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 **6**
1.1 Introduction to Machine Learning 6
1.2 Continual on- line learning 10
1.3 Cloud vs edge inference 16
1.4 Machine Learning on MCU 16
1.4.1 Pruning and quantization 18

2 Hardware **21**
2.1 Gesture recognition hardware 21
2.2 Image classification hardware 24
2.3 Machine learning support 26

3 System implementation **27**
3.1 Basic pipeline of the CL system developed 28
3.2 Algorithms implemented 31
3.2.1 TinyOL 32
3.2.2 TinyOL V2 34
3.2.3 LWF 38
3.2.4 CWR 40
3.2.5 My algorithm 43
3.3 Gesture recognition application 43
3.4 Image classification application 43

4 Experimental setup **44**
4.1 Dataset collection 44
4.1.1 Accelerometer dataset 45

4.1.2 Digits recognition dataset 46
4.2 Frozen model training and evaluation 48
4.2.1 Gesture recognition model 48
4.2.2 Image classification model 51
4.3 STM F401-RE setup for gesture recognition 53
4.4 OpenMV setup for image classification 55

5 Experimental results **61**
5.1 Experiment A: Gesture recognition 61
5.2 Experiment B: Image classification 61

6 Conclusion **71**

List of Figures

1.1 On the right an example of layers inside a ML model. On the left a scheme explaining how a neuron behaves. IMMAGINE NON DEFINITIVA 7
1.2 Summary of characteristics about supervised, unsupervised and reinforcement learning. IMMAGINE NON DEFINITIVA . 9
1.3 Block diagram showing an example of ML model structure: DNN, CNN and autoencoder. IMMAGINE NON DEFINITIVA 11
1.4 Venn Diagram showing the classification of some well known CL strategies. 13
1.5 Block diagram showing the steps of a pruning and quantization procedure. 19

2.1 Hardware used for the CL applications. On the left Nucelo ST32 F401-RE, on the left OpenMV camera. 22
2.2 Hardware used in the gesture recognition application. On the left the sensor shield IKS01A2, on the right the sensor shield mounted on the Nucleo STM32 F401-RE. 23
2.3 Hardware used in the image classification application. On the left the 3D printed tripod, on the right the OpenMV camera mounted on the 3D printed tripod. 26

3.1 Block diagram showing an example of model training 30
3.2 Block diagram describing the algorithm TinyOL. 35
3.3 Block diagram describing the algorithm TinyOL with batches. 35
3.4 Block diagram describing the algorithm TinyOL V2. 37
3.5 Block diagram describing the algorithm TinyOL V2 with batches. 37
3.6 Block diagram describing the algorithm LWF. 40
3.7 Block diagram describing the algorithm CWR. 42

4.1 Motion followed with the sensor for writing letters in the air. . 46
4.2 Stankey diagram showing how the letter dataset is divided . . 47
4.3 Example of images from the MNIST digits dataset 47
4.4 Stankey diagram showing how the MNIST dataset is divided . 48
4.5 Letter recognition model: basic structure and its separation in frozen model and OL classification layer 49
4.6 Results after the Tensorflow training. Top right: variation of accuracy and loss during training and validation. Top right: accuracy of the model in each class. Bottom left: table resuming precision accuracy and F1 score. Bottol right confusion matrix. 50
4.7 Image classification model: basic structure and its separation in frozen model and OL classification layer 51
4.8 Results after the Tensorflow training. Top right: variation of accuracy and loss during training and validation. Top right: accuracy of the model in each class. Bottom left: table resuming precision accuracy and F1 score. Bottol right confusion matrix. 52
4.9 Block diagram showing the communication between laptop and STM Nucleo. 56
4.10 Block diagram showing the communication between laptop and OpenMV camera. 58
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 59
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. 60

5.1 PLACEHOLDER 62
5.2 PLACEHOLDER 62
5.3 PLACEHOLDER 63
5.4 PLACEHOLDER 63
5.5 PLACEHOLDER 64
5.6 PLACEHOLDER 65
5.7 PLACEHOLDER 67
5.8 PLACEHOLDER 67

5.9 PLACEHOLDER 68
5.10 PLACEHOLDER 68
5.11 PLACEHOLDER 69
5.12 PLACEHOLDER 69
5.13 PLACEHOLDER 70

List of Tables

2.1 Nucleo STM F401-RE specifications 23
2.2 OpenMV H7+ specifications 25

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.

The Thesis is organized as follows. The first chapter contains an introduction to the theoretical aspects of Machine Learning (ML) and Continual Learning (CL). At first, basic concepts of Machine Learning are described, then the focus moves towards Continual Learning (CL) where also some state of the art papers are discussed. The chapter then describes some applications of ML on microcontrollers (MCUs) with also a brief explanation of advantages and disadvantages of cloud computing and edge computing. The second chapter briefly explains the hardware used in this study. The study uses two different hardware for two different applications. Therefore, the chapter initially describes the STM32 Nucleo MCU, and later focuses on the OpenMV camera. The chapter concludes with a short explanation of how ML can be applied on tiny devices. In chapter three the system implementation is described. Here, the steps performed for implementing CL training on MCUs are described, followed by an explanation of the basic structure of the TinyOL system. Then, all the algorithms implemented are illustrated in detail with also some considerations about memory and computational power. At last, the general idea of the two applications is demonstrated. The fourth chapter shows in detail the experimental setup. Here, itcan be found all the

information needed for replicating in detail the tests. This chapter describes: i) the collection of the dataset; ii) the training and evaluation of the frozen models; iii) the detailed execution of the tests using the MCU and a laptop. Chapter five contains the results obtained from the training. At first, it provides the detail about the comparison between training performed on a laptop and training performed on an MCU. Then, the the results from all algorithms applied in the experiments of gesture recognition and image classification are explained. The results contain information about accuracy, precision, F1 score, memory used and time of inference. In the last chapter conclusions about the work done and possible future implementations are discussed.

Chapter 1

Related Works

1.1 Introduction to Machine Learning

Machine learning is a branch of Artificial Intelligence (AI) that deals with the creation and training of models that have the ability to learn from data. This technology is a growing field of data science that is gaining popularity thanks to its flexibility to adapt to many problems. Machine learning models can be trained for different purposes such as regression of unknown systems, classification of data, predictions of data behaviour and artificial data generation.
An ML models is a file generated from a training procedure set up with specific characteristics. Training sessions are usually customized to make models learn tasks that can be applied in real world scenarios on data that have characteristics relevant to the training dataset. The simplest ML model, called Deep Neural Network (DNN), is composed of neurons grouped inside layers that are connected in series as displayed in Figure 1.1. The layers can be divided in: input layers, hidden layer and output layers. The input layer is the first of the model. Here the input array (or matrix) is inserted and each neuron of the layer is composed of a value from the input sample. The output layer is the last of the model, this contains the elaborated data. Depending on the use of the model and on its characteristics the output values can represent different types of information, such as bounding boxes positions, percentages representing classification, artificial generated data,... The hidden layers are all the layer that can be found in between the first and the last one. The type of hidden layer can change depending on the application and

the type of elaboration that is applied on the data. The model represented in Figure 1.1 iscomposed of only fully connected layers. This type of layer is characterized by neurons that are connected to every single node from the previous and following layers. Some examples of types are: convolutional layer, which are used to apply the convolution operation on images; pooling layers, which are used for extracting the most relevant features from an image while reducing its size; dropout layers, which are used for randomly nullify input values for avoiding overfitting; normalization layer, which are used for scale the input data to suitable intervals with the aim of removing bias.

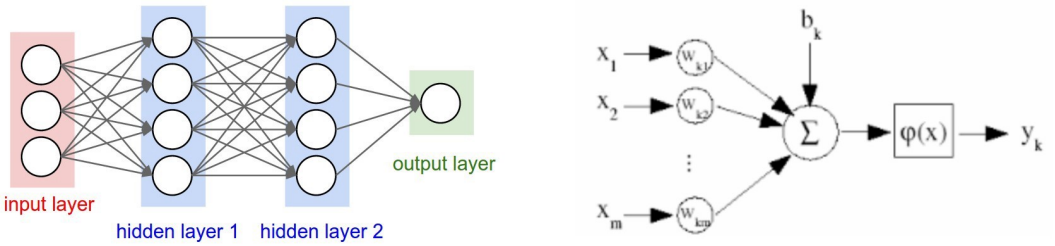


Figure 1.1: On the right an example of layers inside a ML model. On the left a scheme explaining how a neuron behaves. IMMAGINE NON DEFINITIVA

An important role is played by the smallest element in a ML model, the neuron. This component is the one responsible for the evolution of the network and for its learning and adaptation abilities. A neuron, or node, can be seen mathematically as a function that takes many inputs and generates one single output. As shown in Figure 1.1, one node takes all the inputs, multiplies each one for a custom weight, sums them together and adds to it a biased value. Then the result of the addition is fed through an activation function which is used for introducing a non linearity in the model. The type of activation function is usually defined depending on the application of the model. Some well known functions are Softax, ReLU, Sigmoid, Linear, and Hyperbolic Tangent. The most common activation functions in CNN and DNN models are Softmax and ReLU. The Softmax activation function is specifically used in classification layers because it has the ability to convert the values of an array into values that sum up to 1. This function is particularly useful in classification applications because it can be seen as a function that maps an array of values to an array of probabilities. Where each probability is a measure of how confident the model is in assigning an

input to a class. The ReLU activation function, on the other hand, is used only on the hidden layers. Its use is typically preferred to Hyperbolic Tangent or Sigmoid because it introduces less complexity in the computation of the Stochastic Gradient Descend (SGD) and avoids a problem known as Vanishing Gradient [**ReLU˙explanation**].
ML trainings can be categorized in several groups depending on how the training is performed. The main groups are:

- Supervised learning: a type of training where the model is provided with input samples and the desired output. The model is trained to yield the same outcome as the one provided by using a feedback for correcting weights and biases. Supervised learning is commonly used in classification problems (when a model is trained to categorize data in groups) and in regression problems (when a model is trained to find the relation between variables).

- Unsupervised learning: a type of training where the model is provided with only input data. During training the model, by himself, learns to find patterns of the data and is able to categorize them in groups. This is commonly used in problems where the user is unsure about the properties of the dataset.

- Reinforcement learning: a type of training where the model is provided with input data and rewards when outcomes are correct. The goal of the model is to elaborate the input data and compute an output, depending on the rewards obtained the model learns from its mistakes and tries to maximize the reward that it receives from future steps. This type of training is commonly used for minimization of problems such as industrial automation (minimization of robot paths), data processing, and playing chess games.

- Transfer learning: is not a real type of training, but more a technique that is used to transfer already gathered knowledge. This method takes an already trained model and fine-tunes its last portion. The idea usually is t use state-of-the-art models that are trained for generic purposes and fine tune them to be applicable in more specific applications (example: use a state-of-the-art model for object identification and fine tune it to recognize humans in infra-red images)

Figure 1.2 shows the main characteristic of the different training types aforementioned.

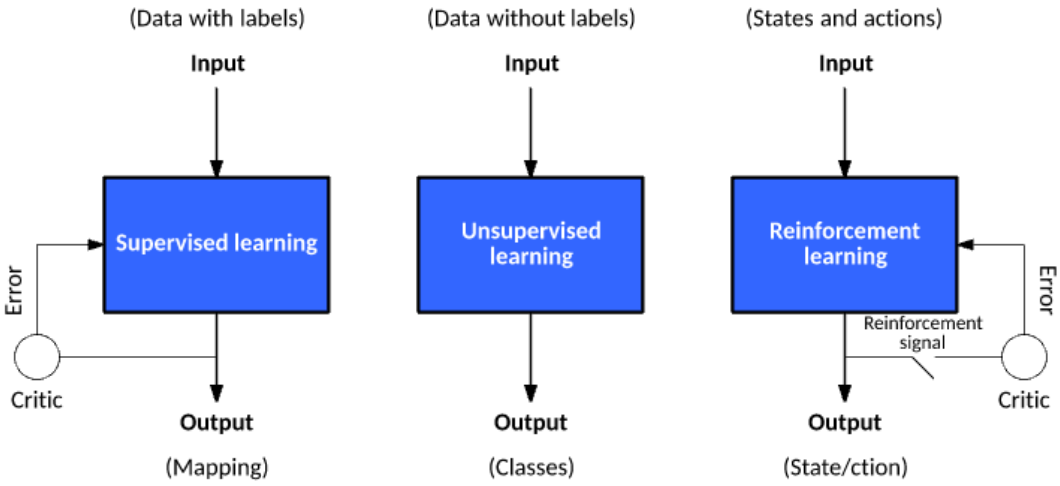


Figure 1.2: Summary of characteristics about supervised, unsupervised and reinforcement learning. IMMAGINE NON DEFINITIVA

A training step is the process that allows a model to change its weight and make them converge toward an optimal state where the error committed by the prediction is minimized. An example of training step is the following. The model receives an input sample and propagates it through all the hidden layers. Once the data reaches the output layer, a prediction is computed and later compared with the ground truth label (if the training is supervised). A loss function is used for computing the error performed by the prediction and then the back propagation is performed. This procedure consists in correcting all the weights and biases of all previous layers with the aim of minimizing the error of future predictions. If the training is performed on a big enough and variegated dataset, the parameters should be now in a condition that allows the model to consistently make predictions that satisfying predictions.

Depending on the types of layer used inside a ML model, several model types can be defined. Different structures can be exploited for different applications, some examples of basic models are DNN, CNN and autoencoders. Models that are composed of only fully connected layers, also called Deep Neural Networks (DNN) are well suited for regression and classification of data. These layers are capable of performing elaboration of data simply by

relying on the use of activation functions and correction over the weights and biases. Layers known as convolutional layers, are well suited for the elaboration of images. They are based on the use of filters that move over an image and perform the convolution operation. This is one of the basic procedure used in image processing for the extraction of features from matrices of data (like pixel values inside an image). Usually these types of layers can be found int the initial portion of a Convolutional Neural Network (CNN) model. These models are usually divided in three part. The first is composed of multiple convolutional layers used for feature extraction. This is followed by flattening layers that are used for transforming matrices in arrays. The last portion is composed of fully connected layers, so basically the third part is a DNN that applies classification elaboration over the features extracted from the images.
Other types of models are autoencoders. These models can be seen as generative models, since their role is to behave as constructor of lost data. An autoencoder is composed of two groups called Encoder and Decoder. The Encoder is just a sequence of fully connected layers with decreasing size, while the Decoder is a sequence of fully connected layers with increasing size. The goal of these types of models is to take an input samples, compress it into a smaller size, in what is called bottleneck, and then reconstruct the compressed array into the original values. If a training is done correctly, the Encoder can be discarded, and the Decoder part can be used as an artificial generator of data.
Figure 1.3 shows some examples of ML model structures aforementioned.

1.2 Continual on- line learning

Applications of machine learning are based on the use of models for the elaboration of data. An important aspect regarding standard ML approaches is the possibility of these models to perform only inference on inputs, this limits the scenarios in which ML applications can be deployed. The models are not capable of adapting to changing context. Context that make up the vast majority if scenarios in which TinyML is employed. To overcome this problem, machine learning strategies can be enhanced with the addition of Continua Learning (CL).
Continual learning is a paradigm of machine learning where models are trained in real time with streams of data presented sequentially. The main

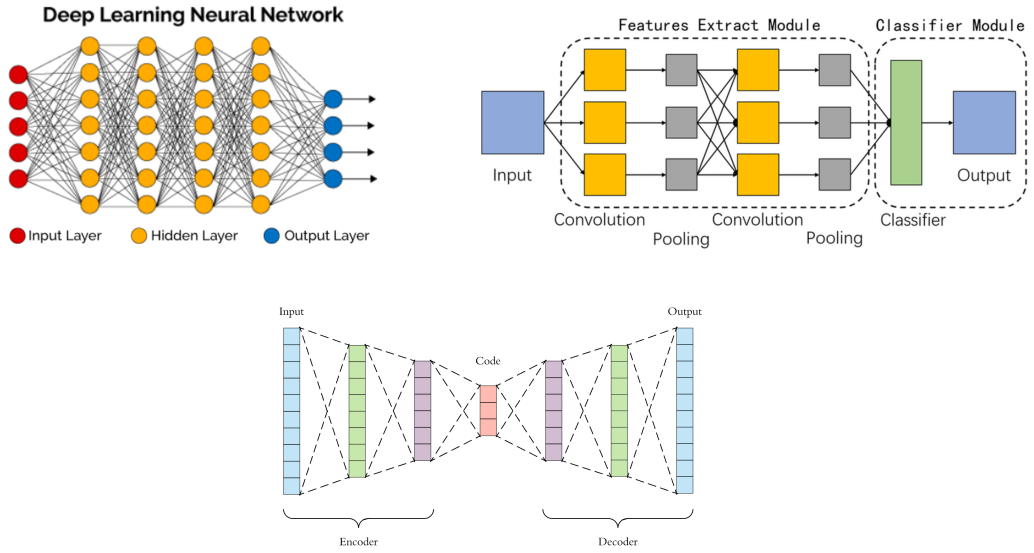


Figure 1.3: Block diagram showing an example of ML model structure: DNN, CNN and autoencoder. IMMAGINE NON DEFINITIVA

characteristic of this method is the introduction of the ability of models to adapt to drift and changes in the context. It is often the case that ML models are trained in ideal and controlled scenarios where static datasets are used. This is not an optimal situation because the models are trained to behave in specific ways only in certain scenarios. By later using these models in real life applications characterized by context drift, it is common to find that models drop in accuracy. These implementation of rigid models trained on specific tasks osmsub optimal, this is the main reason why CL strategies are applied. The main characteristics brought by CL are the ability of models to train in real time over sequential streams of data and the ability of models to recognize new classes and add them to the pool of possible classification groups. These two abilities make for self sustainable models that are able to both fine tune their structure and parameters and at the same time learn new tasks. An important classification that can be applied on data streams is the one proposed in study [], which is the following:

- NI: New Instances, data samples contain new patterns of already known classes (example of image application: classification is done on dogs and cats, inputs present different illumination or different background)

- NC: New Classes, data samples contain data regarding never seen classes. The model is not able to recognize in a different label these kind of samples. (example of image classification: classification is done of dogs and cats, inputs presents images of ducks)

- NIC: New Instances and Classes, data samples containing both patterns of already known classes and samples regarding new classes are present in the scenario. This is the most complete scenario, which also better represents real world applications.

The implementation of CL strategies allows for improvements and enhancement of ML capabilities but it comes with additional problems and challenges. One of the main aspects to be aware of is catastrophic forgetting. Catastrophic forgetting [] is a phenomenon that occurs when a model, trained in real time, starts to drift from the original context, making the model forget the original knowledge in spide of new tasks. This problem is directly correlated to the real time training that tries to minimize the loss function at every step with no constraint. Keep in midn tha ttraditional trainign strategies are applied on batches, and the back propagation that is applied on weight and biases, used for training model is usually dependent on information coming from the batch itself and not single input samples. On the cintrary, in a standard real time training, the back propagation is computed from just a step, so the update to apply on weights depends only on the last sample. This means that the information obtained from the last sample can increase the classification abilities of the model for the recognition of that specific class, but it will destroy/deteriorate the classification abilities of all the others. It is then of extreme importance to develop strategyes that are able to maintain control over this aspect. In today's research many algorithms have been developed with the aim of performing CL with a look over catastrophic forgetting. Summary [] contains a good review of the most relevant and best performing algorithms.
Before speaking in detail about strategies it is important to categorize them. In paper [] and [] the following categorization of CL strategies are proposed:

- 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.

Figure **??** contains a Venn Diagram showing examples of the most known algorithms and their classification.

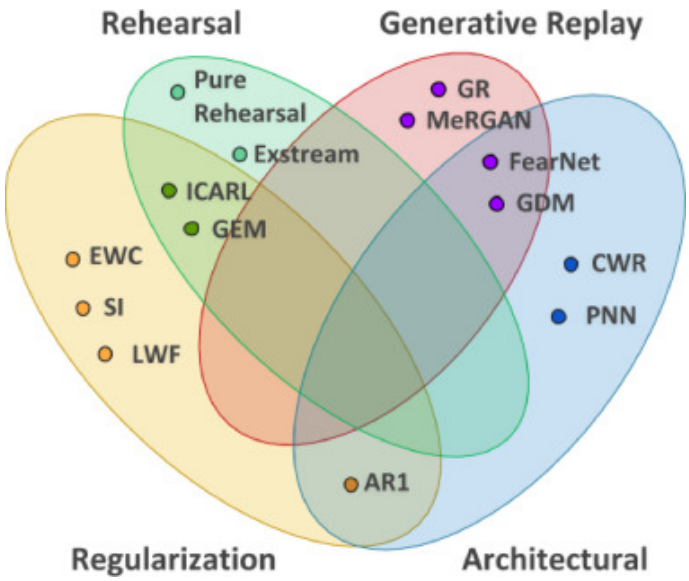


Figure 1.4: Venn Diagram showing the classification of some well known CL strategies.

Note that the most suitable strategies for applications on MCUs are the regularization strategies. This because the strategies rely on the use of par-

ticular loss functions and they do not require any complex use of additional memoryes or compoutations.
Continual learning is an already known and studied topic in the filed of ML. Recently it started to gain attention because of its implememntation of MCU, which opens a new field in TinyML. An important study related to general application of CL is described in []. Here the author's goal is to implement several algorithms and test them to evaualte their ability to overcome catasatrophic forgetting. The study applies already known strategy like LEF [], SI [], EWC [] and it later proposes two custom made strategye called CWR and AR1. Note that CWR+ is an improvement of an already proposed algorithm developed in a previous paper by the same authors.
Another interesting study, not directly realted to algorithms, is the one described in []. Here the authors developed an easy to attach system that aims at balancing real time streams of data. From their point of view, real life applications of CL strategies, mainly for IoT devices, are cìnot characetrized by well balanced streams of that. This means that the input stream is not composed by the same amount of samples for each task, and each task is not composed of same classes. The authors propose a small plugin to be attached in between the raw input stream and the CL model that is able to apply pre processing procedure on the data and make the input stream of data balanced.
One of the most relevant works for this thesis is the paper []. Here the author developed a small framework for an Arduino microcntroller, that is able to perfomr a simple CL (or on line learning) strategy on a ML model with an autoecnoder structure. The system aims at fine tuning the pre trained model and enhance the pool of recognizable classes. The study aplies succesfully the system for the classification of vibration patterns from a PC fan and later improves it with a real application on a industrial machine in paper [].

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 last years especially thanks to the potential demonstrated by applications in several fields such as industrial application, agricultural automation, human-computer interactions, autonomous driving. The use of TinyML allows to introduce the concept of edge computing, where computations are brought

closer to the origin of the data, where devices until now were used only as collectors and transmitters of data. Edge computing brings 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 Internet of Things 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 ML is a computationally heavy procedure it's necessary to be able to optimize it. Stydyes such as [] and [] focused on theis specific characteristic. Their main goals was to develop frameworls for the efficient application of machine learning inference.

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.

1.3.1 Pruning and quantization

When deploying ML models on constrained device there are two main concerns, which are the available memory and the available computational power.

To fight against these two limitations different techniques have been developed. Pruning and quantization are pre processing methods that can be employed on already trained model to reduce their memory footprint and their computational weight. Direct benefits of these strategies are lower power consumption, lower memory bandwidth, less storage required, all with the same performances.
Pruning is a strategy that is adopted after the training of a model. A well trained model is required because, at each step of a pruning procedure, the model of interest is de gradated and so its performances. In most cases, the model developed for the application is bigger than actually required. This is due to the initial definition of a general structure and the following training that touches all the parameters inside the model. The pruning method relies on the fact that, in a ML model, a some parameters are redundant, not relevant or useless for the application. In fact the strategy is to completely remove the connection between neurons that are less important for the application. The goal is to reduce the memory required by the model and the amount of computations for an inference. A pruning step is composed of the following actions: evaluate the importance of each weight based on their influence on the network output and performance, prune the less important weights (bring to 0 their value), re train the model that inevitably lost performance, check if the memory reduction induced is enough, if not perform another pruning step. Figure 1.5 contains a block diagram showing the procedure of a pruning strategy.
An additional improvement brought by pruning is the reduced amount of computations. If done strategically, a pruning process can introduce sparsity (a localized population of zero values) in the weight and bias matrices. If the sparsity is injected with a smart structure, it is then possible to reduce the amount of computations, thus improving the efficiency of inference in terms of time, energy and complexity.

Another method that is often used for compressing ML models is quantization. Quantization is a process that aims at reducing the *bits per weight* metric. The process can be of two types: post-training quantization or training aware quantization [**pruning˙article**]. These two procedures both aim at reducing the amount of memory dedicated to each parameter, but the process for doing so is quite different. In the first case, post-training quantization, the procedure consists in a rounding or type conversion of weights and biases. Common strategies rely on the conversion of the values from

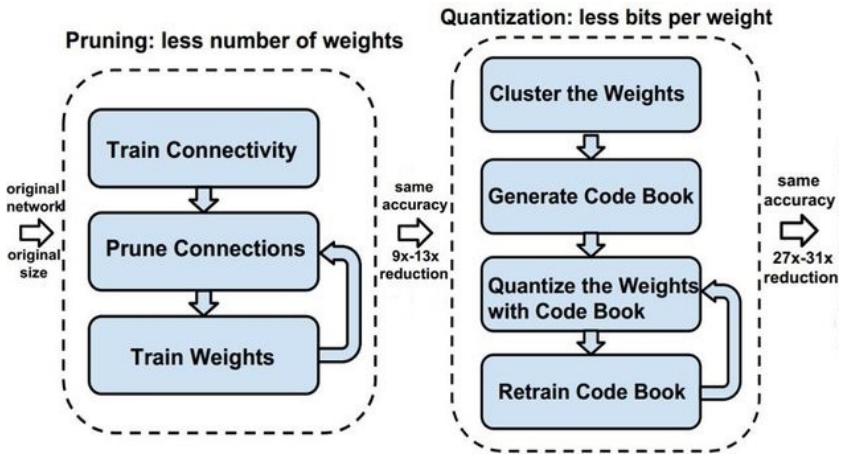


Figure 1.5: Block diagram showing the steps of a pruning and quantization procedure.

floating point 32 to *integer* 8, which is able to reach a compression ratio of $\times 3$. The second method, quantization aware training, is more complex and requires the implementation of quantization strategies that are performed while training. This procedure usually reaches better results but with the added complexity of using a different training procedure. For typical MCU applications, the simple use of post training quantization yields acceptable results both in terms of compression and small accuracy drop.
The implementation of such procedure is quite immediate thanks to the help of libraries like Tensorflow [**tensorflow˙quantization**], which allow to perform the entire procedure with just a couple of lines. Figure 1.5 contains two sequential block, with the most important steps, where pruning and quantization are performed on a ML model.

1.3.2 Cloud vs edge inference

Continual learning is a type of machine learning that is able to overcome context drift. This is an aspect that often characterizes scenarios where TinyML is used. By having devices and models that are deployed for indefinite times in particular places, it is to be expected that the environment in which they lye is subjected to a natural and continuous evolution. This makes the use of CL a great solution that allows models to be self adjusting to these changes and self adapting to new tasks. By using CL strategies in TinyML it is possi-

ble to reduce the maintainance to the devices that would be required because of drop in accuracy.

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, but it requires devices that are capable of sustaining those kinds of computations. In today's market, lots of off-the-shelf MCUs are already equipped with hardware components that make the device suitable for the job. These microcontrollers only need to be correctly equipped with framework and tools that optimize ML computations. In this study, for both applications, devices based on STM microcontrollers were used. The gesture recognition application uses an STM32 Nucleo F401-RE [**nucleo˙datasheet**], a well performing and easy to use development board. The image classification application uses an OpenMV camera [**openmv˙datasheet**], a device equipped with a camera sensor that features an STM32 H7 MCU and is programmable in MicroPython. Figure 2.1 shows both devices: on the left the Nucleo F401-RE, and on the right the OpenMV camera. The reasons that brought to the selection of these two MCUs depends mainly on the quick availability for the Nucleo development board and the uniqueness of characteristics for the OpenMV camera.

2.1 Gesture recognition hardware

The gesture recognition experiment is carried out with a Nucleo STM32 F401-RE. This device is a 64 bit microcontroller produced by STMicroelectronics that belongs to the $high - performance$ product line. This product is thought to be a flexible and easy to use development board for fast pro-

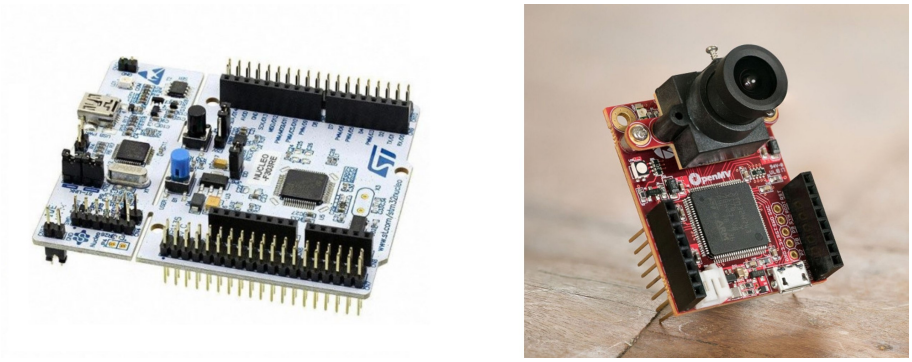


Figure 2.1: Hardware used for the CL applications. On the left Nucelo ST32 F401-RE, on the left OpenMV camera.

totyping, which is also compatible with Arduino Uno shields (the internal GPIOs pin scheme is the same as Arduino Uno). Differently from other devices produced by STMicroelectronics, this MCU does not require the additional debugger/programmer ST-LINK. This component is included in the first part of the device, separated from the rest with visible gaps in the board. The Nucleo can be easily programmed in C or C++ with the the libraries and tools available in the STM32Cube package. This is a powerful software that can be exploited to perform an initial setup of any STM device, such as the definition of all peripherals and basic microcontroller parameters. The STM32Cube package also includes libraries for any device produced by STM. Moreover, using STM32Cube, the Nucleo board STM32 F401-RE fully supports the machine learning extension, namely STM-CUBE-AI [**stm˙cube˙ai**]. This toolkit can be used for compressing and loading ML models on the device and later perform optimized and efficient inference in real time.
The main features of the Nucleo development board are summarized in Table 2.1.

The gesture recognition application uses ML algorithms for the analysis of time series data. The time series array contains the values recorded from an accelerometer sensor while an user is moving that sensor to write letters in the air. To acquire that kind of data, the Nucleo needs to be equipped with an accelerometer sensor shield. For doing that, it was decided to use the Nucleo shield IKS01A2 [**shield˙web˙page**], a device that can be mounted

Table 2.1: Nucleo STM F401-RE specifications

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,

easily on the board simply by aligning the GPIO pins. The shield is equipped with a 3D accelerometer, a pressure sensor, a capacitive digital humidity, and temperature sensor. The device communicates with the Nucleo through I2C protocol and is fully supported by the libraries available on STMCube, which make its use quick and easy. Figure 2.2 shows the two devices separately on the left and mounted on the right.

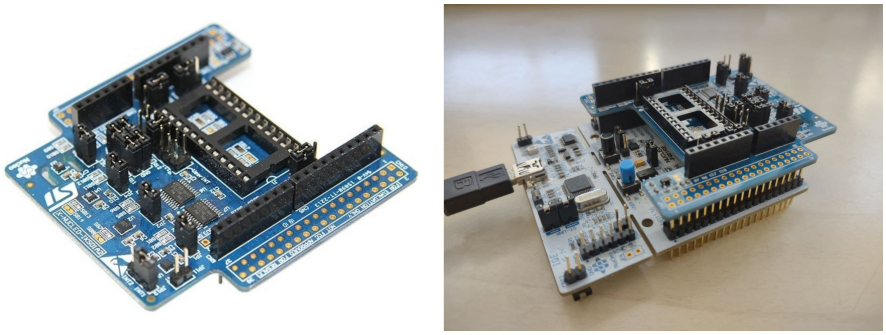


Figure 2.2: Hardware used in the gesture recognition application. On the left the sensor shield IKS01A2, on the right the sensor shield mounted on the Nucleo STM32 F401-RE.

2.2 Image classification hardware

For the image classification 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. Then, the idea has 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 which permit the camera to interact with external devices such as servo motors, sensors or other MCUs. The GPIOs can be exploited for controlling robots, drones, and machine learning systems. 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**].
Moreover, the camera comes 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 and sensors, telephoto lens, super telephoto lens, ultra wide lens, WI-fi shield, LCD shield, and motor shield. The possibility to add these different components or modification make the device a very well suited solution for many problems, application, and especially prototyping .
Table 2.2 summarizes the device specifications.

Because the application of this study requires a camera pointing to a screen, a small 3D printed support was created. An already available project from Thingiverse [**tripod˙link**] was modified to make the mounting of the OpenMV camera possible. 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.

Table 2.2: OpenMV H7+ specifications

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

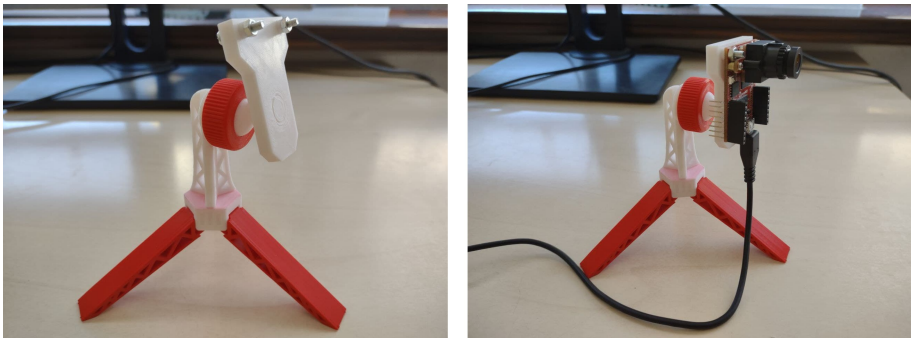


Figure 2.3: Hardware used in the image classification application. On the left the 3D printed tripod, on the right the OpenMV camera mounted on the 3D printed tripod.

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 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

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 the Introduction, continual learning can lead to many improvements if exploited correctly in IoT applications. The main feature of Continual Learning (CL) is the ability of ML models to modify weights and biases to learn from the environment and continue to be relevant in the application. The application of CL on MCUs is still a quite new idea that only gained popularity in recent years. Applications regarding CL have already been explored before, but their applications on tiny devices is still pretty new. Some relevant studies that applied CL on MCU are [**ren2021tinyol**], [**ren2021synergy**] and [**sudharsan2021train++**].
In this thesis, the idea was to develop a CL system similar to what was proposed for the TinyOL system [**ren2021tinyol**]. The paper proposes a light framework, to be deployed on an Arduino, that can be attached to a pre-trained model for the application of on line learning (or continual learning). The goal is to classify vibration patterns of a PC fan and be able to add dynamically new classes of vibrations when detected. The study shows that such a method is feasible and leads to good performances even on constrained devices. The CL strategy takes place only in the last layer of the model, which is added to the pre-trained model and is trained every time a new sample is received.
The idea developed in this thesis started from the implementation of a similar system. The idea was to develop a system that allows a NN or CNN model to perform continual learning, specifically for classification problems. In this implementation the OL system is not connected to the last layer of

the model, but it completely substitutes the last layer of the pre-train model. The continual learning is then applied on this last portion, which allows to enhance the classification abilities. In the following section, the creation, basic pipeline, and application of the new TinyOL system are explained.

3.1 Basic pipeline of the CL system developed

TinyML always featured application regarding only efficient and optimized inference. To apply CL, or real time trainings, on embedded devices it is necessary to develop from scratch frameworks that permit the implementation CL strategies. For this study, two off-the-shelf hardware are used and one framework is created. At first the framework was developed on a powerful device and later it was adapted to be loaded on two MCUs. The framework requires to be attached as the last layer of a pre-trained model, so it should always be paired with hardware that support toolkits for ML inference in real time.
As mentioned before, the main idea was to develop a system starting from the TinyOL framework. My version aims at enhancing the classification abilities of the model, allowing to: i) fine tune and update the classification weights and biases, which, over time, lead to better performing models that better follow the context drift; ii) enlarge the size of the classification layer, adding the ability of the model to recognize new classes, thus improving the flexibility of the model and reducing the necessity to perform maintenance. The applications seen here are always in supervised settings. This means that at every training step the ground truth label is known and is provided to the algorithm, which then uses it to compute the prediction error and perform back propagation for the parameters update.
The basic idea of the developed system is to refresh and update the weights every time a new samples is received. This can be performed by implementing the standard ML training strategy, which consists in computing the prediction error and propagating it back to the weights. The update to apply on the weight and biases depends directly on the influence that the specific weight or bias had on the prediction outcome. To compute this influence it is required to know the details of the model structure (connection between nodes, activation functions, loss function). Once this knowledge is available,

it is possible to compute a feedback rule that allows to find the weight that better optimizes the loss function. In the applications seen in this study, the idea was to apply CL on classification problems. IN typical NN and CNN models, the classification operation is performed in the last layer, which uses a $Softmax$ activation function. This fucntion is specifically used in these applications because its definition allows to obtain a distribution of percentages (that sums up to 100) from an array of values. By having such specific applications, it is possible to compute always in the same way the rules required for the application of CL on such models.
Now consider a simple and short model composed of few fully connected layers, all with size 10. Consider also that the classification layer (the last one) is composed of 5 nodes with Softmax as activation function. Since the application of CL in this study wants to train only the last layer, the part of the model before the last layer can be considered as a grey box (or frozen model) that outputs an array of 10 values. It can be said at this point that the update rule required for the real time training depends on: the Softmax function, the output of the grey box, the loss function, and the error between prediction and label. Figure 3.1 contains a block diagram showing the model situation just described.

The basic formulas used for the propagation of errors in a classification layer are the following:

$$z_i = \sum_i w_{ij}\, x_j + b_i \tag{3.1}$$

$$y_i = Softmax(z_i) = \frac{e^{z_j}}{\sum_j = 1^k e^{z_j}} \tag{3.2}$$

$$L_i^{CROSS} = -\frac{1}{n}\sum_k [\; y\, ln(y_i) \;+\; (1 - t_i)\, ln(1 - y_i)] \tag{3.3}$$

$$where\;\; i = 0, 1.., n \;\; and \;\; j = 0, 1, .., m$$

Where equation 3.1 is the formula that describes the propagation of inputs though a ML node, equation 3.2 is the definition of the Softmax function, equation 3.3 is the definition of the categorical cross entropy loss function, n is the number of classes known by the mode and m is the number of outputs of the grey box (or number of nodes of the frozen model's last layer).
Knowing the entire mathematical description of the classification layer, it is

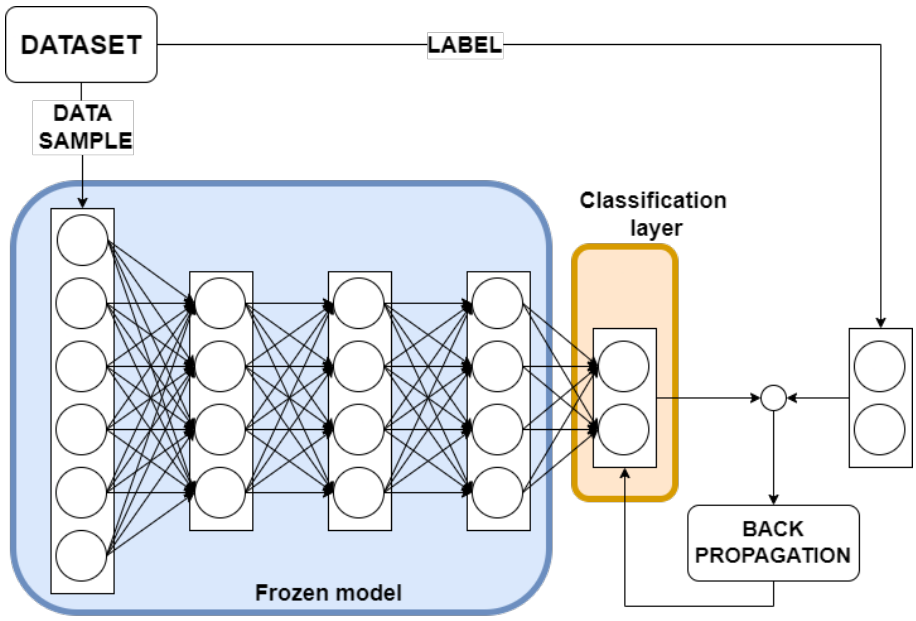


Figure 3.1: Block diagram showing an example of model training

possible to compute the relation between weights and prediction error. To do so, it is necessary to compute the derivative of the error with respect to the weight of interest.

$$\frac{\partial L_i^{CROSS}}{\partial w_{ij}} = \frac{\partial L_i^{CROSS}}{\partial S_i} \cdot \frac{\partial S_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}} \tag{3.4}$$

Each block is then computed as:

$$\frac{\partial z_i}{\partial w_{ij}} = \frac{\partial(\ \sum_i w_{ij}\, x_j + b_i)}{\partial w_{ij}} = x_i \tag{3.5}$$

$$\begin{aligned}\frac{\partial S_i}{\partial z_i} &= \frac{e^{z_i} \sum e^{z_i} - e^{z_i} e^{z_i}}{(\ \sum e^{z_i})^{\ 2}} \\ &= \frac{e^{z_i}}{\sum e^{z_i}} - \frac{(e^{z_i})^2}{(\sum e^{z_i})^2} \\ &= Softmax(z_i) - (Softmax(z_i))^2 \\ &= Softmax(z_i) - (1 - Softmax(z_i)) \end{aligned} \tag{3.6}$$

$$\tag{3.7}$$

The steps required for the computations of the last block are a bit more complex:

Then it is immediate to apply the relation weight-error inside a feedback loop for the correction of the parameters. The final formula that defines the back propagation on weights and biases of a classification layer that uses Softmax as activation function and categorical cross entropy as loss function is:

$$w_{ij} = w_{ij} - l_{rate} \cdot (\ y_i - t_i)\ \cdot x_j \tag{3.8}$$

$$b_i = b_i - l_{rate} \cdot (\ y_i - t_i) \tag{3.9}$$

Where w_{ij} is the weight in the position i,j of the matrix, b_i is the i-th bias of the array, l_{rate} is the learning rate, which is a parameter that defines how fast the correction modifies the weight, y_i is the i-th value inside the array generated by the frozen model and t_i is the i-th value of ground truth label. Note that the label in this case is a hot-one encoded array. This means that the array is filled with zeros and contains a 1 only in the position that represents the correct class.

3.2 Algorithms implemented

The study started from the implementation of this strategy for performing CL. Later the study was expanded and other state-of-the-art methods were

applied. As explained in Chapter 1, regularization approaches are strategies that exploit the addition of loss terms in the update rule. Thanks to this, it is possible to have some control over the weight update. In today's research some strategies for CL training have been proposed mainly with the aim of contrasting catastrophic forgetting, but these were never applied on MCUs. The best performing state-of-the-art regularization strategies are Elastic Weight Consolidation (EWC), Synaptic Intelligence (SI) and Learn Without Forgetting (LWF) [**li2017learning**]. In another paper the authors propose another method called Copy Weight with Reinit (CWR) [**lomonaco2017core50**], which was later improved with the CWR+ version in paper [**maltoni2019continuous**], where also a new method, called AR1, was proposed. In this study only some of the aforementioned strategies were implemented, namely LWF, CWR and TinyOL, together with small variations that aim at using batches of data.
In these applications, the algorithms are all applied with the same system. The framework substitutes the classification layer of a pre-trained model, called frozen model. The frozen model is used only as a feature extractor which provides only an array of elaborated data. All the strategies implemented are applied only on the classification layer.

3.2.1 TinyOL

The TinyOL method applied on MCU, as already mentioned, has been initially implemented in paper [**ren2021tinyol**]. The strategy is straight forward and it follows the basic ML training step, which consists in computing the error from the prediction and propagating it on the weights and biases through stochastic gradients descend (SGD). Its implementation consists in a couple of for loops for updating the biases and the weights. This said, the weights update rule for this algorithm are:

$$w_{i,j} = w_{i,j} - \alpha(y_i - t_i) \cdot x_i \tag{3.10}$$

$$b_i = b_i - \alpha(y_i - t_i) \tag{3.11}$$

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

Where y_i is the i-th value in prediction array obtained from the OL layer, t_i is the i-th value of the true label array, 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 all strategies implemented, the value n can change dynamically since the maximum amount of possible classes is not known a priori, or at least it is not known in a real life application.
Also a variation of this method has been implemented. The variation takes into consideration the possibility to apply a back propagation that depends on a group of samples (a batch) and not just from 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 the batch size has on ML training dynamics. The base strategy of the variation is to compute a back propagation that depends on a batch of inputs and not just from the last sample recorded. This helps the model to be less vulnerable to noisy data and outliers. To implement an algorithm with this variation, it is necessary to maintain memory of all the samples from the batch. This requires the allocation of double the amount of memory used for the standard version. This is done by allocating two additional matrices, one called W, which contains the data of old samples related to the weights, the other is called B, which contains the data of old samples related to the biases. These matrices are used as a cumulative memory of the back propagation applied by each training step. Every time a new sample is received and elaborated, the upgrade for each weight and bias is computed and added in the correct spot inside matrices B and W. When a batch is finished, the average back propagation update is computed and applied on the actual matrices of weights and biases. Note that during a batch the OL system performs an inference with the frozen model's output by using matrices w and b, computes the error, computes the back propagations to be applied on W and B, adds the updates in matrices W and B. During an entire batch the weights used for inference are kept constant. When a batch is finished the average is computed and the update rule is applied. Then contents of W and B are deleted and restored to 0.
So the update rule at each inference step becomes:

$$W_{i,j} = W_{i,j} + \alpha(y_i - t_i) \cdot x_i \tag{3.12}$$
$$B_i = B_i + \alpha(y_i - t_i) \tag{3.13}$$

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

$$w_{i,j} = w_{i,j} - \frac{1}{batch\_size} \cdot W_{i,j} \tag{3.14}$$

$$b_i = b_i - \frac{1}{batch\_size} \cdot B_i \tag{3.15}$$

Both methods are based on the same basic principle which is quite simple. The method TinyOL requires the use of only one weight matrix and one bias array and 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 frozen model's last layer, represented by the value m. This makes the memory allocated from the method be equal to a total of $(n \times m + n \times 1) \times 4\, bytes$. The method changes the layer parameters each time a new sample is received, with no constraints. This aspect is the main problem that concerns the TinyOL strategy and makes it vulnerable to catastrophic forgetting. By allowing such a high flexibility to adapt to any kind of data, the model is not protected at all from catastrophic forgetting. The model learns from every single sample that it receives, no matter if the sample is noisy, an outlier or wrongly labelled.
On the other hand the method TinyOL with mini batches exploits the same approach but applies a back propagation that is dictated by the average update 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.
Figures 3.2 and 3.3 contain block diagram showing how the two methods behave.

3.2.2 TinyOL V2

The TinyOL V2 algorithms is based on the same idea of the original TinyOL. A little intuitable modification was applied in the method with the aim of contrasting catastrophic forgetting. The idea is to contrast the drift that affects the original weights by completely removing the possibility to update those weights. The algorithm applies the same rule seen before, but only on the parameters that represent new classes. The update rules become:

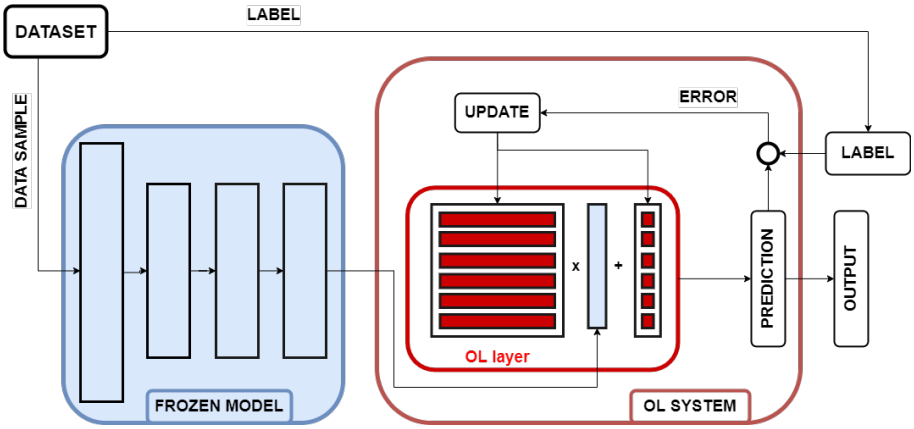


Figure 3.2: Block diagram describing the algorithm TinyOL.

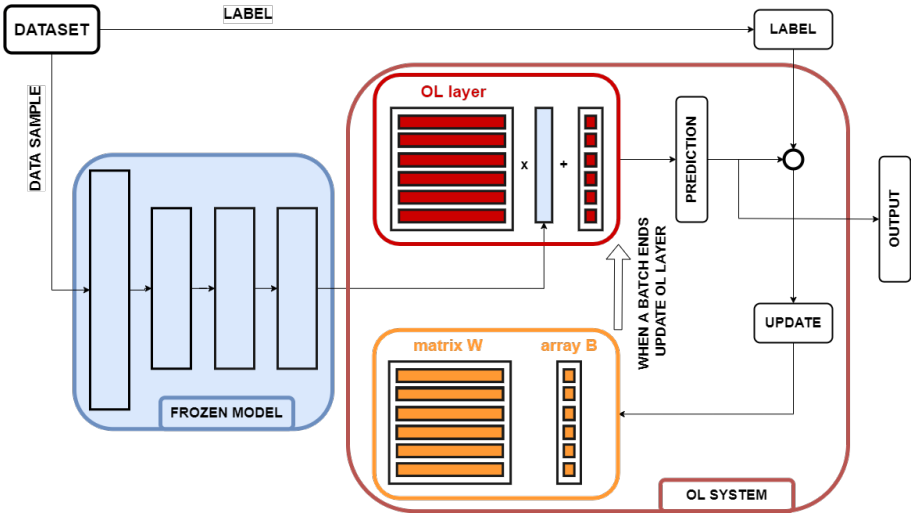


Figure 3.3: Block diagram describing the algorithm TinyOL with batches.

$$w_{i,j} = w_{i,j} - \alpha(y_i - t_i) \cdot x_i \tag{3.16}$$

$$b_i = b_i - \alpha(y_i - t_i) \tag{3.17}$$

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

The only difference with respect to the TinyOL method is the iterator i, which goes from p to n, where p represents the position of the first unknown

class.
Also in this case the variation that uses batches was implemented. Again the method requires the use of two additional matrices called W and B, which contain the cumulative back propagation computed at each step. As before, the algorithm follows the same rules as TinyOL with batches, but with the iterator i going from p to n. The rule that defines the standard behaviour during training is:

$$W_{i,j} = W_{i,j} + \alpha(y_i - t_i) \cdot x_i \tag{3.18}$$
$$B_i = B_i + \alpha(y_i - t_i) \tag{3.19}$$

And at the end of every batch the weights and biases are updated using the average back propagation saved in matrices W and B.

$$w_{i,j} = w_{i,j} - \frac{1}{batch\_size} \cdot W_{i,j} \tag{3.20}$$
$$b_i = b_i - \frac{1}{batch\_size} \cdot B_i \tag{3.21}$$
$$where\ \ i = p, p+1.., n\ \ and\ \ j = 0, 1, .., m$$

As seen for the TinyOL with batches, TinyOL V2 with batches allows the model to learn from a bigger group of samples. This ability should help the model to avoid over fitting, outliers, and noisy data.
In conclusion, TinyOL V2 is a simple method that differs from the original strategy only because the update is restricted to the new parameters. By forcing the update on just a portion of the weight and biases, the context drift that would irreversibly modify the original weights, thus forgetting the original knowledge, is completely removed. This helps the algorithm in contrasting catastrophic forgetting but also reduces the ability of the model to perform fine tuning on those classes. Another negative aspect regards the general behaviour of the model. By having a training strategy that updates only a portion of weights, it is not possible to create a model that behaves as optimizer of the loss function. This means that at the end of the training the model is composed of two parts that behave differently at every iteration. One portion of the weights behaves as the original model, while another part of the weights behaves as the most recent version of the model. These two part, when computing a prediction, cannot make the model converge towards

an optimized prediction.
The method TinyOL V2 requires the same amount of memory 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 with reduced size of $(n-p) \times m$ and $(n-p) \times m$. Figures 3.4 and 3.5 contain block diagram showing the pipeline of the two methods.

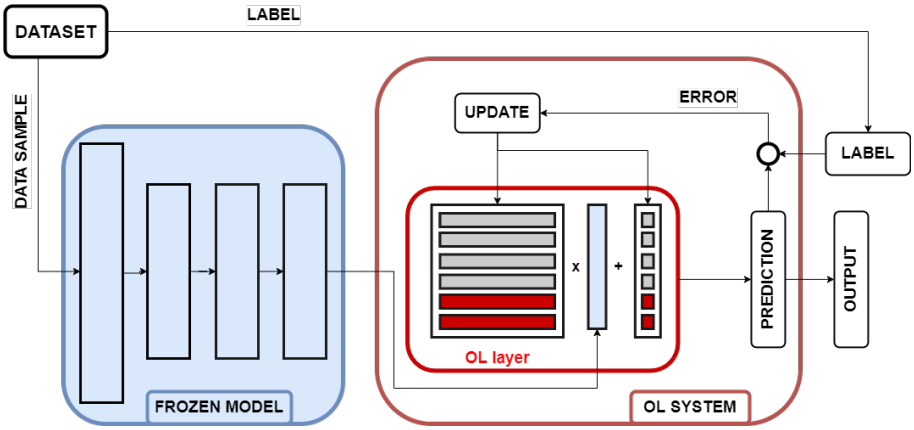


Figure 3.4: Block diagram describing the algorithm TinyOL V2.

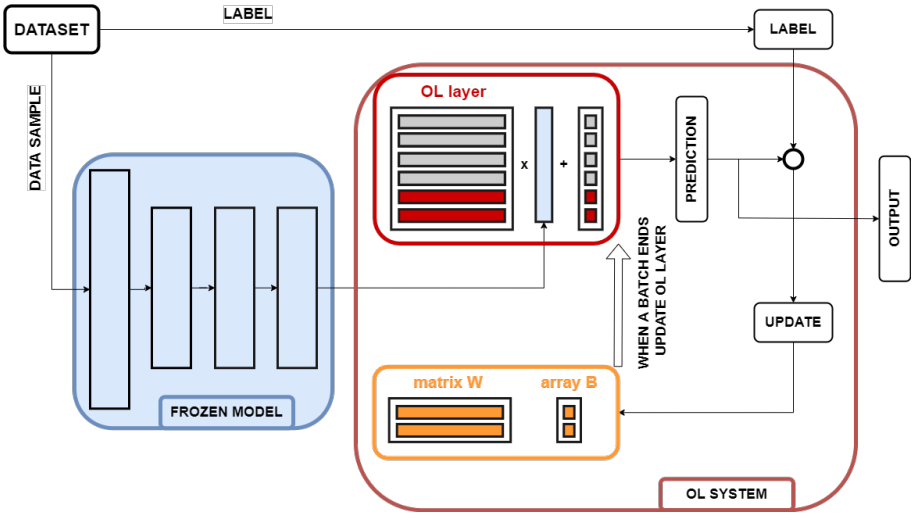


Figure 3.5: Block diagram describing the algorithm TinyOL V2 with batches.

3.2.3 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 paired with a double architecture used for combining old and new knowledge in the back propagations. The double architecture models refers to the fact that two ML model are required. Differently from the application in **??**, this study applies the OL system only on the classification layer. This means that the double architectures is composed of only two classification layers and not two entire models. The first one is called training layer, tl, while the second is called copy layer, cl. The role of tl is to be continuously updated at each training step with the particular LWF back propagation rule, while cl is just a copy of the original frozen model's classification layer (computed by Tensorflow on the laptop training session). So the two layers represent opposite behaviours, tl is the up to date model that evolves accordingly to the LWF update rule while cl represents the original model that remains constant and has no knowledge about new classes.
The back propagation rule can be easily obtained by performing SGD on the $\mathcal{L}_{LWF}$ loss function expressed in Equation 3.22. The function aims at generating an update that depends on a dynamic average computed between the errors committed by both classification layers. Note that the LWF strategy requires two predictions, which means also twice the computation. The final update rule can be easily obtained by computing the derivative of $\mathcal{L}_{LWF}$ with respect to the weights and biases. The result turns out to be just the average of the back propagations.

$$\mathcal{L}_{LWF}(y_i, z_i, t_i) = (1-\lambda) \cdot L_{cross}(y_i, t_i) + \lambda \cdot L_{cross}(y_i, z_i) \tag{3.22}$$

$$w_{i,j} = w_{i,j} - \alpha \cdot x_i \cdot [\ (y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda] \tag{3.23}$$

$$b_i = b_i - \alpha \cdot [\ (y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda] \tag{3.24}$$

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

Where y_i is the i-th element of the prediction array obtained from the layer tl, z_i is the i-th element of the prediction array obtained from the layer cl, t_i is the i-th element of 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. Their application used an evolution that followed a discrete function proportional to the number of batches encountered. In this case, λ needs to be dependent on the number of samples encountered. The update of the loss function weight was found experimentally and is the following:

$$\lambda = \frac{100}{100 + \text{prediction\_counter}} \tag{3.25}$$

Another important note to be said is that Equation 3.22 follows the variation proposed in [**maltoni2019continuous**], where the loss functions $\mathcal{L}_{LWF}$ is not a sum between *categorical cross entropy* and *knowledge distillation* but a sum of two *categorical cross entropy*. This is a little modification that allows for an easier implementation without dropping in performance.
Also in this case a version for integrating upgrades depending on batches is proposed. This time the method does not maintain cl constant for the entire training, but rather updates its values every time a batch is finished. At this point, the algorithm becomes a fusion between new knowledge and knowledge coming from an old version of the layer. The size of a batch is defined by the value k and this value defines how old cl is. The update rules in this case are:

$$w_{i,j}^{TL} = w_{i,j}^{TL} - \alpha \cdot x_i \cdot [\ (y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda] \tag{3.26}$$

$$b_i^{TL} = b_i^{TL} - \alpha \cdot [\ (y_i - t_i)(1-\lambda) + (y_i - z_i)\lambda] \tag{3.27}$$

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

And at the end of one batch (once every k values are elaborated) the parameters of cl are updated in the following way:

$$w_{i,j}^{CL} = w_{i,j}^{TL} \tag{3.28}$$

$$b_i^{CL} = b_i^{TL} \tag{3.29}$$

Where $w_{i,j}^{CL}$ and b_i^{CL} are respectively the weights and biases of cl, while $w_{i,j}^{TL}$ and b_i^{TL} are the weights and biases of tl.
This method, being different from the previous, requires also a different λ rule. Experimentally it has been found the following rule to be well working:

$$\lambda = \begin{cases} 1 & prediction\_counter \leq batch\_size \\ \frac{\text{batch\_size}}{\text{prediction\_counter}} & prediction\_counter > batch\_size \end{cases} \tag{3.30}$$

Both LWF methods require the same amount of memory, which is two times the amount required for a classification layer, so $(n \times m) \times 2$ and $(n \times 1) \times 2$. Both methods are quite easy to implement. Their strength is defined in the value λ and in the particular update rule. The two methods differ only because of the update on 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 double the computation. Figure 3.6 contains a block diagram that explains the pipeline of the LWF method.

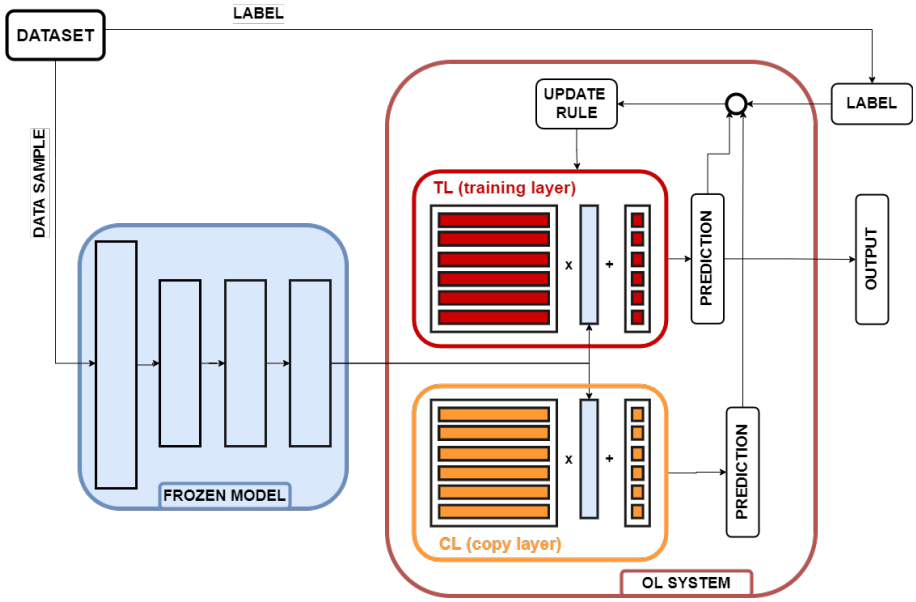


Figure 3.6: Block diagram describing the algorithm LWF.

3.2.4 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 training layer, tl, and consolidated layer,

cl. The idea for this algorithm is to train the two layers in two different ways. For *tl*, the same strategy used in TinYOL is applied. On the other hand *cl* is updated once every batch with a particular update rule. The back propagations for *tl*, that happens at every step is:

$$w_{ij}^{TL} = w_{ij}^{TL} + \alpha(y_i^{TL} - t_i) \cdot x_i \tag{3.31}$$

$$b_i^{TL} = b_i^{TL} + \alpha(y_i^{TL} - t_i) \tag{3.32}$$

While the back propagation for *cl*, that happens once every batch, is the following:

$$w_{ij}^{CL} = \frac{w_{ij}^{CL} \cdot updates_i + w_{ij}^{TL}}{updates_i + 1} \tag{3.33}$$

$$b_i^{CL} = \frac{b_i^{CL} \cdot updates_i + b_i^{TL}}{updates_i + 1} \tag{3.34}$$

$$w_{ij}^{TL} = w_{ij}^{CL} \tag{3.35}$$

$$b_i^{TL} = b_i^{CL} \tag{3.36}$$

Where w_{ij}^{TL} and b_i^{TL} are the weights and biases of the training layer, w_{ij}^{CL} and b_i^{CL} are the weights and biases of the consolidated layer, $updates_i$ is an array that behaves as a counter of labels encountered, y^{TL} is the prediction obtained from *tl*, and α is the learning rate.
The use of two classification layers that update in this way makes the framework behave as a system composed of a short term memory and a long term memory. Because of the continuous update at every batch, *tl* is the short term memory. This layer learns from every single sample that is received, and at the end of a batch it gets corrected with the parameters from *cl*. On the other hand, *cl* behaves as the long term memory. This because it never gets reset or cleaned and it gets updated only once every batch with information coming from the short term memory. Note that the weighted average method depends on the number of times that a specific label appeared in the training batch.
Another important aspect of this method regards the amount of computations required at each step. While training, the methods requires only to perform one inference from *tl*. In fact no prediction from *cl* are actually useful at any point during training. The inference obtained from *cl* should be

performed only when actively requested. Since *cl* represents the long term memory, its prediction is to be considered the more reliable and generally more accurate between the two. Is to be noted that, for this study case, inferences from *cl* are required only during the testing that is performed at the end of the training. In real life scenarios these predictions should be performed only when actively requested. This because the amount of computation required for one step are doubled, and the only improvement obtained would be the info given to the user.
CWR is an easy to implement method. Its strengths are hidden in the double architecture and the update rule that make possible the merging of short term memories and long term memories. The amount of memory required for this algorithm is: two weight matrices of size $n \times m$, 2 bias arrays of size $n \times 1$, and one array that keeps track of the labels encountered of size $n \times 1$ (is called updates in equation 3.33). The amount of computations can change if an inference is required, making it double. Figure 3.7 contains a block diagram that shows the pipeline of the CWR algorithm.

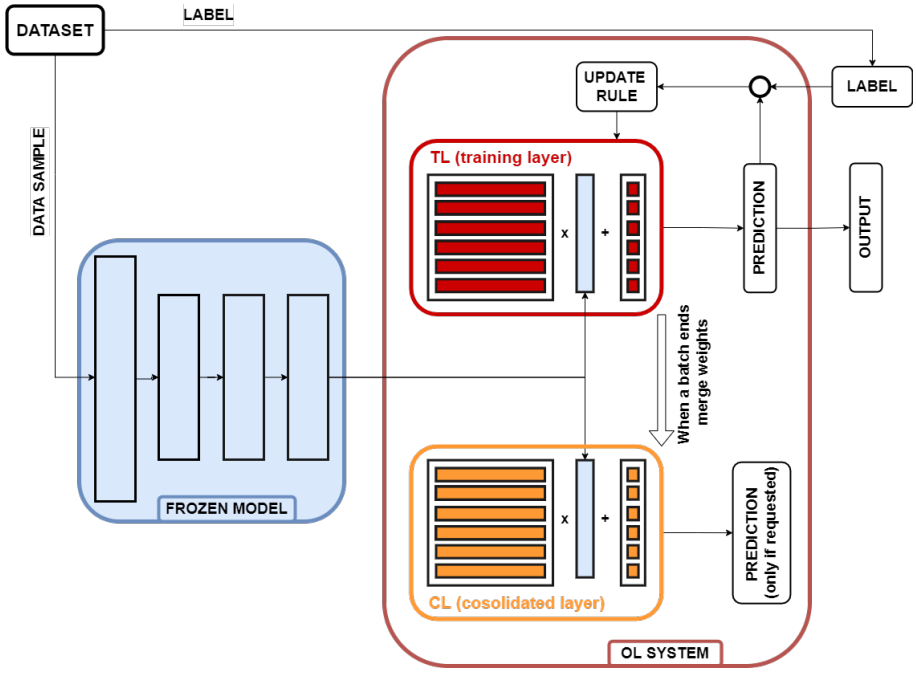


Figure 3.7: Block diagram describing the algorithm CWR.

3.2.5 My algorithm

And

3.3 Gesture recognition application

3.4 Image classification application

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 **??**.
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.

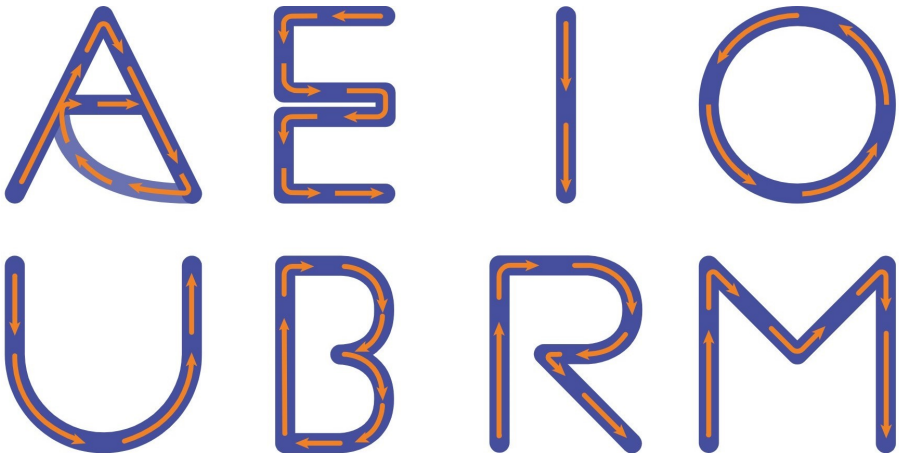


Figure 4.1: Motion followed with the sensor for writing letters in the air.

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

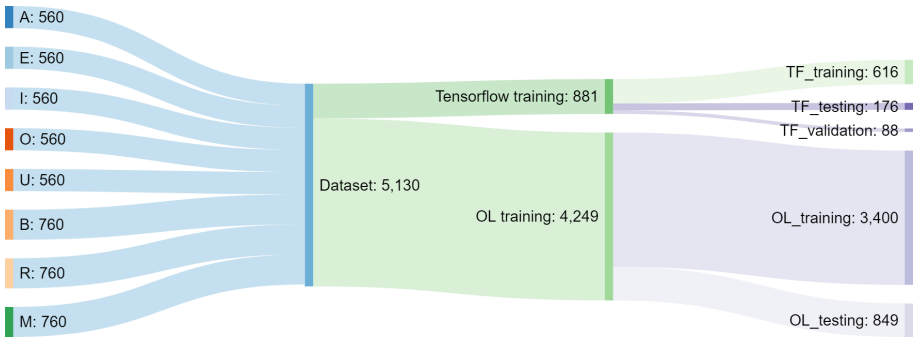


Figure 4.2: Stankey diagram showing how the letter dataset is divided

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 4.3: Example of images from the MNIST digits 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.

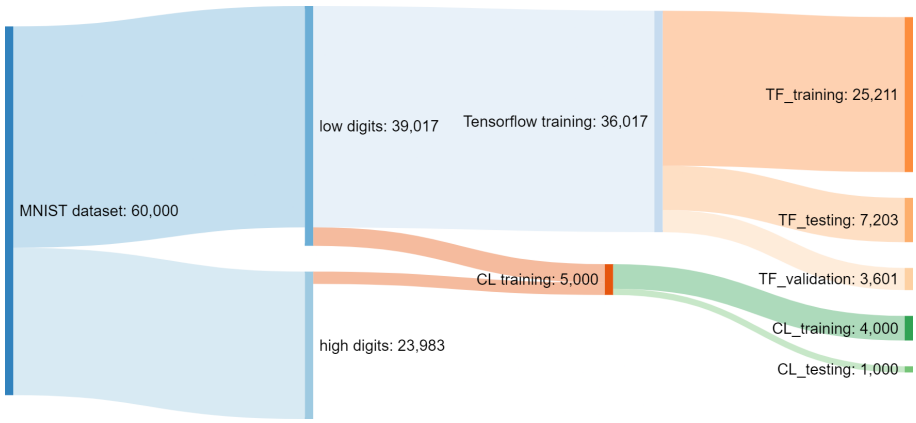


Figure 4.4: Stankey diagram showing how the MNIST dataset is divided

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.

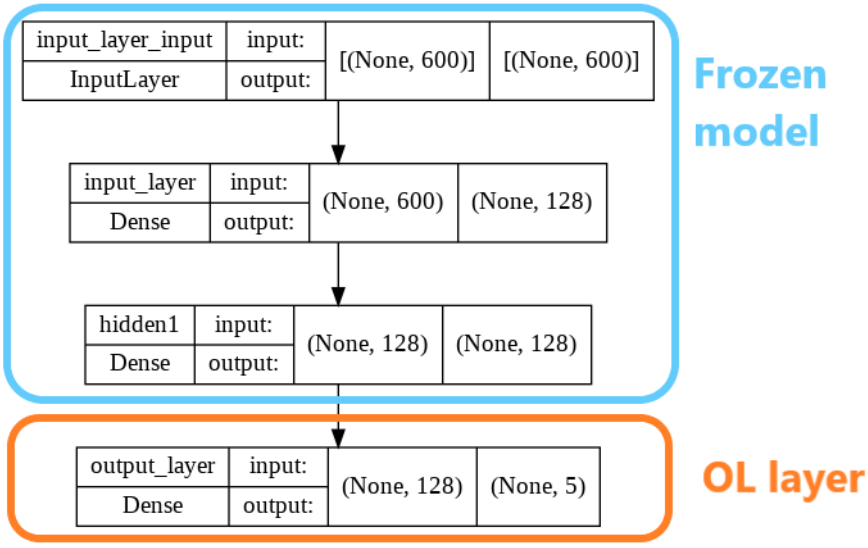
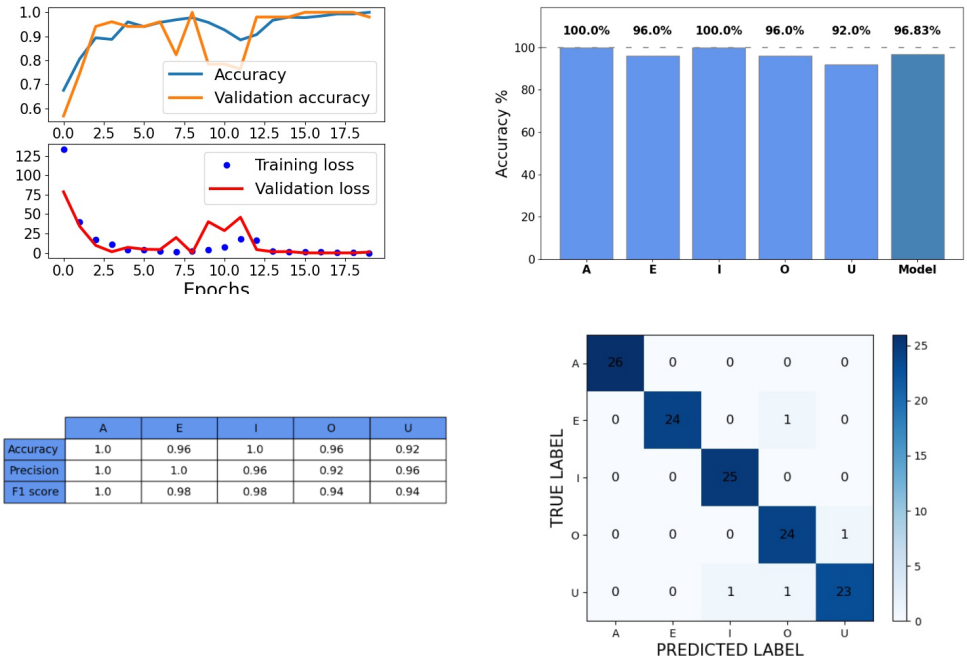


Figure 4.5: Letter recognition model: basic structure and its separation in frozen model and OL classification layer

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.h5* file which contains



	A	E	I	O	U
Accuracy	1.0	0.96	1.0	0.96	0.92
Precision	1.0	1.0	0.96	0.92	0.96
F1 score	1.0	0.98	0.98	0.94	0.94

Figure 4.6: Results after the Tensorflow training. Top right: variation of accuracy and loss during training and validation. Top right: accuracy of the model in each class. Bottom left: table resuming precision accuracy and F1 score. Bottol right confusion matrix.

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 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.

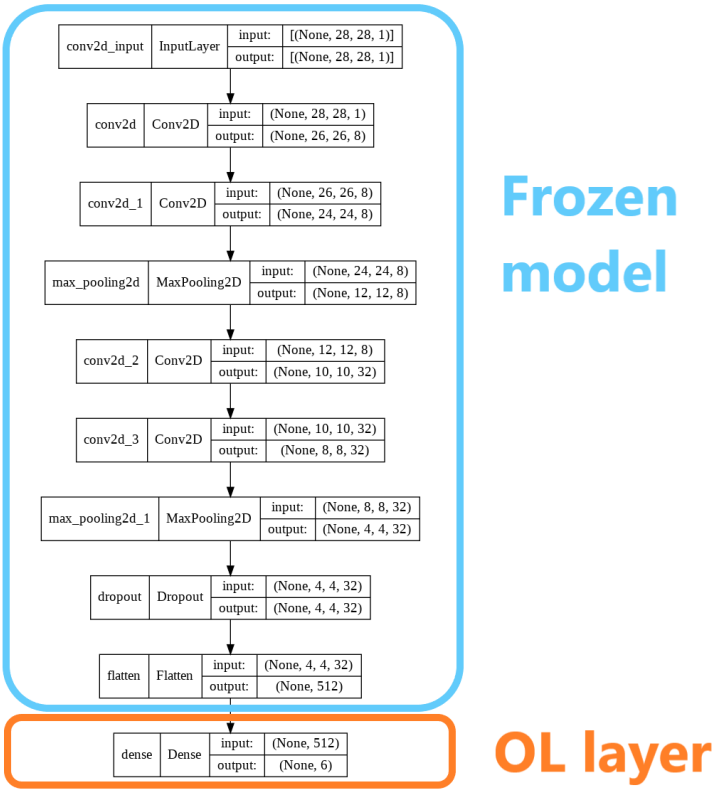
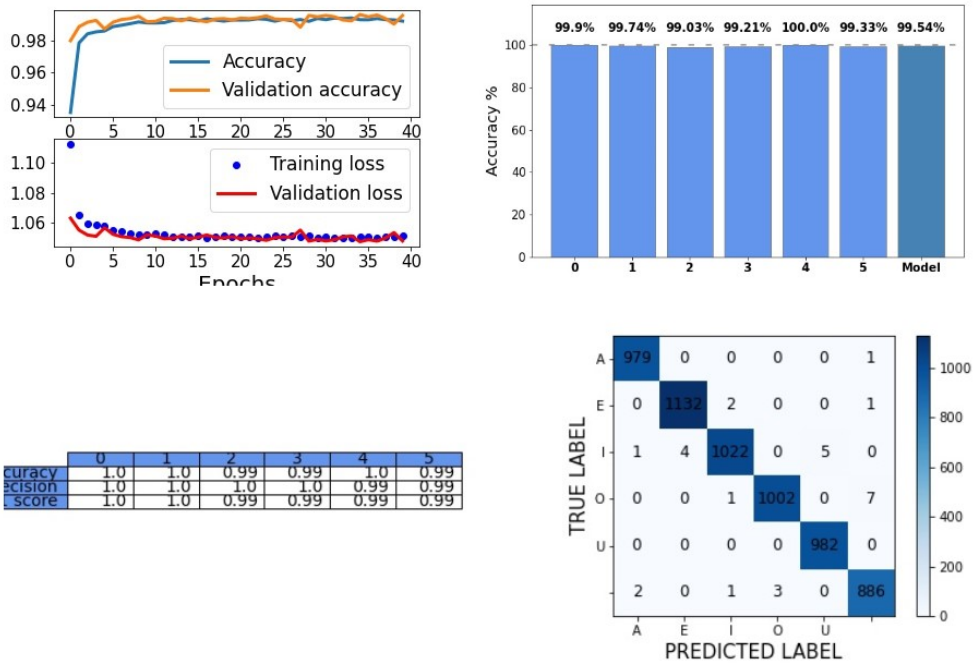


Figure 4.7: Image classification model: basic structure and its separation in frozen model and OL classification layer

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 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.



	0	1	2	3	4	5
curacy	1.0	1.0	0.99	0.99	1.0	0.99
ecision	1.0	1.0	1.0	1.0	0.99	0.99
score	1.0	1.0	0.99	0.99	0.99	0.99

Figure 4.8: Results after the Tensorflow training. Top right: variation of accuracy and loss during training and validation. Top right: accuracy of the model in each class. Bottom left: table resuming precision accuracy and F1 score. Bottol right confusion matrix.

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 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. Specifically, it 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 two software are exploited to add ML

capabilities to the microcontroller. 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 from 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, namely STM-CUBE-AI. This toolkit enhances 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 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, 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. Then the infinite while loop begins, and if the "received sample flag" is triggered, the frozen model inference is 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 adopted strategy. 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 ex-

periment, 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, and 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 to 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 test 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.

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 camera MCU through OpenMV IDE.

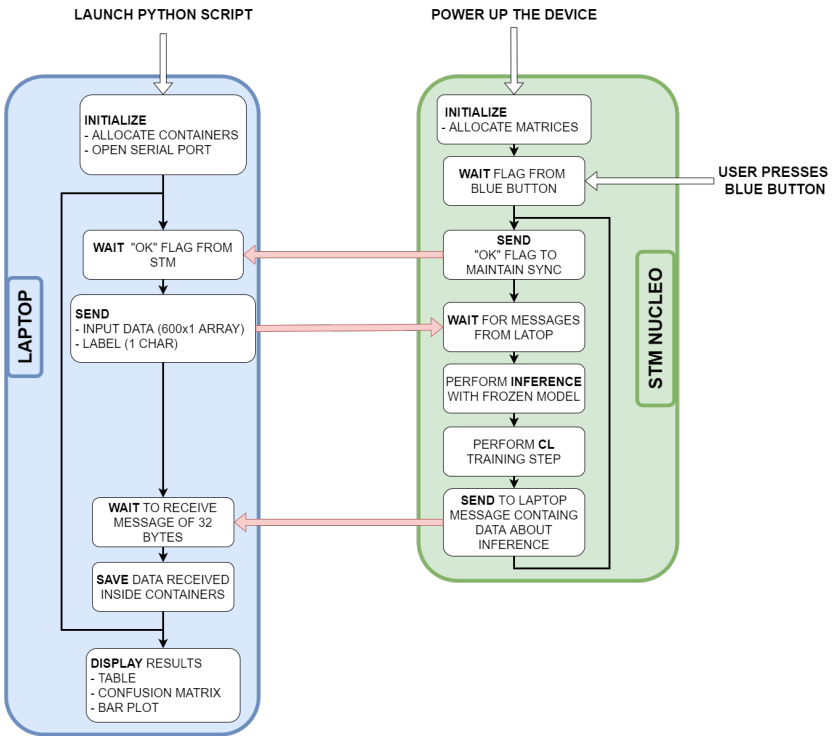


Figure 4.9: Block diagram showing the communication between laptop and STM Nucleo.

The system developed for the application of 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. 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. Moreover, problems regarding the allocation of big matrices often block 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 a small increase of time for the inference, but it is actually required to obtain satisfying reults.
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. As before, a Python

app was developed for the creation of fast, reliable and repeatable training sessions. Here, the idea is not to send data via USB connection, but display on a screen images from the MNIST dataset while the camera is pointing at it. The USB connection is required anyway because the app needs to maintain sync with the camera and 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 is divided into elemental blocks. At first, the app loads the dataset and extracts from it only the required number 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 MNIST digits where the camera is 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 where 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 size of 4 chars was chosen because it is a fast and unique way for sending a command that is also easy to read from the code point of view. 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 pointing 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, compresses the image, and sends it via UART to the laptop. This mode is used for understanding what the camera will see 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 contains just an X char. During this state, the app on the laptop slowly shows all 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 un sync the devices. The label this time is received and transformed into a hot one encoded array for the computation of the back propagation.

Once the training and the pseudo testing are performed, the app stops showing digit images and the OpenMV camera stores the results inside its SD card. Then, the results are manipulated by a script that transforms the raw data written in the SD card into a table a confusion matrix and bar plots. Figure 4.10 contains a block diagram summarizing the flow of the experiment.

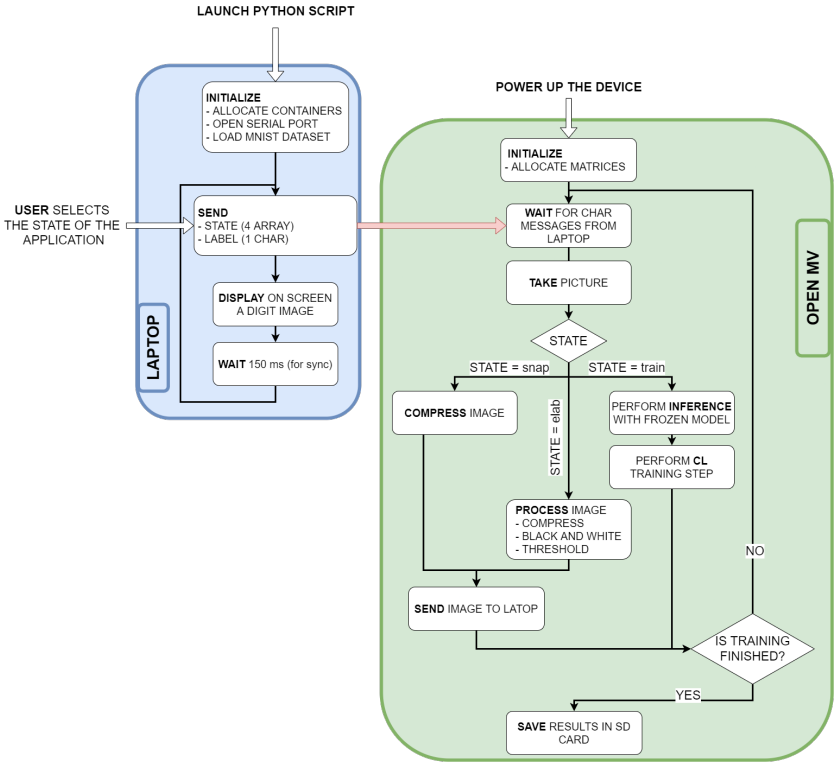


Figure 4.10: Block diagram showing the communication between laptop and OpenMV camera.

Figure 4.11 shows the OpenMV camera pointed to a computer screen in different states. Top right figure shows the camera in *idle* mode, meaning

no stream is received by the laptop. Top left figure represents the camera in *snap* mode, where compressed images are streamed to the laptop. Bottom left figure contains the camera in *elab* mode, where compressed and processed images are streamed to the laptop. Bottom right figure shows the camera in *train* mode, meaning no stream is available and the camera is performing inference and CL training.

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

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. In *elab* mode the image is captured, gray scaled, binarized with threshold, compressed, and sent.

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

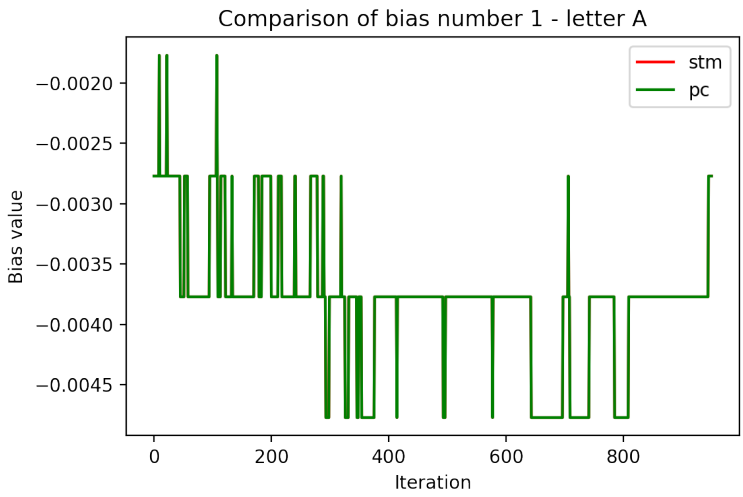


Figure 5.1: PLACEHOLDER

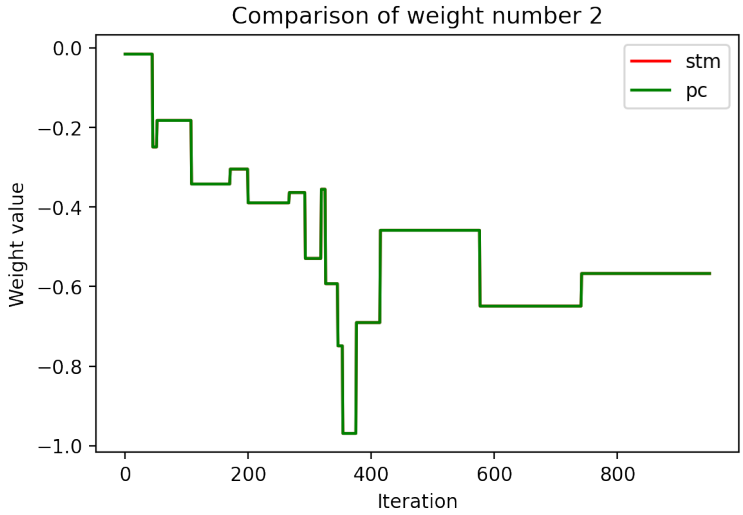


Figure 5.2: PLACEHOLDER

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

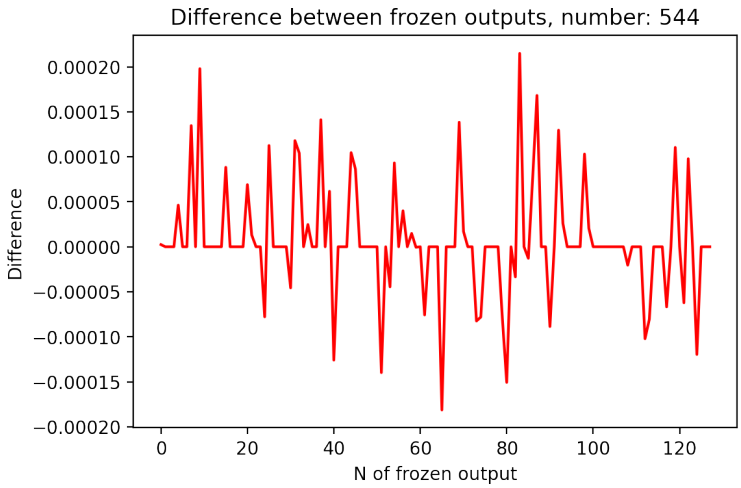


Figure 5.3: PLACEHOLDER

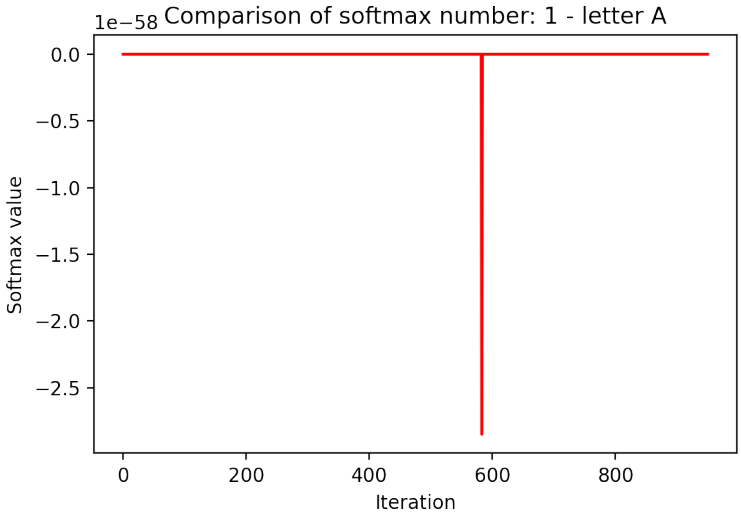


Figure 5.4: PLACEHOLDER

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-

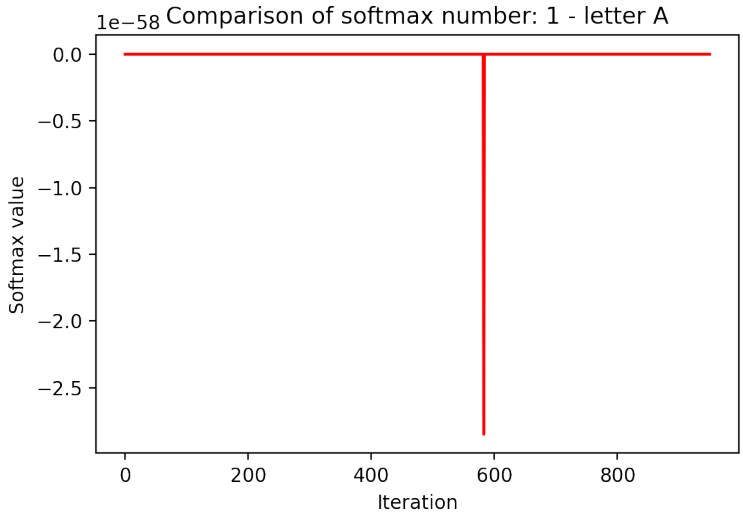


Figure 5.5: PLACEHOLDER

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

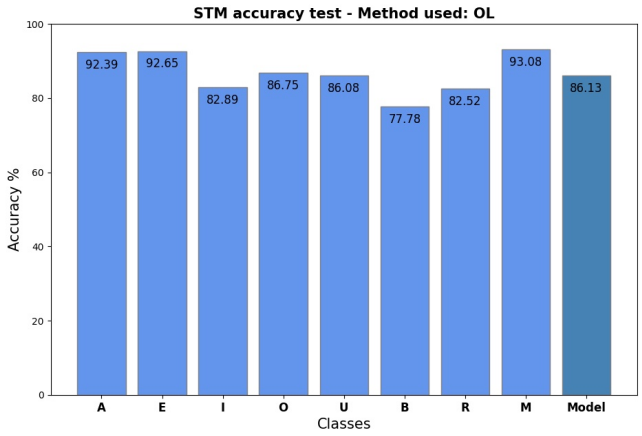


Figure 5.6: PLACEHOLDER

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		

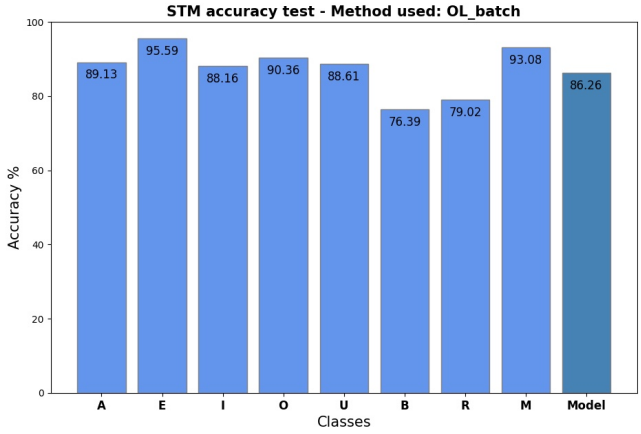


Figure 5.7: PLACEHOLDER

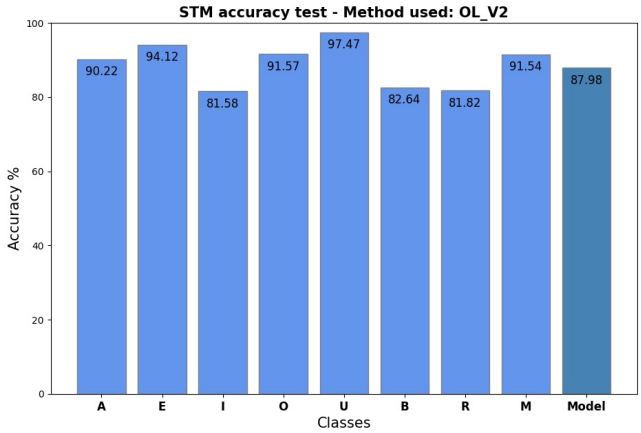


Figure 5.8: PLACEHOLDER

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

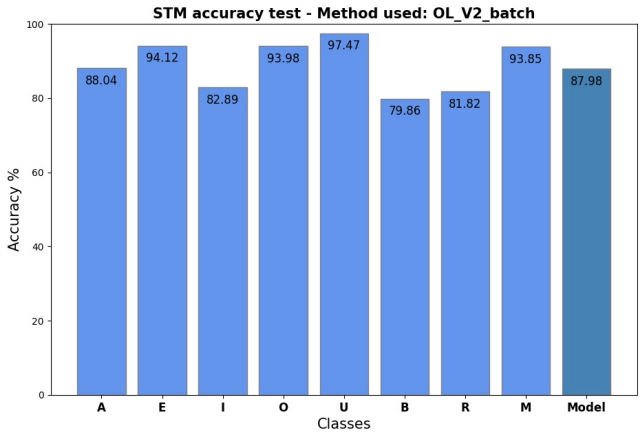


Figure 5.9: PLACEHOLDER

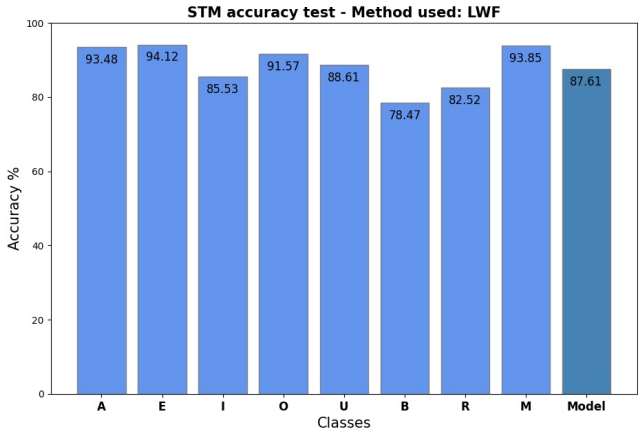


Figure 5.10: PLACEHOLDER

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

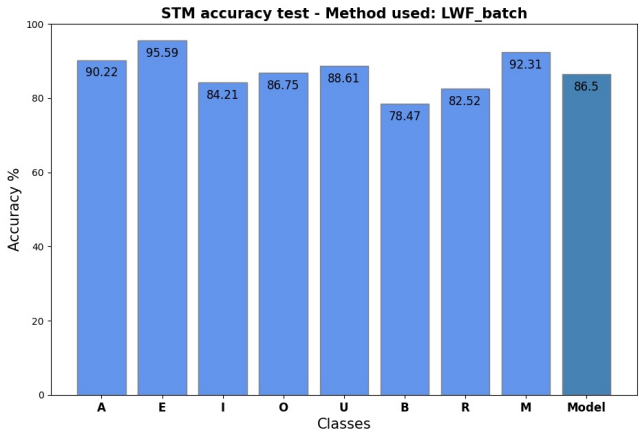


Figure 5.11: PLACEHOLDER

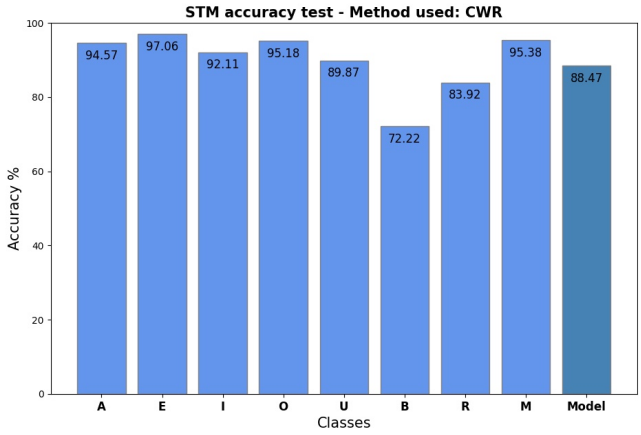


Figure 5.12: PLACEHOLDER

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.

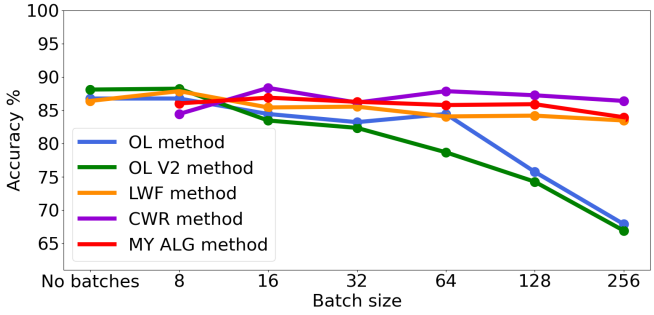


Figure 5.13: PLACEHOLDER

Chapter 6

Conclusion

LA BIBLIOGRAFIA NON FUZNIONA