UNIVERSITÀ
DI TRENTO
Dipartimento di
Ingegneria Industriale

Master's Degree in
Mechatronics Engineering

FINAL DISSERTATION

Comparison of incremental online learning algorithms for gesture and visual smart sensors

Supervisor
Davide Brunelli

Co-Supervisor
Andrea Albanese

Student
Alessandro Avi 214579

Academic Year 2020/2021

"Frase"

Mario Rossi

Contents

Introduction **1**

1 Related Works **5**
1.1 Machine learning in general 5
1.2 Cloud vs edge inference 5
1.3 Machine Learning on MCU 5
1.4 Continual-on line learning 7
1.5 Pruning and quantization 9

2 Hardware **10**
2.1 Gesture recognition hardware 10
2.2 Image classification hardware 12
2.3 Machine learning support 13

3 System implementation **15**
3.1 Basic pipeline of the CL system developed 15
3.2 Algorithms implemented 18
3.3 Gesture recognition application 26
3.4 Image classification application 26

4 Experimental setup **28**
4.1 Dataset collection 28
4.1.1 Accelerometer dataset 29
4.1.2 Digits recognition dataset 30
4.2 Frozen model training and evaluation 32
4.2.1 Gesture recognition model 32
4.2.2 Image classification model 34
4.3 STM F401-RE setup for gesture recognition 37

4.4 OpenMV setup for image classification 39

5 Experimental results **43**
5.1 Experiment A: Gesture recognition 43
5.2 Experiment B: Image classification 43

6 Conclusion **53**

List of Figures

2.1 PLACEHOLDER - da cambiare 11
2.2 PLACEHOLDER - da cambiare 12
2.3 PLACEHOLDER - da cambiare 14

3.1 Place holder - OL 21
3.2 Place holder - OLV2 22
3.3 Place holder - LWF 25
3.4 Place holder - CWR 27

4.1 Motion of the accleerometer (NON SI VEDONO LE FRECCE ARANCIO) 30
4.2 Stankey diagram showing how the letter dataset is divided . . 31
4.3 Example of images of digits in the MNIST dataset 31
4.4 Stankey diagram showing how the MNIST dataset is divided . 32
4.5 Letter recognition model: basic structure and its separation in frozen model and OL classification layer 33
4.6 Letter recognition training: on the left the variation of loss and validation; on the right the accuracy for each class 34
4.7 Image classification model: basic structure and its separation in frozen model and OL classification layer 35
4.8 PLACEHOLDER 36
4.9 PLACEHOLDER 39
4.10 PLACEHOLDER 41
4.11 OpenMV camera pointing to a screen while in different states of the training. Top left is *idle* mode, top right is *snap* mode, bottom left is *elab* mode, bottom right is *trai* mode 42

4.12 Point of view of the OpenMV camera. On the left the original image taken with no compression and no elaboration, in the center an image taken in *snap* mode, on the right an image taken in *elab* mode. 42

5.1 PLACEHOLDER 44
5.2 PLACEHOLDER 44
5.3 PLACEHOLDER 45
5.4 PLACEHOLDER 45
5.5 PLACEHOLDER 46
5.6 PLACEHOLDER 47
5.7 PLACEHOLDER 49
5.8 PLACEHOLDER 49
5.9 PLACEHOLDER 50
5.10 PLACEHOLDER 50
5.11 PLACEHOLDER 51
5.12 PLACEHOLDER 51
5.13 PLACEHOLDER 52

List of Tables

2.1 PLACEHOLDER - da cambiare 11
2.2 OpenMV H7+ specifications 13

Introduction

Machine learning (ML) applications on small devices, known as TinyML, is becoming more and more popular. The usage of this type of technology on micro controllers (MCU) is becoming more and more indispensable and helpful in several fields such as industrial applications, agricultural automation, autonomous driving, and human-machine interaction. One of the main fields in which TinyML is well suited is Internet of Things (IoT). Here, machine learning is applied on small devices and it can be exploited to revolutionize the basics of IoT networks.
The ability of embedded systems to perform high-level and smart data elaboration makes it possible for the IoT pipeline to change from cloud computing to edge computing. This transformation comes with great benefits and additional challenges. First of all, the traffic on IoT networks is drastically reduced. In fact, by performing inferences and predictions directly on the edge the raw data gets compressed into smaller sequences that are dense of information, reducing the quantity of data moving in the IoT networks. This allows to diminish the energy consumption dedicated to the entire system as well as the system responsiveness and efficiency. Edge computing lowers the traffic in IoT systems but also reduces the computational weight in cloud servers. This results in reduced times of computation and communication between edge and cloud which, if combined with the ability of the MCU to perform autonomous decision, reduces the latency of real time applications improving the overall experience. Moreover, IoT network privacy issues can be addressed by reducing transmitted data, and consequently reducing the possibility to have unwanted interceptions. At last the usage of ML on small devices allows to better customize the device, and make the devices better suited for specific jobs.
Of course, the application of such a technology comes with a cost, which is the increased complexity and higher amount of vulnerabilities. Thus, it is

necessary to set up robust systems that are able to ensure the system security (due to the high number of vulnerable nodes), and high performances, no matter the limitation of the device. It is in fact known that the main downsides of embedded systems and small MCUs are their limited hardware, small memories, and low-capacity batteries. Another important aspect concerning TinyML is the training and deployment of the model, which is typically performed on a powerful device and later loaded on the MCU using compression strategies. The main challenge is how the compression of big and well performing models is performed, especially because the model needs to maintain high accuracy with a low memory footprint. The creation of efficient and optimized compression strategies has been one of the main focus of recent research in the TinyML.
Another relevant challenge for the application of ML in IoT systems comes directly from the environment in which IoT smart nodes are deployed. Depending on the specific application, it is usually the case that the context in which an IoT device works is not characterized by a static behaviour. Meaning that the phenomenon to be monitored is able to change or evolve over time, thus data recorded can consequently evolve and change its main features. This can make difficult using ML models because they only perform inference and lack the ability to adapt to changing scenarios. It is clear how devices, set up in this way, are vulnerable to the context drift aforementioned. By training ML models for a specific context and later deploying them in the real world, it is expected a drop in accuracy which can make the application itself not reliable. It is then obvious how an application of simple ML inference on such environments is not the best solution. To contrast this issue, it is necessary to implement the so called Continual Learning (CL) algorithms. CL is a machine learning approach that allows ML models to perform training in real time and continually keep up-to-date the model weights. The implementation of this method comes with new challenges and limitations which are mainly related to memory management and strategies for the implementation of real time training which also keep in consideration the optimization of resources.

Continual learning methods lead to a real time training based on the data incoming. This allows the model to change and fine tune its weights and structure to better contrast the context drift. An additional feature that can be easily added to CL is the ability to recognize never seen classes. This, if paired with the model's ability to extend its structure, allows to create

a flexible model that is able to allocate new weights and biases for better predictions. An important problem that tackles basic applications of CL is catastrophic forgetting. Catastrophic forgetting is a phenomenon that occurs when model trained in real time overfits new data. This makes the knowledge related to past tasks be replaced by new knowledge, thus forgetting the initial scenario which leads to a reduction of the model performances over time. This aspect can be reduced by applying preventive mechanism inside the back propagation that control the parameters update.
The implementation of CL in industrial applications is not a new topic in the research world, but its implementation on tiny devices is just started to become more and more popular. One common application is CL in industrial scenarios, mainly for monitoring purposes on heavy machines. The main contributions of this study concern the application of CL in two different applications. The objective is to understand if CL is a feasible solution for TinyML and if its use is actually effective for the generation of autonomous and self adapting models. In this study, a light framework that is easy to connect to a pre trained classification model was developed. The system substitutes the last layer and continually performs updates on weights and biases, and also extends its shape for flexible adaptation to new classes. The system is able to use different state-of-the-art strategies that are tested and compared in two experiments with the aim of understanding if it is possible to: i) maintain or improve the accuracy of the model; ii)contrast catastrophic forgetting; iii) digest and learn classes of never seen data. Both experiments concern the application of ML for the classification of data coming from different sensors.
The first application regards the analysis of accelerometer data. In this experiment the user holds the accelerometer sensor in its hand and records a time series of accelerations while drawing letters in the air. The idea is to apply ML to classify the data and recognize the letters written. The model created is initially trained for the recognition of the pattern that characterize the five vowels. Later CL is applied to the experiment and the model is exposed to new data representing three new consonants. The aim of the experiment is to let the ML model learn new patterns by performing a real time training. The experiment can be considered a simplification of a real world applications, but it is a clear example of how a CL model can behave in these scenarios. This application can be extended in a real-life scenario such as the monitoring of vibration patterns of heavy industrial machinery.
The second application concerns the experimentation of CL on a CNN model

applied on an OpenMV camera for the visual recognition of digits from the MNIST dataset. The idea consists of initially train the model to recognize only the digits from 0 to 5 and later use the CL framework developed for applying a real time training on the remaining digits. This second experiment can be extended to applications where a camera is used for a visual control of defects on products in a production pipeline.
The work carried out in this study shows that the application of CL on tiny devices is possible. Even though the CL strategies are applied only on the last layer the results are satisfying and in both examples all the classes were correctly digested by the model. These tests show that a model equipped with a CL system is able to expand its knowledge and learn more classes, specifically 3 for the letters example and 4 for the digits example. The devices are able to maintain a reasonable accuracy at the end of the trainings that drop from the original frozen model accuracy by only 10.7%. The study performed is a good example that shows the capabilities of these tiny devices. It proves that machine learning applied on MCUs is a technology that has a huge potential and deserves more attention. CL can lead to smarter, more efficient, better performing systems in the IoT field and in industrial applications.

UNA VOLTA DEFINITA UNA STRUTTURA CHIARA E PRECISA PER IL RESTO DELLA TESI SCRIVERE 4/5 RIGHE PER OGNI CAPITOLO/SEZIONE

Chapter 1

Related Works

In this chapter some information about the application of machine learning on tiny devices are given. Then a brief introduction to continual learnin is done with a following explanation of the most relevant state-of-the-art studies regarding continual learning in TinyML world.

1.1 Machine learning in general

1.2 Cloud vs edge inference

1.3 Machine Learning on MCU

TinyML is a fast growing research area that aims at applying ML on limited devices like micro controllers. This technology has found a rapid grow in the last years especially thanks to the potential demonstrated by its application in several fields like industrial application, agricultural automation, human-computer interactions, autonomous driving. The use of TinyML in all these fields allows to introduce the concept of edge computing, where computations are brought closer to the origin of the data, to devices that up until now have been used only as collector and transmitters of data. Edge computing bring to lots of advantages like low-latency , better privacy, security and reliability to the network end-user. It allows to move the computations from the cloud to the device itself, which brings to higher throughput and improved responsiveness in applications. Speaking about responsiveness, which is a huge deal for real time applications, edge computing permits to decrease the

network traffic. This feature comes from the fact that the use of machine learning directly on the device allows to lift a big portion of computation weight given to the central server, thus reducing the amount of data that has to be exchanged on the network, which is also compressed in size and dense in information.
One of the main fields in which TinyML is particularly fit and can be exploited with all its potential is for sure the world of IoT. Here the application of edge computing brings to lots of advantages already described before. Of course the implementation of such systems comes with a cost, which in this case is related to the increased complexity on which the MCU works and its limited resources. Machine learning is a field of computer science that is known to be energy and resource demanding. Using such a technology on these tiny devices is a real challenge which already has seen interesting applications and improvements in the research world. The main characteristic of MCU is for sure their limited dimensions and power consumption. This is usually a nice feature that allows to develop small systems that can live for very long period of times in harsh environments without the need of maintenance. Lot of focus has also given to the implementation of energy harvesting systems that, not only use very low quantities of energy but also are able to extract energy from the environment in which they live. Limited dimensions has also drawbacks, which are limited memories and limited computational power, all features that do not really match with the application of machine learning. In the last decade (??) the research around TinyML revolved around the implementation of efficient framework from both point of view of memory use and power consumption. Some well known systems for deployment of ML models on tiny devices developed by big companies are Tensorflow Lite **??**, STM32 CUBE AI **??**, PyTorch mobile **??**. All these frameworks are used for training a model on a powerful system and later load a compressed version of the model on the MCU. The main concerns is the compression of the model in such a way that it doesn't drop in accuracy even with reduced weights or quatnized values. The second aspect that concerns the use of these tools is the inferenfce performed on the device. Being that procedure computationally heavy it's necessary to be able to optimize the computations. Some research studies focused soecifically on this.

The main challenge of TinyML is for sure the successful application of ML on such resource constrained systems. These are in fact designed to be deployed in difficult to reach places and for running for very long times. This implies that the devices should be battery power or equipped with energy

harvesting hardware and their power consumption should be limited and optimized. Other limitations concern the limited computational power, which is directly connected to the CPU frequency and the battery management and the available memory. The latter is a very important topic for TinyML. It's in fact known that the application of ML on any type of device requires the usage of great amounts of memory, it's then a big challenge to be able to deploy these systems with very limited memories.
The application of ML on MCUs, mobile devices or in general on the edge of IoT systems it's a great advantage that can bring to some improvmenets. The key advantages are:

- privacy: by having the data directly processed on the node there is no change of violating the privacy policies since the possibility of interception is totally nulled
- latency: by elaborating data directly on the edge the work load of processing that should be performed by the cloud is limited and so is the transmission of the data itself. This brings to limited time delays and allows the device to perform decisions in real time, improving the performances of real time applications.
- energy efficiency: the transmission of huge quantities of data from the edge to the cloud takes a big portion of the energy consumption of an IoT system. Even if the application of NN is energy intensive it is an order of magnitude less, thus an improvement.

1.4 Continual-on line learning

Until recent times the application of ML on MCUs has always been focused on the creation of intelligent small system that maintain good performance with reasonable consumption, limited time of inference and long lifetimes. A major negative aspect of the TinyML solutions is their focus on the inference of streams of data. Which almost always requires the usage of powerful machines for the training of NN models that are later deployed on the MCU. This results in the creation of a static network which is not able to adapt to the data and adjust to different scenarios. The solution to this problem is the creation of a Continual Learning system.
CL systems are a variation of the tipical pipeline of ML. The main focus of

CL systems is to be able to continuously update the model in order to adapt its structure and parameters to overcome context drift, be able to recognize appearance of new patterns and to avoid catastrophic forgetting. The latter is a problem that is directly introduced by the nature of the paradigm itself. By having a model that is continuously updated with a feedback loop that is directly dependent on the current erross i'ts clear how it's immediate to update the model in such a way that the old tasks are forgetten for the sake of learning the new ones. This could be seen also as a over fitting of the model on the new tasks and if oc course to be avoided. Different algorithms have different ways for contrasting this phenomenon.
In todays literature several CL algoprithms and strategies have been already proposed. A well organized summary is proposed in **lesort2020continual**, where the most relevant methods are briefly classified in 4 categories, originally proposed by [**maltoni2019continuous**].

- Architectural: these algorithms are based on the usage of particular types of structures and architectures. Some common methods are weight-freezing, layer activation or dual-memories-models that try to imitate logn term memory and short term memory.

- Regularization: this group contains all those approaches that base their ability to retain past memories on the application of particular loss functions. In these loss functions usually a term is added with the aim of performing a feedback that considers both the old knowledge and tries to learn the new data.

- Rehearsal strategies: in these strategies past informations are periodically revisited by the model. This is done for strengthening the old knowledge and connections. Notice that this methods is not well suited for application on MCUs mainly because of the restricted memories.

- Generative Replay: this methods implement similar strategies of the rehearsal. This time the data that is repeated in the models is not actually old data saved in the memory but it's actually data artificially generate by the model itself.

The type of strategies that better suits an application on MCU are for sure the regularization methods and the architectural methods. Both these groups require little to no extra computation with respect to a simple ML

application, thus their strength is intrinsic in the update rules adopted. Some of the most important methods from the state of the art are, LWF, PNN, CWR, EWC, SI.

1.5 Pruning and quantization

Chapter 2

Hardware

In this chapter the hardware used to carry out the experiments is described. The application of ML on MCU does not require specific types of hardware. In today's market lots of off-the-shelf microcontrollers are already capable of performing this type of computations, so it is quite easy to select a suitable device for this application. Machine learning on microcontrollers is known to be resource heavy, especially from the point of view of memories and computational power. Most big companies that propose their product have different families of devices that are designed for different purposes. In this study, for both applications, devices based on ST microcontrollers have been used. The gesture recognition application uses an STM32 Nucleo F401-RE, a well performing and easy to use development board. The image classification application uses an OpenMV camera, a device equipped with a camera sensor that uses an STM32 H7 MCU and is programmable in MicroPython. Figure 2.1 shows both devices: on the left the Nucleo F401-RE; on the right the OpenMV camera. The reasons that brought to the selection of these two devices depends mainly on the on the quick availability and ease of use for the Nucleo development board and the uniqueness of characteristics for the OpenMV camera.

2.1 Gesture recognition hardware

The experiment is carried out with a Nucleo STM32 F401-RE. This type of device is an easy to use development board that can be programmed in C or C++. It fully supports the extension pack STM-CUBE-AI [**stm˙cube˙ai**]

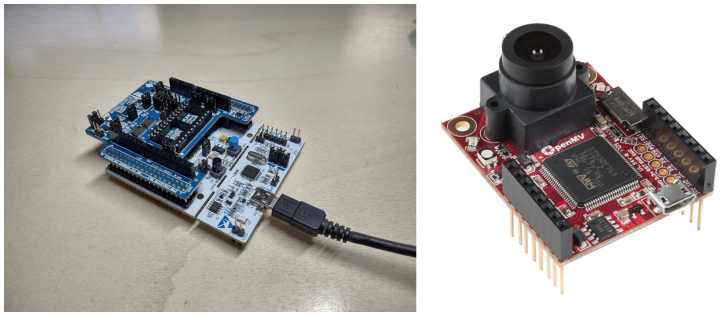

Figure 2.1: PLACEHOLDER - da cambiare

Processor	ARM 32-bit Cortex-M4 CPU 84 MHz
Memory	SRAM: 96 kB Flash: 512 kB
Phisical attributes	Weight: ??g Length, Width, Height: ??x??x3??mm
Peripherals	50 GPIOs, SPI, UART, I2C, DAC/ADC, PWM, Timers,

Table 2.1: PLACEHOLDER - da cambiare

developed by STM used for loading and compressing models and performing machine learning inference in real time.
The main features of the development board are summarized in Table **??**.

The application for gesture recognition is based on the analysis of times-series arrays. The time series represent the accelerations recorded while a sensor is moved around by a user that writes letters in the air. To acquire that kind of data, the Nucleo needs to be equipped with an accelerometer sensor. For doing that, it has been decided to use the Nucleo shield IKS01A2 [**shield˙web˙page**], which is a device that can be mounted easily on the board simply by aligning the GPIO pins. The shield is equipped with a 3D accelerometer, a pressure sensor and a capacitive digital humidity and temperature sensor. The shield communicates with the Nucleo through I2C protocol and it is fully supported by additional STM libraries that make its use quick and easy. Figure 2.2 shows the two devices separately on the left

and mounted on the right.

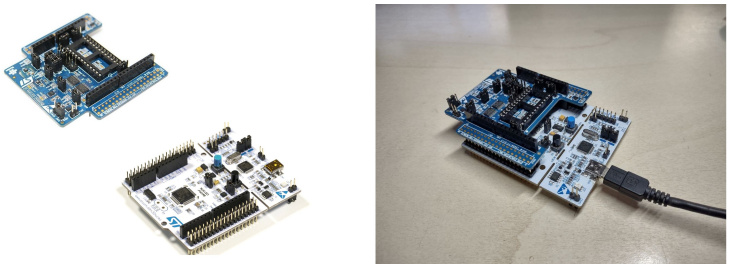

Figure 2.2: PLACEHOLDER - da cambiare

2.2 Image classification hardware

For the second experiment an OpenMV cam H7 plus [**abdelkader2017openmv**] [**openmv˙web˙page**] is used. This device is an affordable and expandable small board equipped with an STM MCU and a camera sensor with interchangeable lens. The OpenMV camera started as a project [**openmv˙project**] back in 2013 due to a lack of affordable, small, powerful and easy to use breakout board with cameras. The idea then developed into a kickstarter that gained popularity until reaching what is today, an established product that aims at becoming the standard device for machine vision applications. The device mounts the camera sensor OV5640 and an STM32 H7 MCU and it can be programmed easily with MicroPython through the dedicated OpenMV IDE. The board is equipped with 16 GPIOs thanks to which it can interact with external devices like servo motors, sensors or other MCUs. The GPIOs can be exploited for controlling robots, drones or for machine learning purposes. Some examples of successful applications of OpenMV cameras in robotic systems are: the design of a control system for rolling a ball [**zhou2019design**] and the design of a tracking system [**wei2020design**]. The camera comes also with the possibility to mount different lenses, sensors and additional modular components. From the official website a complete list is available, some examples are: IR lenses or sensors, telephoto lens, super telephoto lens, ultra wide lens, WI-fi shield, LCD shield, motor shield, ecc.. The possibility to add all these different components or modification make the device a very well suited solution for many problems, application

Processor	ARM 32-bit Cortex-M7 CPU w/ Double Precision FPU 480 MHz (1027 DMIPS)
Memory	SDRAM: 32 MBs SRAM: 1 MB Flash: ext 32 MB, int 2 MB Expandable with SD cart
Resolution	Grayscale: 640x480 max RGB565: 320x240 max Grayscale JPEG: 640x640 max RGB565 JPEG: 640x480 max
Phisical attributes	Weight: 19g Length, Width, Height: 45x36x30 mm
Lens	Focal length: 2.8mm Aperture: F2.0 Format: 1/3” HFOV: 70.8°, VFOV:55.6°
Peripherals	GPIOs, interrupts, SPI, UART, I2C, DAC/ADC, PWM, LEDs, removable camera module

Table 2.2: OpenMV H7+ specifications

and especially for prototipation.
Table **??** summarizes the specifications of the device used in this study.

Since the application of this study sees the use of a camera that points to a screen, a small 3D printed support was created. An already available project [**tripod˙link**] have been modified to make it possible to mount the OpenMV camera on the support itself. The *stl* files can be found in the GitHub repository of this project [**github˙repo**]. Figure 2.3 shows the OpenMV mounted on the 3D printed tripod while pointing to the computer screen during a CL session.

2.3 Machine learning support

In this study applications it is requires to apply machine learning on the MCUs. Even if both examples use microcontrollers developed by ST, the

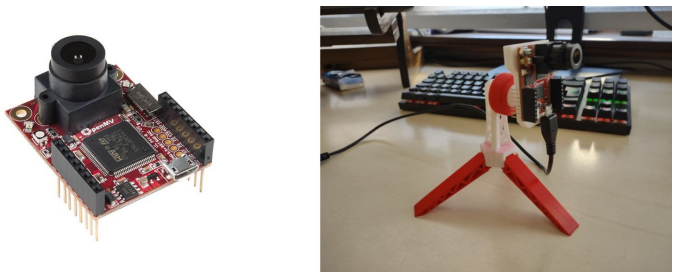

Figure 2.3: PLACEHOLDER - da cambiare

procedure for implementing ML capabilities is not the same.
For the Nucleo application it is required to use ST-CUBE-AI [**stm˙cube˙ai**], an extension pack developed and supported by ST itself. This toolkit can be added quite easily from the CubeMX, a software developed by ST that helps with automatic generation of chunks of code. In this case the addition of the AI pack allows the user to not be bothered by the generation of routines and other tools required for optimized machine learning inference. By using this pack it is possible to compress, load and run ML models in an easy and immediate way with just some function calls.

Chapter 3

System implementation

Machine learning is a branch of artificial intelligence (AI) and computer science that focuses on the usage of algorithms and huge quantities of data to generate models that are able to perform regression of grey mox models.

3.1 Basic pipeline of the CL system developed

Continual learning is the application of real time training on a model with the aim of generating self adjusting systems that are able to learn from incoming data streams. As already mentioned in Chapter , continual learning can lead many improvements if exploited in IoT applications. The main feature of Continual Learning (CL) is the ability of ML models to adapt weights and biases to learn the environment and continue to be relevant in the application.
In order to apply such a technology on embedded devices, it is necessary to develop a framework that permits the implementation of these kind of algorithms and strategies. For this study, two off-the-shelf hardware are used and a small framework is developed. The framework should always be paired with other toolkits that allow to perform machine learning inference easily. The framework developed should be attached to the last portion of a pre trained model. So it is necessary to use some built in tools for performing efficiently the inference and later apply the continual learning on the last portion of the model.
In this study the main idea was to develop something similar to paper

TinyOL. The idea is to attach the OL system to the last layer of the pre trained model in order to enhance just the classification aspect of the ML model. The base functions required by the system are basically two: be able to update the weights of the classification layer each time the model shows an error in the prediction array and be able to enlarge the size of the last layer at will, specifically when new classes are detected that should require a new label. The applications seen here are supervised machine learning trainings. This means that at every training step the griund truth label is known and is provided to the algorithm, which will then use these info to compute the error and back propagate this for updating the layer. In fact the feature that allows the model to recognize new classes is performed simply by checking if the label recieved is inside the pool of already known classes.
The basic idea of continual learning consists in being able to continually refresh and update the weights and biases of the layer or layers of interest. This is performed as in standard trainings by computing the error performed by the inference and by propagating its error back in the weights themselves. In orrder to do this it's necessary to know the basic structire of the model wich gives informatiosn about the math used by the nodes. If the entire path of computation from the weight of interest to the final prediction is known then it's possible to back propagate the error to the parameter. In this specific study the idea has always been to use CL in classification problems. By havinsuch a specifica field of application it's possil e to simply the CL ide to a restricted group of models. Since the problems regard only classification and this specifica application want sto update only the lsat layer the problem becomes a simple study of the back propagation over softmx layers and nodes. Let's consider a simple model composed of few layer, and asume that the last layer brings 100 nodes of the hidden layer 2 to just 5 nodes with softmax application function. It's then possible to comnsider everythin that happens before the softmax layer a grey box that simply gives us 100 output values. The basic formulas used ina softmax layer are the following:

$$z_i = \sum_i w_{ij}x_j + b_i y_i = softmax(z_i) = \frac{e^{z_j}}{\sum_j = 1^k e^{z_j}} \text{where i = 1, 2, ..., k j = 1, 2, ..., nwhere k = nu} \tag{3.1}$$

Where equation x is the formula that describes the the basic computation of a node of a ML layer, equation 2 is the softmax function directly applied on the ouput computed by the node. The complete computation for the

output of the OL layer then becomes:

$$y_i = softmax(z_i) = \frac{e^{\sum_j=1^n w_{ij}x_j+b_i}}{\sum_j = 1^k e^{\sum_j=1^n w_{ij}x_j+b_i}} \tag{3.2}$$

Once the prediction is computed with the softmax function applied on the output of the nodes, it is possible to use the loss function and compute the error. In common machine learning application the list of possible loss fuicntion that can be used is long and the different costs can be selected for different reasons and applications. In this case it has been decided to use a *categorical cross entropy*, which has the following definition:

$$cost = -\frac{1}{n}\sum_k [yln(y_i) + (1-t_i)ln(1-y_i)] \tag{3.3}$$

Where n is the total number of training data, y_i is the output obtained from the softmax function, t_i is the true label.
At this point the entire path of computations from the output of the frozen model to the error committed by the prediction is known. This allows for the computation of the derivatives iof such error which are necessary for the back propagation, where the weights and biases of previous layers are updated with a proportional dependancy on the derivativo of the error with respect to every single weight. The computations performed for the final update rule are the following:

Given that the error E is computed as: $E = cross(y_i) = cross(soft(z_i))$

(3.4)

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial S_i} \cdot \frac{\partial S_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}}$$

(3.5)

$$\frac{\partial z_i}{\partial w_{ij}} = X_j$$

(3.6)

$$\frac{\partial S_i}{\partial z_i} = \quad = \frac{e^{z_i}}{(\sum e^{z_i})^2} - \frac{(e^{z_i})^2}{(\sum e^{z_i})^2}$$

(3.7)

(3.8)

Then it's immediate to apply this on the weifht in order to apply an actual training step. The final formula that defines the back proapgation on weights and biases of a softmax layer with the usage of a categorical cross entrioyp loss function is:

$$w = w - lrate \tag{3.9}$$

This is the rule to be followd for the application a basic training step in real time. This is also the same idea explored by TinyOL, which simply applies this rule to an autoencoder model for the recognition of patterns.

3.2 Algorithms implemented

In this study not only the basic training strategy has been implemented and tested but also other regularization approaches have been used. As explained in Chapter 1, regularization approaches are strategies that exploit the addition of loss term to the update, thanks to which it is possible to have some sort of control over the weight update. In today's research some well known state-of-the-art strategies for CL are Elastic Weight Consolidation (EWC), Synaptic Intelligence (SI) and Learn Without Forgetting (LWF) [**li2017learning**]. In a recent paper Copy Weight with Reinit (CWR) [**lomonaco2017core50**] has been proposed and later upgraded with some improvements under the name CWR+ [**maltoni2019continuous**]. In the same paper the authors propose a new strategy called AR1 [**maltoni2019continuous**], which exploits more layers of a model for performing the CL training. In this study only some of the aforementioned were implemented, namely LWF, CWR and TinyOL, together with small variations for the implementation of CL with batches.
In the applications for this study all methods are applied in the same OL general block system, which simply is attached to a pre trained model, called frozen model. The frozen model is treated as a grey box which simply performs feature extraction on the input array and provides the last layer with useful elaborated data. The OL training is then applied on the weights of the classification layer following the update rule imposed by the algorithm of choice.

TinyOL

The TinyOL method applied on MCU has been initially implemented in paper [**ren2021tinyol**]. Its implementation is quite easy since it consists in just some for loops applied correctly on the weights and biases. As explained before, this method uses the basic rule of the training step which consists of propagating the error committed at inference to the weight of interest by using SGD. Keep in mind that the activation function of the last layer is Softmax and the loss function is always Categorical Cross Entropy. This said the weights update rule for this algorithm are:

$$\mathcal{L}_{cross}(y_i, t_i) = -\sum_i t_i log(y_i) + (1 - t_i) \cdot log(1 - y_i)$$

$$w_{i,j} = w_{i,j} - \alpha(y_i - t_i) \cdot x_i$$

$$b_i = b_i - \alpha(y_i - t_i)$$

$$\text{where i=0,1..,n and j=0,1,..,m}$$

Where y_i is the prediction obtained from the OL layer, t_i is the true label, α is the learning rate (tuned by the user), $w_{i,j}$ are the weights of the OL layer, b_i are the biases of the OL layer, n is the max amount of classes known by the OL system and m is the height of the last layer of the frozen model. Note that in this entire study the number n can change dynamically since the maximum amount of possible classes is not known a priori, or at least it ism not known in a real life scenario.
Also a variation of this method has been implemented. The variation takes into consideration the possibility to use batches of data for computing the back propagation and not simply the last sample received. The idea for implementing such a variation came thanks to the article [**batch˙size˙medium**], where the author explores the impact that batch size has on the training dynamics. The base idea of the variation is that by using a batch of samples bigger than 1 the back propagation computed should not relay on just the last sample but on the average computed on the group. This should help the model to be less vulnerable to noisy data and outliers. In order to apply the algorithm with this variation it's necessary to store the data generated from the previous samples. This requires the allocation of double the amount of memory required for the standard version, which is done with the creation of the matrix W and B. In order to reduce the amount of computation at each

training step at first the inference is performed, after this the back propagation for each single weight and bias is computed and its value is added inside the matrices W and B. When the batch finishes the average on each weight and bias is computed and applied on the actual parameter that was kept constant during the entire inference of the batch. This time the update rule at each inference step becomes:

$$W_{i,j} = W_{i,j} + \alpha(y_i - t_i) \cdot x_i$$

$$B_i = B_i + \alpha(y_i - t_i)$$

And at the end of every batch the update applied on the real weights is:

$$w_{i,j} = w_{i,j} - \frac{1}{batch\_size} \cdot W_{i,j}$$

$$b_i = b_i - \frac{1}{batch\_size} \cdot B_i$$

Said this both methods are quite simple and are based on the same basic principle. The method TinyOL requires the usage of only one weight matrix and one bias array. their dimension depend on two parameters, the number of classes known by the OL system represented by the value n and the size of the last layer of the frozen layer, represented by the value m. This makes the memory allocated from the method be equal to a total of (nxm+nx1)*4 bytes. The method is able to change the layer parameters each time a new sample is received with no constraints. This feature is the main problem that concerns this strategy. In fact it allows for a lot of flexibility but it doesn't provide a protection from catastrophic forgetting catastrophic forgetting. By allowing any possible modification it's immediate for the model to follow what the stream of data is imposing to it, thus it's quite easy to train on noisy or outliers data or even be guided towards context drift.
On the other hand the method TinyOL with mini batches exploits the same approach seen by the standard TinyOL method but receives a back propagation update that is dictated by the average computed from a group of k samples. Depending on the value of k the group can be considered to be a more or less good representation of the data received. In any case this method should be able to better contrast catastrophic forgetting, noisy data and outliers.

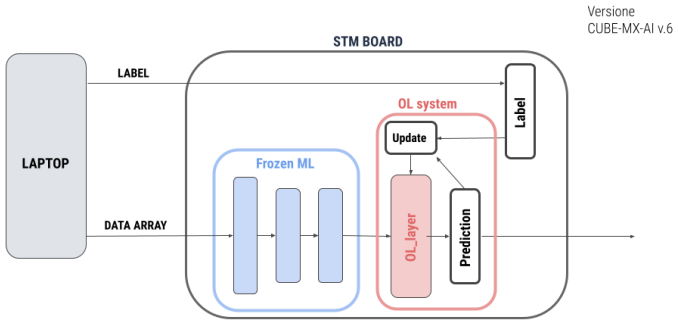

Figure 3.1: Place holder - OL

TinyOL V2

The TinyOL V2 algorithms is based on the same idea of the original TinyOL. A little intuible modification is applied in the method with the aim of contrasting catastrophic forgetting. The idea is to remove the possibility to have a drift over the original weights by removing completely this possibility. This algorithm in fact applies the same exact update rule on its parameters but it applies it only on the weights that represent the new classes. Which simply means that the rule now becomes:

$$w_{i,j} = w_{i,j} - \alpha(y_i - t_i) \cdot x_i$$

$$b_i = b_i - \alpha(y_i - t_i)$$

where i=p,p+1..,n and j=0,1,..,m

The only difference is in fact the iterator i, which goes from p to n, where p represents the position of the first unknown class.
Also in thsi case the variation of the method for working on batches has been implemented. Again the algorithm is the same as TinyOL with batches but with the iterator i that goes from p to n.

$$W_{i,j} = W_{i,j} + \alpha(y_i - t_i) \cdot x_i$$

$$B_i = B_i + \alpha(y_i - t_i)$$

And at the end of every batch the update applied on the real weights is:

$$w_{i,j} = w_{i,j} - \frac{1}{batch\_size} \cdot W_{i,j}$$

$$b_i = b_i - \frac{1}{batch\_size} \cdot B_i$$

where i=p,p+1..,n and j=0,1,..,m

In conclusion TinyOL V2 is a simple method that differs from the original only because of a small difference in the update rule. By forcing the update on only a portion of the weight and biases the behaviour of the catastrophic forgetting that tries to modify the original knowledge and make the model drift from that context is completely removed. This helps the algorithm by contrasting catastrophic forgetting but also reduces the ability of the model to perform fine tuning on those specific classes. Another negative aspect regards the general behaviour of the final mode. By having a training strategy that updates only a portion of weights it's clear how the model itself cannot be optimized to reduce the loss function. This means that at the end of the training the model will be composed of two parts that will not work together to find the best prediction. Like before the TinyOL V2 with mini batches allows the model to learn from a bigger group of samples, this should help the model to avoid over fitting, outliers and noisy data.
The method TinyOL V2 requires the same amount of memory that was used by TinyOL, which means a matrix of size $n \times m$ and an array of size $n \times 1$. On the other hand the method TinyOL V2 with batches requires an additional matrix and array but this time with a reduced size of $(n-p) \times m$ and $(n-p) \times m$ since only the weight of the new classes require an update.

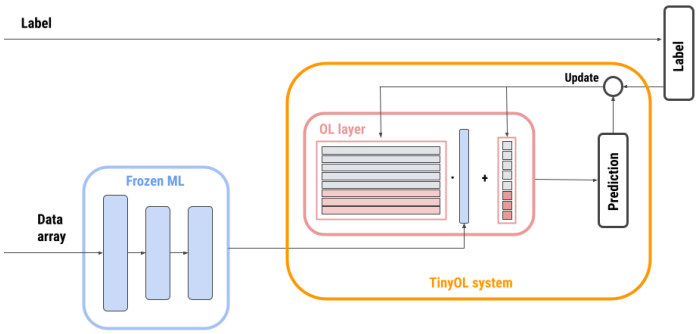

Figure 3.2: Place holder - OLV2

LWF

The LWF strategy is a regularization approach introduced in [**li2017learning**] and later applied with small variation in [**maltoni2019continuous**]. The main idea of the method is to contrast catastrophic forgetting by applying a smart loss function and double architecture that is able to combine old and new knowledge in the back propagation of the parameters. The double architecture models refers to the fact that two models are required in order to use this method. In this case the entire OL system has been applied only on the last layer of the model so the double architecture in this case is composed only of a double classification layer. The first is called tl, training layer while the second is called cl, copy layer. The role of tl is to be continuously updated at each training step with the LWF back propagation rule, while cl is a layer that contains a copy of all the original weights computed in the Tensorflow training, thus it represents the original knowledge of the model. The back propagation rule is based on the idea of fusing the weight updates that the two layer would apply. The fusion of these two updates is done with a weighted average that changes dynamically as the training continues. This of course implies that both layers produce a prediction, which means double computation for the OL system.
At this point the only major difference with respect to the TinyOL method is the double inference and the computation of the weighted back propagation. The update to be applied can be computed quite easily again by using SGD which turn out to be a simple weighted sum of two back propagation:

$$\mathcal{L}_{LWF}(y_i, z_i, t_i) = (1-\lambda) \cdot L_{cross}(y_i, t_i) + \lambda \cdot L_{cross}(y_i, z_i)$$

$$w_{i,j} = w_{i,j} - \alpha \cdot x_i \cdot [(y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda]$$

$$b_i = b_i - \alpha \cdot [(y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda]$$

$$\text{where i=0,1..,n and j=0,1,..,m}$$

Where y_i is the prediction array obtained from the layer tl, $z_i$is the prediction array obtained from the layer cl, t_i is the ground truth label, λ is the variable weight that defines which prediction has more decisional power.
The back propagation is composed of two parts, the first defined by tl and the second defined by cl. The value λ plays a very important role in this update. As explained in [**maltoni2019continuous**] its value cannot stay constant because it would be suboptimal. In their application the value

follows a discrete function linear with the number of batches encountered, while in oiur case its value needed to be dependant on the only value known in a OL application, the amount of samples elaborated. The update of the loss function weight is the following:

$$\lambda = \frac{100}{100 + \text{prediction\_counter}}$$

Another important note to be said is that in the update rule the implementation follows the variation proposed in [**maltoni2019continuous**], where the loss functions L_{LWF} used in the weighted average are not a balance between categorical cross entropy and knowledge distillation but a balance between two categorical cross entropy. This is a little modification that allows for an easier implementation without ruining the performance.
Also in this case a version that integrates batches is proposed. This time the method simply updates the values of *cl* every time a batch is finished. The algorithm this time simply becomes a fusion between old and knew knowledge where the old knowledge is refresh once in a while. In this way the model can be seen as a model that performs a weighted average in between a fast learning memory and a memory that stops in time. The size of a batch is defined by the value k.

$$\mathcal{L}_{LWF}(y_i, z_i, t_i) = (1-\lambda) \cdot L_{cross}(y_i, t_i) + \lambda \cdot L_{cross}(y_i, z_i)$$

$$w_{i,j}^{TL} = w_{i,j}^{TL} - \alpha \cdot x_i \cdot [(y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda]$$

$$b_i^{TL} = b_i^{TL} - \alpha \cdot [(y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda]$$

$$\text{where i=0,1..,n and j=0,1,..,m}$$

And at the end of a batch (once every k values are elaborated):

$$w_{i,j}^{CL} = w_{i,j}^{TL}$$

$$b_i^{CL} = b_i^{TL}$$

This method, being different from the previous, requires also a different λ rule. Experimentally it has been found to be well working an update rule defines as follows:

$$\lambda = \begin{cases} 1 & prediction\_counter \leq batch\_size \\ \frac{\text{batch\_size}}{\text{prediction\_counter}} & prediction\_counter > batch\_size \end{cases} \tag{3.10}$$

Both the LWF methods require the same amount of memory, which is a double prediction layer of size $n \times m$ and $n \times 1$. Both methods are quite easy to implement and their strength is defined in the value λ and in their update rule. They differ only from the weight updated applied on the matrix cl. A negative aspect that characterizes these two methods is the amount of computation required, which can be a problem for tiny devices. By having two layers and the need of two prediction is of course needed the double of computation.

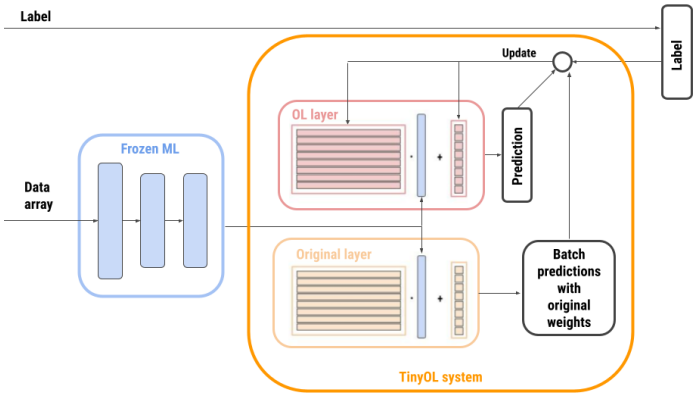

Figure 3.3: Place holder - LWF

CWR

CWR is an architectural approach that exploits the usage of two classification layers and a weighted back propagation rule for performing OL. Again the two classification layers are called tl, training layer and cl, consolidated layer. The idea is to perform training at each step on tl with the same method as TonyOL and at the end of each batch update cl with a particular rule. The back propagations for the cl at the end of a batch are tge same for biases and weights and are the following:

$$cw_{i,j} = \frac{cw_{i,j} \cdot updates_i + tl_{i,j}}{updates_i + 1}$$

$$tw_{i,j} = cw_{i,j}$$

Where $tw_{i,j}$ are the weights and biases of the training layer, $cw_{i,j}$ are the weights and biases of the consolidated layer, and $updates_i$ is an array that

behaves as a counter of labels encountered.
By using two classification layer that update differently the method tries to replicate the short term memory and long term memory architecture that characterizes biological brains ??? ESSERE SICURO. The layer tl behaves as the short term memory since it gets updated at every single training step, and at each batch it gets reset to the correct values. The layer cl behaves as a long term memory since it never gets reset or cleaned and it gets updated only once every batch with a weighted average. This weighting method depends on the number of times that a specific label appeared in the training batch.
Another important aspect of this method is how the prediction is computed and used. While performing only training the method requires only a prediction performed by tl since this needs to get updated by its error. However if an actual prediction is requested to the model also the cl layer should perform the computation and provide an inference. In fact the inference obtained by the consolidated layer is to be considered more relevant and reliable since it is produced by the long term memory. In the case a prediction is required the method needs a double prediction, one from tl and the other from cl. Again as said for LWF this is not optimal because of the limits of tiny devices.
CWR is a method easy to implement. Its strength are hidden in the double architecture and the update rule that make it possible to merge short term memories and long term memories and also contrast catastrophic forgetting. The memory required for this algorithm is: to weight matrices of size $n \times m$, 2 bias arrays of size $n \times 1$, one array that keeps track of the labels encountered of size $n \times 1$. The amount of computations can change depending if a simple training step is performed or if an inference is required.

My algorithm

And

3.3 Gesture recognition application

3.4 Image classification application

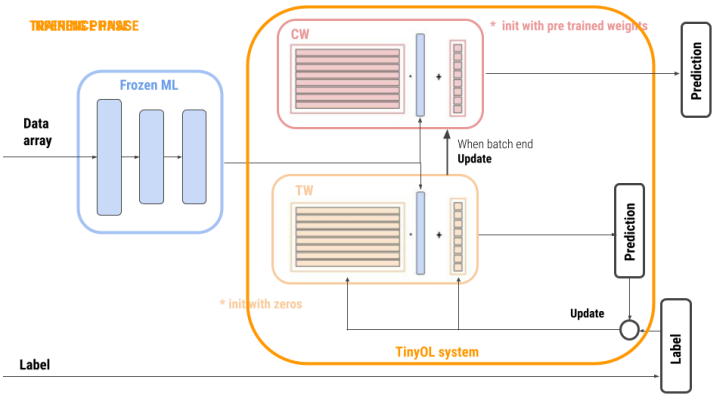

Figure 3.4: Place holder - CWR

Chapter 4

Experimental setup

In this chapter, the practical aspects of the experiments are described. Initially, the study was developed entirely on the laptop using Python code with the aim of understanding the theoretical behaviour of the methods and the capabilities of CL. Later, the same principle and basic pipeline were ported to the STM Nucleo and the OpenMV camera, respectively. All applications use the same general workflow: i) train the base model on a powerful device for the classification of the basic classes using Tensorflow; ii) manipulate and compress the model if necessary, then load it on the MCU of interest; iii) attach the OL system to the frozen model, then define the basic training parameters and the desired CL strategy.
In this chapter, all the most important steps for a good setup of the experiments are described. Specifically, these steps are: the dataset collection, the training of the frozen models, the implementation of the OL system on the MCUs, and the application of the entire system on the device with a short explanation of how the CL trainings are performed.

4.1 Dataset collection

To create and train ML models, it is necessary to have big datasets. In this study, the two applications explored elaborate two types of data. The Nucleo F410-RE application uses time series of data recorded from an accelerometer, while the OpenMV application uses images containing digits from the MNIST dataset **??**.

4.1.1 Accelerometer dataset

For the gesture recognition application the dataset was created from scratch. Datasets of this type, containing accelerometer data representing gestures are not very common, so it was necessary to collect it. It was decided to collect time series of 3D accelerometer data of letters written in the air from the user as a proof of concept of a gesture recognition system. The dataset collection was carried out with the hardware described in Chapter 2.
The dataset is composed of 8 different letters, which are A,E,I,O,U and B,R,M. The vowels compose the original classes that are first learned by the frozen model, while the three consonants are the additional classes that are learned later by the CL system. The collection of the dataset is performed by connecting the MCU to a laptop via UART protocol (USB cable). The laptop behaves as a power provider and real-time storage for the data stream. To collect the sensor data, it is used a small script that controls UART and I2C communication with some timers and GPIOs. When the Nucleo detects a GPIO interrupt, the user specifies the label. Then the code records data from the sensors with a frequency of 100 Hz for 2 seconds. Meanwhile, the values are also streamed via USB to the laptop which stores data using a serial communication software (MobaXTerm).
To make the dataset be composed of samples that better resemble real-life applications, a NIC scenario was artificially imposed. This means that the recorded samples contain both new classes (the consonants) and new pattern of known classes. The latter was introduced by performing motion paths with accentuated characteristics. Some examples are: the accentuated oval shape of the letter O, the speed at which the sensor moves for the letter I, the general size/width/height of all letters, and the radius of the curves for letter R and B. Figure 4.1 shows the general path that was followed while drawing the letters in the air.

All the samples received by the MCU are saved in a table format in a text file. The columns of the table contain: the number of samples recorded, the label of the sample and three columns for the accelerations recorded from X, Y, Z axis. Considering that the MCU was set to work at a sampling frequency of 100 Hz for 2 seconds a single sample is composed of 600 values (200 for each axis). The final shape of the dataset is 5130 samples, where the vowels have on average 560 samples each, while the consonants have around 760 samples each.

Figure 4.1: Motion of the accleerometer (NON SI VEDONO LE FRECCE ARANCIO)

Once the dataset is collected, post-processing is performed. This consists of a simple reshape, shuffle and subdivision. To perform the training on the ML architecture, all samples are reshaped by stacking all the rows into a single array from a matrix 3×300 to an array 1×600. Then, a subdivision of the dataset is performed. Given that the dataset is needed for two trainings (frozen model training and CL training), it is required to separate it correctly. The frozen model can recognize only vowels, thus its dataset is composed only of vowels. This dataset counts a total of 881 samples with 176 samples from each vowel. The OL model, on the other hand, is trained on all letters, so its dataset contains the remaining vowels and all the consonants.
After this, both datasets are divided in training and testing portions, which is done with the usual 80-20% rule. Figure 4.2 shows how the dataset are divided and balanced for the two trainings regarding the gesture recognition application.

4.1.2 Digits recognition dataset

For the image classification application the well-known MNIST dataset was used. The MNIST dataset is a publicly available large collection of images of handwritten digits. The dataset is well known in academic and research for its small size of images and large quantity of samples. It is composed of 60000 images for training and 10000 images for testing. The images are gray scaled and have a size of 28×28 pixels. In today's research, the dataset is commonly used for benchmarking ML training, while in academic it is used for basic training of classification and generative models. Figure 4.3 contains some sample images that of the MNIST dataset.

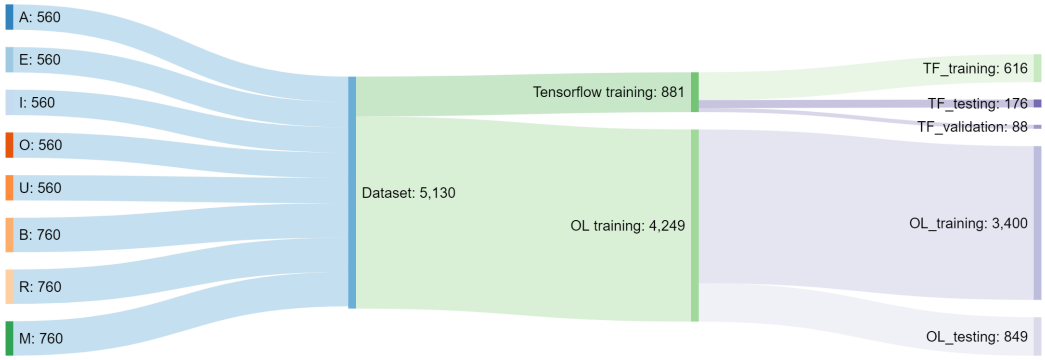

Figure 4.2: Stankey diagram showing how the letter dataset is divided

Figure 4.3: Example of images of digits in the MNIST dataset

For this application purpose, the dataset requires pre-processing. Considering that the goal of the frozen model is to correctly recognize the digits from 0 to 5 it is necessary to separate the dataset into two groups, *low_digits* and *high_digits*. By doing this, the *low_digits* group is composed of 36017 samples, while the *high_digits* group is composed of 23989 samples. For the training of the frozen model, the entire *low_digits* group is used for a Tensorflow training, which is further also separated in train, test, and validation. For this reason, the common 70-20-10 rule was used. On the other hand, for the training of the CL model only 5000 samples from the *high_digits* group were used. This because from the experience previously obtained from the latter application, it was demonstrated that 500 samples for each class are more than enough for a correct CL session. The CL dataset is then separated in training and testing with the rule 80-20% for the CL application.
Figure 4.4 shows how the dataset is divided for the training of the frozen model and CL model.

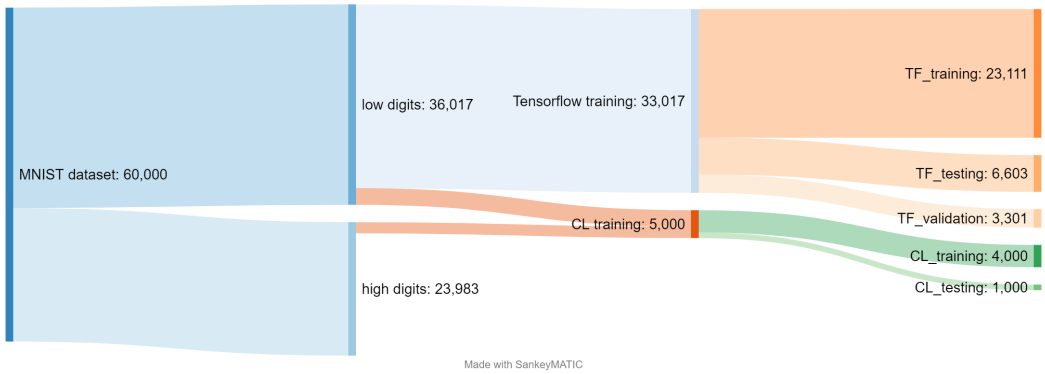

Figure 4.4: Stankey diagram showing how the MNIST dataset is divided

4.2 Frozen model training and evaluation

Once the datasets are created and post-processed, the training of the frozen models can be performed. Both models require small structures in order to be loaded on small MCUs. In both applications the models perform a classification, which can be achieved simply by imposing the last layer's activation function as a Softmax. All the other layers are used only for feature extraction and their structure and characteristics depend on the type of data to elaborate. The trainings of the frozen models and their manipulation was carried out with Tensorflow library and Python.

4.2.1 Gesture recognition model

In the gesture recognition application, the model elaborates a time series of accelerometer data. The model's structure is composed of only fully connected layers, which makes the structure very simple. Typical applications of ML on time series use LSTM types of models, which are structures well suited for the elaboration of time dependent signals. In this case, the results obtained from a structure composed of only fully connected layers brought to satisfying results, so the model's structure was kept as is. The layer sizes are 600 for the input, 128 for the hidden layers, and 5 for the classification layers. The activations functions are, Softmax for the classification layer, and ReLu for all the other layers. Figure 4.5 shows a plot that contains the basic structure of the model used.

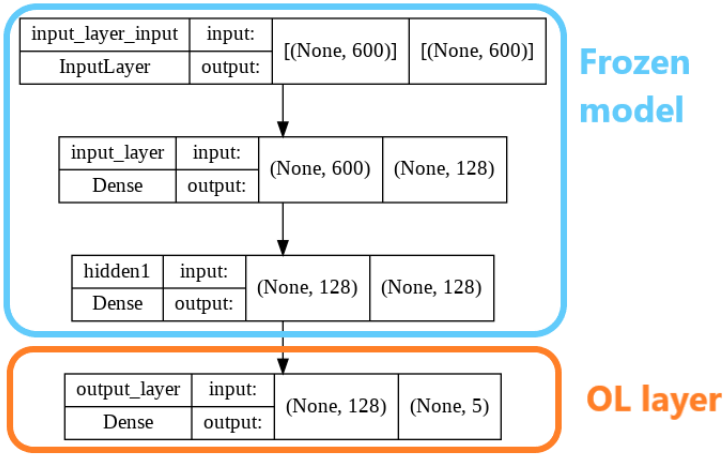

Figure 4.5: Letter recognition model: basic structure and its separation in frozen model and OL classification layer

The output layer's shape consists of 5 nodes because the model predicts 5 classes (the vowels). The number of layers and total parameters is quite low (94,085) respect to typical machine learning models; this makes it very suited for applications on MCUs.
The main training parameters are: *Adam* optimizer, *categorical cross entropy* as loss function, 20 epochs, and a batch size of 16. The accuracy obtained from the testing is 96.83%. Figure 4.6 contains two plots. On the left is shown the variation of the accuracy and loss during training, while on the right is shown the accuracy of each class at the end of the Tensorflow training.

The last step consists of the preparation of the frozen models and the exportation of the model itself. This is necessary because of the particular actions that are performed on the last layer by the OL system. Because it is required to have total control over the weights and biases of the classification layer, it is necessary to have the model separated into two parts. The first one is called the frozen model and is just a *model.h*5 file which contains the truncated version of the trained model (classification model is removed). This portion of the model, once loaded on the MCU, is manipulated by the inference tools and can be used as a grey box. The output of this grey

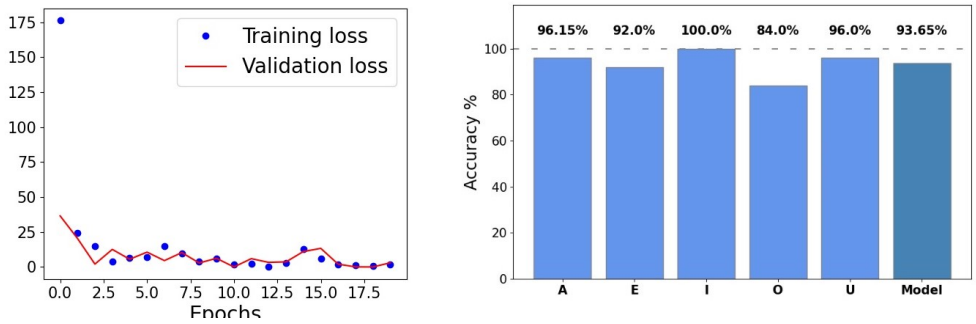

Figure 4.6: Letter recognition training: on the left the variation of loss and validation; on the right the accuracy for each class

box is just the results from the feature extraction and it can be forwarded in the classification layer for the prediction. The second part of the model exportation is the classification layer. The weights and biases of this layer are not exported as before, but are saved in a text file in matrix form. By doing this, it is possible to later load them in the MCU's RAM, which allows the OL system to edit and manipulate parameters and the layer shape. Figure 4.5 shows how the base model was divided into two main parts.

4.2.2 Image classification model

In the image classification application, a Convolutional Neural Network (CNN) architecture was used. These model types are specifically created for the elaboration of images and their main feature is the presence of convolutional layers for the feature extraction followed by NN layers for the classification. Figure 4.7 contains a plot that shows the structure of the model.

The model contains two sequential blocks of two convolutional layers followed by Max Pooling. This type of structure allows the ML model to perform initially a feature extraction over the image and to flatten the output matrix to an array. The fully connected layer is used to elaborate the array and later feed the data to the classification layer, where the Softmax activation function is used for providing the probability of each class.
The output consists of 6 classes because the frozen model is trained for the recognition of the *low_digits* group of images (i.e. digits from 0 to 5). Note

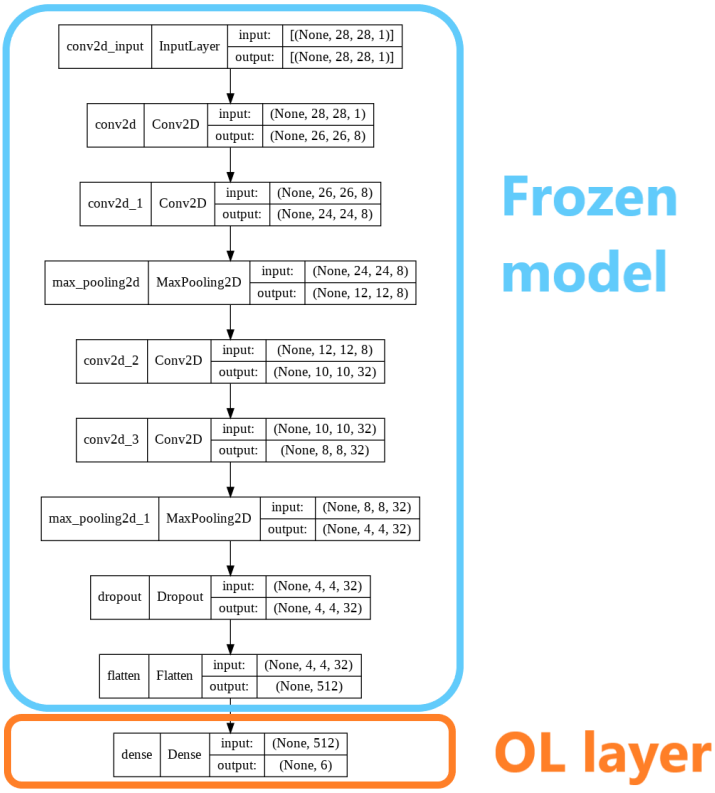

Figure 4.7: Image classification model: basic structure and its separation in frozen model and OL classification layer

also how despite having a more complex structure the number of parameters in the model is not much higher when compared to the previous application. This allows to have a small model that can be easily deployed on constrained MCUs for enabling fast inference.
The relevant training characteristics are: *Adam* as optimizer, *categorical cross entropy* as loss function, 30 epochs, and batch size of 64. The final accuracy obtained from the testing of the model is of 99.35%. Figure 4.8 shows on the right the behaviour of the accuracy and loss during training, and on the left the accuracy of the model for each class of the starting dataset.

Another important step that was performed for the OpenMV application is the pruning and quantization of the model. Usually, to deploy models with a high memory footprint it is necessary to apply compression techniques such

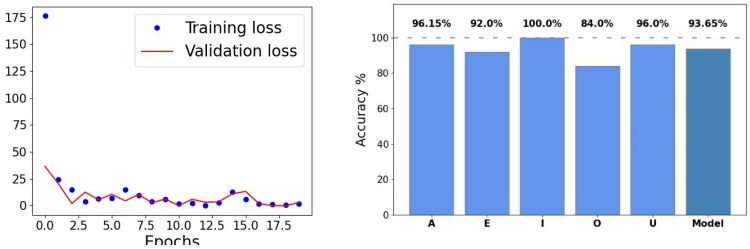

Figure 4.8: PLACEHOLDER

ass combination of pruning and quantization. Pruning is a method that reduces the number of connections in a NN model by setting to 0 redundant or non-relevant weights. This helps to reduce the memory occupied by the model, and if done strategically, can reduce the number of computations required for inference. By injecting forced sparsity inside the weight matrix, it is possible to reduce the computations needed, thus improving the inference efficiency.
Note that the Flash memory of the OpenMV is capable of storing models with bigger sizes than the one created, so pruning and quantization are not actually required. However, to demonstrate these model capabilities, it was decided to apply compression techniques anyway. The pruning and quantization procedure was carried out with Tensorflow, which makes the process as easy as a training setup. The main characteristics of the pruning are: *Adam* as optimizer, *categorical cross entropy* as loss function, 5 epochs, batch size of 32, initial sparsity of 0.5 and final sparsity of 0.8. After pruning the model shows an accuracy of 99.6%.
After this step, it is also possible to introduce the quantization operation. This is carried out almost automatically by Tensorflow simply by calling the correct functions. The model size along the different steps of the compression are: 230 kB after the first Tensorflow training, 86 kB after pruning and 64 kB after quantization.
As in the previous application, the last step consists of the exportation of the model. The frozen model is exported as *model.h*5, while the classification layer parameters are exported in text file and later loaded in the MCU.

4.3 STM F401-RE setup for gesture recognition

In this section the setup needed for a correct gesture recognition application is described. Here is described how the CL system was implemented on the Nucleo board, hoe the ML model was deployed on the MCU and how the communication laptop-MCU works.
In the gesture recognition application, to add ML capabilities to the microcontroller, two software are exploited. The first, STM-CUBE-MX, is a piece of software used for automatically generating chunks of code. This helps the user by lifting a lot of work required for the initial setup of the device. Thanks to this software it is possible to use the UI and easily define the main characteristics of the MCU that is being programmed. Starting frokm the definition of the main clock and the essential characteristics of all the peripherals enabled to the parameters of the communication protocols. The second piece of software is just an extension pack of STM-CUBE-MX, and is called STM-CUBE-AI. This toolkit enhances the capabilities of the MCU by adding the possibility to use machine learning models on the device. The main advantages brought by the extension pack are the possibility to automatically load and process ML models on the flash memory and run optimized routines for inference on the same model. This toolkit also allows the user to compress the model at deploy time. In this case no compression is applied and the model is loaded as a *model.h*5 file.
Once the ML abilities have been added to the MCU it is possible to incorporate the CL system inside the code. The system, as already explained in Chapter 3.1, is attached at the end of the frozen model, and it is composed mainly of functions for the elaboration of the frozen model's output, management of memory and management of the data stream. The entire system is written in C code and it is almost entirely contained in one single library. The entire project is available on GitHub [**github˙repo**].
The basic structure of the code applied on the MCU is the following. At first all the most important parameters and variables are created. Depending on the algorithm defined, the right amount of memory is allocated. The weight and bias matrices are then filled with the parameters that have been previously generated by the Tensorflow training. Note that for the application on the Nucleo development board, the file is actually a C library, which contains all the weights and biases written inside a matrix. After all have been set up,

the infinite while loop begins. In here, if the "received sample flag" was triggered, the frozen model inference ism performed. Once the feature extraction is done, the output is fed to the OL layer, which will: propagate it through the classification layer, compute the prediction, compare the prediction with the label, compute the error and back propagate the error on the weights using the strategy adopted. After this, a small message (32 bytes) containing the most relevant informations about the training step is generated and sent back through UART.
Once the CL system and the ML model are loaded correctly on the MCU, the experiment can begin. To perform a fast, reliable and repeatable experimentation it was decided to develop a small app that controls autonomously the data stream towards the MCU. The app was developed in Python and is executed from the laptop, which stays in sync with the MCU and sends through UART (USB cable) one sample at a time. The script is quite simple and it follows the logic line: load the dataset of accelerometer array data, open the serial port with the specified properties, initialize containers for storing information, start an infinite while loop where the communication laptop-MCU is continuously repeated.
When the app is launched, the MCU should already be connected to the laptop, otherwise the serial port cannot be opened. Once the script is launched and the initial setup is done, the app waits for an acknowledgement from the MCU (a message of 2 chars), which signals the laptop that the device is ready to receive data and start the training. Once the ACK (acknowledgement) signal is received, the app sends a sample from the dataset composed of an array of 600 values and the label. The MCU then performs the routine aforementioned. Once the message is received by the laptop, a new sample is sent and the communication starts again. The procedure continues in this way until the training portion of the dataset is completed. After this, the pseudo testing begins. The only difference here, is that the message received by the laptop is stored and used later for the generation of plots and tables regarding the testing of the CL model.
A complete training procedure lasts for about 10 minutes. At the end of the communication of all samples the python scripts automatically stops the transmission and generates some plots. Figure 4.9 contains a block diagram that summarizes the steps of the app and the communication.

Figure 4.9: PLACEHOLDER

4.4 OpenMV setup for image classification

The application on the OpenMV camera is quite similar. Once the dataset, the model, the last layer and the system are prepared, everything can be deployed on the MCU. This time, because of the custom board and custom firmware, the toolchain for the deployment of the model is different. Thanks to it, it is possible to include the ML model inside the MCU files and generate from scratch a new firmware that contains the model parameters and structure. This allows to later use built-in-tools for efficient, optimized and fast inference directly from the MicroPython code. Once the firmware is generated, it can be loaded on the MCU and is immediate to apply machine learning.
The system developed for the application of the CL strategies is basically the same as the one explained in the previous example. Initially the code allocates the necessary parameters and memory, and later it enters in an infinite while loop. In here the code continuously waits for a new sample which is then fed to the built-in-tools that perform the frozen model inference. Then, the output is propagated through the CL system. The only major difference in this application is the management of the memory for the matrices. Some problems regarding the allocation of big matrices often blocked the code. It

was decided to not have the entire classification layer inside one single matrix, but rather separate it in smaller sections. This can lead to small increases of time for the inference, but it is actually required in order to have resulsts at all.
Once the code is loaded in the *main.py* file and the library *TinyOL.py* is loaded on the MCU, the experiment can be carried out. Once again a Python app was developed for the creation of fast, reliable and repeatable training sessions. The idea here is not to send data via USB connection, but rather display on a screen the images from the MNIST dataset while the camera is pointing at it. The USB connection is required in any case because the app needs to maintain sync with the camera and also send labels representing the image displayed. This time the app requires the use of Tensorflow (for loading the dataset), OpenCV (for displaying the images on screen) and PySerial (for opening a serial port with the MCU).
The app once again is divided in elemental blocks. At first the app loads the dataset and extract from it only the required amount of samples. Then the serial port is opened with the correct characteristics. After that, the app opens two windows, where the first is used for displaying the MNIST digits where the camera should be pointed at, while the second is used for displaying in real time the point of view of the camera. At last, a while loop starts and here the communication laptop-MCU is done. At every single step the laptop sends two messages to the camera. The first contains a small 4 char command which defines the state of the OpenMV camera, the second message is just the label of the image displayed on the screen. Every time these two messages are received, the camera takes a picture and performs actions depending on its state. The possible states are defined by the user (through the python app) and are:

- *snap* mode: the camera takes a photo, compresses it and sends it via UART to the laptop. This state is used for understanding where the camera is point and if the digit is inside the point of view of the OpenMV. The label received is not used and should be a char containing an X.

- *elab* mode: the camera takes a photo, applies a gray scale filter, applies a binarization, compressed the image and sends it via UART to the laptop. This mode is used for understanding what the camera will see when also basic image manipulation is applied. These manipulations

are used in the training for transforming the coloured image into a black and white image. Also in this case the label is discarded and it should contain just an X char. During this state, the app on the laptop slowly shows all the digits from 0 to 9. This permits the user to better point the camera towards the screen.

- *trai* mode: the camera takes a photo, applies inference on the image and later feeds the output to the CL system. No transmission of image is performed towards the laptop because it is easy to de synchronize the devices. The label this time is received an transformed into a hot one encoded array for the computation of the back propagation.

Once the training and pseudo testing are performed, the app stops showing digits images and the OpenMV camera stores the results inside its SD. The results can then be amanipulated by another script that will transformt he data in table, plots and confusion matrices. Figure 4.10 contains a block diagram summarizing the flow of the experiment.

Figure 4.10: PLACEHOLDER

Figure 4.11 shows the OpenMV camera pointed to a computer screen in all the different states. Top right is the camera in *idle* mode, no stream is received by the laptop. Top left is the camera in *snap* mode, compressed images are streamed to the laptop. Bottom left is the camera in *elab* mode, compressed and processed images are streamed to the laptop. Bottom right is the camera in *train* mode, no stream is available and the camera is performing inference and CL training.

Figures 4.12 shows a better comparison between images taken in the *snap* and *elab* mode. As mentioned before in the *snap* state the photo is captured, compressed and sent, while in *elab* mode the image is captured, gray scaled, a threshold is applied and later compressed and sent.

Figure 4.11: OpenMV camera pointing to a screen while in different states of the training. Top left is *idle* mode, top right is *snap* mode, bottom left is *elab* mode, bottom right is *trai* mode

Figure 4.12: Point of view of the OpenMV camera. On the left the original image taken with no compression and no elaboration, in the center an image taken in *snap* mode, on the right an image taken in *elab* mode.

Chapter 5

Experimental results

5.1 Experiment A: Gesture recognition

5.2 Experiment B: Image classification

This section contains and explains the results obtained from the test performed. Initially a description about the comparison between simulation and real application is performed with the aim of understanding if the training on the nucleo evolves as the simulation on the laptop. Then the results obtained from the application from the Nucleo are discussed and finally the results from the OpenMV application are described.
To understand if the Nucleo STM32 F401-RE is able to perform a real time ML training a study concerning the history of its parameters is done. The idea is to record the variation of the most important parameters from the OL layer at every training step and then compare its evolution to the same parameter evolutioin but recorded from the simulation carried out on the laptop. The parameters of interest are the biases of the OL layer, the predictions obtained from Softmax, the output of the frozen model, 10 weights picked randomically from the weight matrix. The evolution of the parameters is then displayed in a plot with the aim of observing how and if the history recorded from the laptop differs from the history recorded from the MCU. This is done qualitatively simply by looking at superimposition of the two lines. Figure 5.3 shows one example of comparison for the frozen model outpus, figure 5.1 shows the comparison of the bias evolution, figure 5.2 shows the comparison of the weights evolution, figure 5.4 shows the difference of

the predictions obtained from Softmax. The plots displayed above are

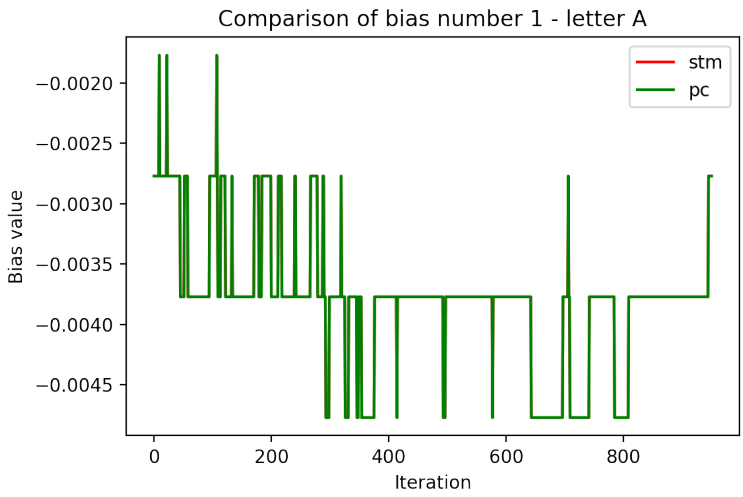

Figure 5.1: PLACEHOLDER

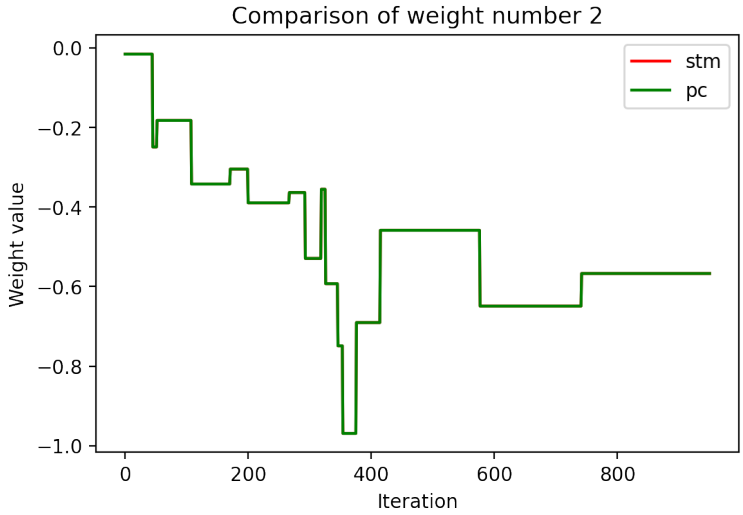

Figure 5.2: PLACEHOLDER

examples of the evolution performed, but they are rappresentative of the behaviour of all parameters. It's clear from the results how the two applications are very close, differencing from each other by just a small magnitude and for

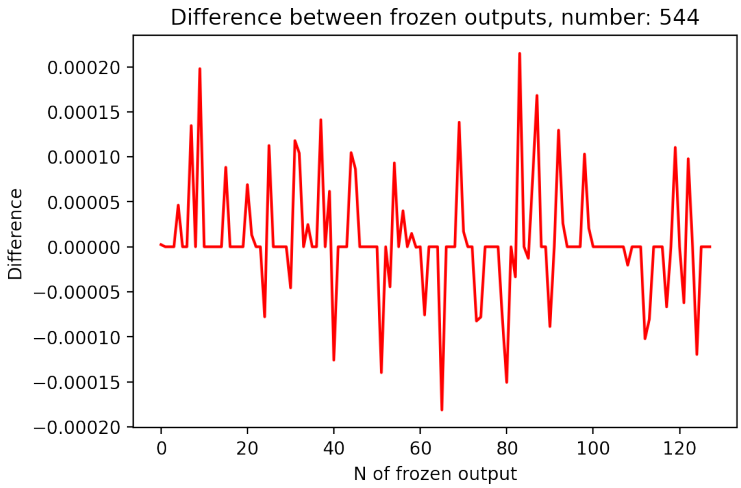

Figure 5.3: PLACEHOLDER

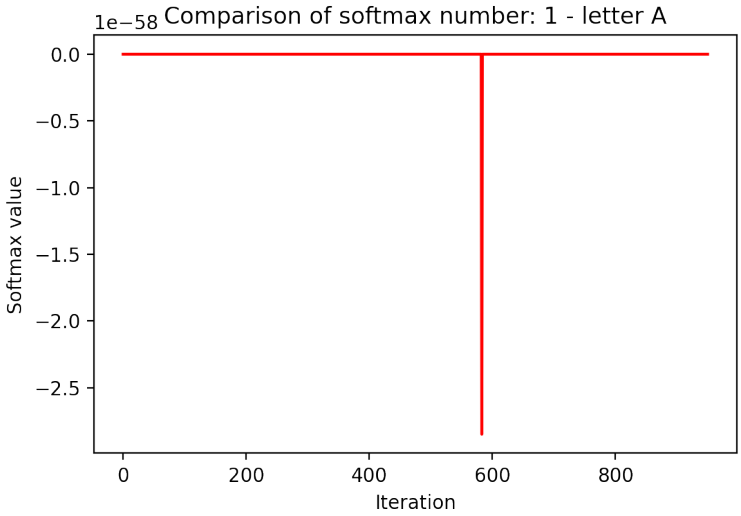

Figure 5.4: PLACEHOLDER

very few training step. Only one difference can be noted in figure x, where the dofference of the softmax prediction is dofferent from zero but still very little. This error in fact doesn't introduce any problem in the evolution since in the following steps the error goes back to 0.
One of the main concerns was about the feature extraction performed by the

frozen model. Because of the limited resources of the MCU usually models are compressed to be later loaded on the device. In this case no pruning or quantization have been applied, so the frozen model loaded on the MCU and laptop are exactly the same. The concern regards how the prediction is carried out by the X-CUBE-AI tool on the MCU compared with Tensorflow on the laptop. Figure 5.5 contains two examples of comparison of frozen model outputs. The x axis contains the iterator representing the i-th difference computed between the i-th value from the Tensorflow and STM output from the frozen model. On the left the difference that contains the biggest error is displayed, while on the right is displayed the sample that contains the second biggest error. It's clear how the plot on the left is not a correct representation of the MCU behaviour since it has a magnitude far too high when compared to the second biggest error. Thanks to this study it is possi-

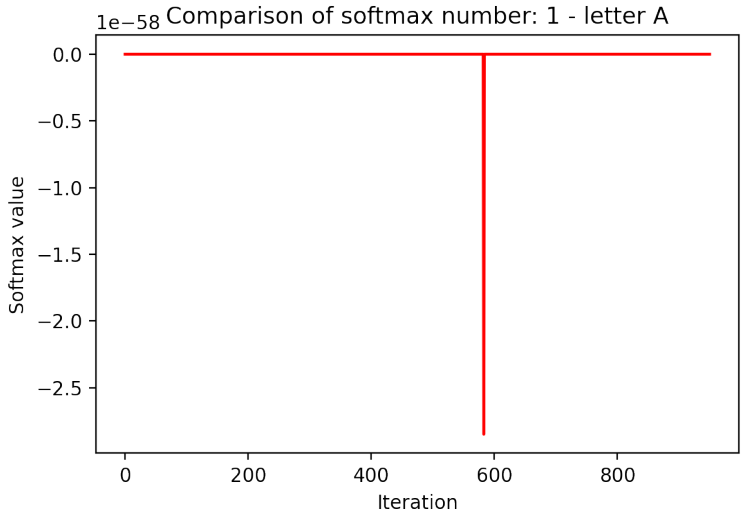

Figure 5.5: PLACEHOLDER

ble to conclude that a training performed on such a small device is actually possible and in terms of accuracy and precision it is as reliable as a training performed on a powerful device. Because of these plots it is then possible to conclude that the evolution of the model on the MCU is reliable and correct. A model trained on such device is subject to the same exact evolution that would affect the model in the case the training were to be carried on a laptop. From this point on all the experiment and results are obtained from MCUs.

Speaking about the gesture recognition experiment once the training have been carried out it's possible to display the accuracy of every single algorithm for every single class. As mentioned in section REFERENCE SECTION the testing is performed on the last 20 % of the dataset, so on a total of XXX samples. The bar plots containing the accuracies from every strategy together with their confusion matrices are displayed in Figures 5.6 5.7 5.8 5.9 5.10 5.11 5.12. From these plots it is clear how all methods are quite good

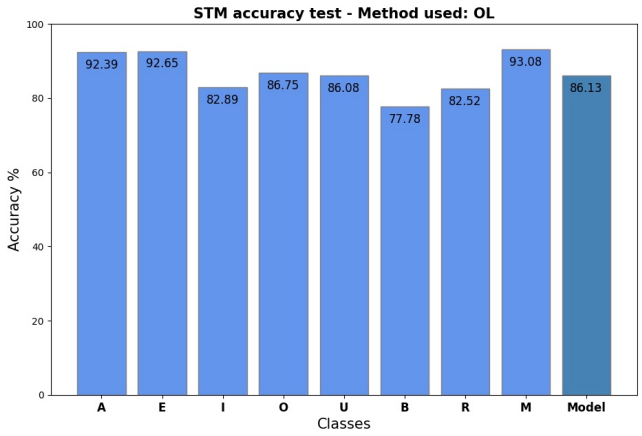

Figure 5.6: PLACEHOLDER

in digesting new classes and completely fuse them in the classification layer. No method in fact shows a bad learning on a specific class exept for the letter B that in all methods sees a lower accuracy when compared to others. It' easy to see from the cofusion matrix that the letter is not learned in a wrong way but rather the letter is easily confused with the letter R. Most probably this is because the path that has been followed when the dataset has beenc reated differs for just the leg of letter R.
Table x and table x contain other important results from the experiment.

In table x the overall accuracy of the model with specific strategies is displayed. From here it's clear how the algorithm CWR performs the best with an accuracy of xx %. All methods perform quite good with the lowest accuracy being xx & from the method OL, which is just a drop of xx % with respect to the accuracy obtained from the training of the frozen model performed with Tensorflow. Speaking about the time required for a training step the total time can be split in two portions. The first concerns the

	Accuracy %	**Average time inference frozen model in ms**	**Average time inference OL layer in ms**	**Maximum allocated RAM in kB**
OL	86.13	10.65	0.99	26.1
OL batch	86.26	10.65	1.54	29.8
OL V2	87.98	10.65	1.03	26.1
OL V2 batch	87.98	10.65	1.11	29.8
LWF	87.61	10.65	3.45	29.9
LWF batch	86.5	10.65	3.26	29.9
CWR	88.47	10.65	2.11	29.9
MY ALG	86.87	10.65	3.54	29.9

Algorithm	Parameter	Class								batch size	learning rate
		A	E	I	O	U	B	R	M		
OL	Accuracy	0.92	0.93	0.83	0.87	0.86	0.78	0.83	0.93	16	16
	Precision	0.92	0.93	0.85	0.87	0.86	0.78	0.83	0.92		
	F1 score	0.92	0.93	0.84	0.87	0.86	0.78	0.83	0.92		
OL batch	Accuracy	0.89	0.96	0.88	0.90	0.89	0.76	0.79	0.93	16	16
	Precision	0.93	0.97	0.89	0.91	0.85	0.75	0.78	0.93		
	F1 score	0.91	0.96	0.88	0.90	0.87	0.75	0.78	0.93		

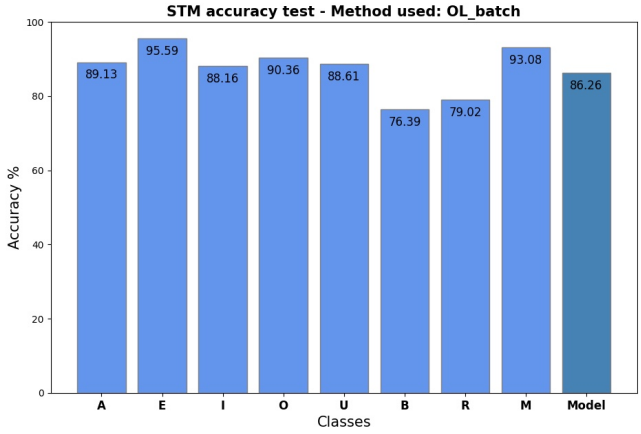

Figure 5.7: PLACEHOLDER

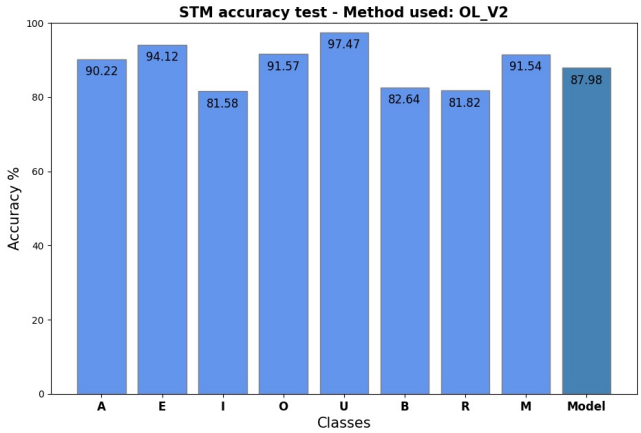

Figure 5.8: PLACEHOLDER

inference obtainde by the frozen model, which is of course constant for all strategies and takes 10.65 ms. The other part of the training step time is the time taken by the OL layer which contains the time required for the inference of the OL layer and the following computation of the back propagation and update of the layer's weights. From the table is clear how the faster methods are TinyOL and TinyOL v2, which are the only methods that do require only one OL layer, thus reducing the amount of computations require. On

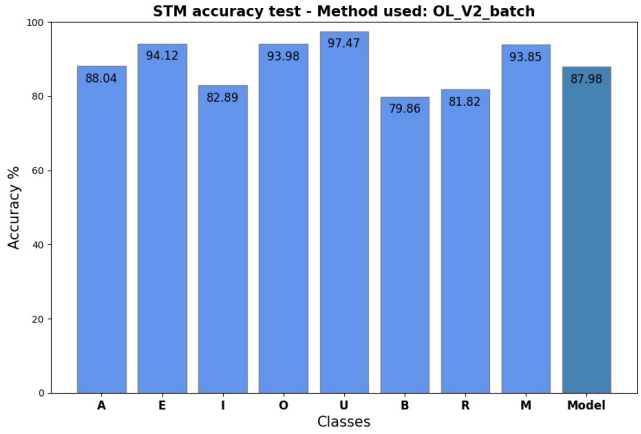

Figure 5.9: PLACEHOLDER

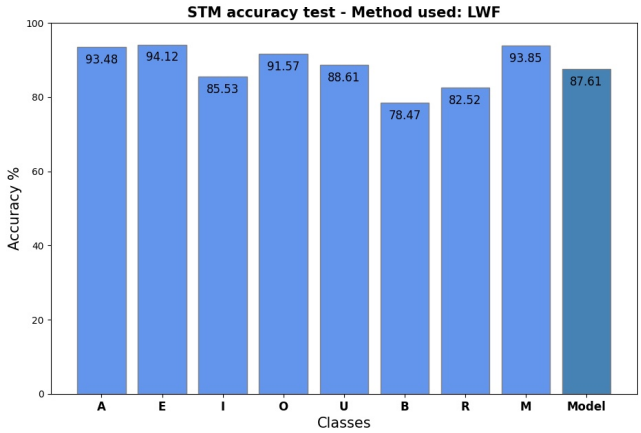

Figure 5.10: PLACEHOLDER

the other hand the slowest methods are LWF, LWF batch and MY ALG, which all require a double inference from the two classification layer. The time is in fact more than double the time required by all the other strategies. The last column concerns the amount of RAM allocated by the strategies. This value shows that the TinyOL and TinyOL v2 are the lightest methods since they require only the allocation of 1 weights matrix and 1 bias array. All the other methods require a very similar amount of RAM since they all

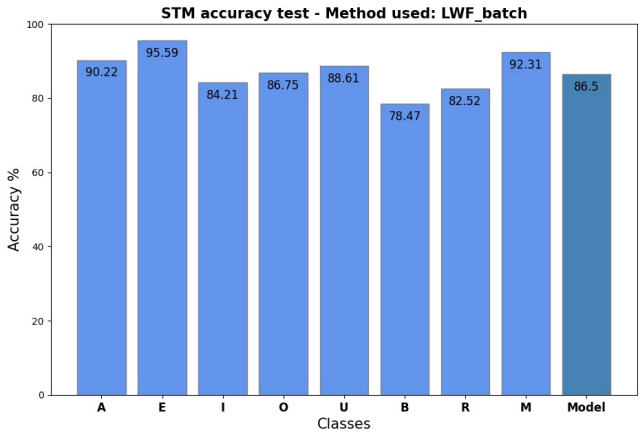

Figure 5.11: PLACEHOLDER

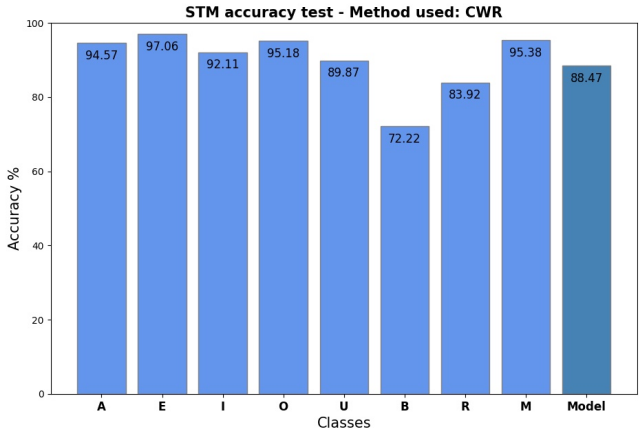

Figure 5.12: PLACEHOLDER

work with double memories. A little difference of just 100 bytes is due to the allocation of some additional values particular for some strategies.
Another important study that has been carried out is the study of variation of the accuracy while changing the batch size. This study was of particular interest thanks to **??**, a blog post that studies in detail the impact that the batch size has on a ML training. The results are show in figure 5.13. Here is clear how the only methods that drop their accuracy quite a bit are TinyOL

and TinyOL v2, while all the other are able to mantain their accuracy quite constant.

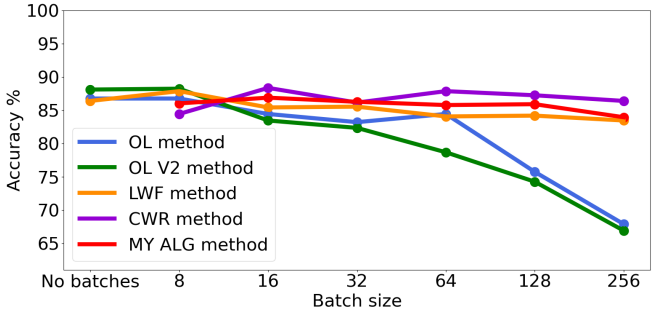

Figure 5.13: PLACEHOLDER

Chapter 6

Conclusion

LA BIBLIOGRAFIA NON FUZNIONA