On-Device Personalization for Human Activity Recognition on STM32

Michele Craighero, Davide Quarantiello, Beatrice Rossi, Diego Carrera, Pasqualina Fragneto, Giacomo Boracchi

***Abstract*—Human Activity Recognition (HAR) is one of the most interesting application for machine learning models running on low cost and low-power devices, such as microcontrollers (MCUs). As a matter of fact, MCUs are often dedicated to performing inference on their own acquired data, and any form of model training and update is delegated to external resources. We consider this mainstream paradigm a severe limitation, especially when privacy concerns prevent data sharing, thus model personalization, which is universally recognized as beneficial in HAR. In this work, we present our HAR solution where MCUs can directly fine-tune a deep learning model using locally acquired data. In particular, we enable training functionalities for 1-D Convolutional Neural Networks (CNN) on STM32 microcontrollers and provide a software tool to estimate the memory and computational resources required to accomplish model personalization.**

***Index Terms*—Human Activity Recognition, On-Device Learning, Model Personalization, Microcontrollers, STM32.**

I. Introduction

Recently, we have witnessed a broad diffusion of Internet of Things (IoT) devices equipped with tiny *microcontroller units* (MCUs) and an increasing interest in *Tiny Machine Learning* (TinyML [1]) research to leverage machine learning models on low-power devices. Human Activity Recognition (HAR) is among the most frequently addressed problems in the TinyML domain [2]. The mainstream paradigm in HAR consists in classifying (e.g., by a 1D-CNN) segments of *Inertial Measurement Units* (IMUs) recordings, which are easy to gather in wearable devices mounting accelerometers and gyroscopes. Model training is exclusively performed on a server using many annotated data and large computational resources. Once trained, the model is possibly optimized e.g., by distillation, quantization, or pruning [2], and then deployed to the MCU, which is only in charge of inference. Not surprisingly, the vast majority of TinyML solutions support only *inference* on MCUs. Examples include Tensorflow Lite Micro [3] from Google and X-CUBE AI [4] from STMicroelectronics.

On-device learning (ODL) solutions for MCUs are scarce in the literature, and a few frameworks expose only training functionalities for *dense* layers [5]. This is somehow in contrast with modern CNNs that favor convolutional ones. Moreover, TinyTL [6] and Train++ [7], are purely algorithmic studies and do not provide any implementation to be used in HAR. The only framework that specifically addresses CNNs training on MCUs is [8], which introduces an algorithm-system co-design that combines quantizations and sparse updating techniques to enable the training of an image classifier on STM32 MCUs with minimal memory requirements (256KB). However, none of these frameworks, including [8], were used to investigate

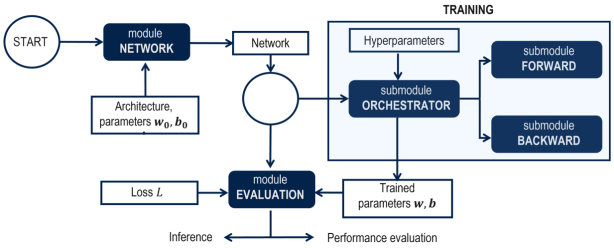

Fig. 1: Proposed framework for HAR personalization.

model *personalization* in HAR by fine-tuning all the layers of a CNN directly on the MCU. We believe this is a relevant problem for HAR, where model personalization is often key to compensate for subjects' heterogeneity resulting in very different signals for the same activity. Moreover, performing model personalization directly on the device is key to prevent sharing of confidential data.

In this paper we perform for the first time model personalization for HAR directly on a STM32 microcontroller. To this purpose, we implemented both a software framework that enables to fine-tune on the MCU all the layers of a 1-D CNN, and also a tool to estimate the memory footprint and the computational resources required for the personalization. Our framework can also be used for addressing other problems in HAR, namely: *i*) enabling continuous learning to counteract *concept drift*, and *ii*) enabling *Federated Learning* (FL) mechanisms [9] in a fully distributed manner.

Our experiments, performed on real-world HAR datasets, confirm that model personalization in HAR is very beneficial and that it is possible to retrain 1-D CNNs satisfying the strict computational and memory constraints of the STM32L496ZG MCU. In addition, we demonstrate a trade-off between model accuracy and computational requirements, since performing a full retraining of the model is beneficial in terms of F1-score, but at the same time it has a greater impact on the memory and the energy consumption. This analysis provides insights on when to schedule model personalization on the device.

II. Problem Definition

We address HAR as a multi-class classification problem, where the input $s \in \mathbb{R}^{n \times z}$ is a segment of z samples from n time series acquired by inertial sensors, and the output y is a label corresponding to a human activity. We assume a *general* training set TR of data from different users is provided to train a classifier $\mathcal{C}$, which associates each s to a label $\hat{y} = \mathcal{C}(s)$. State-of-the-art classifiers for HAR [2] are 1D-CNNs.

The general classifier $\mathcal{C}$ can poorly recognize activities from users not present in TR. Therefore, model personalization for

a user i consists in fine-tuning the parameters θ_0 of the general classifier $\mathcal{C}$ using a *local* training set TR_i containing data from user i. In particular, our goal is to train a *local* classifier $\mathcal{C}_i$ with parameters θ_i directly on a MCU, using θ_0 as initialization and TR_i as training set.

III. A Framework for HAR Personalization

Figure 1 illustrates our framework to personalize HAR models on STM32 MCUs. The framework is developed in C programming language and composed of 3 modules:

a) Network: instantiates the local classifier $\mathcal{C}_i$ from the architecture specifications and the initial parameters θ_0, which can be imported from a pretrained HAR classifier $\mathcal{C}$, or randomly set. The output of this module is the classifier itself, which can be fed into the Evaluation or Training modules.

b) Training: HAR personalization is performed by a few iterations of *backpropagation*. The goal of the backpropagation algorithm is to find parameters θ^* that minimize a loss function L on the local training set TR_i (more details on that will be given in Section III-A). This module implements backpropagation via gradient descent through 3 submodules: the *Orchestrator* governing the iterative training procedure by invoking alternatively the *Forward* and the *Backward* submodules. The *Orchestrator* allows specifying training hyperparameters, such as the number of epochs, the learning rate, and the batch size. The *Forward* module performs the forward pass of the backpropagation by implementing the forward expressions reported in Table I for the most popular layers of 1D-CNN. During the forward pass, the values of the activations in the neurons of C_i need to be stored. The *Backward* module performs the backward pass of the backpropagation, implementing the backward expressions reported in Table I.

c) Evaluation: this module performs inference and during training computes $L(\theta)$, namely the value of the loss function corresponding to parameters θ. We also use this module to assess the effectiveness of personalization, by comparing the accuracy of $\mathcal{C}$ and $\mathcal{C}_i$.

A. Gradients Computation

We illustrate the implementation of backpropagation in the Training module. As in Tensorflow API, we decompose the backpropagation step (a *forward pass* followed by a *backward pass*) into a sequence of operations performed layer by layer and combined using the chain rule of derivatives. Each layer is associated with a layer function f and parameters $\theta = [\mathbf{w}, \mathbf{b}]$. Starting from the layer's input $\mathbf{x}$ and the parameters θ, the function f computes the value of the output activation $\mathbf{a}$, which will become the input $\mathbf{x}$ of the subsequent layer.

During the backward pass, starting from the last layer, we compute the *gradient* of the loss L with respect to the network parameters $\mathbf{w}, \mathbf{b}$, and the so-called downstream error $\frac{\partial L}{\partial \mathbf{x}}$ as:

$$\frac{\partial L}{\partial \mathbf{w}} = \sum_{i=1}^{M} \frac{\partial L}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{w}}, \quad \frac{\partial L}{\partial \mathbf{b}} = \sum_{i=1}^{M} \frac{\partial L}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{b}}, \quad \frac{\partial L}{\partial \mathbf{x}} = \sum_{i=1}^{M} \frac{\partial L}{\partial a_i} \frac{\partial f_i}{\partial \mathbf{x}}, \quad (1)$$

where f_i and a_i are respectively the i^{th} component of the layer function f and of the unrolled $\mathbf{a}$ tensor, and M is the number of units of the layer. The downstream error is then passed back to the previous layer as $\frac{\partial L}{\partial \mathbf{a}}$. Table I reports the explicit expressions for the forward and backward pass of most popular layers of a CNN, namely *Conv1D, Dense, AvgPool1D, GlobalAvgPool1D, and Flatten*. By combining those expressions, it is possible to derive the overall forward pass and to compute the *loss* L and its derivative.

As an example we illustrate the computation of a *Conv1D* layer with *C* channels, *N* input units, *F* filters, and kernel size *K*. The (j, m)-th element of the output $\mathbf{a}$ is defined as:

$$a_{j,m} = \sum_{c=1}^{C} \sum_{k=1}^{K} x_{c,m+k-1} w_{j,c,k} + b_j, \quad (2)$$

where $j \in \{1, \ldots, F\}$, $m \in \{1, \ldots, M\}$ and $M = N - K + 1$. Note that in (2) we treat *convolution* as *cross-correlation* as in Tensorflow, a customary practice when filters are being learned. According to the chain rule, we have:

$$\frac{\partial L}{\partial \mathbf{x}} = \sum_{j=1}^{F} \sum_{m=1}^{M} \frac{\partial L}{\partial a_{jm}} \frac{\partial f_{jm}}{\partial \mathbf{x}}. \quad (3)$$

In particular, since $\frac{\partial L}{\partial \mathbf{a}}$ is passed from the next layer during backpropagation, we just have to compute the local gradients of the layer function with respect to the input, namely:

$$\frac{\partial L}{\partial x_{in}} = \sum_{j=1}^{F} \sum_{k=1}^{K} \frac{\partial L}{\partial a_{j,n-k+1}} w_{jik}. \quad (4)$$

In (4), the activation index $n-k+1$ ranges from $2-K$ to N, for a total of $N+K-1$ terms for each channel j, whereas $\frac{\partial L}{\partial a}$ has size $F \times (N - K + 1)$. If $K > 1$, we apply a 0 padding to $\frac{\partial L}{\partial \mathbf{a}}$ by adding $F \cdot (K-1)$ zeros to both sides of $\frac{\partial L}{\partial \mathbf{a}}$ along its second dimension. The index k in (4) has opposite signs in the two terms of the convolution ($-k$ in $\frac{\partial L}{\partial a}$ and $+k$ in w), thus we obtain a flipped kernel. The final result is expressed as:

$$\frac{\partial L}{\partial \mathbf{x}} = \text{conv}\left(\text{pad}\left(\frac{\partial L}{\partial \mathbf{a}}\right), \text{flip}(\mathbf{w})\right), \quad (5)$$

where *conv* still represents *cross-correlation*.

Finally, to compute the gradient of a batch of input segments we resort *gradient accumulation*. This maintains fixed the memory footprint during the training regardless the batch size.

B. Estimating Resources

Our framework is equipped with a tool that estimates *i*) the *memory footprint* and *ii*) the *CPU load* to personalize a HAR classifier on an STM32 MCU. This tool is very valuable during prototyping to properly size embedded ML applications to the MCU capabilities.

Our tool computes the memory footprint as the amount of memory required by model personalization. Model training requires storing training samples, network parameters, activations, gradients, and errors computed at each layer during backpropagation. In particular, all the activations and the downstream errors $\frac{\partial L}{\partial \mathbf{a}}$ of the layers that we want to train must be stored in memory to be used during the backward pass to compute gradients. Our tool can estimate a-priori the memory usage of training by multiplying the total amount of saved variables by their bit precision. The tool also estimates the CPU load by counting the total number of operations

Layer	Forward pass	Backward pass			Parameters	
		Input	Weights	Bias	Weights	Bias
Dense	$\mathbf{a} = \mathbf{w}^T \cdot \mathbf{x} + \mathbf{b}$	$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{w} \cdot \frac{\partial L}{\partial \mathbf{a}}$	$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{x} \cdot \left(\frac{\partial L}{\partial \mathbf{a}}\right)^T$	$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{a}}$	$M \cdot N$	N
Conv1D	$\mathbf{a} = \text{conv}(\mathbf{x}, \mathbf{w}) + \mathbf{b}$	$\frac{\partial L}{\partial \mathbf{x}} = \text{conv}\left(\frac{\partial L}{\partial \mathbf{a}}, \text{flip}(\mathbf{w}), \text{full}\right)$	$\frac{\partial L}{\partial \mathbf{w}} = \text{conv}\left(\frac{\partial L}{\partial \mathbf{a}}, \mathbf{x}\right)$	$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{a}}$	$F \cdot C \cdot K$	F
Activation	$a_i = \frac{e^{x_i}}{\sum_{i=1}^{N} e^{x_i}}$	$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{a} - \mathbf{t}$	-	-	-	-
AvgPool1D	$a_{jm} = \frac{1}{p} \sum_{i=0}^{p} x_{ji}$	$\frac{\partial L}{\partial x_{in}} = \frac{1}{p} \frac{\partial L}{\partial a_{ij}}$	-	-	-	-
GlobalAvgPool1D	$a_j = \frac{1}{N} \sum_{i=0}^{N} x_{ji}$	$\frac{\partial L}{\partial x_{ij}} = \frac{1}{N} \frac{\partial L}{\partial a_{in}}$	-	-	-	-
Flatten	$a = \text{vec}(x)$	$\frac{\partial L}{\partial \mathbf{x}} = \text{reshape}(\frac{\partial L}{\partial \mathbf{a}})$	-	-	-	-

TABLE I: Forward and Backward expressions for layers implemented in the Training module. N and M are the width of the previous and the current layer respectively, F is the number of filters of the current layer, K is the kernel size, C is the number of input channels, and p is the stride of the AvgPool1D layer. The backward passes of weights and biases are computed only for *Dense* and *Conv1D* since the other considered layers do not train these parameters.

performed by the backpropagation algorithm in the forward and in backward passes, as well as in the parameters' update phase. The number and the type of operations performed depend on the type of layers and are derived from an analysis of the expressions in Table I.

IV. Experiments and Results

Our experiments are meant to assess the benefits of model personalization in HAR. In particular, in the considered settings we have that: i) training on user-specific data improves the accuracy of a pretrained model and ii) personalization of a pretrained model outperforms a classifier trained only on the target user. Finally, we show that enabling the full retraining of the classifier on MCU is beneficial with respect to transfer learning in terms of accuracy, although it requires more computational resources.

Our target MCU is the STM32L496ZG, an ultra-low-power MCU produced by STMicroelectronics with an ARM Cortex-M4 core and 320 KB of sRAM. We flash the firmware using the development board NUCLEO-L496ZG that is equipped with such MCU.

A. Datasets and Architecture

The general dataset TR we consider is the *Wireless Data Mining* (WISDM) dataset [10], which is used to pretrain a general classifier $\mathcal{C}$. The local training set TR_i are from *ST dataset*, which is collected in STM facilities using a SensorTile.box [11]. Their characteristics are:

- WISDM dataset: 36 users, 6 activities (*walking, jogging, ascending stairs, descending stairs, sitting*, and *standing*), sampling frequency 20 Hz;
- ST dataset: 3 users, 3 activities (*walking, ascending stairs* and *descending stairs*), sampling frequency 27 Hz.

We notice that the activities in the ST dataset are a subset of those in WISDM. Both datasets have been collected using tri-axial accelerometers, but with different sensors, thus the ST dataset has been resampled to $20Hz$. We underline that, after the resampling, the two datasets have approximately the same number of samples for each user (around 30k). The classifier takes as input from 1 to 5 seconds of recording, which correspond to the following input sizes: $(20, 3), (40, 3), (60, 3), (80, 3), (100, 3)$, where 3 is the number of axis of the accelerometers.

We adopt a 1D-CNN made of 4 blocks: 1) Conv1D with $F = 32$ filters and kernel size $K = 3$ with ReLU, AvgPool1D, 2) Conv1D of $F = 64$ filters and kernel size $K = 3$ with ReLU, AvgPool1D, GlobalAvgPool1D, 3) Dense of $M = 50$ units with ReLU and 4) Dense of $M = 6$ units with Softmax. The total number of parameters θ to train is around 10000. We select the SGD optimizer with a learning rate of 0.01 and a batch size of 32. Facing a multi-class classification task, we adopt the *Categorical Cross-Entropy* as loss function.

B. Model Personalization

At first, we show that in the considered settings, model personalization improves the performance of a pretrained model. The first experiment is entirely conducted on the WISDM dataset by a Leave-One-Subject-Out (LOSO) approach. For each user $i = 1, \ldots, 36$ we define a training (TR_i) and test (TS_i) sets and pretrain a general classifier $\mathcal{C}$ from the other 35 users. We personalize each local classifier $\mathcal{C}_i$ by retraining all the layers (*Full* personalization) of $\mathcal{C}$ using TR_i. As a comparison, we consider the *Transfer Learning (TL)* which can be pursued by standard TinyML frameworks [5] and that retrains only the last 2 dense layers of $\mathcal{C}$. For each user i we assess $\mathcal{C}_i$ on TS_i, and we show in Table II the F1-score averaged over all the users. We also consider *No Pers.* as the performance of the classifier $\mathcal{C}$. Both personalization approaches improve the performance of $\mathcal{C}$, and *Full* personalization always reaches the highest F1-score, for all the input sizes. This confirms that enabling the retraining of all the network layers is highly beneficial in the HAR scenario.

We also assess the benefits of model personalization over each user of the ST dataset for both *TL* and *Full* personalization. As a customary procedure for improving convergence in fine tuning, we first retrain for 2 epochs the last dense layer freezing all the others. We then retrain all the network's layers *(Full)* or training the last 2 dense layers only *(TL)*. Moreover, we train user specific classifiers from data of each user (denoted as *No Pretrain*) starting from a random initialization. Table II reports the F1-scores averaged on the 3 users of the ST dataset. We note that the while *TL* achieves lower or comparable performance w.r.t. classifier *No Pretrain*, *Full* personalization always achieves the highest F1-score, independently from the chosen input size. This confirms that enabling the retraining of all the network's layers is highly

Input size	WISDM Dataset			ST Dataset		
	No Pers.	TL	Full	No Pretrain.	TL	Full
(100,3)	0.813	0.955	**0.969**	0.936	0.938	**0.960**
(80,3)	0.815	0.963	**0.973**	0.944	0.939	**0.962**
(60,3)	0.819	0.962	**0.978**	0.938	0.931	**0.966**
(40,3)	0.808	0.958	**0.974**	0.951	0.929	**0.965**
(20,3)	0.804	0.948	**0.967**	0.945	0.911	**0.959**

TABLE II: F1 score on WISDM and ST datasets.

Input size	Memory Footprint		Time per batch		Power per batch	
	TL	Full	TL	Full	TL	Full
(100,3)	115KB	189 KB	14.25 s	48.10 s	2.67 mW	2.68 mW
(80,3)	102KB	165 KB	11.27 s	38.05 s	2.68 mW	2.69 mW
(60,3)	91KB	131 KB	8.29 s	28.00 s	2.66 mW	2.68 mW
(40,3)	79KB	122 KB	5.31 s	17.95 s	2.64 mW	2.67 mW
(20,3)	63KB	98 KB	2.33 s	7.90 s	2.65 mW	2.65 mW

TABLE III: Computational resources required by Full and TL personalization for different input sizes.

beneficial even when personalization is performed on data from a different dataset, which is common in HAR scenarios.

C. Computational Resources' Evaluation

Here we assess the computational resources required to perform model personalization on the MCU. At first, we use our tool to estimate the memory footprint of both *TL* and *Full* personalization. As reported in Table III, the input size has a great impact on the memory footprint: the smallest input size $(20, 3)$ requires about half of the memory with respect to the largest size $(100, 3)$. However, all the tested cases are within the memory limitations of our selected device, since they use less than 320 kB.

Table III reports the time required to process a batch of 32 segments for each input size for both Full and TL. Also in this case we observe that increasing the input size results in larger execution time. In particular, $(100, 3)$ requires more than 5 times the time required by input size of $(20, 3)$. However, as shown in Table II, an input size of $(20, 3)$ is enough for reaching a very high accuracy.

Finally, we use the X-NUCLEO-LPM01A power shield to measure the average power required to process a batch of 32 samples. The power shield is attached to the development board to measure the current absorbed during the execution of the training. Since the voltage provided is equal to 3.3 V, we can easily derive the power consumed. We note that the power is the same for both *TL* and *FUll* personalization procedures for any input size. However, the energy (power $\times$ time) required to process a single batch is higher for the *Full* personalization, since it scales linearly with the time.

This analysis is very valuable as it gives an estimate of the computational resources required during the training, allowing to schedule model personalization depending on the power availability. For example, *Full* personalization could be performed only when the battery device is recharging, while *TL* could be run when the device relies on its own battery. Finally, our framework can be used to adjust the input size and select the number of training epochs for the chosen MCU, to reach the best trade-off between the accuracy of the model and the usage of resources.

V. Conclusions

We present a HAR solution to fine-tune and personalize a deep learning model directly on a STM32 MCU using locally acquired data. In particular, we develop a framework to retrain 1-D CNNs satisfying the strict computational and memory constraints of the STM32L496ZG MCU. Our experiments shows that the Full personalization of the CNN achieves a better accuracy than Transfer Learning, which is what existing frameworks allow, although it requires more energy.

Future work concerns extending our framework to support more layers and different optimization strategies. We will also adapt the framework to be used on MicroProcessors like those of the MP1 series from STMicroelectronics, which can be equipped with a small GPU.

Acknowledgments

We would like to thank Filippo Augusti and Marco Longoni for their invaluable support in the set up of the X-NUCLEO-LPM01A for the power consumption measurements.

References

[1] Tinyml. [Online]. Available: https://www.tinyml.org

[2] M.-K. Yi and S. O. Hwang, "Smartphone based human activity recognition using 1d lightweight convolutional neural network," in *International Conference on Electronics, Information, and Communication*, 2022.

[3] 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*, 2020.

[4] STMicroelectronics. Artificial intelligence (ai) software expansion for stm32cube. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

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

[6] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 11 285–11 297.

[7] B. Sudharsan, P. Yadav, J. G. Breslin, and M. I. Ali, "Train++: An incremental ml model training algorithm to create self-learning iot devices," in *Proceedings of the 18th IEEE International Conference on UIC*, 2021.

[8] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," *arXiv:2206.15472*, 2022.

[9] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 2017, pp. 1273–1282.

[10] J. R. Kwapisz *et al.*, "Activity recognition using cell phone accelerometers," in *Proceedings of the Fourth International Workshop on Knowledge Discovery from Sensor Data*, 2010, pp. 10–18.

[11] Human activity recognition using cnn in keras for sensortile. [Online]. Available: https://github.com/ausilianapoli/HAR-CNN-Keras-STM32/blob/master/Dataset.csv