4 format (4 bits for the fractional part) is 0001 1110 in binary. $\log_2 e$ with in Q4.4 format is 0001 0111 in binary, or 1.4375 in decimal. To compute $e^{1.875}$, first the exponentiation base is changed from e to 2:

$$e^{1.875} = 2^{1.875 \times \log_2 e} = 2^{1.875 \times 1.4375} = 2^{2.6953125} \tag{H.1}$$

After the multiplication, the result now has 8 bits for the fractional part. Thus, it is scaled back to 4 bits for the fractional part, 0010 1011 or 2.6875 in decimal.

The power-of-two is first computed for the integer part after removing the fractional part. The integer part is 2 in decimal or 0010 in binary, producing a result of $2^2 = 4$ in decimal or 0100 in binary. Putting back an empty 4-bit fractional part, the intermediate result is now 0100 0000 in Q4.4 format. These operations can be implemented with a bit shift to the left.

Then, the intermediate result is multiplied by the n-th roots of 2. For the fractional part of 1011, the n-th roots are $\sqrt[2^1]{2}$, $\sqrt[2^3]{2}$, and $\sqrt[2^4]{2}$. After converting to fixed-point numbers (round to the nearest), they are 1.4375, 1.0625 and 1.0625 in decimal, respectively. This conversion rounds to the nearest. So the result is:

$$\begin{aligned} 2^{2.6875} &= ((2^2 \times \sqrt[2^1]{2}) \times \sqrt[2^3]{2}) \times \sqrt[2^4]{2} \\ &\approx ((4 \times 1.4375) \times 1.0625) \times 1.0625 \\ &= 5.75 \times 1.0625 \times 1.0625 \\ &= 6.109375 \times 1.0625 \\ &\approx 6.0625 \times 1.0625 \\ &= 6.44140625 \\ &\approx 6.4375 \end{aligned} \tag{H.2}$$

which is encoded as 0110 0111 in Q4.4 format. The intermediate results are rounded down when scaling back the fixed-point number after each multiplication.

With floating-point numbers, the `expf()` function would give a result of 6.520819. This example is given with 8-bit numbers for illustration purposes, however the actual implementation is used with 32-bit numbers.

Bibliography

[1] INSEE. *Tableaux de l'économie française: Population par âge*. 2020. URL: https://www.insee.fr/fr/statistiques/4277619 (visited on 10/21/2022).

[2] Yu Wang, Gu-Yeon Wei, and David Brooks. "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning." In: (2019). DOI: 10.48550/arXiv.1907.10701.

[3] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. "MCUNet: Tiny Deep Learning on IoT Devices." In: *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS 2020)*. 2020. DOI: 10.48550/arXiv.2007.10319.

[4] Liangzhen Lai and Naveen Suda. "Enabling Deep Learning at the IoT Edge." In: *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2018)*. 2018, pp. 1–6. DOI: 10.1145/3240765.3243473.

[5] Roland Kromes, Adrien Russo, Benoît Miramond, and François Verdier. "Energy consumption minimization on LoRaWAN sensor network by using an Artificial Neural Network based application." In: *Proceedings of the 2019 IEEE Sensors Applications Symposium (SAS 2019)*. 2019, pp. 1–6. DOI: 10.1109/SAS.2019.8705992.

[6] Precedence Research. *Microcontroller Market*. URL: https://www.precedenceresearch.com/microcontroller-mcu-market (visited on 10/13/2022).

[7] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. 1955. URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.

[8] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity." In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[9] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386. DOI: 10.1037/h0042519.

[10] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[11] Yann LeCun et al. "Backpropagation applied to handwritten zip code recognition." In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[12] Adam Paszke et al. "Automatic differentiation in pytorch." In: *31st Conference on Neural Information Processing Systems (NeurIPS 2017), Autodiff Workshop*. 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ.

[13] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 2016, pp. 265–283. DOI: 10.5555/3026877.3026899.

[14] Roy Frostig, Matthew Johnson, and Chris Leary. “Compiling machine learning programs via high-level tracing.” In: *Proceedings of the 2018 Systems for Machine Learning Conference (SysML 2018)*. Vol. 4. 2018. URL: https://mlsys.org/Conferences/doc/2018/146.pdf.

[15] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. “ImageNet: A large-scale hierarchical image database.” In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR 2009. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks.” In: *Proceedings of the 26th Conference on Neural Information Processing Systems (NeurIPS 2012)*. Vol. 25. 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[17] Tom Brown et al. “Language Models are Few-Shot Learners.” In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[18] Jaime Sevilla et al. *Parameter, Compute and Data Trends in Machine Learning*. 2022. URL: https://docs.google.com/spreadsheets/d/1AAIebjNsnJj%5C_uKALHbXNfn3%5C_YsT6sHXtCU0q7OIPuc4/.

[19] Md Zahangir Alom et al. “A State-of-the-Art Survey on Deep Learning Theory and Architectures.” In: *Electronics* 8.3 (2019). DOI: 10.3390/electronics8030292.

[20] Laith Alzubaidi et al. “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions.” In: *Journal of Big Data* 8 (53 2021). DOI: 10.1186/s40537-021-00444-8.

[21] Ubahnverleih. *Nvidia Jetson Nano 2 Development Kit*. CC0 1.0. URL: https://commons.wikimedia.org/wiki/File:Nvidia_Jetson_Nano_2_Development_Kit_15_14_39_352000.jpeg.

[22] Ahmet Ali Süzen, Burhan Duman, and Betül Şen. “Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN.” In: *Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA 2020)*. 2020, pp. 1–5. DOI: 10.1109/HORA49412.2020.9152915.

[23] Microchip. *8-bit PIC Microcontroller Peripheral Integration Quick Reference Guide*. DS30010068G. 2020. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/30010068G.pdf.

[24] STMicroelectronics. *STM32H7 series*. BRSTM32H70920. 2020. URL: https://www.st.com/resource/en/brochure/brstm32h7.pdf.

[25] *Zephyr*. URL: https://www.zephyrproject.org/ (visited on 07/20/2022).

[26] Lauranne Choquin and Fred Piry. *Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm*. 2020. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-custom-instructions-wp.pdf.

[27] Andrew Waterman, Yunsup Lee, Rimas Avizienis, Henry Cook, David Patterson, and Krste Asanovic. “The RISC-V instruction set.” In: *IEEE Hot Chips 25 Symposium*. HCS 2013. Poster. 2013. DOI: 10.1109/HOTCHIPS.2013.7478332.

[28] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. “XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions.” In: *Proceedings of the 23rd Design, Automation & Test in Europe Conference & Exhibition*. DATE 2020. 2020, pp. 186–191. DOI: 10.23919/DATE48585.2020.9116529.

[29] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit.” In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA 2017. 2017, pp. 1–12. DOI: 10.1145/3079856.3080246.

[30] Norman P. Jouppi et al. “Ten Lessons From Three Generations Shaped Google’s TPUv4i : Industrial Product.” In: *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*. ISCA 2021. 2021, pp. 1–14. DOI: 10.1109/ISCA52012.2021.00010.

[31] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. “An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks.” In: (2021). DOI: 10.48550/arXiv.2102.10423.

[32] Eitan Medina and Eran Dagan. “Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor.” In: *IEEE Micro* 40.2 (2020), pp. 17–24. DOI: 10.1109/MM.2020.2975185.

[33] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. “Survey of Machine Learning Accelerators.” In: *Proceedings of the 2020 IEEE High Performance Extreme Computing Conference*. HPEC 2020. 2020, pp. 1–12. DOI: 10.1109/HPEC43674.2020.9286149.

[34] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” In: *Proceedings of the 3rd International Conference on Learning Representations*. ICLR 2015. 2015. DOI: 10.48550/arXiv.1409.1556.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” In: *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR 2016. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[36] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.” In: *Proceedings of the 9th International Conference on Learning Representations*. ICLR 2021. 2021. DOI: 10.48550/arXiv.2010.11929.

[37] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size.” In: (2016). arXiv: 1602.07360 [cs.CV].

[38] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” In: (2017). arXiv: 1704.04861 [cs.CV].

[39] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices.” In: *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2018. 2018, pp. 6848–6856. DOI: 10.1109/CVPR.2018.00716.

[40] Tan M. and Le Q. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” In: *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*. Vol. 97. 2019, pp. 6105–6114. DOI: 10.48550/arXiv.1905.11946. arXiv: 1905.11946 [cs.CV].

[41] Sergey Zagoruyko and Nikos Komodakis. *Wide Residual Networks*. 2016. DOI: 10.48550/ARXIV.1605.07146. arXiv: 1605.07146 [cs.CV].

[42] Irwan Bello et al. “Revisiting resnets: Improved training and scaling strategies.” In: NeurIPS 2021 34 (2021), pp. 22614–22627. URL: https://openreview.net/forum?id=dsmxf7FKiaY.

[43] Barret Zoph and Quoc V. Le. “Neural Architecture Search with Reinforcement Learning.” In: *Proceedings of the 5th International Conference on Learning Representations*. ICLR 2017. OpenReview.net, 2017. DOI: 10.48550/arXiv.1611.01578.

[44] Mingxing Tan et al. "Mnasnet: Platform-aware neural architecture search for mobile." In: *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2019. 2019, pp. 2820–2828. DOI: 10.1109/CVPR.2019.00293.

[45] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. "Efficient Neural Architecture Search via Parameters Sharing." In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. ICML 2018. 2018, pp. 4095–4104. DOI: 10.48550/arXiv.1802.03268.

[46] Song Han, Jeff Pool, John Tran, and William 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*. NeurIPS 2015. 2015, pp. 1135–1143.

[47] Kohei Yamamoto and Kurato Maeno. *PCAS: Pruning Channels with Attention Statistics*. 2018. DOI: 10.48550/arXiv.1806.05382.

[48] G. Hacene, C. Lassance, V. Gripon, M. Courbariaux, and Y. Bengio. "Attention Based Pruning for Shift Networks." In: *Proceedings of the 25th International Conference on Pattern Recognition*. ICPR 2020. 2021, pp. 4054–4061. DOI: 10.1109/ICPR48806.2021.9412859.

[49] Ramchalam Kinattinkara Ramakrishnan, Eyyub Sari, and Vahid Partovi Nia. "Differentiable Mask for Pruning Convolutional and Recurrent Networks." In: *Proceedings of the 17th Conference on Computer and Robot Vision*. CRV 2020. 2020, pp. 222–229. DOI: 10.1109/CRV50864.2020.00037.

[50] Yang He, Yuhang Ding, Ping Liu, Linchao Zhu, Hanwang Zhang, and Yi Yang. "Learning Filter Pruning Criteria for Deep Convolutional Neural Networks Acceleration." In: *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2020. 2020, pp. 2006–2015. DOI: 10.1109/CVPR42600.2020.00208.

[51] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding." In: *4th International Conference on Learning Representations*. ICLR 2016. 2016. DOI: 10.48550/arXiv.1510.00149.

[52] Lucas Beyer, Xiaohua Zhai, Amélie Royer, Larisa Markeeva, Rohan Anil, and Alexander Kolesnikov. "Knowledge distillation: A good teacher is patient and consistent." In: *Proceedings of the 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2022. 2022. DOI: 10.48550/ARXIV.2106.05237.

[53] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. *A White Paper on Neural Network Quantization*. 2021. DOI: 10.48550/ARXIV.2106.08295.

[54] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. 2018. DOI: 10.48550/ARXIV.1806.08342.

[55] Yoshua Bengio, Nichloas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. arXiv:1308.3432. 2013.

[56] Geoffrey Hinton. *Neural networks for machine learning*. Coursera video lectures. 2012.

[57] Ruihao Gong et al. "Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks." In: *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV 2019)*. 2019, pp. 4851–4860. DOI: 10.1109/ICCV.2019.00495.

[58] Z. He and D. Fan. "Simultaneously Optimizing Weight and Quantizer of Ternary Neural Network Using Truncated Gaussian Approximation." In: *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2019. 2019, pp. 11430–11438. DOI: 10.1109/CVPR.2019.01170.

[59] Eunho Lee and Youngbae Hwang. "Layer-Wise Network Compression Using Gaussian Mixture Model." In: *Electronics* 10.1 (2021). DOI: 10.3390/electronics10010072.

[60] S. Vogel, R. B. Raghunath, A. Guntoro, K. Van Laerhoven, and G. Ascheid. "Bit-Shift-Based Accelerator for CNNs with Selectable Accuracy and Throughput." In: *Proceedings of the 22nd Euromicro Conference on Digital System Design*. DSD 2019. 2019, pp. 663–667. DOI: 10.1109/DSD.2019.00106.

[61] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H. Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. "Power-of-Two Quantization for Low Bitwidth and Hardware Compliant Neural Networks." In: *Proceedings of the 2022 tinyML Research Symposium*. tinyML 2022. 2022. DOI: 10.48550/ARXIV.2203.05025.

[62] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Training deep neural networks with low precision multiplications*. 2015. DOI: 10.48550/ARXIV.1412.7024.

[63] J. L. Holt and T. E. Baker. "Back propagation simulations using limited precision calculations." In: *Proceedings of the 1991 International Joint Conference on Neural Networks*. Vol. ii. IJCNN 1991. 1991, pp. 121–126. DOI: 10.1109/IJCNN.1991.155324.

[64] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the speed of neural networks on CPUs." In: *25th Conference on Neural Information Processing Systems, Deep Learning and Unsupervised Feature Learning Workshop*. NeurIPS 2011. 2011. URL: https://research.google/pubs/pub37631/.

[65] Alex Krizhevsky. *Convolutional Deep Belief Networks on CIFAR-10*. 2010. URL: https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf.

[66] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. "Accurate and Efficient 2-bit Quantized Neural Networks." In: *Proceedings of the 2nd Conference on Machine Learning and Systems*. Vol. 1. MLSYS 2019. 2019, pp. 348–359. URL: https://proceedings.mlsys.org/paper/2019/hash/006f52e9102a8d3be2fe5614f42ba989-Abstract.html.

[67] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. "Learned Step Size Quantization." In: *Proceedings of the 8th International Conference on Learning Representations*. ICLR 2020. 2020. DOI: 10.48550/arXiv.1902.08153.

[68] Miloš Nikolić et al. *BitPruning: Learning Bitlengths for Aggressive and Accurate Quantization*. 2020. DOI: 10.48550/ARXIV.2002.03090.

[69] Stefan Uhlich et al. "Mixed Precision DNNs: All you need is a good parametrization." In: *Proceedings of the 8th International Conference on Learning Representations*. ICLR 2020. 2020. DOI: 10.48550/arXiv.1905.11452.

[70] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks." In: *Proceedings of the 30th Conference on Neural Information Processing Systems*. NeurIPS 2016. 2016, pp. 1–9. URL: https://papers.nips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html.

[71] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." In: *Proceedings of the 14th European Conference on Computer Vision*. ECCV 2016. 2016, pp. 525–542. DOI: 10.1007/978-3-319-46493-0_32.

[72] Yichi Zhang, Zhiru Zhang, and Lukasz Lew. "PokeBNN: A Binary Pursuit of Lightweight Accuracy." In: *Proceedings of the 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR 2022. 2022. DOI: 10.48550/arXiv.2112.00133.

[73] N. J. Cotton, B. M. Wilamowski, and G. Dundar. "A Neural Network Implementation on an Inexpensive Eight Bit Microcontroller." In: *Proceedings of the 12th International Conference on Intelligent Engineering Systems*. INES 2008. 2008, pp. 109–114. DOI: 10.1109/INES.2008.4481278.

[74] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *Proceedings of the 27th International Conference on Machine Learning*. ICML 2010. 2010, pp. 807–814.

[75] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. *Hello Edge: Keyword Spotting on Microcontrollers*. 2018. DOI: 10.48550/arXiv.1711.07128.

[76] R. David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2020. DOI: 10.48550/arXiv.2010.08678.

[77] STMicroelectronics. *STM32Cube.AI*. URL: https://www.st.com/content/st_com/en/stm32-ann.html (visited on 03/19/2021).

[78] Google. *TensorFlow Lite for Microcontrollers supported operations*. URL: https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/micro_mutable_op_resolver.h (visited on 07/13/2022).

[79] François Chollet et al. *Keras*. 2015. URL: https://keras.io (visited on 07/13/2022).

[80] Google. *TensorFlow Lite 8-bit quantization specification*. URL: https://www.tensorflow.org/lite/performance/quantization_spec (visited on 03/19/2021).

[81] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In: *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2018)*. 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[82] STMicroelectronics. *X-CUBE-AI Documentation, Supported Deep Learning toolboxes and layers*.

[83] STMicroelectronics. *NanoEdge AI Studio*. URL: https://www.st.com/en/development-tools/nanoedgeaistudio.html (visited on 03/19/2021).

[84] CEA-List. *N2D2*. URL: https://github.com/CEA-LIST/N2D2 (visited on 07/19/2022).

[85] CEA-List. *N2D2 Post-training quantization*. URL: https://cea-list.github.io/N2D2-docs/quant/post.html (visited on 07/19/2022).

[86] CEA-List. *N2D2 Quantization-Aware Training*. URL: https://cea-list.github.io/N2D2-docs/quant/qat.html (visited on 07/19/2022).

[87] Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak. "LSQ+: Improving Low-Bit Quantization Through Learnable Offsets and Better Initialization." In: *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. CVPR 2020. 2020. URL: https://openaccess.thecvf.com/content_CVPRW_2020/html/w40/Bhalgat_LSQ_Improving_Low-Bit_Quantization_Through_Learnable_Offsets_and_Better_Initialization_CVPRW_2020_paper.html.

[88] Qing Jin, Linjie Yang, and Zhenyu Liao. *Towards Efficient Training for Neural Network Quantization*. 2019. arXiv: 1912.10207 [cs.CV].

[89] CEA-List. *Deep Green, Plateforme de deep learning ouverte et souveraine pour l'embarqué*. 2022. URL: https://anr.fr/fileadmin/aap/2022/france2030-ami-IA-deepgreen.pdf.

[90] Jianjia Ma. *A higher-level Neural Network library on Microcontrollers (NNoM)*. Version v0.4.2. Oct. 2020. DOI: 10.5281/zenodo.4158710. URL: https://doi.org/10.5281/zenodo.4158710.

[91] Raphael Zingg and Matthias Rosentha. "Artificial Intelligence on Microcontrollers." Embedded World 2020. 2020. URL: https://raw.githubusercontent.com/InES-HPMM/Artificial_Intelligence_on_Microcontrollers/master/Artificial_Intelligence_on_Microcontrollers.pdf.

[92] Usman Ali Butt. "On the deployment of Artificial Neural Networks (ANN) in low cost embedded systems." MA thesis. Politecnico di Torino, 2021. URL: https://webthesis.biblio.polito.it/19692/1/tesi.pdf.

[93] Jon Nordby. *emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices*. 2019. DOI: 10.5281/zenodo.2589394. URL: https://doi.org/10.5281/zenodo.2589394.

[94] Fouad Sakr, Francesco Bellotti, Riccardo Berta, and Alessandro De Gloria. "Machine Learning on Mainstream Microcontrollers." In: *Sensors* 20.9 (2020). DOI: 10.3390/s20092638.

[95] Tony Givargis. "Gravity: An Artificial Neural Network Compiler for Embedded Applications." In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. ASP-DAC 2021. 2021, pp. 715–721. DOI: 10.1145/3394885.3431514.

[96] 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." In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4403–4417.

[97] Apache TVM. *microTVM: TVM on bare-metal*. URL: https://tvm.apache.org/docs/topic/microtvm/index.html (visited on 07/19/2022).

[98] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI 2018. 2018, pp. 579–594.

[99] Renesas Electronics. *e-AI Solution*. URL: https://www.renesas.com/us/en/application/key-technology/artificial-intelligence/e-ai (visited on 07/20/2022).

[100] NXP Semiconductors. *eIQ ML Software Development Environment*. URL: https://www.nxp.com/design/software/development-software/eiq-ml-development-environment:EIQ (visited on 07/20/2022).

[101] tinyML Foundation. *tinyML Sponsors*. URL: https://www.tinyml.org/sponsors/ (visited on 07/20/2022).

[102] Dawon Kim and Yosoon Choi. "Applications of Smart Glasses in Applied Sciences: A Systematic Review." In: *Applied Sciences* 11.11 (2021). DOI: 10.3390/app11114956.

[103] Anna Syberfeldt, Oscar Danielsson, and Patrik Gustavsson. "Augmented Reality Smart Glasses in the Smart Factory: Product Evaluation Guidelines and Review of Available Products." In: *IEEE Access* 5 (2017), pp. 9118–9130. DOI: 10.1109/ACCESS.2017.2703952.

[104] Jingping Nie et al. "SPIDERS+: A light-weight, wireless, and low-cost glasses-based wearable platform for emotion sensing and bio-signal acquisition." In: *Pervasive and Mobile Computing* 75 (2021), p. 101424. DOI: https://doi.org/10.1016/j.pmcj.2021.101424.

[105] Alexis Arcaya-Jordan, Alain Pegatoquet, and Andrea Castagnetti. "Smart Connected Glasses for Drowsiness Detection: a System-Level Modeling Approach." In: *Proceedings of the 14th IEEE Sensors Applications Symposium*. SAS 2019. 2019, pp. 1–6. DOI: 10.1109/SAS.2019.8706022.

[106] Alexis Arcaya Jordan, Alain Pegatoquet, Andrea Castagnetti, Julien Raybaut, and Pierre Le Coz. "Deep Learning for Eye Blink Detection Implemented at the Edge." In: *IEEE Embedded Systems Letters* 13.3 (2021), pp. 130–133. DOI: 10.1109/LES.2020.3029313.

[107] Shinji Niwa, Mori Yuki, Tetsushi Noro, Shunsuke Shioya, and Kazutaka Inoue. "A Wearable Device for Traffic Safety - A Study on Estimating Drowsiness with Eyewear, JINS MEME." In: *SAE 2016 World Congress and Exhibition*. 2016. DOI: 10.4271/2016-01-0118.

[108] Justine Hellec, Frédéric Chorin, Andrea Castagnetti, and Serge S. Colson. "Sit-To-Stand Movement Evaluated Using an Inertial Measurement Unit Embedded in Smart Glasses–A Validation Study." In: *Sensors* 20.18 (2020). DOI: 10.3390/s20185019.

[109] Justine Hellec, Frédéric Chorin, Andrea Castagnetti, Olivier Guérin, and Serge S. Colson. "Smart Eyeglasses: A Valid and Reliable Device to Assess Spatiotemporal Parameters during Gait." In: *Sensors* 22.3 (2022). DOI: 10.3390/s22031196.

[110] Florenc Demrozi, Graziano Pravadelli, Azra Bihorac, and Parisa Rashidi. "Human Activity Recognition Using Inertial, Physiological and Environmental Sensors: A Comprehensive Survey." In: *IEEE Access* 8 (2020), pp. 210816–210836. DOI: 10.1109/ACCESS.2020.3037715.

[111] Djamila Romaissa Beddiar, Brahim Nini, Mohammad Sabokrou, and Abdenour Hadid. "Vision-based human activity recognition: a survey." In: *Multimedia Tools and Applications* 79 (2020), pp. 30509–30555. DOI: 10.1007/s11042-020-09004-3.

[112] D. Anguita, Alessandro Ghio, L. Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. "A Public Domain Dataset for Human Activity Recognition using Smartphones." In: *Proceedings of the 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. ESANN 2013. 2013.

[113] Federico Cruciani et al. "Comparing CNN and Human Crafted Features for Human Activity Recognition." In: *2019 IEEE SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*. 2019, pp. 960–967. DOI: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00190.

[114] Xiangyu Jiang, Yonggang Lu, Zhenyu Lu, and Huiyu Zhou. "Smartphone-Based Human Activity Recognition Using CNN in Frequency Domain." In: *Proceedings of the 2nd Asia-Pacific Web and Web-Age Information Management Joint International Conference on Web and Big Data*. APWeb-WAIM 2018. 2018, pp. 101–110. DOI: 10.1007/978-3-030-01298-4_10.

[115] Jorge-Luis Reyes-Ortiz, Luca Oneto, Alessandro Ghio, Albert Samá, Davide Anguita, and Xavier Parra. "Human Activity Recognition on Smartphones with Awareness of Basic Activities and Postural Transitions." In: *Proceedings of the 24th International Conference on Artificial Neural Networks and Machine Learning*. ICANN 2014. 2014, pp. 177–184. DOI: 10.1007/978-3-319-11179-7_23.

[116] Jorge-L. Reyes-Ortiz, Luca Oneto, Albert Samà, Xavier Parra, and Davide Anguita. "Transition-Aware Human Activity Recognition Using Smartphones." In: *Neurocomputing* 171 (2016), pp. 754–767. DOI: https://doi.org/10.1016/j.neucom.2015.07.085.

[117] Attila Reiss and Didier Stricker. "Introducing a New Benchmarked Dataset for Activity Monitoring." In: *Proceedings of the 16th International Symposium on Wearable Computers*. ISWC 2012. 2012, pp. 108–109. DOI: 10.1109/ISWC.2012.13.

[118] Oresti Baños, Miguel Damas, Héctor Pomares, Ignacio Rojas, Máté Attila Tóth, and Oliver Amft. “A benchmark dataset to evaluate sensor displacement in activity recognition.” In: *Proceedings of the 14th ACM Conference on Ubiquitous Computing*. UbiComp 2012. 2012, pp. 1026–1035. DOI: 10.1145/2370216.2370437.

[119] O. Banos and M. A. Toth. *Realistic Sensor Displacement Benchmark Dataset*. 2014. URL: https://archive.ics.uci.edu/ml/datasets/REALDISP+Activity+Recognition+Dataset.

[120] Daniela Micucci, Marco Mobilio, and Paolo Napoletano. “UniMiB SHAR: A Dataset for Human Activity Recognition Using Acceleration Data from Smartphones.” In: *Applied Sciences* 7.10 (2017). DOI: 10.3390/app7101101.

[121] Daniel Garcia-Gonzalez, Daniel Rivero, Enrique Fernandez-Blanco, and Miguel R. Luaces. “A Public Domain Dataset for Real-Life Human Activity Recognition Using Smartphone Sensors.” In: *Sensors* 20.8 (2020). DOI: 10.3390/s20082200.

[122] Daniel Roggen et al. “Collecting complex activity datasets in highly rich networked sensor environments.” In: *Proceedings of the 7th International Conference on Networked Sensing Systems (INSS)*. INSS 2010. 2010, pp. 233–240. DOI: 10.1109/INSS.2010.5573462.

[123] Gary M. Weiss, Kenichi Yoneda, and Thaier Hayajneh. “Smartphone and Smartwatch-Based Biometrics Using Activities of Daily Living.” In: *IEEE Access* 7 (2019), pp. 133190–133202. DOI: 10.1109/ACCESS.2019.2940729.

[124] Avgoustinos Filippoupolitis, William Oliff, Babak Takand, and George Loukas. “Location-Enhanced Activity Recognition in Indoor Environments Using Off the Shelf Smart Watch Technology and BLE Beacons.” In: *Sensors* 17.6 (2017). DOI: 10.3390/s17061230.

[125] Sébastien Faye, Nicolas Louveton, Sasan Jafarnejad, Roman Kryvchenko, and Thomas Engel. “An Open Dataset for Human Activity Analysis using Smart Devices.” 2017. URL: https://hal.archives-ouvertes.fr/hal-01586802.

[126] Dillam Díaz, Nicholas Yee, Christine Daum, Eleni Stroulia, and Lili Liu. “Activity Classification in Independent Living Environment with JINS MEME Eyewear.” In: *Proceedings of the 16th IEEE International Conference on Pervasive Computing and Communications*. PerCom 2018. 2018, pp. 1–9. DOI: 10.1109/PERCOM.2018.8444580.

[127] Joshua Ho and Chien-Min Wang. “User-Centric and Real-Time Activity Recognition Using Smart Glasses.” In: *Proceedings of the 11th International Conference on Green, Pervasive, and Cloud Computing*. GPC 2016. 2016, pp. 196–210. DOI: 10.1007/978-3-319-39077-2_13.

[128] Pierre Baldi. “Autoencoders, Unsupervised Learning, and Deep Architectures.” In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. Vol. 27. ICML 2011. 2012, pp. 37–49. URL: https://proceedings.mlr.press/v27/baldi12a.html.

[129] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. “Deep Convolutional AutoEncoder-based Lossy Image Compression.” In: *Proceedings of the 33rd Picture Coding Symposium*. PCS 2018. 2018, pp. 253–257. DOI: 10.1109/PCS.2018.8456308.

[130] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. “Extracting and Composing Robust Features with Denoising Autoencoders.” In: *Proceedings of the 25th International Conference on Machine Learning*. ICML 2008. 2008, pp. 1096–1103. DOI: 10.1145/1390156.1390294.

[131] Chong Zhou and Randy C. Paffenroth. “Anomaly Detection with Robust Deep Autoencoders.” In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD 2017. 2017, pp. 665–674. DOI: 10.1145/3097983.3098052.

[132] Lyes Khacef, Laurent Rodriguez, and Benoît Miramond. "Improving Self-Organizing Maps with Unsupervised Feature Extraction." In: *Proceedings of the 27th International Conference on Neural Information Processing*. ICONIP 2020. 2020, pp. 474–486. DOI: 10.1007/978-3-030-63833-7_40.

[133] Qun Liu and Supratik Mukhopadhyay. "Unsupervised learning using pretrained CNN and associative memory bank." In: *Proceedings of the 2018 International Joint Conference on Neural Networks*. IJCNN 2018. 2018, pp. 01–08. DOI: 10.1109/IJCNN.2018.8489408.

[134] Johannes C. Thiele, Olivier Bichler, and Antoine Dupret. "Event-Based, Timescale Invariant Unsupervised Online Deep Learning With STDP." In: *Frontiers in Computational Neuroscience* 12 (2018). DOI: 10.3389/fncom.2018.00046.

[135] Teuvo Kohonen. "The self-organizing map." In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. DOI: 10.1109/5.58325.

[136] Lyes Khacef, Benoît Miramond, Diego Barrientos, and Andres Upegui. "Self-organizing neurons: Toward brain-inspired unsupervised learning." In: *Proceedings of the 2019 IEEE International Joint Conference on Neural Networks (IJCNN 2019)*. 2019, pp. 1–9. DOI: 10.1109/IJCNN.2019.8852098.

[137] James MacQueen. "Some methods for classification and analysis of multivariate observations." In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 1967, pp. 281–297.

[138] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*. KDD 1996. 1996, pp. 226–231. URL: http://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf.

[139] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. "Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection." In: *ACM Transactions on Knowledge Discovery from Data* 10.1 (2015). DOI: 10.1145/2733381.

[140] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. "OPTICS: Ordering Points to Identify the Clustering Structure." In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD 1999. 1999, pp. 49–60. DOI: 10.1145/304182.304187.

[141] Hande Alemdar, Tim van Kasteren, and Cem Ersoy. "Using Active Learning to Allow Activity Recognition on a Large Scale." In: *Proceedings of the 2nd International Joint Conference on Ambient Intelligence*. AmI 2011. 2011, pp. 105–114. DOI: 10.1007/978-3-642-25167-2_12.

[142] Pengzhen Ren et al. "A survey of deep active learning." In: *ACM Computing Surveys* 54.9 (2021), pp. 1–40. DOI: 10.1145/3472291.

[143] Paola Ariza, Enrico Vicario, Emiro De la Hoz, Marlon Piñeres-Melo, Ana Oviedo Carrascal, and Fulvio Patara. "Unsupervised Human Activity Recognition Using the Clustering Approach: A Review." In: *Sensors* 20.9 (2020). DOI: 10.3390/s20092702.

[144] R. Nawaratne, D. Alahakoon, D. De Silva, and X. Yu. "HT-GSOM: Dynamic Self-organizing Map with Transience for Human Activity Recognition." In: *Proceedings of the 17th IEEE International Conference on Industrial Informatics*. INDIN 2019. 2019, pp. 270–273. DOI: 10.1109/INDIN41052.2019.8972260.

[145] Sayandeep Bhattacharjee, Swapnil Kishore, and Aleena Swetapadma. "A Comparative Study of Supervised Learning Techniques for Human Activity Monitoring Using Smart Sensors." In: *Proceedings of the 2nd International Conference on Advances in Electronics, Computers and Communications*. ICAECC 2018. 2018, pp. 1–4. DOI: 10.1109/ICAECC.2018.8479436.

[146] C.A Ronao and S Cho. “Deep convolutional neural networks for human activity recognition with smartphone sensors.” In: ICONIP 2015. 2015, pp. 46–53. DOI: 10.1007_978-3-319-26561-2_6-citation.

[147] Y. Li, D. Shi, B. Ding, and D. Liu. “Unsupervised feature learning for human activity recognition using smartphone sensors.” In: *Mining Intelligence and Knowledge Exploration* (2014), pp. 99–107. DOI: 10.1007/978-3-319-13817-6_11.

[148] S. Wan, L. Qi, X. Xu, and al. “Deep Learning Models for Real-time Human Activity Recognition with Smartphones.” In: *Mobile Networks and Applications* 25 (2 2019), pp. 743–755. DOI: 10.1007/s11036-019-01445-x.

[149] Nilay Tufek, Murat Yalcin, Mucahit Altintas, Fatma Kalaoglu, Yi Li, and Senem Kursun Bahadir. “Human Action Recognition Using Deep Learning Methods on Limited Sensory Data.” In: *IEEE Sensors Journal* 20.6 (2020), pp. 3101–3112. DOI: 10.1109/JSEN.2019.2956901.

[150] Duarte Folgado Patrícia Bota Joana Silva and Hugo Gamboa. “A Semi-Automatic Annotation Approach for Human Activity Recognition.” In: *Sensors* 19.3 (2019). DOI: 10.3390/s19030501.

[151] B. Romera-Paredes, M. S. H. Aung, and N. Bianchi-Berthouze. “A one-vs-one classifier ensemble with majority voting for activity recognition.” In: *Proceedings of the 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. ESANN 2013. 2013, pp. 443–448. URL: https://www.esann.org/sites/default/files/proceedings/legacy/es2013-124.pdf.

[152] M. Kästner, M. Strickert, and T. Villmann. “A sparse kernelized matrix learning vector quantization model for human activity recognition.” In: *Proceedings of the 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. ESANN 2013. 2013, pp. 449–454. URL: https://www.esann.org/sites/default/files/proceedings/legacy/es2013-123.pdf.

[153] A. Reiss, G. Hendeby, and D. Stricker. “A competitive approach for human activity recognition on smartphones.” In: *Proceedings of the 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. ESANN 2013. 2013, pp. 455–460. URL: https://www.esann.org/sites/default/files/proceedings/legacy/es2013-122.pdf.

[154] Charissa Ann Ronao and Sung-Bae Cho. “Human activity recognition using smartphone sensors with two-stage continuous hidden Markov models.” In: *Proceedings of the 10th International Conference on Natural Computation*. ICNC 2014, pp. 681–686. DOI: 10.1109/ICNC.2014.6975918.

[155] Felix Berkhahn, Richard Keys, Wajih Ouertani, Nikhil Shetty, and Dominik Geißler. “Augmenting Variational Autoencoders with Sparse Labels: A Unified Framework for Unsupervised, Semi-(un)supervised, and Supervised Learning.” In: (2019). DOI: 10.48550/ARXIV.1908.03015.

[156] Aaqib Saeed, Tanir Ozcelebi, and Johan Lukkien. “Multi-Task Self-Supervised Learning for Human Activity Detection.” In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. IMWUT 3.2 (2019). DOI: 10.1145/3328932.

[157] Setareh Rahimi Taghanaki, Michael J Rainbow, and Ali Etemad. “Self-Supervised Human Activity Recognition by Learning to Predict Cross-Dimensional Motion.” In: *Proceedings of the 2021 International Symposium on Wearable Computers*. ISWC 2021. 2021, pp. 23–27. DOI: 10.1145/3460421.3480417.

[158] Steven C.H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. “Online learning: A comprehensive survey.” In: *Neurocomputing* 459 (2021), pp. 249–289. DOI: https://doi.org/10.1016/j.neucom.2021.04.112.

[159] Vincenzo Lomonaco and Davide Maltoni. “CORe50: a New Dataset and Benchmark for Continuous Object Recognition.” In: *Proceedings of the 1st Annual Conference on Robot Learning*. Vol. 78. CoRL 2017. 2017, pp. 17–26. URL: https://proceedings.mlr.press/v78/lomonaco17a.html.

[160] M. De Lange et al. “A Continual Learning Survey: Defying Forgetting in Classification Tasks.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.07 (2022), pp. 3366–3385. DOI: 10.1109/TPAMI.2021.3057446.

[161] Anthony Robins. “Catastrophic Forgetting, Rehearsal and Pseudorehearsal.” In: *Connection Science* 7.2 (1995), pp. 123–146. DOI: 10.1080/09540099550039318.

[162] Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, Francesco Conti, and Luca Benini. “A TinyML Platform for On-Device Continual Learning With Quantized Latent Replays.” In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (2021), pp. 789–802. DOI: 10.1109/JETCAS.2021.3121554.

[163] Baosheng Zhang, Yuchen Guo, Yipeng Li, Yuwei He, Haoqian Wang, and Qionghai Dai. “Memory Recall: A Simple Neural Network Training Framework Against Catastrophic Forgetting.” In: *IEEE Transactions on Neural Networks and Learning Systems* 33.5 (2022), pp. 2010–2022. DOI: 10.1109/TNNLS.2021.3099700.

[164] Bernard Ans and Stéphane Rousset. “Avoiding catastrophic forgetting by coupling two reverberating neural networks.” In: *Comptes Rendus de l’Académie des Sciences - Series III - Sciences de la Vie* 320.12 (1997), pp. 989–997. DOI: https://doi.org/10.1016/S0764-4469(97)82472-9.

[165] N. P. Rougier and Y. Boniface. “Dynamic Self-Organising Map.” In: *Neurocomputing* 74 (2011), pp. 1840–1847. DOI: 10.1016/j.neucom.2010.06.034.

[166] Sebastian Farquhar and Yarin Gal. “Towards Robust Evaluations of Continual Learning.” In: *35th International Conference on Machine Learning, Lifelong Learning: A Reinforcement Learning Approach Workshop*. ICML 2018. 2018. DOI: 10.48550/arXiv.1805.09733.

[167] Ian J. Goodfellow, Mehdi Mirza, Xia Da, Aaron C. Courville, and Yoshua Bengio. “An Empirical Investigation of Catastrophic Forgeting in Gradient-Based Neural Networks.” In: *Proceedings of the 2nd International Conference on Learning Representations*. ICLR 2014. 2014. DOI: 10.48550/arXiv.1312.6211.

[168] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. “The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks.” In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: 10.1109/TNNLS.2020.3044364.

[169] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. “TinyOL: TinyML with Online-Learning on Microcontrollers.” In: *Proceedings of the 2021 International Joint Conference on Neural Networks*. IJCNN 2021. 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533927.

[170] Hiroki Matsutani, Mineto Tsukada, and Masaaki Kondo. “On-Device Learning: A Neural Network Based Field-Trainable Edge AI.” In: (2022). DOI: 10.48550/ARXIV.2203.01077.

[171] Nan-ying Liang, Guang-bin Huang, P. Saratchandran, and N. Sundararajan. “A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks.” In: *IEEE Transactions on Neural Networks* 17.6 (2006), pp. 1411–1423. DOI: 10.1109/TNN.2006.880583.

[172] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. "On-Device Training Under 256KB Memory." In: (2022). DOI: 10.48550/ARXIV.2206.15472.

[173] Diogo Costa, Miguel Costa, and Sandro Pinto. "Train Me If You Can: Decentralized Learning on the Deep Edge." In: *Applied Sciences* 12.9 (2022). DOI: 10.3390/app12094653.

[174] Nil Llisterri Giménez, Marc Monfort Grau, Roger Pueyo Centelles, and Felix Freitag. "On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning." In: *Electronics* 11.4 (2022). DOI: 10.3390/electronics11040573.

[175] Bharath Sudharsan, Piyush Yadav, John G. Breslin, and Muhammad Intizar Ali. "Train++: An Incremental ML Model Training Algorithm to Create Self-Learning IoT Devices." In: *Proceedings of 2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation*. SmartWorld/SCALCOM/UIC/ATC/IOP/SCI 2021. 2021, pp. 97–106. DOI: 10.1109/SWC50871.2021.00023.

[176] Jon J. Pimentel, Brent Bohnenstiehl, and Brevan M. Baas. "Hybrid Hardware/Software Floating-Point Implementations for Optimized Area and Throughput Tradeoffs." In: *IEEE Transactions on Very Large Scale Integration (VLSI)* 25 (2017), pp. 100–113. DOI: 10.1109/TVLSI.2016.2580142.

[177] Arm Ltd. "Cortex-M4 instructions." In: *Cortex-M4 Technical Reference Manual*. 2010.

[178] Arm Ltd. "FPU instruction set." In: *Cortex-M4 Technical Reference Manual*. 2010.

[179] Pierre-Emmanuel Novac, Alain Pegatoquet, and Benoît Miramond. *MicroAI, a software framework for end-to-end deep neural networks training, quantization and deployment onto embedded devices*. 10.5281/zenodo.5507397. Version 1.0. 2021.

[180] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019. 2019, pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[181] Paulius Micikevicius et al. "Mixed Precision Training." In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. 2018.

[182] Arm Ltd. "4.7.9 Q-Format." In: *ARM Developer Suite AXD and armsd Debuggers Guide*. ARM DUI 0066D Version 1.2. 2001.

[183] Nenad Markuš. *Fusing batch normalization and convolution in runtime*. URL: https://nenadmarkus.com/p/fusing-batchnorm-and-conv/ (visited on 10/07/2022).

[184] Pierre-Emmanuel Novac, Alain Pegatoquet, and Benoît Miramond. "Déclaration d'Invention DI-14851-01 du logiciel MicroAI." DI-14851-01 (France). Apr. 2021. URL: https://hal.archives-ouvertes.fr/hal-03595177.

[185] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoît Miramond, and Vincent Gripon. "Quantization and Deployment of Deep Neural Networks on Microcontrollers." In: *Sensors* 21.9 (2021). DOI: 10.3390/s21092984.

[186] *Tom's Obvious Minimal Language (TOML)*. https://toml.io/. 2021.

[187] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. "torch.fx: Practical Program Capture and Transformation for Deep Learning in Python." In: *Proceedings of the 5th Conference on Machine Learning and Systems (MLSys 2022)*. Vol. 4. 2022, pp. 638–651. URL: https://proceedings.mlsys.org/paper/2022/hash/ca46c1b9512a7a8315fa3c5a946e8265-Abstract.html.

[188] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*. Vol. 37. 2015, pp. 448–456. DOI: 10.5555/3045118.3045167.

[189] *Jinja2*. https://palletsprojects.com/p/jinja/. 2021.

[190] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. "mixup: Beyond Empirical Risk Minimization." In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. 2018. URL: https://openreview.net/forum?id=r1Ddp1-Rb.

[191] Lyes Khacef, Laurent Rodriguez, and Benoît Miramond. *Written and Spoken Digits Database for Multimodal Learning*. 10.5281/zenodo.3515935. 2019.

[192] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. arXiv:1804.03209. 2018.

[193] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. "The German Traffic Sign Recognition Benchmark: A multi-class classification competition." In: *Proceedings of the 2011 International Joint Conference on Neural Networks (IJCNN 2011)*. 2011, pp. 1453–1460. DOI: 10.1109/IJCNN.2011.6033395.

[194] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. "CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices." In: *IEEE Transactions on Circuits and Systems (CAS) II: Express Briefs* 67 (2020), pp. 871–875. DOI: 10.1109/TCSII.2020.2983648.

[195] Eunhyeok Park et al. "Big/little deep neural network for ultra low power inference." In: *Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS 2015)*. 2015, pp. 124–132. DOI: 10.1109/CODESISSS.2015.7331375.

[196] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured Pruning of Deep Convolutional Neural Networks." In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–18. DOI: 10.1145/3005348.

[197] Pierre-Emmanuel Novac, Alain Pegatoquet, Benoît Miramond, and Christophe Caquineau. *UCA-EHAR, a dataset for human activity recognition using smart glasses*. 10.5281/zenodo.5659336. Version 1.0. 2021.

[198] Pierre-Emmanuel Novac, Alain Pegatoquet, Benoît Miramond, and Christophe Caquineau. "UCA-EHAR: A Dataset for Human Activity Recognition with Embedded AI on Smart Glasses." In: *Applied Sciences* 12.8 (2022). DOI: 10.3390/app12083849.

[199] *Dear ImGui*. https://github.com/ocornut/imgui. 2021.

[200] Chompinha. *Training process of Self-organizing map on* 8 × 8 *rectangular grid two-dimensional data set, using Euclidean distance*. CC BY-SA 4.0. URL: https://commons.wikimedia.org/wiki/File:TrainSOM.gif.

[201] Lyes Khacef, Vincent Gripon, and Benoît Miramond. "GPU-based Self-Organizing-Maps for Post-Labeled Few-Shot Unsupervised Learning." In: *Proceedings of the 27th International Conference on Neural Information Processing (ICONIP 2020)*. Vol. 12533. 2020, pp. 404–416. DOI: 10.1007/978-3-030-63833-7_34.

[202] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of Machine Learning Research* 9.11 (2008), pp. 2579–2605. URL: https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf.

[203] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. "Very deep convolutional neural networks for raw waveforms." In: *Proceedings of the 42nd IEEE International Conference on Acoustics, Speech and Signal Processing*. ICASSP 2017. 2017, pp. 421–425. DOI: 10.1109/ICASSP.2017.7952190.

[204] ABDK Consulting. *ABDK Libraries for Solidity*. URL: https://github.com/abdk-consulting/abdk-libraries-solidity (visited on 09/16/2022).

[205] *libfixmath*. URL: https://code.google.com/archive/p/libfixmath/ (visited on 10/06/2022).

[206] Johannes Partzsch et al. “A fixed point exponential function accelerator for a neuromorphic many-core system.” In: *Proceedings of the 2017 IEEE International Symposium on Circuits and Systems*. ISCAS 2017. 2017. DOI: 10.1109/ISCAS.2017.8050528.

[207] Imourane Abdoulaye. “Minimisation de la consommation énergétique dans un réseau de capteurs sans fil en utilisant une application basée sur des réseaux de neurones.” Stage Master 2 ESTEL. Université Côte d’Azur, 2022. URL: https://hal.archives-ouvertes.fr/hal-03778784.

[208] Alexis Arcaya Jordan. “Energy consumption optimization on smart connected glasses under QoS constraints.” PhD thesis. Université Côte d’Azur, 2021. URL: https://tel.archives-ouvertes.fr/tel-03336598.

[209] Wei Fang et al. *SpikingJelly*. URL: https://github.com/fangwei123456/spikingjelly (visited on 08/19/2022).

[210] Loïc Cordone, Benoît Miramond, and Philippe Thierion. “Object Detection with Spiking Neural Networks on Automotive Event Data.” In: *Proceedings of the 2022 International Joint Conference on Neural Networks*. IJCNN 2022. 2022. DOI: 10.1109/IJCNN55064.2022.9892618.

[211] Nassim Abderrahmane, Benoît Miramond, Erwann Kervennic, and Adrien Girard. “SPLEAT: SPiking Low-power Event-based ArchiTecture for in-orbit processing of satellite imagery.” In: *Proceedings of the 2022 International Joint Conference on Neural Networks*. IJCNN 2022. 2022. DOI: 10.1109/IJCNN55064.2022.9892277.

[212] Lyes Khacef, Bernard Girau, Nicolas Rougier, Andres Upegui, and Benoît Miramond. “Neuromorphic hardware as a self-organizing computing system.” In: *2018 IEEE World Congress on Computational Intelligence, Neuromorphic Hardware In Practice and Use Workshop*. WCCI 2018. 2018. DOI: 10.48550/ARXIV.1810.12640.

[213] Claudio Sousa. “A parallel SCALP simulator that focus on communication accuracy.” Bachelor’s Thesis. hepia, 2019. URL: https://sitehepia.hesge.ch/diplome/ITI/2019/documents/Sousa-478.

[214] Pierre-Emmanuel Novac. *Digital electronic design of a neuromorphic architecture for multimodal associations*. Internship report. Université Côte d’Azur, 2019. URL: https://hal.archives-ouvertes.fr/hal-02288753.

[215] *torchvision*. URL: https://github.com/pytorch/vision (visited on 09/09/2022).