POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Neural Network Training on Embedded Systems: a feasibility study

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Davide Quarantiello**

Student ID: 939618
Advisor: Prof. Giacomo Boracchi
Co-advisors: Diego Carrera
Academic Year: 2021-2022

Abstract

The deployment of embedded systems in an ever-growing range of application contexts has determined a surge of interest in empowering the devices with more and more advanced, personalized Artificial Intelligence (AI) functionalities.
Current frameworks enable embedded systems to effectively perform AI inference by converting pre-trained models into reference suitable for resource-constrained devices. However do not empower embedded systems with the ability to train models directly on the device. In current solutions, the training process relies on a cloud-based paradigm. This strategy has several drawbacks: it does not preserve data privacy, it requires a stable connection and, moreover, since the training procedure uses the data collected from many devices, the trained models are not tailored for the specific device. For all these reasons, the need to empower embedded devices with training on device functionalities is clearly emerging.
The attractiveness of the training on device is not only limited to tackle the aforementioned challenges, but can also enable more complex frameworks like Federated Learning where several devices contribute to learn a common model. In the thesis, developed from a collaboration with STMicroelectronics, we propose a framework which enables embedded systems to effectively perform the training of neural networks on device.
The problem has been addressed from different perspectives allowing not only to design and implement the desired framework more easily, but also to set up a feasibility study about the neural network training on STM32 microcontrollers. Specifically starting from a formal reorganization of backpropagation calculus using computational graphs, we develop a framework which enables embedded systems to train neural networks directly on device. Moreover we design a tool for evaluating the computational resources required for running the training process. Finally, we set several experiments aimed to evaluate the enhancement of models performances due to the training on device in a real use case.

Keywords: TinyML, Training On Device, Neural Network, Backpropagation, Embedded Systems, Microcontrollers, STM32

Abstract in lingua italiana

L'utilizzo dei sistemi *embedded* in contesti applicativi sempre più ampi ha reso necessaria l'integrazione dell'intelligenza artificiale (IA) sui dispositivi stessi.
L'attuale progresso tecnologico consente di usare modelli basati sull'IA per effettuare predizioni. Infatti, tramite appositi framework, è possibile convertire modelli pre-addestrati in versioni compatibili con i sistemi *embedded* per essere poi utilizzate per fare inferenza. Il principale problema di questo approccio è che i modelli risultano essere statici in quanto non possono essere modificati dagli stessi dispositivi. Per poter addestrare nuovamente il modello si ricorre ad un paradigma basato sul *cloud*, strategia che tuttavia presenta diversi svantaggi. In primo luogo non tutela la privacy dei dati, inoltre è richiesto che i dispositivi siano necessariamente provvisti di una connessione. Un'ulteriore criticità da sottolineare è rappresentata dal fatto che i modelli sono addestrati utilizzando i dati raccolti da diversi dispositivi, per cui i modelli addestrati su tali dati possono essere non altrettanto efficaci quando vengono impiegati su un dispositivo diverso da quelli utilizzati per la raccolta di dati.
L'addestramento di modelli direttamente sui dispositivi *embedded*, senza ricorrere al *cloud*, consente di risolvere tali problematiche. I modelli possono apprendere continuamente, diventando sempre più performanti e, soprattutto, personalizzati. In questo modo vengono sfruttate al massimo le potenzialità dell' IA. Inoltre l'addestramento sul dispositivo rappresenta anche una funzione abilitante per l'apprendimento distribuito, come l'apprendimento federato, in cui diversi dispositivi contribuiscono all'apprendimento di un modello comune.
Nella tesi, nata dalla collaborazione con STMicroelectronics, abbiamo sviluppato un framework che consente ai sistemi *embedded* di addestrare reti neurali direttamente sul dispositivo.
Abbiamo affrontato il problema da diverse prospettive, il che ci ha permesso non solo di progettare e implementare più facilmente il framework desiderato, ma anche di impostare uno studio di fattibilità sull' addestramento di reti neurali sui microcontrollori STM32. In particolare, partendo da una riorganizzazione formale della backpropagation utilizzando grafi computazionali, abbiamo sviluppato il framework che consente ai sistemi *embed-*

ded di addestrare reti neurali direttamente sul dispositivo. Inoltre abbiamo progettato uno strumento per valutare le risorse computazionali necessarie per eseguire il processo di addestramento. Infine abbiamo condotto diversi esperimenti per valutare l'effettivo miglioramento delle prestazioni dei modelli grazie all'addestramento sul dispositivo considerando un caso d'uso reale.

Parole chiave: TinyML, Addestramento sul dispositivo, Reti Neurali, Backpropagation, Sistemi Embedded, Microcontrollori , STM32

Contents

1 | Introduction

1.1. Artificial Intelligence on Embedded Systems

The deployment of embedded systems in an ever-growing range of application contexts has determined a surge of interest in empower the devices with more and more intelligent, personalized Artificial Intelligence (AI) functionalities.
Since AI algorithms are capable of extract data from unstructured information, they are particularly useful in contexts where the embedded systems are usually employed. These AI applications would be impractical or even impossible to deploy in a centralized cloud due to issues related to bandwidth and privacy. With embedded AI, devices have the ability to run AI models at the device level and then directly use the results to perform an appropriate task.
The most critical drawback of the effective implementation of AI models on embedded systems is that the AI algorithms are generally computationally and memory very intensive, conversely embedded systems are resource-constrained devices. Thus designing a lightweight AI models became a key aspect of accomplishing AI functionalities on embedded devices. Several techniques have been proposed to reduce the size of AI models such as model compression, pruning and quantization, however maintaining high the accuracy of the models while applying these techniques is challenging. The field of research that tackle these challenges is known as Tiny Machine Learning (TinyML). However, all the solutions which are proposed, start with the assumption that the model to optimize is already trained and it will be used by the embedded systems to perform only inference.
Current frameworks enable embedded systems to effectively perform AI inference by reducing the size of pre-trained models with the aforementioned techniques. However, also these frameworks do not provide functionalities for training the model directly on device. The main drawback is that, once the model is deployed to the embedded systems, it becomes a static object that cannot be modified more. These prediction-only models work properly if environment around the embedded systems is fairly static, however, since in the real world an environment incessantly changes over time, the pre-trained models may not perform well because they do not adapt to the new environmental conditions and

become outdated.

1.2. Training on Device

In current solutions the training process relies on a centralized cloud-based paradigm: the data collected from the devices are transmitted to the cloud where the training of the model is performed. The trained model is then send back from the cloud to the device. This strategy has several drawbacks: it does not preserve data privacy since possibly sensitive data needs to be transmitted to the cloud and indefinitely stored there, it requires high bandwidth since data must be transmitted over a stable connection. Moreover cloud-based models are typically trained using data collected from many different devices, and they can lose accuracy when applied to a new device whose data are never transmitted to the cloud. For all these reasons, the need to empower embedded devices with training on device functionalities is clearly emerging. Training on device allows the embedded systems to train models directly on device using local data. The enablement of this functionalities has several advantages. First of all the AI models can learn and respond to new scenarios, learning from local data means that the resulting trained models are specifically tailored for each devices providing more personalized AI functionalities. Another huge benefit of training on device is that the data have not be sent to the cloud and so there is no more need of a fast and a highly reliable connection. As a result of the processing of data offline also the protection and the privacy of data is ensured. The attractiveness of the training on device is not limited only to tackle the aforementioned challenges, but can also enable more complex frameworks like Federated Learning [7] where several devices contributes to learn a common model.
Besides the benefits, training on device has the several limitations and challenging aspects. The main issue is the resource requirements which are more and more demanding then the inference on device.

1.3. Related Work

Most of the solutions currently present in the literature in the field of AI functionalities on embedded systems focus on methods aiming at optimizing and compressing models trained on the cloud to make them suitable for embedded devices. This research area is know as Tiny Machine Learning (TiniyML)[11].
Proprietary frameworks such as TFLite Micro from Google [5], CMSIS-NN from Arm [9], X-CUBE-AI from ST [3] allow to effectively enable embedded systems to perform inference by converting pre-trained models into optimized versions that can be run on

resource-constrained environment. However these frameworks do not provide functionalities for training the model directly on device. The research about training on device is highly fragmented and most of the proposed solutions are related to the training of non-deep ML models or to the implementation of specific use cases.
In [6] authors propose a method based on k-nearest neighbor (KNN), however the KNN does not require a training procedure because it incrementally learn by simply adding the new samples in it's knowledge base.
In [10],[13] the focus is on the Multi Layer Perceptron (MLP). Two alternatives to BP are proposed which are faster and computational less intensive compared to BP, but they are specifically designed for training a MLP with a single hidden layer.
The only works on BP are [8],[12] however the proposed methods are based on Transfer Learning and aiming at training only on the last layer of the network. Moreover in [4] the authors propose a novel transfer learning method that reduce the BP memory requirements by updating only biases.

1.4. Objectives and contributions

From a collaboration with STMicroelectronics we propose a framework which enables embedded systems to effectively perform the training of neural networks on device.
We have addressed the problem from different perspectives, specifically we consider the algorithmic, the software and the applicability point of view. This re-formulation of the problem allow us not only to design and implement the desired framework more easily, but also to set up a feasibility study on the training on STM32 microcontrollers. The considered perspectives indeed space from a theoretical to a practical point of view, which are key aspect equally important, each of one presents challenging aspects to be solved. The contribution provided by the thesis can summarized as follows:

- Algorithmic: since training methods are grounded in their algorithmic implementations, an important part for enabling training on device is making the algorithm efficient. We propose a formal reorganization of backpropagation (BP) calculus using Computational Graphs aiming at simplify the implementation of the training on embedded systems.
- Software: current AI frameworks for embedded devices do not support training on device functionalities. So we design and develop a framework in C which enables embedded devices to train arbitrary networks.
- Applicability: computing resources represents the main bottleneck for the training

on device. We design a tool for evaluating and predicting the resource required for running the training on a device.

The actual effectiveness of the developed framework needs to be verified empirically in a real scenario. An additional provided contribution is related to this task.

- We propose an experimental evaluation of the personalization on a Human Activity Recognition (HAR) use case which highlights the enhancement of model performances due to the training on device.

Note that the implementation of the framework leverages the BP reformulation while the estimation of required computing natural arises from the effective implementation of the framework. Finally the experimental evaluation on a real use case is made possible by both the developed framework and tool for the resource consumption estimation.

1.5. Thesis structure

The structure of the thesis follows the way in which the problem has been addressed. Specifically we have that in:

- **Chapter 2**: the problem of training a neural network is introduced from a theoretical perspective.
- **Chapter 3**: the re-formulation of backpropagation using computational graph is explained.
- **Chapter 4**: the functionalities and logic of developed framework are described.
- **Chapter 5**: the process of designing the tool for evaluating the required computing resources is described.
- **Chapter 6**: the experiment on the Human Activity Recognition task are proposed.

2 | Neural Network Optimization

Most of deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing a function $f(\mathbf{x})$ by altering $\mathbf{x}$. Most problems are posed in terms of minimizing $f(\mathbf{x})$. The function f to minimize is called interchangeably *objective function*, *cost function* or *loss function* and returns a value that indicates how good is a deep learning algorithm. Specifically a lower value of f indicates a better performance of the algorithm and thus we are interested into minimize it.

2.1. One-Dimensional Gradient Descent

Gradient descent in one dimension is an excellent example to explain why the Gradient Descent algorithm is used to minimize an *objective function*. Consider a continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$. By applying the first-order Taylor expansion we obtain

$$f(x + \eta) \approx f(x) + \epsilon f'(x) \tag{2.1}$$

The derivative is therefore useful to optimize a function because it tells how to change input in order to make a small improvement in the output. For example, we know that that $f(x - \eta \text{ sign } (f'(x))) < f(x)$ for small enough *learning rate* η. Thus we can reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative.

$$x = x - \eta f'(x) \tag{2.2}$$

This means that, if we iteratively apply the Eq.(2.2) the value of function $f(x)$ might converge to a minimum. This technique is called *Gradient Descent* and the progress of optimizing $f(x)$ over x can be plotted as follows

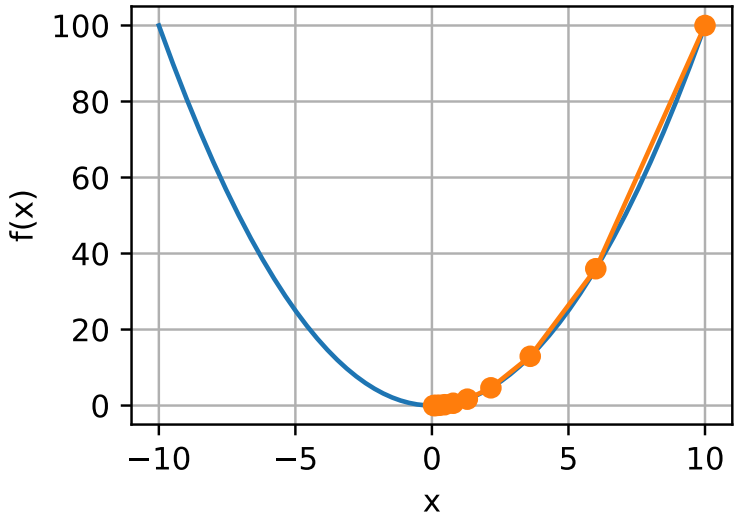

Figure 2.1: Gradient Descent applied on $f(x)$

The *learning rate* η is a hyper-parameter that determines the step size at each iteration while moving toward a minimum of the *objective function*. It controls how much to change the model in response to the estimated error each time the model weights are updated. Since it influences to what extent newly acquired information overrides old information, it metaphorically represents the speed at which a machine learning model learns. Choosing the *learning rate* is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

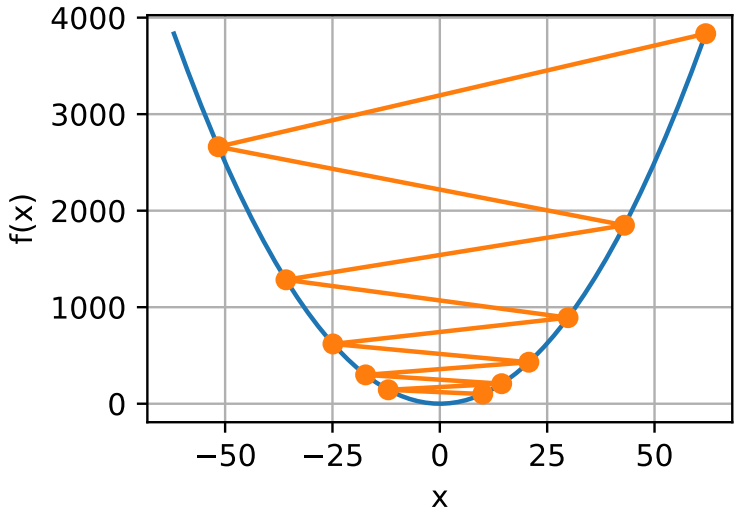

Figure 2.2: Learning rate changes the step size of the update

2.2. Multivariate Gradient Descent

Consider the case in which we have that $\mathbf{x} = [x_1, \cdots, x_n]^T$ and so the *objective function* $f(\mathbf{x}) : \mathbf{R}^N \to \mathbf{R}$. Its gradient is multivariate too. Specifically It is a vector consisting of n partial derivatives

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n}\right]^T \tag{2.3}$$

each ones indicates the rate of change of f at x with respect to the input x_i. We can reduce $f(\mathbf{x})$ by moving $\mathbf{x}$ in small steps with the opposite sign of the gradient $\nabla f(\mathbf{x})$.

$$\mathbf{x} = \mathbf{x} - \eta \nabla f(\mathbf{x}) \tag{2.4}$$

Algorithm 2.1 Single step of Gradient Descent(GD)

Require: Initial parameters $\mathbf{x}$

Require: Learning rate η

Require: Batch size b

1: Compute the gradient estimate: $\nabla f(\mathbf{x})$

2: Apply update: $\mathbf{x}' = \mathbf{x} - \eta \nabla f(\mathbf{x})$

The progress of optimizing $f(\mathbf{x})$ over $\mathbf{x}$ can be plotted as follows.

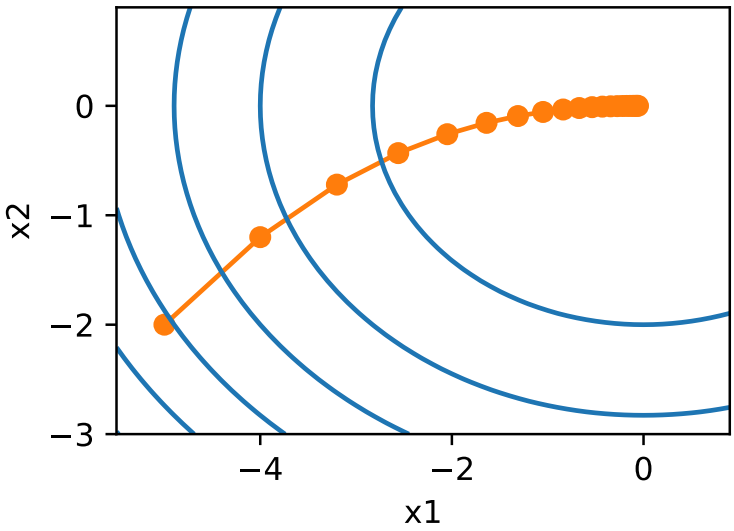

Figure 2.3: Different steps of Gradient Descent applied on function $f(\mathbf{x})$

2.3. Optimization in Deep Learning

Gradient descent is used in the context of neural network optimization in order to find the parameters $\boldsymbol{\theta}$ that reduce the expected generalization of the loss function, also known as *risk*.

$$\overline{C}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},y)\sim p_{data}} \left[L(f(\mathbf{x}, \boldsymbol{\theta}), y) \right] \tag{2.5}$$

where:

- $\boldsymbol{\theta}$ are the network parameters;
- p_{data} is the distribution that generates data;
- L is the per-example loss function;
- $f(\mathbf{x}, \boldsymbol{\theta})$ is the predicted output when the input is $\mathbf{x}$;
- y is the target output;

In general, the Eq. (2.5) cannot be computed because the distribution p_{data} is unknown. However, since we know a subset of data sampled from p_{data}, we can use the average value of the losses computed on the available samples.

$$\overline{C}(\boldsymbol{\theta}) \approx C(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i) \tag{2.6}$$

The (2.6) is known as *empirical risk* and represent the *objective function* which we want to minimize. Since we are approximating the *risk* also the result of minimization is an approximation.

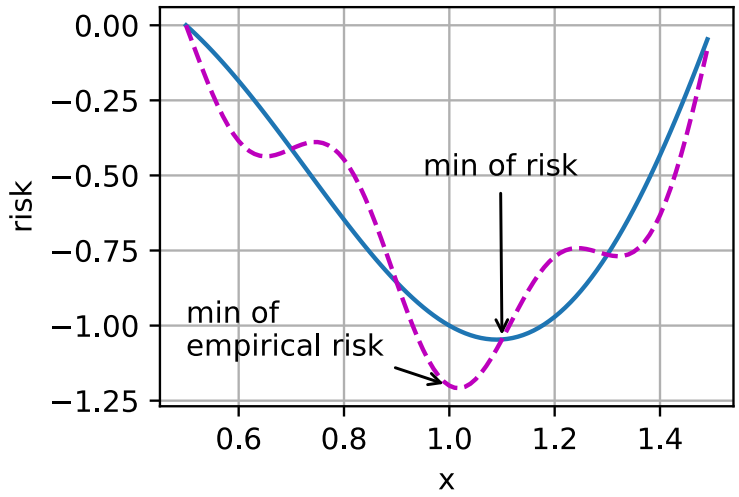

Figure 2.4: Comparison between empirical risk and risk optimization

For minimizing the *empirical risk* we assume that

$$\nabla C(\boldsymbol{\theta}) = \nabla \mathbb{E}_{(\mathbf{x},y)\sim p_{data}} \left[L(f(\mathbf{x}, \boldsymbol{\theta}), y) \right] = \mathbb{E}_{(\mathbf{x},y)\sim p_{data}} \left[\nabla L(f(\mathbf{x}, \boldsymbol{\theta}), y) \right] \tag{2.7}$$

Then parameters are updated as follows

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \frac{\eta}{N} \sum_{i=1}^{N} \nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i) \tag{2.8}$$

There are three variants of *Gradient Descent*, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

2.3.1. Batch Gradient Descent

In Batch Gradient Descent(BGD), all the training data is taken into consideration to take a single step of Gradient Descent. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. The computational cost for each step is $\mathcal{O}(n)$, where n is the number of samples. Therefore, when the training set is very large, the time to take a single of the loss function becomes prohibitively long. Moreover it performs redundant computations for large training set, as it recomputes gradients for similar examples.

Algorithm 2.2 Batch Gradient Descent(BGD)

Require: Initial parameters $\boldsymbol{\theta}$
Require: Learning rate η
Require: Batch size b
1: **while** stopping criterion not met **do**
2: Compute the gradient estimate: $\nabla c(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i)$
3: Apply update: $\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla C(\boldsymbol{\theta})$
4: **end while**

Recall that the standard error of the mean estimated from n samples is given by $\frac{\sigma}{\sqrt{n}}$, where σ is the true standard deviation of the value of the samples. The denominator shows that there are less than linear returns to using more examples to estimate the gradient and thus it's possible to reduce the number of samples used without affect the standard error too much.

2.3.2. Stochastic Gradient Descent

In Stochastic Gradient Descent(SDG), we consider just one example at a time to take a single step. Specifically at each iteration we sample an example (x_i, y_i) from training set $[(x_1, y_1), \cdots, (x_n, y_n)]$ and compute the gradient $\nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i)$ on this sample. Since we are considering just one example at a time the computational cost for each step drops from $O(n)$ of 4.3 to the constant $O(1)$.

Algorithm 2.3 Stochastic Gradient Descent(SGD)

Require: Initial parameters $\boldsymbol{\theta}$
Require: Learning rate η
1: **while** stopping criterion not met **do**
2: Sample an example (x_i, y_i) from training set $[(x_1, y_1), \cdots, (x_n, y_n)]$.
3: Compute the gradient estimate: $\nabla C(\boldsymbol{\theta}) = \nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i)$
4: Apply update: $\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla C(\boldsymbol{\theta})$
5: **end while**

Stochastic Gradient Descent by means of the natural variation of gradients over examples is able to dislodge the parameters from local minima.

2.3.3. Mini-batch Gradient Descent

In Mini-batch Gradient Descent(mBGD), we consider a mini-batch of examples at a time to take a single step. Specifically at each step we sample a mini-batch of b example from training set $[(x_1, y_1), \cdots, (x_n, y_n)]$ with $b << n$ and compute the gradient $\nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i)$ on this sample. The computational cost for each step drops from $O(n)$ of 4.3 to $O(b)$.

Algorithm 2.4 Mini-batch Gradient Descent(mBGD)

Require: Initial parameters $\boldsymbol{\theta}$
Require: Learning rate η
Require: Batch size b
1: **while** stopping criterion not met **do**
2: Sample a mini-batch of b examples from training set $[(x_1, y_1), \cdots, (x_n, y_n)]$.
3: Compute the gradient estimate: $\nabla c(\boldsymbol{\theta}) = \frac{1}{b} \sum_{i=1}^{b} \nabla L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i)$
4: Apply update: $\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla C(\boldsymbol{\theta})$
5: **end while**

It represents a good balance between BGD and SGD in terms of efficiency, robustness

and stochasticity that lead to escape from local minimum.

3 | Backpropagation Computation

When we use a feedforward network to compute an output y given an input $\mathbf{x}$, information flows forward through the network. This is called *forward propagation*. During training, forward propagation continues until it computes a scalar cost $L(f(\mathbf{x}, \boldsymbol{\theta}), y)$ that in the following we will denote simply as C. The *backpropagation* algorithm, often called *backprop*, allows the information from the cost to flow *backward* through the network in order to compute the gradient of the cost with respect to the network parameters $\boldsymbol{\theta}$.

3.1. Computational Graphs

To describe the backpropagation algorithm it is helpful to use *computational graph* language. Computational graph is a formalism used to express mathematical expressions by means of a directed graph. There exist many ways of formalizing computational graphs: here we use nodes to indicate operations, and edges to indicate variables, which may be a scalar, matrix, or tensor.

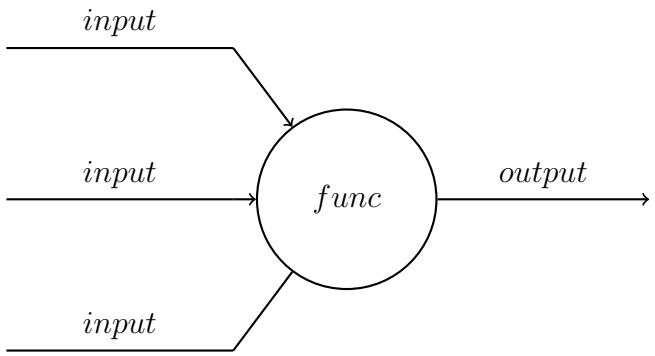

Figure 3.1: A general computational graph

By means of computational graphs it is possible to break down complex functions into a sequence of simpler operations, each one is represented by a node.

3.2. Chain Rule

The chain rule is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Backpropagation widely exploits the chain rule, with a specific order of operations which can be effectively represented using a computational graph. Let x be a real number, and let f and g both be functions from real numbers to real numbers. Suppose that $y = g(x)$ and $z = f(y) = f(g(x)) := F(x)$. Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{3.1}$$

Obviously the composition of functions can be formalized also by means of a computational graph as:

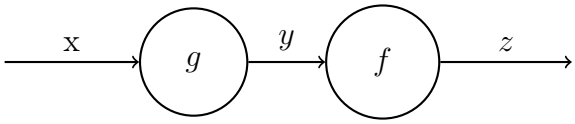

Figure 3.2: Computational graph of function $F(x)$

as well as the chain rule:

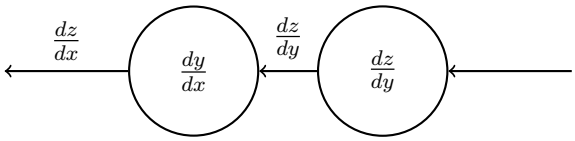

Figure 3.3: Computational graph of chain rule applied to the function $F(x)$

The chain rule can be generalized beyond the scalar case. Let $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$, let g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and let f maps from $\mathbb{R}^n$ to $\mathbb{R}$, then the *generalized chain rule* states that:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{3.2}$$

3.3. Computation Flow

Let us now consider a generic layer l belonging to the neural network we want to train. The computational graph represented below highlights the computation performed during

the forward pass: The dashed edges denote variables used only by the current layer l, while straight edges are variables that traverse the network. The input x comes from the previous layer $l-1$ and it is combined with the weights w and biases b of the layer l by means of a function f, which depends on the type of layer (convolutional, linear, etc.). The result a of this computation is then propagated as input to the next layer $l+1$.

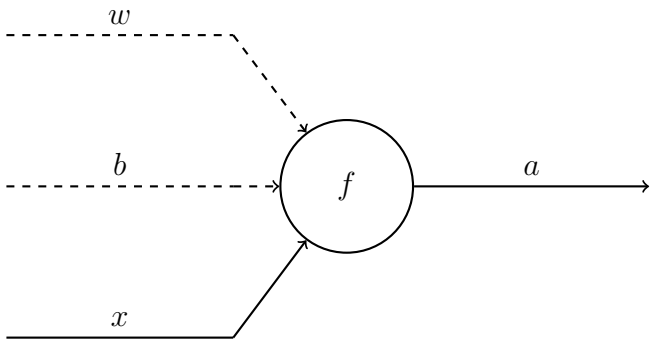

Figure 3.4: Computational graph of the forward pass computation in a parametric layer

Note that, in a neural networks, there are also non parametric layers, namely layers for which weights and biases are not defined. Examples of these layers are pool layers, flatten, and activation layers, which we treat as standalone layers in this document. For these layers the computational graph can be simplified as follows.

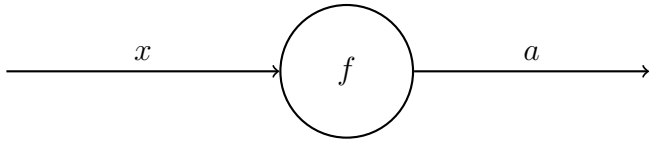

Figure 3.5: Computational graph of the forwars pass computation in a non-parametric layer

Similarly to the forward pass, we can exploit computational graphs also to represent computations performed during the backward pass. The computational graph for back-propagation through a parametric layer is represented in the above figure. Differently from the forward pass where we have multiple inputs to the function f, in backward pass we have multiple outputs. These outputs are actually the results of different computations that however, to simplify the notation, we encompass in one single node. The so called *error* $\frac{\partial C}{\partial a}$, coming from layer $l+1$, is the input of the node which computes the so called *local gradients* $\frac{\partial f}{\partial w}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial x}$, namely the partial derivatives of the node function with respect to the inputs. The outputs of the node are respectively the gradient with respect to the weights $\frac{\partial C}{\partial w}$, the gradient with respect to the biases $\frac{\partial C}{\partial b}$ and the new error $\frac{\partial C}{\partial x}$ which is

backward propagated to the layer $l-1$. Since we are dealing with two variables defined as error, to avoid misunderstandings, we refer to $\frac{\partial C}{\partial a}$ as *downstream error*, since comes from backward, and $\frac{\partial C}{\partial x}$ as *upstream error*, since goes upward.

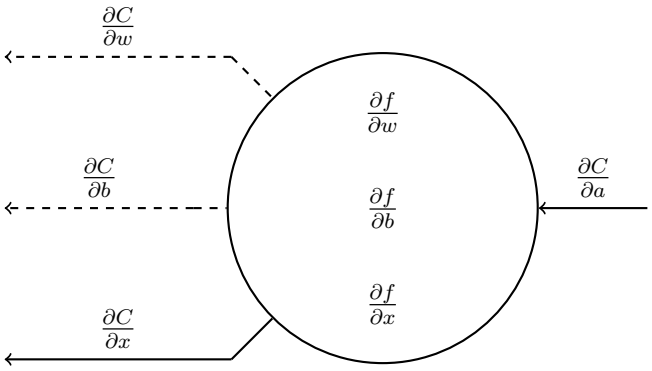

Figure 3.6: Computational graph of the backward pass computation in a parametric layer

Like in the case of the forward pass, for non parametric layers the computational graph of backpropagation can be simplified as follows.

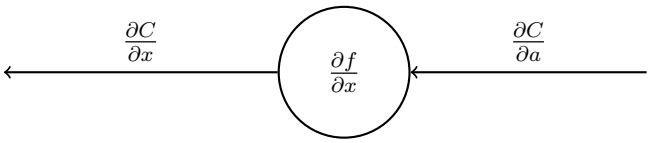

Figure 3.7: Computational graph of the backward pass computation in a non-parametric layer

Once the overall working flow of backpropagation is introduced, let us now go deeper into details of the computation. Indeed, depending on the type of layer, the specific computation as well as the dimension of variables involved in the process change. In the following we describe the specific backpropagation computation for the most common type of layers.

3.3.1. Dense Layer

Let us consider a dense layer l of M units and assume that the previous layer $l-1$ has N units, namely:

Forward Pass

- Input $\mathbf{x} \in \mathbb{R}^N$

- Weights $\mathbf{w} \in \mathbb{R}^{N \times M}$
- Biases $\mathbf{b} \in \mathbb{R}^{M}$
- Output $\mathbf{a} \in \mathbb{R}^{M}$.

In dense layers, the layer function f is defined as:

$$\mathbf{a} = f(\mathbf{x}, \mathbf{w}, \mathbf{b}) = \mathbf{w}^T \cdot \mathbf{x} + \mathbf{b} \tag{3.3}$$

Backward pass

- Downstream error $\frac{\partial C}{\partial \mathbf{a}} \in \mathbb{R}^{M}$
- Gradients wrt weights $\frac{\partial C}{\partial \mathbf{w}} \in \mathbb{R}^{N \times M}$
- Gradients wrt biases $\frac{\partial C}{\partial \mathbf{b}} \in \mathbb{R}^{M}$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{N}$
- Local gradients wrt weights $\frac{\partial f}{\partial \mathbf{w}} \in \mathbb{R}^{M \times N \times M}$
- Local gradients wrt biases $\frac{\partial f}{\partial \mathbf{b}} \in \mathbb{R}^{M \times M}$
- Local gradients wrt input $\frac{\partial f}{\partial x} \in \mathbb{R}^{M \times N}$

In order to compute the gradients of the cost C with respect to network parameters $\mathbf{w}, \mathbf{b}$ as long as the upstream error $\frac{\partial C}{\partial \mathbf{x}}$, generalized chain rule of Eq.(3.2) is exploited.

$$\frac{\partial C}{\partial \mathbf{w}} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{w}} \tag{3.4}$$

$$\frac{\partial C}{\partial \mathbf{b}} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{b}} \tag{3.5}$$

$$\frac{\partial C}{\partial \mathbf{x}} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{x}} \tag{3.6}$$

The *downstream error* $\frac{\partial C}{\partial \mathbf{a}}$ is assumed to be known from computations of layer $l+1$, thus we have only to compute the local gradients. We start to consider the partial derivative of f with respect to single variable as follows:

$$\frac{\partial f_i}{\partial w_{nm}} = \frac{\partial}{\partial w_{nm}} \sum_{j=1}^{N} w_{ji} x_j + b_i = \sum_{j=1}^{N} \delta_{ji} \delta_{mi} x_j = \delta_{mi} x_n \tag{3.7}$$

where $\delta_{i,j}$ denotes the *Kronecker delta* such that $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ otherwise. In Eq.(3.7) we remove the sum over j since the product goes to zero for all j different from i.

$$\frac{\partial f_i}{\partial b_m} = \delta_{im} \tag{3.8}$$

$$\frac{\partial f_i}{\partial x_n} = \frac{\partial}{\partial x_n} \sum_{j=1}^{N} w_{ji} x_j + b_i = \sum_{j=1}^{N} w_{ji} \delta_{jn} = w_{ni} \tag{3.9}$$

Once the formulae to compute *local gradients* are derived, it is possible to use them to compute gradients for the network parameters and *upstream error*.

$$\frac{\partial C}{\partial w_{nm}} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \delta_{im} x_n = \frac{\partial C}{\partial a_m} x_n \tag{3.10}$$

Similarly we can derive the gradient with respect to the (m)-th bias and the (n)-th *upstream error*a s follows:

$$\frac{\partial C}{\partial b_m} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \delta_{im} = \frac{\partial C}{\partial a_m} \tag{3.11}$$

$$\frac{\partial C}{\partial x_n} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} w_{ni} = \frac{\partial C}{\partial a_i} w_n \tag{3.12}$$

Without loss of generality we can extend the (3.10)-(3.12) to the general case:

$$\frac{\partial C}{\partial \mathbf{w}} = \mathbf{x} \cdot \left(\frac{\partial C}{\partial \mathbf{a}} \right)^T \tag{3.13}$$

$$\frac{\partial C}{\partial \mathbf{b}} = \frac{\partial C}{\partial \mathbf{a}} \tag{3.14}$$

$$\frac{\partial C}{\partial \mathbf{x}} = \mathbf{w} \cdot \frac{\partial C}{\partial \mathbf{a}} \tag{3.15}$$

Note that the use of *transpose* it is used for justify for a dimension.

3.3.2. Activation Layer

Let us consider an activation layer l of M units, namely:

Forward Pass

- Input $\mathbf{x} \in \mathbb{R}^M$

- Output $\mathbf{a} \in \mathbb{R}^M$.

The computation depends on the type of activation function f. Here we consider the *ReLU* and the *Softmax* cases. The ReLU is an element-wise function (namely, the i-component of the output a_i depends only on the i-component of the input x_i) defined as:

$$a_i = f(x_i) \begin{cases} x_i & \text{if } x_i > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.16}$$

The *Softmax* on the contrary is not an element-wise function (namely, the i-component of the output a_i depends on the whole input vector $\mathbf{x}$) defined as:

$$a_i = f(\mathbf{x}) = \frac{e^{x_i}}{\sum_{i=1}^{M} e^{x_i}}. \tag{3.17}$$

Observe that the Softmax is a rather special case as activation functions commonly used are actually defined element-wise.

Backward Pass

- Downstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^M$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^M$
- Local gradients wrt input $\frac{\partial f}{\partial x} \in \mathbb{R}^{M \times M}$

Let f be th ReLU activation function defined as in Eq.(3.16). The upstream error $\frac{\partial C}{\partial \mathbf{x}}$ can be computed element-wise exploiting the chain rule:

$$\frac{\partial C}{\partial x_m} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \frac{\partial f_i}{\partial x_m}. \tag{3.18}$$

The downstream error $\frac{\partial C}{\partial \mathbf{a}}$ is assumed to be known from computations of layer $l+1$, while the local gradient is computed as:

$$\frac{\partial f_i}{\partial x_m} = \begin{cases} 1 & \text{if } a_m > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.19}$$

Let now f be the Softmax activation function defined as in Eq.(3.17). Typically, this activation function is employed only in combination with the *Categorical Cross-Entropy* loss function in multi-class classification tasks. In this case, it can be demonstrated that

the computation of the upstream error can be simplified to:

$$\frac{\partial C}{\partial \mathbf{x}} = \mathbf{a} - \mathbf{t} \tag{3.20}$$

where $\mathbf{t}$ is a one hot encoded vector for the labels, so that $\sum_i t_i = 1$, and $t_m + \sum_{i \neq m} t_i = 1$.

3.3.3. Convolutional Layer

Let us consider a one dimension convolutional layer defined by the following hyperparameters: the kernel size K, the number of filters F, the number of input channels C and the of input units M.

Forward Pass

- Input $\mathbf{x} \in \mathbb{R}^{C \times M}$
- Weights $\mathbf{w} \in \mathbb{R}^{F \times C \times K}$
- Biases $\mathbf{b} \in \mathbb{R}^{F}$
- Output $\mathbf{a} \in \mathbb{R}^{F \times (M-K+1)}$.

In convolutional layers the layer function f is defined as:

$$\mathbf{a} = f(\mathbf{x}, \mathbf{w}, \mathbf{b}) = \text{conv}(\mathbf{x}, \mathbf{w}) + \mathbf{b}. \tag{3.21}$$

In particular the (j, m)-th element of the output $\mathbf{a}$ is defined as:

$$a_{jm} = \sum_{c=1}^{C} \sum_{k=1}^{K} x_{c,m+k-1} w_{jck} + b_j \tag{3.22}$$

Note that in (3.22) we treated *convolution* as *cross-correlation*.

Backward Pass

- Downstream error $\frac{\partial C}{\partial \mathbf{a}} \in \mathbb{R}^{F \times (M-K+1)}$
- Gradients wrt weights $\frac{\partial C}{\partial \mathbf{w}} \in \mathbb{R}^{F \times C \times K}$
- Gradients wrt biases $\frac{\partial C}{\partial \mathbf{b}} \in \mathbb{R}^{F}$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{C \times M)}$
- Local gradients wrt weights $\frac{\partial f}{\partial \mathbf{w}} \in \mathbb{R}^{F \times (M-K+1) \times N \times M \times F \times C \times K}$

- Local gradients wrt biases $\frac{\partial f}{\partial \mathbf{b}} \in \mathbb{R}^{F\times(M-K+1)\times M\times F}$
- Local gradients wrt input $\frac{\partial f}{\partial x} \in \mathbb{R}^{F\times(M-K+1)\times N\times C\times M}$

According to the generalized chain rule, the gradients with respect to the network parameters and the upstream error can be computed as:

$$\frac{\partial C}{\partial \mathbf{w}} = \sum_{j=1}^{F}\sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}}\frac{\partial f_{jm}}{\partial \mathbf{w}} \tag{3.23}$$

$$\frac{\partial C}{\partial \mathbf{b}} = \sum_{j=1}^{F}\sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}}\frac{\partial f_{jm}}{\partial \mathbf{b}} \tag{3.24}$$

$$\frac{\partial C}{\partial \mathbf{x}} = \sum_{j=1}^{F}\sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}}\frac{\partial f_{jm}}{\partial \mathbf{x}} \tag{3.25}$$

Again, since the downstream error is known, in order to perform computations in Eq.(3.23)-(3.25) we just have to compute the local gradients. Considering the derivative of f with respect to element-wise variables, we have that the (j, m)-th local gradient with respect to the (i, n)-th input is given by:

$$\frac{\partial f_{jm}}{\partial x_{in}} = \frac{\partial}{\partial x_{in}}\left(\sum_{c=1}^{C}\sum_{k=1}^{K} x_{c,m+k-1}w_{j,c,k} + b_j\right) \tag{3.26}$$

$$= \sum_{c=1}^{C}\sum_{k=1}^{K} \delta_{i,c}\delta_{n,m+k-1}w_{j,c,k} \tag{3.27}$$

$$= \sum_{k=1}^{K} \delta_{n,m+k-1}w_{j,i,k} \tag{3.28}$$

where $\delta_{i,j}$ denotes the *Kronecker delta* such that $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ otherwise. In Eq.(3.27) we remove the sum over c since the product goes to zero for all c different from i. Note that also the index of the weights change accordingly. Similarly we can derive the (j, m)-th local gradient with respect to the (j, i, n)-th weight as follows:

$$\frac{\partial f_{jm}}{\partial w_{qin}} = \frac{\partial}{\partial w_{qin}} \left(\sum_{c=1}^{C} \sum_{k=1}^{K} x_{c,m+k-1} w_{j,c,k} + b_j \right) \tag{3.29}$$

$$= \sum_{c=1}^{C} \sum_{k=1}^{K} \delta_{q,j} \delta_{i,c} \delta_{n,k} x_{c,m+k-1} \tag{3.30}$$

$$= \sum_{k=1}^{K} \delta_{q,j} \delta_{n,k} x_{i,m+k-1} \tag{3.31}$$

$$= \delta_{q,j} x_{i,m+n-1}, \tag{3.32}$$

Since the product goes to zero for all c different from i and for all n different from k we can remove the two summation. Note that again the index of x changes accordingly. The computation of the (j, m)-th local gradient with respect to the (j)-th bias is more simple:

$$\frac{\partial f_{jm}}{\partial b_q} = \frac{\partial}{\partial b_q} \left(\sum_{c=1}^{C} \sum_{k=1}^{K} x_{c,m+k-1} w_{j,c,k} + b_j \right) = \delta_{q,j} \tag{3.33}$$

Once we have computed local gradients, we can first compute global gradients and the upstream error for element-wise variables and then generalize them as vectors as follows:

$$\frac{\partial C}{\partial x_{in}} = \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \frac{\partial f_{jm}}{\partial x_{in}} \tag{3.34}$$

$$= \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \sum_{n=1}^{K} \delta_{n,m+k-1} w_{jik} \tag{3.35}$$

$$= \sum_{j=1}^{F} \sum_{k=1}^{K} \frac{\partial C}{\partial a_{j,n-k+1}} w_{jik} \tag{3.36}$$

The product goes to zero for all values of n except for $n = m + k - 1$ which expressed in terms of m is $m = n - k + 1$. So we can remove the sum over m and change the index of a accordingly. Note that the last Equation coincides the full convolution of the downstream error with the flipped kernel. So we can re-write it as:

$$\frac{\partial C}{\partial \mathbf{x}} = \text{conv} \left(\frac{\partial C}{\partial \mathbf{a}}, \text{flip}(\mathbf{w}), \text{full} \right) \tag{3.37}$$

Similarly the gradients with respect to the parameters can be computed as:

$$\frac{\partial C}{\partial w_{qin}} = \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \frac{\partial f_{jm}}{\partial w_{qin}} \tag{3.38}$$

$$= \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \delta_{q,j} x_{i,m+n-1} \tag{3.39}$$

$$= \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{qm}} x_{i,m+n-1} \tag{3.40}$$

The summation over j is removed since the product goes to zero for all j different from q and the index of a is changed accordingly. Note that the last Equation is the convolution of the downstream error with the input. So we can re-write is as:

$$\frac{\partial C}{\partial \mathbf{w}} = \text{conv}\left(\frac{\partial C}{\partial \mathbf{a}}, \mathbf{x}\right) \tag{3.41}$$

For the biases the computation is more simple:

$$\frac{\partial C}{\partial b_q} = \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \frac{\partial f_{jm}}{\partial b_q} \tag{3.42}$$

$$= \sum_{j=1}^{F} \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{jm}} \delta_{q,j} \tag{3.43}$$

$$= \sum_{m=1}^{M-K+1} \frac{\partial C}{\partial a_{q,m}} \tag{3.44}$$

Again the summation over j is removed since the product goes to zero for all j different from q and the index of a is changed accordingly.

3.3.4. Pool Layer

Let us consider a *pool layer* with *stride* p, therefore we have:

Forward Pass

- Input $\mathbf{x} \in \mathbb{R}^{F \times N}$
- Output $\mathbf{a} \in \mathbb{R}^{F \times (N/p)}$

The computation f depends on the considered type of pool function. A popular pool function is the *Average Pool* which calculates the average value for each patch of the input, which for one dimensional case is:

$$a_{jm} = f(\mathbf{x}) = \frac{1}{p}\sum_{i=0}^{p} x_{ji} \tag{3.45}$$

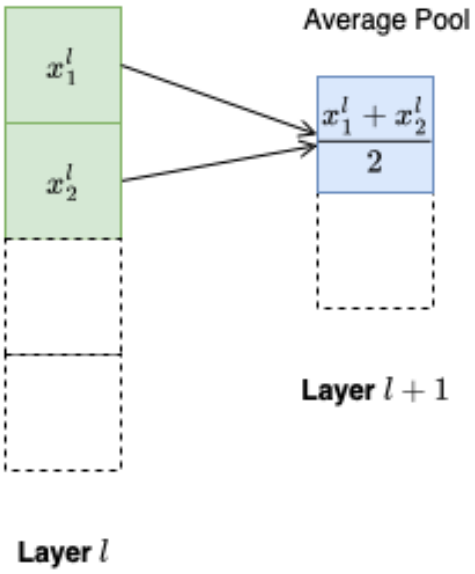

Figure 3.8: Forward Pass of Average Pool layer

Backward Pass

- Downstream error $\frac{\partial C}{\partial \mathbf{a}} \in \mathbb{R}^{F\times(N/p)}$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{F\times N}$

Regardless to the considered pool function, pool layers perform a down-sampling operation. On the contrary, in the backward pass the inverse computation in performed. Let us consider again the *Average Pool*, than in the backward pass we have:

$$\frac{\partial C}{\partial x_{in}} = \frac{1}{p}\frac{\partial C}{\partial a_{ij}} \qquad \text{with } j = [0, \cdots, p] \tag{3.46}$$

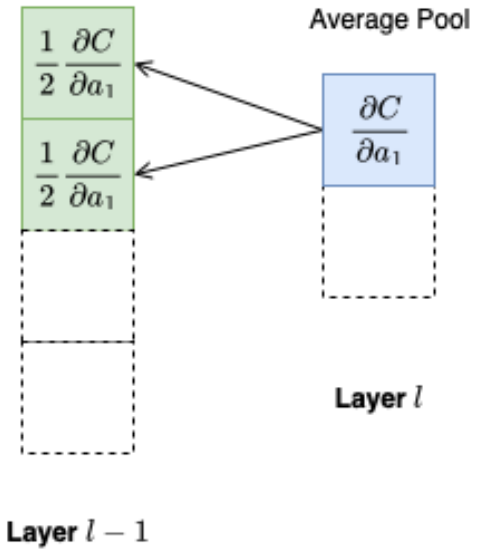

Figure 3.9: Backward Pass of Average Pool layer

3.3.5. Global Pool Layer

Let us consider a *global pool layer*, therefore we have:

Forward Pass

- Input $vx \in \mathbb{R}^{F \times N}$
- Output $va \in \mathbb{R}^{F}$

In this case the layer function f is defined as:

$$a_j = f(\mathbf{x}) = \frac{1}{N} \sum_{i=0}^{N} x_{ji} \tag{3.47}$$

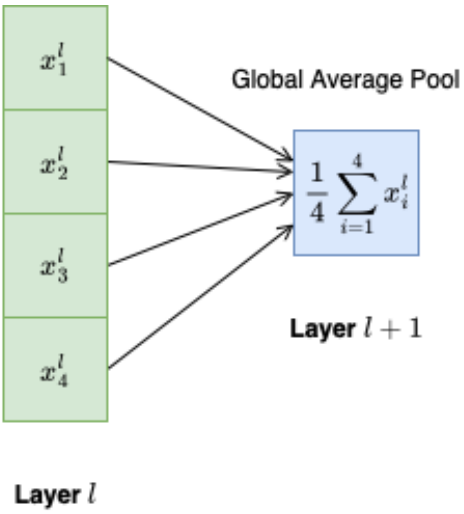

Figure 3.10: Forward Pass of Global Average Pool layer

Backward Pass

- Downstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^F$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{F \times N}$

$$\frac{\partial C}{\partial x_{ij}} = \frac{1}{N}\frac{\partial C}{\partial a_{in}} \qquad \text{with } n = [0, \cdots, N] \tag{3.48}$$

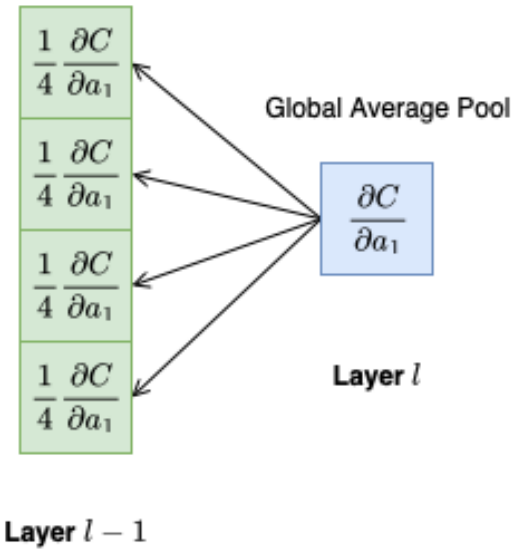

Figure 3.11: Backward Pass of Global Average Pool layer

3.3.6. Flatten Layer

Considering a Flatten layer, we can state that:

Forward Pass

- Input $\mathbf{x} \in \mathbb{R}^{F \times N}$
- Output $\mathbf{a} \in \mathbb{R}^{F \cdot N}$

In this case, the layer function f acts a *vectorization* of the input, which consists in a linear transformation that converts matrices into column vectors, namely:

$$\mathbf{a} = f(\mathbf{x}) = vec(\mathbf{x}) \tag{3.49}$$

Specifically, the vectorization of the input matrix $\mathbf{x}$ is the vector obtained by stacking $\mathbf{x}$ column-wise.

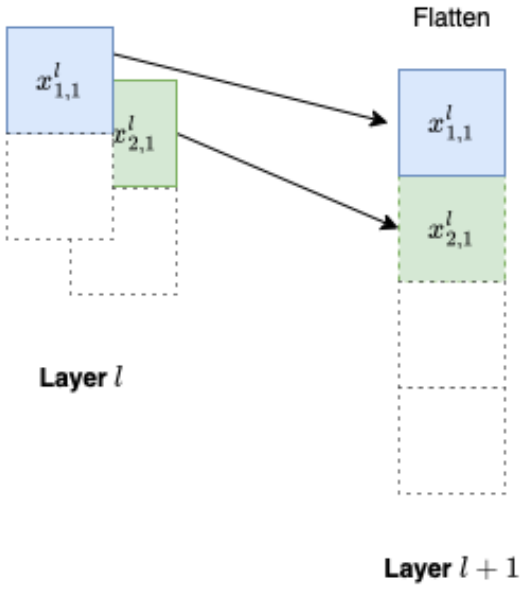

Figure 3.12: Forward Pass of Flatten layer

Backward Pass

- Downstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{F \cdot N}$
- Upstream error $\frac{\partial C}{\partial \mathbf{x}} \in \mathbb{R}^{F \times N}$

In the backward pass the inverse operation is performed.

$$\frac{\partial C}{\partial \mathbf{x}} = vec^{-1}(\frac{\partial C}{\partial \mathbf{a}}). \tag{3.50}$$

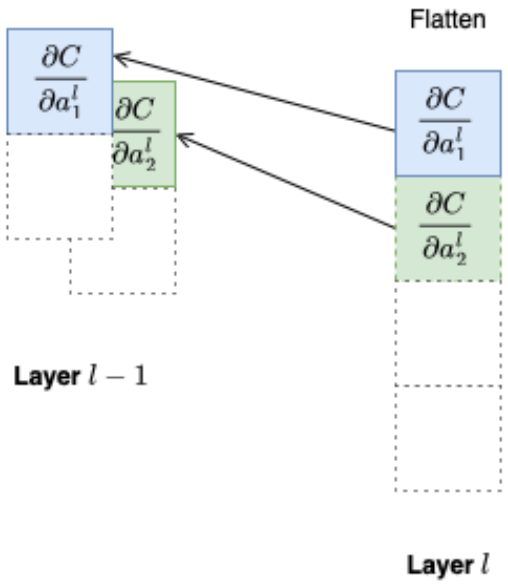

Figure 3.13: Backward Pass of Flatten layer

4 | Proposed Framework

4.1. Implemented Functionalities

Once the formulation of backpropagation is re-framed using computational graph which provide a more algorithmic view of the entire procedure, we proceed to the development of a framework in C with should satisfied the following functionalities:

- Define the topology of the network,
- Initialize network parameters randomly or with a specific configuration,
- Train the network using Mini-batch gradient descent,
- Freeze layers making them non trainable,
- Evaluate the performances of network.

4.2. Framework Structure

In order to decoupling the complexity of the problem different modules are created each one addresses different task, specifically we have implemented

- *Network Module* responsible of the network creation and initialization.
- *Training Module* responsible of overall training procedure.
- *Forward Module* responsible to perform the forward pass.
- *Backward Module* responsible to perform the backward pass.
- *Evaluation Module* responsible of the evaluate the performances network.

Specifically the network is defined by means of the *network module*, in which different functions allows the creation of the different layer of the network. Each layer is represented as structure that holds relevant information like the parameters, the activation and the gradients of such layer. Once the network is defined, each layer can automatically retrieve its *activations* or its *gradients* by invoking *forward* or *backward* modules which

implements the explicit expression provided in the Chapter 3 for each type of layer considered.
The overall training procedure is handled by the *training module* which allows to specify several custom training hyper-parameters such as the number of epochs, the learning rate and the batch-size.

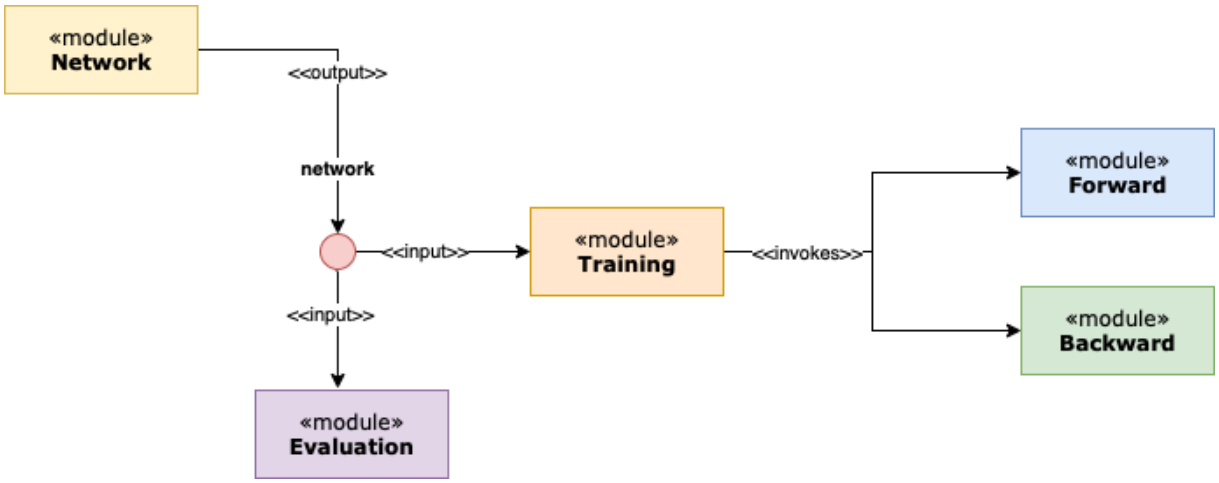

Figure 4.1: Modules of proposed framework

Algorithm 4.1 Training algorithm

Require: Network $\mathcal{N}$
Require: Training data $\mathcal{D}$
Require: Batch Size bs
Require: Number of Epochs ne
Require: Learning rate lr

```
for all epoch e in epochs ne do
    for all batch b in training data D do
        for all sample s in batch b do
            forward(N, d)
            backward(N, d)
        end for
        update(N)
    end for
end for
```

The logic of the *training module* is summarized in the above pseudo code. Once a neural network is created and the different training hyper-parameters are defined, the algorithm

iterate over training samples in the batch and perform forward and backward passes. During the different iterations the gradients computed for each sample of the batch are accumulated and then averaged when the update is performed. The *training module* rely on *forward module* and *backward module* which are called each time a new samples is presented to the network. The core of the training algorithm is encapsulated in *forward* and *backward module*, whose logic is very similar and is summarized in the following pseudo codes.

Algorithm 4.2 Forward algorithm

Require: Network $\mathcal{N}$
Require: Training sample s
for all layer l of network starting from the first **do**
  if layer.type == Convolutional **then**
    forward conv(l, s)
  end if
  if layer.type == Average Pool **then**
    forward avgpool(l)
  end if
  if layer.type == Global Average Pool **then**
    forward gap(l)
  end if
  if layer.type == Flatten **then**
    forward flatten(l)
  end if
  if layer.type == Dense **then**
    forward dense(l, s)
  end if
end for

As we can see the forward module iterate over the layers of the network and based on the type of the layer during the single step of the iteration the specific function which compute the *activations* of the layer is invoked. Since Convolutional and Dense layers may be the first layer of the network must have access to the values of the sample s.

Algorithm 4.3 Backward algorithm

Require: Network l
Require: Training sample s
for all layer l of network starting from the last **do**
  if layer l is trainable **then**
    if layer.type == Convolutional **then**
      backward conv(l,s)
    end if
    if layer.type == Average Pool **then**
      backward avgpool(l)
    end if
    if layer.type == Global Average Pool **then**
      backward gap(l)
    end if
    if layer.type == Flatten **then**
      backward flatten(l)
    end if
    if layer.type == Dense **then**
      backward dense(l,s)
    end if
  end if
else
  break
end for

Similarly the *backward module* iterate over the layers of the network and, based on the type of the layer during the single step of the iteration, the specific function and for compute the *error* and eventually to compute the *gradient* is invoked. The loop is break when a non trainable layer is encountered.

The proposed framework is written without the support of specific libraries and in a way that it can also be run on micro-controllers (MCUs). The only constraint regards the computations to be performed in floating point, which require a Floating Point Unit (FPU) to be performed.

4.3. Assessment on STM32L4R9

The target platform for validating the developed framework is the STM32L4R9 [1], an ultra-low-power MCU produced by ST-Microelectronics with an ARM Cortex-M4 core and 640 KB of sRAM. The firmware is compiled and flashed on the device using STM32CubeIDE [2] then it is tested on the baseline of TensorFlow.
Specifically several NN are trained on PC using TensorFlow starting from a given initialization. The same training procedure is then reproduced on the STM32L4R9 using the developed framework. For each of considered models, the parameters at end of the training procedure performed respectively by TensorFlow and the developed framework were the same. Moreover a more rigorous approach has been followed to check the effectiveness of the training procedure on the device: for each pass of gradient descent and for each layer of the network, the computed gradients have been compared with respect to the TensorFlow baseline.
The considered NN for the validation of the developed framework is made up by two Convolutional1D layers with ReLU intervaled by AveragePool1D layer a GlobalAveragePool1D and two Dense layer with ReLU and Softmax respectively. The different NN differ from each other in the number of filters and kernel size in Convolutional layers as well as in the number of units in Dense layers and the input size.
The training procedure on STM32L4R9 has also been monitored in terms of memory footprint and execution time. For training all layers of a *CNN* with around 10000 parameters and an input sample of size (20,3) the execution time is 0.23 seconds per samples and the estimated memory footprint is 97 KB while for a fine-tuning of two last layers of the same model the execution time is 0.003 seconds per samples and the estimated memory footprint is of 65KB.

5 | Required Computing Resources for Training

Since the target platforms for code development are *STM32* devices, which are *MPUs* and so a resource-constrained devices, it is necessary to perform a feasibility analysis on the required computing resources for running the back-propagation algorithm.
The analysis is based on the estimation of memory footprint and CPU load which are the most critical computing resource to take in consideration.

5.1. Memory Footprint Estimation

The word *footprint* refers to the extent of physical dimensions that an object occupies, giving a sense of its size. In this sense *memory footprint* refers to the amount of main memory which a program occupies while it is running.
Our objective is to figure out how much memory requires the program which implement neural network training. Although it represent only an estimation (the real *memory footprint* depends on a lot of other factors) anyhow the data processed by the algorithm will be the heaviest source of memory requirements. Once we know the total amount of data involved in the network training, we can easily retrieve the related *memory footprint* by simply multiplying the total amount by the number of bits used to store the single datum, namely bit precision.
Note that the memory footprint estimation is useful not only to check the feasibility of training a given neural network on a specific device, but also is used to know the exact quantity of memory that we have to reserve in order store data involved in the training process, since memory, in the developed framework, is statically allocated. There are tree types of data processed by the training algorithm:

- Network parameters,
- Training samples,
- Quantities computed during BP.

5.1.1. Network Parameters

The first source of *memory footprint* are the network parameters, namely weights **w** and biases **b**, which is used both in forward/backward passes and during the update phase of parameters themselves. These data are employed beyond the training phase, indeed they are needed for inference. For this motivation it's reasonable consider them as a source of *memory footprint* not mandatory linked to the training itself.

In the table below are reported the formulae for computing the number of parameters for the different parametric layers of a network.

Layer	w	b
Dense	$M \cdot N$	N
Conv1D	$F \cdot C \cdot K$	F

Table 5.1: Memory footprint of the network parameters

Where:

- M is the number of neurons in the previous layer,
- N is the number of neurons in the current layer,
- F is the number of filters in the current,
- C is the number of filters in the previous layer,
- K is the kernel size.

The overall footprint is the sum of the total numbers of parameters multiplied by bit precision B.

For having a more clear idea on what is the memory impact of the network parameters let's consider a practical example. In the table below the columns **w** and **b** report the quantity of values to store, while the column *Footprint* reports the related memory footprint to store such values.

Layer	Output Shape	w	b	Footprint
Input(20,3)	(20,3)	-	-	-
Conv1D(F=32,K=3)	(18,32)	288	32	1 KB
AvgPool1D()	(9,32)	-	-	-
Conv1D(F=64,K=3)	(7,64)	6144	64	24 KB
AvgPool1D()	(3,64)	-	-	-
GlobalAvgPool1D()	(64)	-	-	-
Dense(50)	(50)	3200	50	12 KB
Dense(6)	(6)	300	6	1 KB
TOTAL	-	9932	124	39 KB

Table 5.2: Example of the memory footprint of network parameters

The parameters are threaded as single precision floating point thus B will be of 32 bit.

5.1.2. Training Samples

The second source of *memory footprint* is the data used for training the network.
Usually we do not have to load on memory the whole dataset, assuming that the network is trained with a *mini-batch* approach, it is reasonable to assume that on the memory is loaded a single batch of data at time. Training samples can be employed not only in the training procedure, indeed they could be used for testing purposes and during the inference.
The related memory footprint will be $B \cdot bs \cdot |S|$ where bs is the batch size and $|S|$ is the zie of the single sample.

For having a more clear idea let's consider a pratical example. With batch size of 32 and an input sample of shape $(20, 3)$ stored with single precision floating point, the footprint of a single batch will be 6 KB.

5.1.3. Quantities computed during BP

The last source of *memory footprint* is related to the quantities computed during BP and it is the one that requires more memory.
We know that backpropagation consist in two step, the forward pass in which *activations*

are computed and the backward pass in which *errors* are backpropagated and *gradients* with respect to parameters are computed. Note that all of these variables must be stored on memory for the whole training process. The *activations* must be on memory because it used in the backward pass which start from the last layer and iteratively goes back till the first one, when the related *activations* are used. Analogously the *gradients* must be on memory since they are accumulated as the input in the batch are processed, then they are averaged when the update phase is performed. Moreover since on *MCU* the memory is not allocated dynamically, also *errors* must be on always memory. Thus the *memory footprint* for data produced during backpropagation is the sum of the size of these variables multiplied by bit precision.

In the table below are reported the formulae for computing the size of these variables for the different layers of a network.

Layer	**a**	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$
Dense	N	$M \cdot N$	N	M
Conv1D	$F \cdot (M - K + 1)$	$F \cdot C \cdot K$	F	$C \cdot M$
Flatten	$F \cdot M$	-	-	$F \cdot M$
GloabalAvgPool1D	F	-	-	$F \cdot M$
AvgPool1D	$\frac{F \cdot M}{p}$	-	-	$F \cdot M$

Table 5.3: Memory footprint of quantities computed during BP

Where:

- M is the width oh the previous layer,
- N is the width of the current layer,
- F is the number of filters in the current layer
- C is the number of filters in the previous layer,
- K is the kernel size,
- p is the stride of the average pooling layer.

All parametric layers are usually followed by an activation layer, which apply an activation function element wise. Since the size of the two layers is the same, in the developed framework the output values of parametric layer will be overwritten by the output of following activation layer. Under this consideration for evaluating the footprint we consider

these two layer merged.

For having a more clear idea on what is the memory impact of the quantities processed during backpropagation let's estimate the footprint of the example network.

Layer	Output Shape	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	Footprint
Input(20,3)	(20,3)	-	-	-	-	-
Conv1D(32,3)	(18,32)	576	288	32	576	6 KB
AvgPool1D(2)	(9,32)	288	-	-	288	2 KB
Conv1D(64,3)	(7,64)	448	6144	64	448	28 KB
AvgPool1D(2)	(3,64)	192	-	-	192	2 KB
GlobalAvgPool1D()	(64)	64	-	-	64	1 KB
Dense(50)	(50)	50	3200	50	50	13 KB
Dense(6)	(6)	6	300	6	6	1 KB
TOTAL	-	1664	9932	124	1664	53 KB

Table 5.4: Example of memory footprint of quantities computed during BP

Also in this case we assume that all values are stored as single-precision floating point and so with a bit resolution of 32 bit.
It's important to emphasize that the *gradients* and the *errors* are produced only for trainable layers. If one or more layers of the network are *freezed*, the training for that layers is not performed and thus the related *gradients* and *errors* are not computed nor stored on memory. As a consequence also the related *activations*, which needed no more, can be forgotten.
In the developed framework, instead of allocating the memory for storing all the *activations* of *freezed* layers, it is implemented a method which uses a smaller portion of memory which acts as a over-writable buffer with a further enhancement of memory footprint. Specifically the size of the buffer correspond to the size of the greatest *activations* of *freezed* layers multiplied by two.

In the table below is shown the footprint of the example network with all layers freezed except for the last two Dense layers.

Layer	Output Shape	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	Footprint
Input(20,3)	(20,3)	-	-	-	-	-
Conv1D(F=32,K=3)	-	-	-	-	-	-
AvgPool1D(p=2)	(9,32)	-	-	-	-	-
Conv1D(F=64,K=3)	(7,64)	448	-	-	-	4 KB*
AvgPool1D(p=2)	(3,64)	-	-	-	-	-
GlobalAvgPool1D()	(64)	-	-	-	-	
Dense(50)	(50)	50	3200	50	50	13 KB
Dense(6)	(6)	6	300	6	6	1 KB
TOTAL	-	952	3500	56	56	18 KB

Table 5.5: Example of memory footprint of quantities computed during BP with all layers freezed except for the last two

Note that 4 KB* refers to the footprint of the buffer in which *activations* of *freezed* layers are stored.

5.1.4. Memory Footprint of the Training

Once it is showed how compute the memory footprint for the different types of data involved in the training of a network, highlighting also the case in which some layers are *freezed*, we show the procedure to compute the overall footprint estimation.

Algorithm 5.1 BP Memory Footprint Estimation

Require: Network $\mathcal{N}$
Require: Input Size $\mathcal{I}$
Require: Batch Size bs
Require: Bit Precision $\mathcal{B}$
Parameters Size: $par \leftarrow 0$
Training Data Size: $data \leftarrow bs \cdot \mathcal{I}$
Activations Size: $act \leftarrow 0$
Gradients Weights Size: $wGrad \leftarrow 0$
Gradients Biases Size: $bGrad \leftarrow 0$
Gradients Input Size: $xGrad \leftarrow 0$
Greatest Activations Size: $max \leftarrow 0$
Memory Footprint: $mF \leftarrow 0$
for all layer ℓ in $\mathcal{N}$ **do**
  if ℓ is trainable **then**
    $act \leftarrow act + dim(a^\ell)$
    if ℓ is parametric layer **then**
      $par \leftarrow par + dim(w^\ell) + dim(b^\ell)$
      $wGrad \leftarrow wGrad + dim(w^\ell)$
      $bGrad \leftarrow bGrad + dim(b^\ell)$
    end if
    $xGrad \leftarrow xGrad + dim(x^\ell)$
  end if
  if ℓ is non-trainable **then**
    if $dim(a^\ell) > max$ **then**
      $max \leftarrow dim(a^\ell)$
    end if
    if ℓ is parametric layer **then**
      $par \leftarrow par + dim(w^\ell) + dim(b^\ell)$
    end if
  end if
end for
$mF \leftarrow B \cdot (par + data + act + wGrad + bGrad + xGrad + max)$
return Memory Footprint mF

The Algo. 5.1 traverse the network layer by layer and compute the dimensions (dim) of all data processed by the training process in the considered layer both during forward and

backward passes. The dimensions are cumulated while the network is traversed, and then multiplied by bit precision.

In the following tables we summarize the memory footprint of the whole training process of the previous examples:

Layer	Output Shape	w	b	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	Footprint
Input(32,20,3)	(20,3)	-	-	-	-	-	-	6 KB
Conv1D(32,3)	(18,32)	288	32	576	288	32	576	7 KB
AvgPool1D(2)	(9,32)	-	-	288	-	-	288	2 KB
Conv1D(64,3)	(7,64)	6144	64	448	6144	64	448	52 KB
AvgPool1D(2)	(3,64)	-	-	192	-	-	192	2 KB
GlobalAvgPool1D()	(64)	-	-	64	-	-	64	1 KB
Dense(50)	(50)	3200	60	50	3200	50	50	24 KB
Dense(6)	(6)	300	6	6	300	6	6	2 KB
TOTAL	-	9932	124	1664	9932	124	1664	98 KB

Table 5.6: Example of memory footprint for the whole training process

Layer	Output Shape	w	b	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	Footprint
Input(32,20,3)	(20,3)	-	-	-	-	-	-	6 KB
Conv1D(32,3)	(18,32)	288	32	-	-	-	-	1 KB
AvgPool1D(2)	(9,32)	-	-	-	-	-	-	2 KB
Conv1D(64,3)	(7,64)	6144	64	448	-	-	-	28 KB
AvgPool1D(2)	(3,64)	-	-	-	-	-	-	-
GlobalAvgPool1D()	(64)	-	-	-	-	-	-	-
Dense(50)	(50)	3200	60	50	3200	50	50	24 KB
Dense(6)	(6)	300	6	6	300	6	6	2 KB
TOTAL	-	9932	124	1664	3500	56	56	63 KB

Table 5.7: Example of memory footprint for the whole training process with all layer freezed except for dense

5.2. CPU Load Estimation

In order to figure out how expensive is the implemented training procedure a rough estimate which consist in counting the the total number of operations performed by the algorithm is proposed. Specifically the operations considered are that performed in forward and in backward pass, as well as in parameters' update phase. The number and the type of operations performed depend on the type of layer considered.Since operation are performed using floating point numbers we refer to them as Floating Point Operations (FLOPs)

Given a *Dense Layer* we have that:

- The *activations* in (3.3) perform two computation, the first is matrix-vector product which counts N multiplications and $N-1$ additions for each of M row, the second is a vector-vector sum which counts M addition. The total number of operation is $2 \cdot M \cdot N + M$.
- The *gradients* in (3.13) is a vector-vector outer product which counts $M \cdot N$ multiplications.
- The *gradients* in (3.15) is matrix-vector product which counts M multiplications and $M-1$ additions for each of N row. The total number of operation is $2 \cdot M \cdot N$.

Given an *Activation Layer* we have that:

- The *activations* as $ReLU$ in (3.16) perform an element wise computation which counts one comparison for each element for a total of M comparison, if we consider instead the *Softmax* in (3.17) for each element are performed one exponential for the numerator, M exponential and $M-1$ sum for the denominator plus the division between numerator and denominator, for each of M row are performed $2M$ operation for a total of $2 \cdot M^2$ operations.
- The *gradients* in case we use $ReLU$ in (3.19) counts one comparison for each element for a total of M comparison if we use *Softmax* in (3.20) it is performed a vector-vector subtraction for a total of M operation

Given a *Convolutional Layer* we have that:

- The single *activation* in (3.22) counts CK multiplication $(C-1) \cdot (K-1)$ addition plus one addition. The computation repeated $F \cdot (M-K+1)$ times. The total number of operations is $2 \cdot F \cdot C \cdot K \cdot M$.

- The single partial derivative in (3.36) counts $F \cdot K$ multiplication $(F-1) \cdot (K-1)$ addition. The computation repeated $C \cdot N$ times. The total number of operations is $2 \cdot F \cdot C \cdot N \cdot K$.
- The single partial derivative in (3.40) counts $M - K + 1$ multiplication $M - K$ addition. The computation repeated $F \cdot C \cdot K$ times. The total number of operations is $2 \cdot F \cdot C \cdot M \cdot K$.
- The single partial derivative in (3.44) counts $M - K + 1$ multiplication $M - K$ addition. The computation repeated F times. The total number of operations is $2 \cdot F \cdot M$.

Given a *Average Pooling Layer* we have that:

- The single *activation* in (3.45) counts one multiplication and $p - 1$ addition. The computation repeated $F\lfloor M/p \rfloor$ times for a total of $F \cdot M$ operations.
- The single *error* in (3.46) counts one multiplication. The computation is repeated $F \cdot N$ times.

Given a *Global Average Pooling Layer* we have that:

- The single *activation* in (3.47) counts one multiplication and $M - 1$ addition. The computation repeated F times for a total of FM operations.
- The single *error* in (3.48) counts one multiplication. The computation is repeated F times.

In the following table are summarized the FLOPs for each layer of a network.

Layer	**a**	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$
Dense	$2 \cdot M \cdot N + M$	$M \cdot N$	-	$2 \cdot M \cdot N$
Act	M or $2 \cdot M^2$	-	-	M
Conv1D	$2F \cdot C \cdot K \cdot M$	$2 \cdot F \cdot C \cdot K \cdot N$	$2 \cdot F \cdot M$	$2 \cdot F \cdot C \cdot K \cdot M$
AvgPool1D	$F \cdot M$	-	-	$F \cdot N$
GlobalAvgPool1D	$F \cdot M$	-	-	F

Table 5.8: FLOPs count per layer

So considering the example network we have:

	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	**FLOPs**
Input(20,3)	-	-	-	-	-
Conv1D(F=32,K=3)	11520	10368	1280	11520	34688
AvgPool1d(p=2)	576	-	-	288	846
Conv1D(F=64,K=3)	110592	86000	1152	110592	300000
AvgPool1D(p=2)	448	-	-	192	640
GlobalAvgPool1D()	128	-	-	64	200
Dense(50)	6464	3200	-	6400	1600
Dense(6)	612	300	-	600	1500
TOTAL	130952	100124	2432	199656	360732

Table 5.9: Example of FLOPs count of a network

As we can notice the computation performed in convolutional layers are the most expensive in terms of FLOPs.
By freezing all layers of the network except for the last two, the number of FLOPs can be reduced.

	a	$\frac{\partial C}{\partial \mathbf{w}}$	$\frac{\partial C}{\partial \mathbf{b}}$	$\frac{\partial C}{\partial \mathbf{x}}$	**FLOPs**
Input(20,3)	-	-	-	-	-
Conv1D(F=32,K=3)	11520	-	-	-	11520
AvgPool1d(p=2)	576	-	-	-	576
Conv1D(F=64,k=3)	110592	-	-	-	110592
AvgPool1D(p=2)	448	-	-	-	448
GlobalAvgPool1D()	128	-	-	-	128
Dense(50)	6464	3200	-	6400	1600
Dense(6)	612	300	-	600	1500
TOTAL	130952	3500	-	7000	141452

Table 5.10: Example of FLOPs count of a network with all layers freezed except for last two

6 | Training Neural Network on device for Human Activity Recognition

Once the framework to perform training on device has been developed and assessed on a real device, we want to highlight the benefits that the training on device can bring.
We prove, by means of several experiments, that by training a pre-trained model using data that are never seen during the training, the pre-trained model adapts to the new data and improves its performances. This task is known as personalization.
The experimental evaluation is performed in a simulated environment and validated on the HAR use case.
Note that the objective of the proposed experiments is not addressed to achieve the state of art performances in HAR nor to investigate on best personalization strategies. The experiments are conducted in order to highlight the enhancements provided by training on device functionalities, which also enable the personalization of a pre-trained model.

6.1. Human Activity Recognition

HAR is the problem of predicting the activity performed by a person based on sensor data. A large number of sensors' type can be used to perform motion tracking, recently particular attention is given to wearable sensors, such as accelerometer, gyroscope and magnetometer. This type of sensors can be integrated into embedded systems which, once are attached to a body part of a person, they can be used to monitor the movements performed. Specifically one of the most used sensor in HAR, is the acceleromenter, which measures the acceleration of the movements along the tree axis x, y and z as shown in the image below.

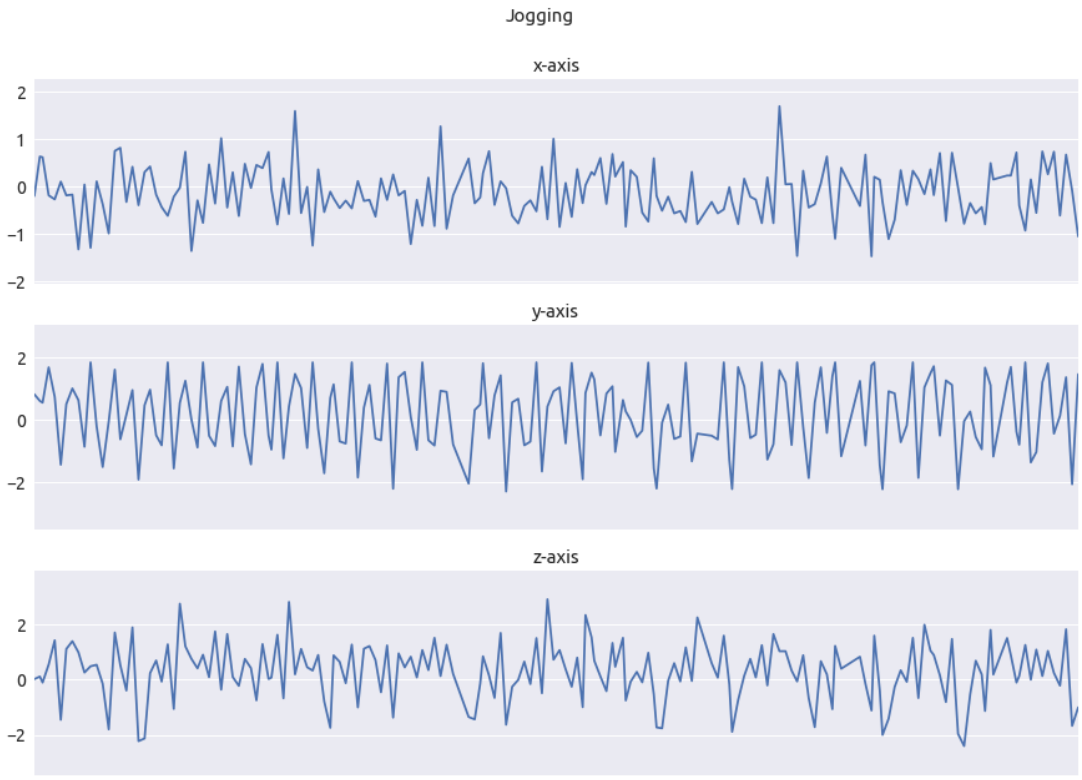

Figure 6.1: Example of the accelerometer raw data related to the Jogging activity

The choice of this use case is perfectly suitable for our task since, in order to track human motion, embedded systems are commonly used, moreover to perform the recognition of activities, which are typically subject-dependent due to the user's physical characteristics, a model personalized on the specific subject movement is more effective then a model trained with the data collected from different subjects. Moreover in this use case the data available for the training on device is possibly very limited, and certainly not enough to train a full model from scratch. The sensed data could be instead enough to further enhanced the performances of a pre-trained model By means of personalization.
In HAR, Deep Learning (DL) methods, in particular CNNs, achieve state-of-the-art results. The benefit of using CNNs is that they can learn from the data directly, with no requirement of an expert-driven approach for extract features from data.
The use of CNN, however, requires that raw sensor data must be prepared in a specific manner in order to fit a model. Usually the preparation consists in the split of the data sequences into subs-sequences called windows, each of them is associated with an activity.

6.2. Proposed Experiments

In the proposed experiments we use a CNN, made up by the following layers: Convolutional 1D with 32 filters and a kernel size of 3 with ReLU, Avgerage Pooling 1D, Convolutional 1D of 64 filters and a kernel size of 3 with ReLU, Avgerage Pooling 1D , Global Avgerage Pooling 1D, Dense of 50 units with ReLU, Dense of 6 units with Softmax. Raw accelerometer data is framed into windows of a given input size before training the network. We consider widows that space from 1 to 5 second which correspond to the following input sizes: (100,3),(80,3),(60,3),(40,3),(20,3) each of which is evaluated in a specific experiment.

Thanks to the Global Avgerage Pooling, the variation of input size does not affects the number of total network parameters which, for each experiment, are initialized with the same configuration.

Two strategies of personalization are taken into account: the first approach consists in the train of all layers in the pre-trained model, we refer to it as full train and the second strategy consists in the train of two last layers of the same model, we refer to it as partial train. Moreover different input size are taken into account. The size of the input data affects the overall memory footprint. By considering different input size it is possible to assess how effective is the personalization of simpler models whose train is supported also by device with very limited memory.

The dataset considered for the experiments is the Wireless Data Mining (WISDM) [14] in which six activities are taken into account: walking, jogging, ascending stairs, descending stairs, sitting, and standing. The WISDM dataset is generated by 36 subjects and the data is collected using a 3-axial accelerometer at 20HZ. The graph below shown the class distribution of the raw-data.

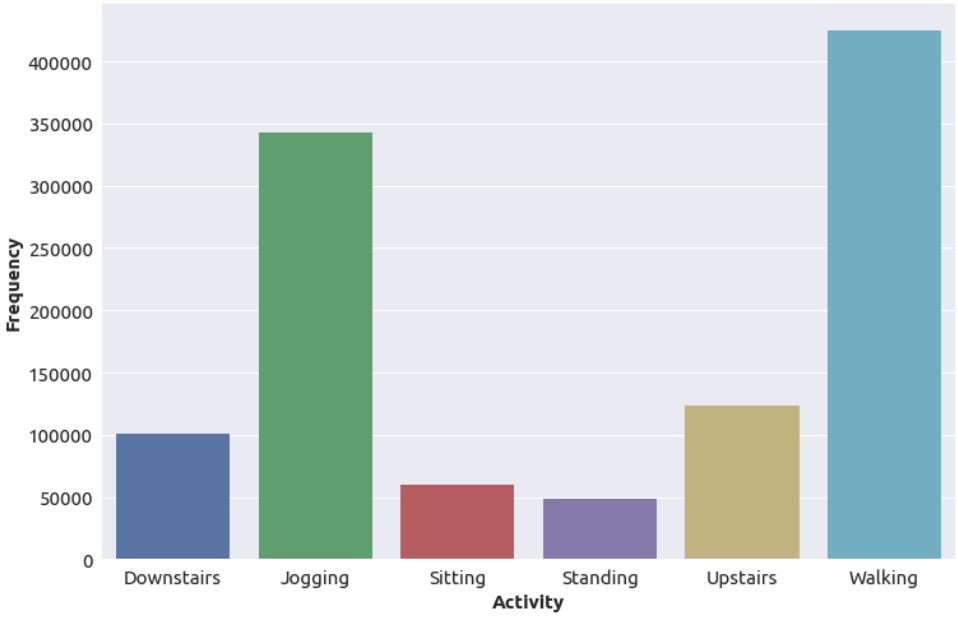

Figure 6.2: Class distribution of WISDM dataset

As can be noticed the classes distribution is highly unbalanced.

The working flow of each experiment can be summarized as follows:

- The neural network is preliminary trained for 100 epochs with a learning rate of 0.01 and a batch size of 32 on the data of all users except one. In this step the pre-trained model is generated.
- The data of the excluded user, which represents the local data used for personalize the pre-trained model, is splitted in two parts, one used for the train and one used for the test.
- The pre-trained model is evaluated on the local test data.
- The pre-trained model is then personalized over the local train data by re-training the full model for 10 epochs with a learning rate of 0.01 and a batch size of 32. Then the resulting model is evaluated on local test data.
- The pre-trained model is personalized over local train data by fine-tuning of the last two layers for 10 epochs with a learning rate of 0.01 and batch size of 32. Then the resulting model is evaluated on the local test data.

Since the pre-trained model is personalized for a specific user and in the WISDM dataset we have 36 users, we perform the personalization for each of them and then we aggregate

performances of each model into a unique representative estimation. Specifically the considered metrics, due to the umbalancess of class distributions in WISDM dataset are Precision, Recall and F1. We compute these metrics for each class and then we perform a weighted average which considers the number of instances of each class.
Although the experimental evaluation is performed in a simulated environment, each single experiment is designed to run also on real devices. For each experiment an analysis of memory footprint related to the training process of the considered model is performed. In the following sections we report the results of each experiment.

6.2.1. Experiment 1: window size of 5 seconds

In the Experiment 1 we segment raw data with a window size of 5 seconds, corresponding to an input size for the model of (100,3). The model is preliminary trained and then evaluated on test data reporting the following results:

Metrics	No train
Precision	0.8454
Recall	0.8245
F1	0.8323

Table 6.1: Evaluation of pre-trained model of Experiment 1

As can be noticed the pre-trained model has not highly generalization capabilities due to the class imbalances of the dataset. This is confirmed by the confusion matrix of the pre-trained model's predictions on the test data.

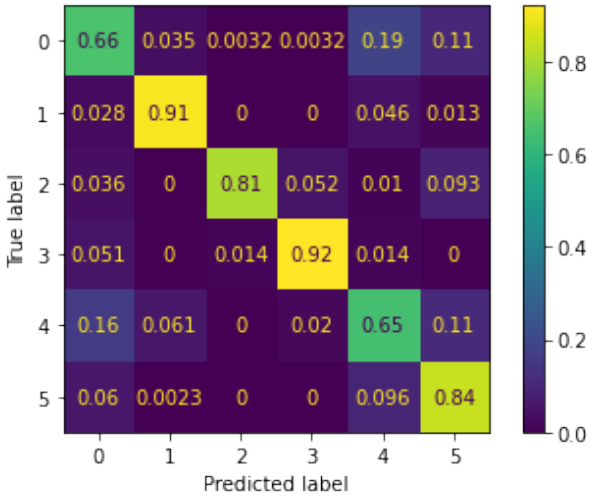

Figure 6.3: Confusion matrix of pre-trained model of Experiment 1

By means of personalization, however, the performances of the pre-trained model are significantly enhanced.

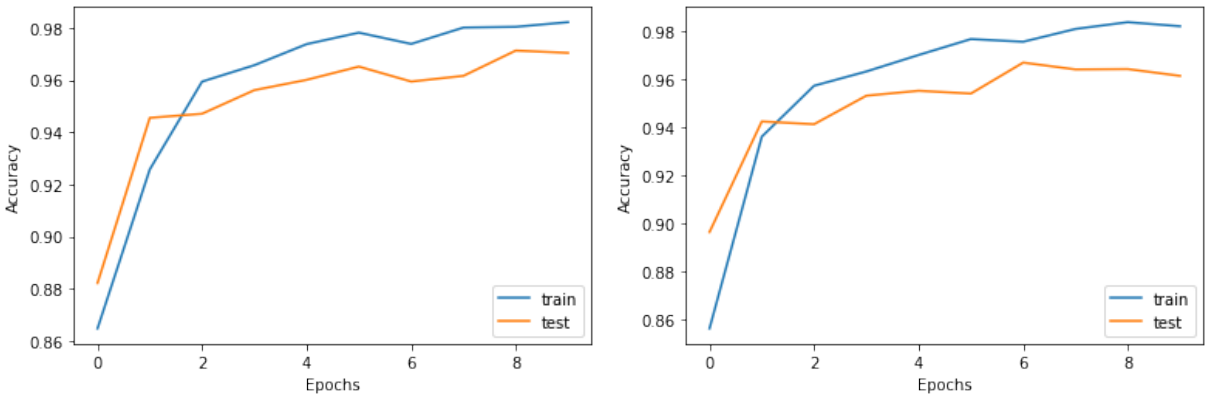

Figure 6.4: Training of the pre-trainied model of Experiment 1 on local data

The above figure shows how the accuracy of the pre-model increases while it is trained with the two personalization approaches. The graph on the left refers to the full training personalization approach, while the graph on the right the partial training. The latter approach, although starts with an higher accuracy, is more prone to over-fitting. Once the re-training of the pre-trained is performed the resulting models have been evaluated.

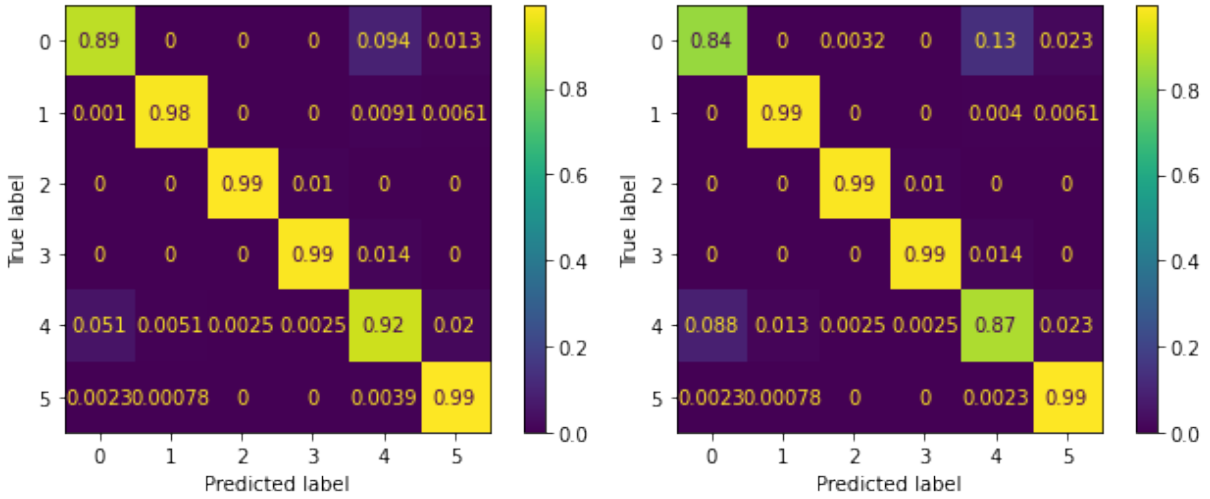

Figure 6.5: Confusion matrix of personalized models of Experiment 1

Metrics	Full	Partial
Precision	0.9718	0.9630
Recall	0.9715	0.9634
F1	0.9716	0.9631

Table 6.2: Evaluation of personalized models of Experiment 1

As we can notice the full re-training of the model reach better results. The re-training of the proposed models with the two personalization approaches requires the following amount of memory:

- Networks parameters: 39 KB
- Training samples in a batch: 38 KB
- Quantities processed by BP Full training: 112 KB
- Quantities processed by BP Partial training: 38 KB

So the full training approach requires a total of 189 KB while the partial training 115 KB.

6.2.2. Experiment 2: window size of 4 seconds

In the Experiment 2 we segment raw data with a window size of 4 seconds, corresponding to an input size for the model of (80,3). The model is preliminary trained and then evaluated on test data reporting the following results:

Metrics	No train
Precision	0.8605
Recall	0.8453
F1	0.8511

Table 6.3: Evaluation of pre-trained model of Experiment 2

As can be noticed the pre-trained model has not highly generalization capabilities due to the class imbalances of the dataset. This is confirmed by the confusion matrix of the pre-trained model's predictions on the test data.

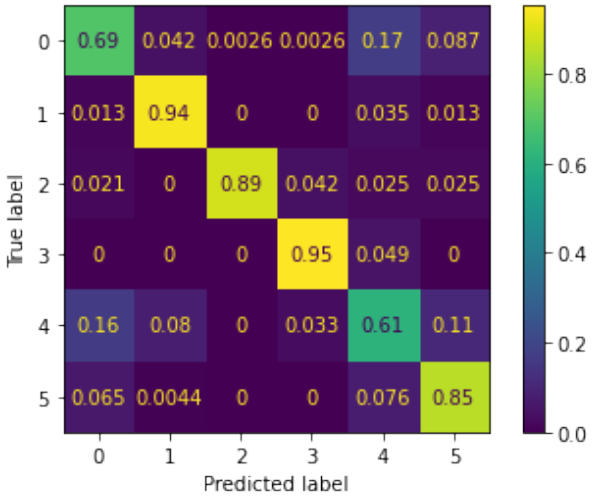

Figure 6.6: Confusion matrix of pre-trained model of Experiment 2

By means of personalization, however, the performances of the pre-trained model are significantly enhanced.

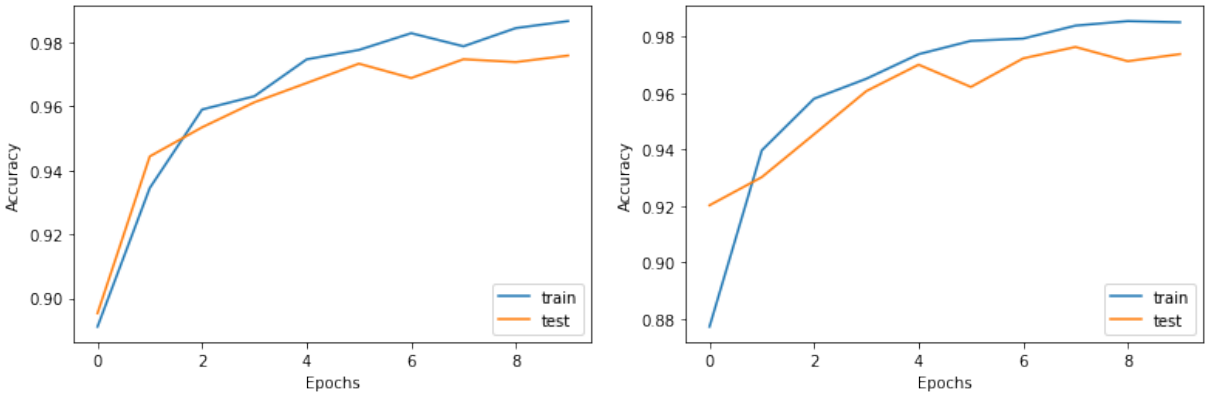

Figure 6.7: Training of the pre-trainied model of Experiment 2 on local data

The above figure shows how the accuracy of the pre-model increases while it is trained with the two personalization approaches. The graph on the left refers to the full training personalization approach, while the graph on the right the partial training. The latter approach, although starts with an higher accuracy, is more prone to over-fitting. Once the re-training of the pre-trained is performed the resulting models have been evaluated.

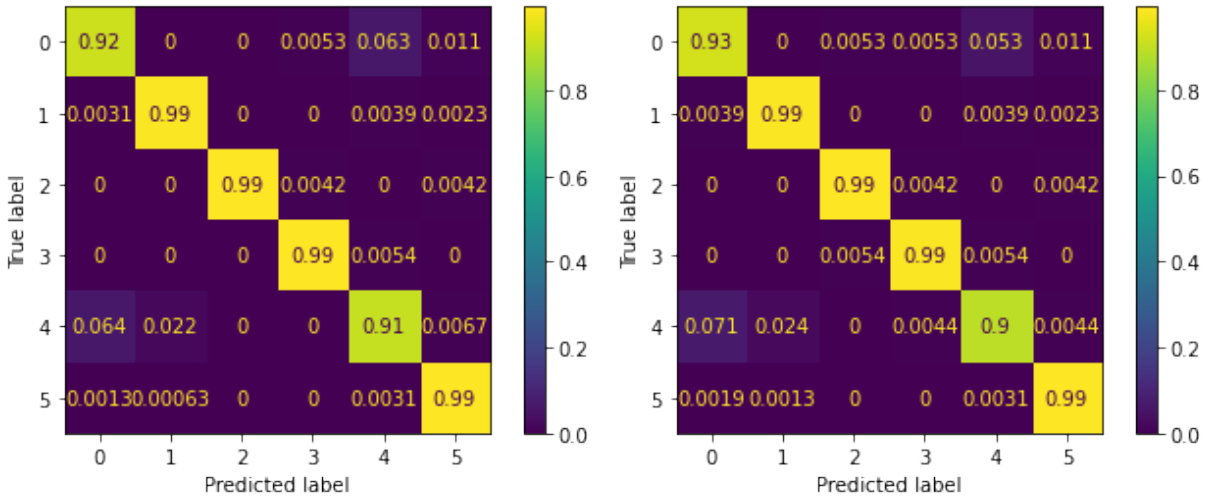

Figure 6.8: Confusion matrix of personalized models of Experiment 2

Metrics	Full	Partial
Precision	0.9770	0.9753
Recall	0.9770	0.9753
F1	0.9770	0.9753

Table 6.4: Evaluation of personalized models of Experiment 2

As we can notice the full re-training of the model reach better results. The re-training of the proposed models with the two personalization approaches requires the following amount of memory:

- Networks parameters: 39 KB
- Training samples in a batch: 29 KB
- Quantities processed by BP Full training: 97 KB
- Quantities processed by BP Partial training: 34 KB

So the first approach requires a total of 165 KB while the second of 102KB.

6.2.3. Experiment 3: window size of 3 seconds

In the Experiment 3 we segment raw data with a window size of 3 seconds, corresponding to an input size for the model of (60,3). The model is preliminary trained and then evaluated on test data reporting the following results:

Metrics	No train
Precision	0.8425
Recall	0.8292
F1	0.8341

Table 6.5: Evaluation of pre-trained model of Experiment 3

As can be seen the pre-trained model has not highly generalization capabilities due to the class imbalances of the dataset. This is confirmed by the confusion matrix of the pre-trained model's predictions on the test data.

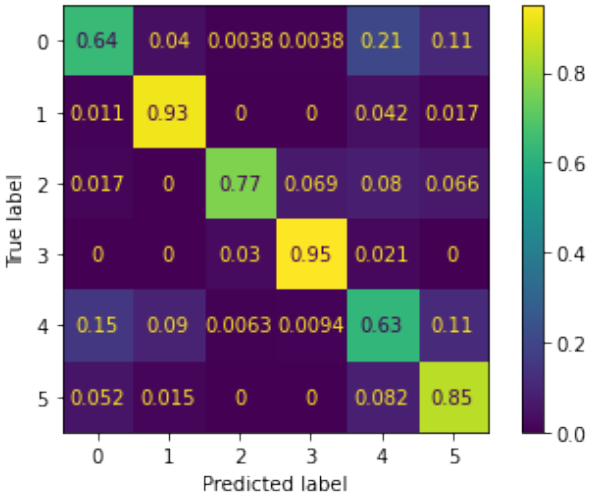

Figure 6.9: Confusion matrix of pre-trained model of Experiment 3

By means of personalization, however, the performances of the pre-trained model are significantly enhanced.

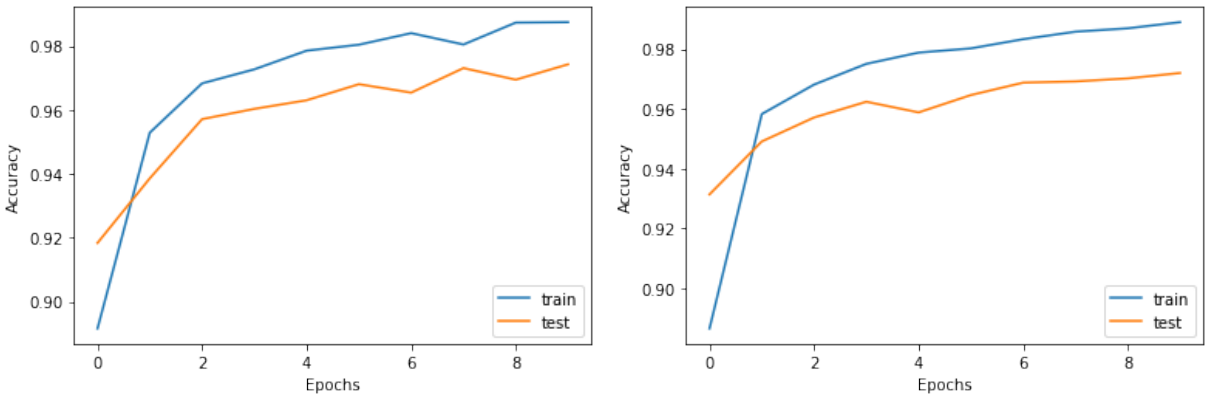

Figure 6.10: Training of the pre-trainied model of Experiment 3 on local data

The above figure shows how the accuracy of the pre-model increases while it is trained with the two personalization approaches. The graph on the left refers to the full training personalization approach, while the graph on the right the partial training. Once the re-training of the pre-trained is performed the resulting models have been evaluated.

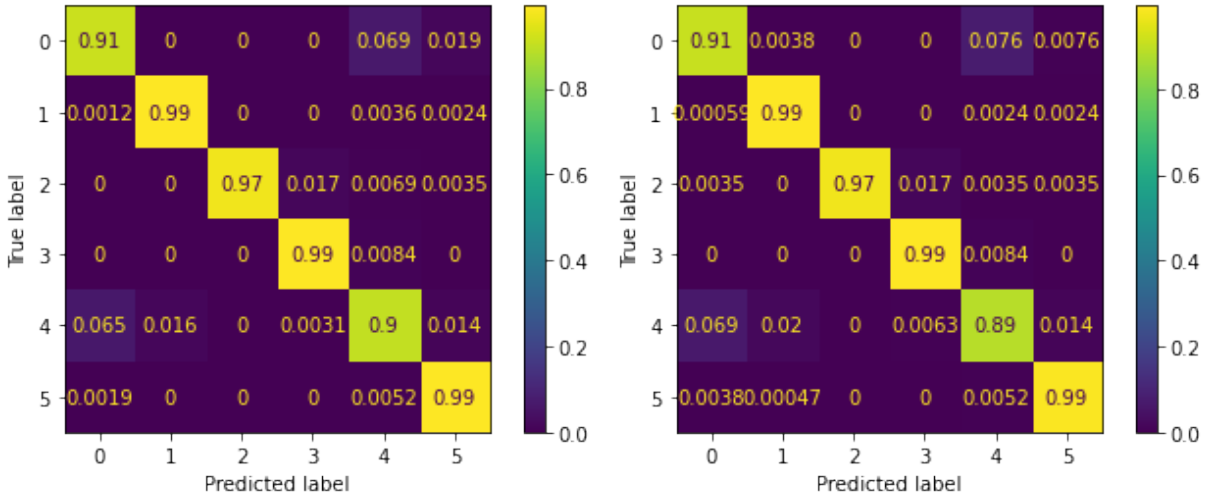

Figure 6.11: Confusion matrix of personalized models of Experiment 3

Metrics	Full	Partial
Precision	0.9718	0.9736
Recall	0.9718	0.9736
F1	0.9718	0.9736

Table 6.6: Evaluation of personalized models of Experiment 3

As we can notice the partial re-training of the model reach better results. The re-training of the proposed models with the two personalization approaches requires the following amount of memory:

- Networks parameters: 29 KB
- Training samples in a batch: 20 KB
- Quantities processed by BP Full training: 82 KB
- Quantities processed by BP Partial training: 14 KB

So the first approach requires a total of 131 KB while the second of 91 KB.

6.2.4. Experiment 4: window size of 2 seconds

In the Experiment 4 we segment raw data with a window size of 2 seconds, corresponding to an input size for the model of (40,3). The model is preliminary trained and then evaluated on test data reporting the following results:

Metrics	No train
Precision	0.8332
Recall	0.7918
F1	0.8056

Table 6.7: Evaluation of pre-trained model of Experiment 4

As can be seen the pre-trained model has not highly generalization capabilities due to the class imbalances of the dataset. This is confirmed by the confusion matrix of the pre-trained model's predictions on the test data.

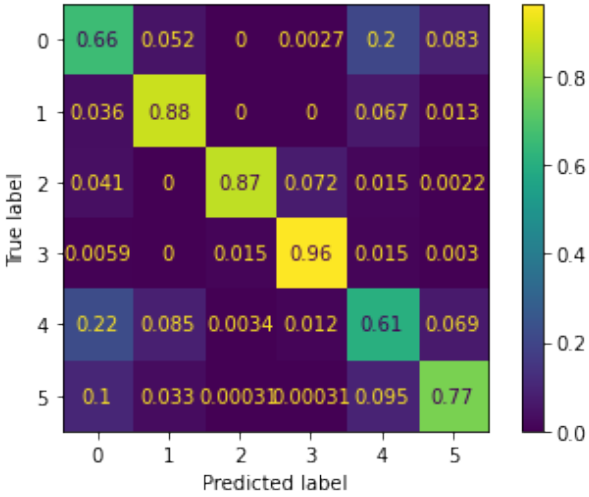

Figure 6.12: Confusion matrix of pre-trained model of Experiment 4

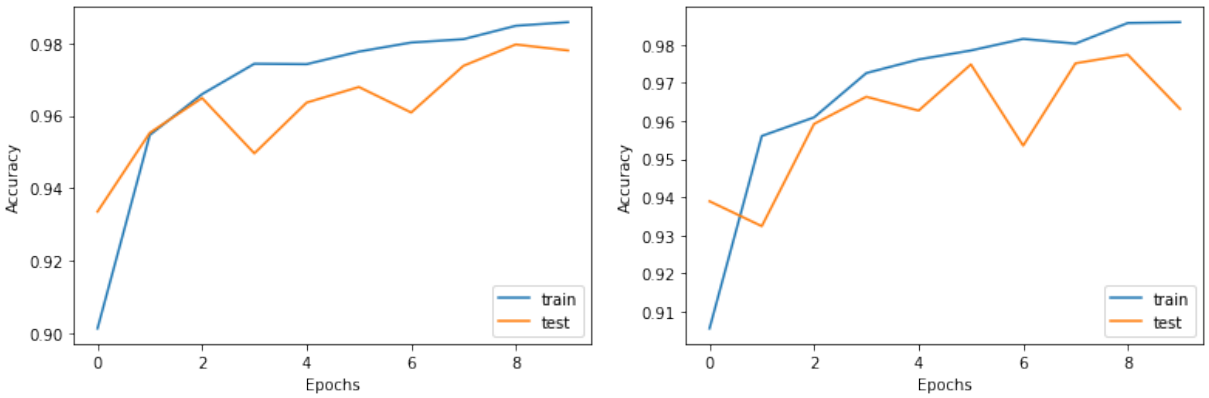

Figure 6.13: Training of the pre-trainied model of Experiment 4 on local data

The above figure shows how the accuracy of the pre-model increases while it is trained

with the two personalization approaches. The graph on the left refers to the full training personalization approach, while the graph on the right the partial training. Once the re-training of the pre-trained is performed the resulting models have been evaluated.

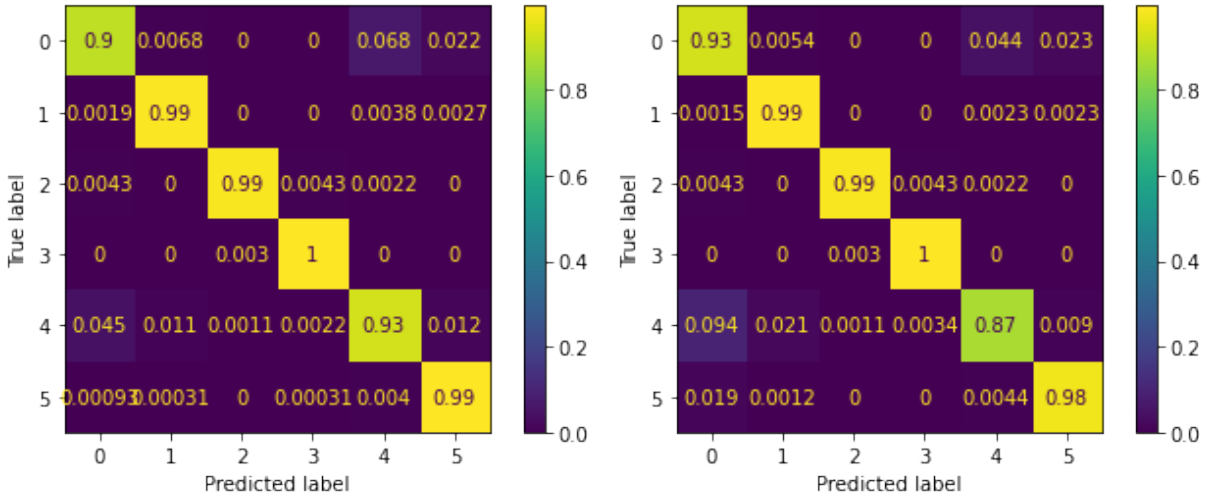

Figure 6.14: Confusion matrix of personalized models of Experiment 4

Metrics	Full	Partial
Precision	0.9780	0.9669
Recall	0.9781	0.9674
F1	0.9780	0.9677

Table 6.8: Evaluation of personalized models of Experiment 4

As we can notice the full re-training of the model reach better results. The re-training of the proposed models with the two personalization approaches requires the following amount of memory:

- Networks parameters: 39 KB
- Training samples in a batch: 16 KB
- Quantities processed by BP Full training: 67 KB
- Quantities processed by BP Partial training: 24 KB

So the first approach requires a total of 122 KB while the second of 79 KB.

6.2.5. Experiment 5: window size of 1 second

In the Experiment 5 we segment raw data with a window size of 1 second, corresponding to an input size for the model of (20,3). The model is preliminary trained and then evaluated on test data reporting the following results:

Metrics	No train
Precision	0.7987
Recall	0.7835
F1	0.7885

Table 6.9: Evaluation of pre-trained model of Experiment 5

As can be seen the pre-trained model has not highly generalization capabilities due to the class imbalances of the dataset. This is confirmed by the confusion matrix of the pre-trained model's predictions on the test data.

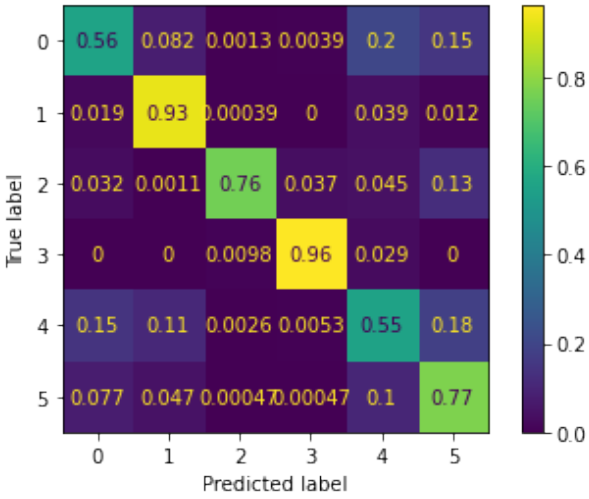

Figure 6.15: Confusion matrix of pre-trained model of Experiment 5

By means of personalization, however, the performances of the pre-trained model are significantly enhanced.

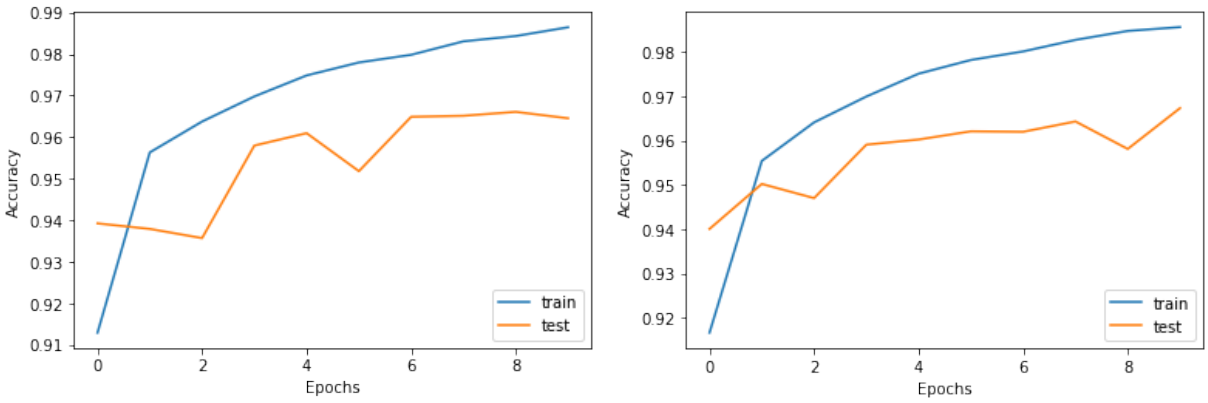

Figure 6.16: Training of the pre-trainied model of Experiment 5 on local data

The above figure shows how the accuracy of the pre-model increases while it is trained with the two personalization approaches. The graph on the left refers to the full training personalization approach, while the graph on the right the partial training. Once the re-training of the pre-trained is performed the resulting models have been evaluated.

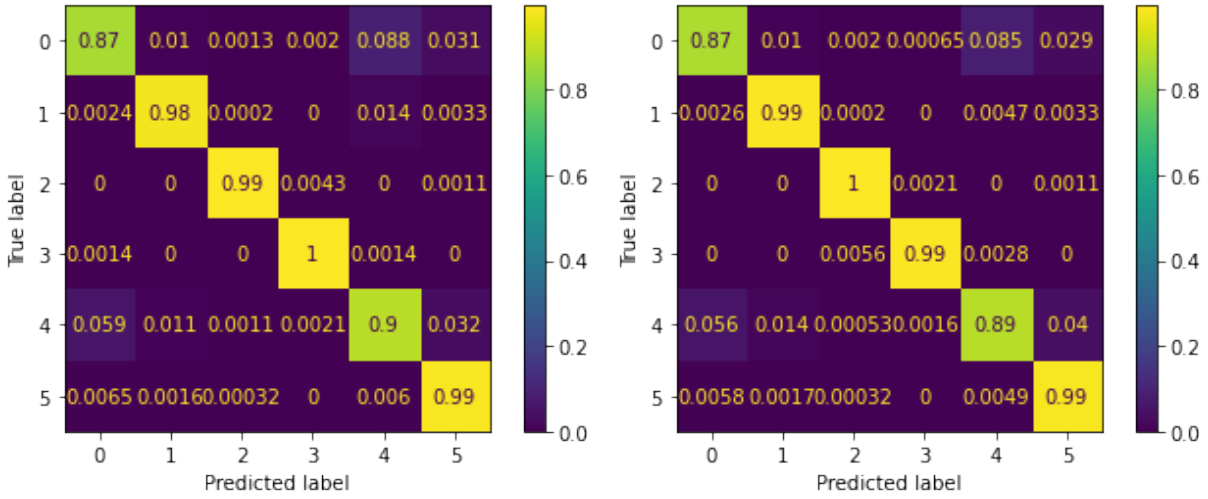

Figure 6.17: Confusion matrix of personalized models of Experiment 5

Metrics	Full	Partial
Precision	0.9637	0.9663
Recall	0.9635	0.9666
F1	0.9636	0.9665

Table 6.10: Evaluation of personalized models of Experiment 5

As we can notice the partial re-training of the model reach better results. The re-training of the proposed models with the two personalization approaches requires the following amount of memory:

- Networks parameters: 39 KB
- Training samples in a batch: 6 KB
- Quantities processed by BP Full training: 52 KB
- Quantities processed by BP Partial training: 18 KB

So the first approach requires a total of 97 KB while the second of 63 KB.

6.3. Summary of results

In the following tables are summarized the memory footprint of each considered models and the accuracy of the models evaluated on test data before and after the personalization.

Input shape	Full	Partial
(100,3)	189KB	115KB
(80,3)	165KB	102KB
(60,3)	131KB	91KB
(40,3)	122KB	79KB
(20,3)	97KB	63KB

Table 6.11: BP Memory Footprint

As showed in Table 6.11 the input size has a great impact on the memory footprint. The train of only the two last layer of the network reduce significantly the memory required such that the footprint of partial train of the model with the bigger input shape is comparable with the footprint of the full train of the model with the smaller input shape.

Input shape	No train	Full	Partial
(100,3)	0.8245	0.9715	0.9634
(80,3)	0.8453	0.9770	0.9754
(60,3)	0.8292	0.9718	0.9736
(40,3)	0.7918	0.9780	0.9673
(20,3)	0.7835	0.9635	0.9666

Table 6.12: F1 of the proposed model

The results in Table 6.12, which reports the F-beta scores of the models, indicate that the personalization of the pre-trained model on local data significantly improves it's performances. Even if the pre-trained model does not have an high generalization capability by re-training it on device the model achieves very good results.

7 | Conclusions and Future Developments

The work done in the thesis show that is possible to empower embedded systems with training on device functionalities. The developed framework is featured with the flexibility to perform the training on the device of arbitrary neural networks, provided that the computational resources of the empowered device is sufficient to run the training process. In this regards a tool for predicting the required computational resources for the training on device has been developed. Different experiment on personalization of a pre-trained model on HAR use case show the enhancement which can be obtained thanks to training on device. Indeed the results of experiments show that the accuracy of the model increases and the pre-trained model adapts to new data and improves its performance over time.

There are tree main future directions in which this work can go.
The first is related to an extension of the functionalities provided by the framework itself, such as the support of more types of layers and the possibility to use other optimizers.
The second future direction is related to the application of optimization and compression techniques in order to reduce the amount of memory required to run training on-device.
The last one is the FL which provides a means to train models in a distributed scenario without sharing the local training data.
Moreover from a STMicroelectronics perspective a future work will be the integration of training on device functionalities in the STM32Cube software ecosystem.

Bibliography

[1] Stm32l4 - arm cortex-m4 ultra-low-power mcus. URL https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html.

[2] Integrated development environment for stm32 products. URL https://www.st.com/en/development-tools/stm32cubeide.html.

[3] X-cube-ai - ai expansion pack for stm32cubemx. URL https://www.st.com/en/embedded-software/x-cube-ai.html.

[4] H. Cai, C. Gan, L. Zhu, and S. Han. Tiny transfer learning: Towards memory-efficient on-device learning. *CoRR*, abs/2007.11622, 2020. URL https://arxiv.org/abs/2007.11622.

[5] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems. *CoRR*, abs/2010.08678, 2020. URL https://arxiv.org/abs/2010.08678.

[6] S. Disabato and M. Roveri. Incremental on-device tiny machine learning. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, AIChallengeIoT '20, page 7–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381345. doi: 10.1145/3417313.3429378. URL https://doi.org/10.1145/3417313.3429378.

[7] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. A. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and

S. Zhao. Advances and open problems in federated learning. *CoRR*, abs/1912.04977, 2019. URL http://arxiv.org/abs/1912.04977.

[8] K. Kopparapu and E. Lin. Tinyfedtl: Federated transfer learning on tiny devices. *CoRR*, abs/2110.01107, 2021. URL https://arxiv.org/abs/2110.01107.

[9] L. Lai, N. Suda, and V. Chandra. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR*, abs/1801.06601, 2018. URL http://arxiv.org/abs/1801.06601.

[10] H. Matsutani, M. Tsukada, and M. Kondo. On-device learning: A neural network based field-trainable edge ai, 2022. URL https://arxiv.org/abs/2203.01077.

[11] V. Rajapakse, I. Karunanayake, and N. Ahmed. Intelligence at the extreme edge: A survey on reformable tinyml, 2022. URL https://arxiv.org/abs/2204.00827.

[12] H. Ren, D. Anicic, and T. A. Runkler. Tinyol: Tinyml with online-learning on microcontrollers. *CoRR*, abs/2103.08295, 2021. URL https://arxiv.org/abs/2103.08295.

[13] M. Tsukada, M. Kondo, and H. Matsutani. A neural network based on-device learning anomaly detector for edge devices. *CoRR*, abs/1907.10147, 2019. URL http://arxiv.org/abs/1907.10147.

[14] G. M. Weiss, J. W. Lockhart, T. T. Pulickal, P. T. McHugh, I. H. Ronan, and J. L. Timko. Actitracker: A smartphone-based activity recognition system for improving health and well-being. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 682–688, 2016. doi: 10.1109/DSAA.2016.89.

List of Figures

List of Tables

List of Symbols

Variable	Description
$\mathcal{N}$	neural network
$\boldsymbol{\theta}$	neural network parameters
$\mathbf{w}$	weights
$\mathbf{b}$	biases
$\mathbf{x}$	input
$\mathbf{a}$	activations
$\mathcal{L}$	loss function
$\overline{C}(\boldsymbol{\theta})$	risk
$C(\boldsymbol{\theta})$	empirical risk
$\frac{\partial f}{\partial \mathbf{w}}$	local gradient of weights
$\frac{\partial f}{\partial \mathbf{b}}$	local gradient of biases
$\frac{\partial f}{\partial \mathbf{x}}$	local gradient of input
$\frac{\partial C}{\partial \mathbf{w}}$	gradient of weights
$\frac{\partial C}{\partial \mathbf{b}}$	local gradient of biases
$\frac{\partial C}{\partial \mathbf{a}}$	downstream error
$\frac{\partial C}{\partial \mathbf{a}}$	upstream error

Acknowledgements

I would first like to thank my thesis advisor Giacomo Boracchi for offering me the opportunity to get in touch with STMicroelectornics and for providing support and guidance throughout this project.
I would like to express my deepest gratitude to the members of the technical staff of the company: Beatrice, Diego and Pasqualina and for my internships' colleague Elisabetta. Their passionate participation in every step of the project and their continuous feedback have been fundamentals for the work.
I'm extremely grateful to my parents Marialusia and Vincenzo, my sister Elisa and my grandmother who have supported me and offered deep insight into the study.
I am gratefully indebted to Carla, she provides me with unconditional support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.
I would like to extend my sincere thanks to my classmates Amedeo and Francesco with whom I shared my enthusiasm and passions during the time spent at the University. Lastly, I'd like to mention my best friends Cristian, Davide, Erica, Laura, Mattia, Simone and Stefano who have always been present during the course of my studies.