Memory-Efficient CMSIS-NN with Replacement Strategy

Fouad Sakr
Department of Electrical, Electronic and Telecommunication Engineering and Naval Architecture - University of Genoa
School of Electronic Engineering and Computer Science - Queen Mary University of London
Genoa, Italy
f.sakr@qmul..ac.uk

Francesco Bellotti
Department of Electrical, Electronic and Telecommunication Engineering and Naval Architecture - University of Genoa
Genoa, Italy
francesco.bellotti@unige.it

Riccardo Berta
Department of Electrical, Electronic and Telecommunication Engineering and Naval Architecture - University of Genoa
Genoa, Italy
riccardo.berta@unige.it

Alessandro De Gloria
Department of Electrical, Electronic and Telecommunication Engineering and Naval Architecture - University of Genoa
Genoa, Italy
alessandro.degloria@unige.it

Joseph Doyle
School of Electronic Engineering and Computer Science - Queen Mary University of London
London, United Kingdom
j.doyle@qmul.ac.uk

2021 8th International Conference on Future Internet of Things and Cloud (FiCloud) | 978-1-6654-2574-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FICLOUD49777.2021.00050

***Abstract*— Microcontroller Units (MCUs) are widely used for industrial field applications, and are now ever more being used also for machine learning on the edge, because of their reliability, low cost, and energy efficiency. Due to the MCU resource limitations, the deployed ML models need to be optimized particularly in terms of memory footprint. In this paper, we propose an in-place computation strategy to reduce memory requirements of neural network inference. The strategy exploits the MCU single-core architecture, with sequential execution. Experimental analysis using the CMSIS-NN library on the CIFAE-10 dataset shows that the proposed optimization method can reduce the memory required by a NN model by more than 9%, without impacting the execution performance nor accuracy. The amount of reduction further increases with deeper network architectures.**

Keywords*— *Microcontroller Units, deep learning, edge computing, sequential execution, CMSIS-NN, memory reduction.

I. Introduction

Convolutional Neural Network (CNN) are well established machine learning (ML) models, particularly used for image recognition and classification. Recent advances in edge computing have made it possible to ever more move the inference task from the cloud to embedded devices on the edge. Edge computing offers advantages in terms latency, bandwidth, energy, privacy [1]. However, edge devices have limited computational power and on-chip memory. Several solutions have been devised in order to overcome the memory size limitation. For example, model compression techniques such as parameter pruning and quantization [2], binarization [3], low-rank factorization [4], and knowledge distillation [5] are widely used to reduce the model size. Further alternatives involve changing the execution order of the network's operations by requiring the inference software to follow a specific order [6].

A common class of edge devices is represented by microcontroller units (MCUs) [7], that are widely available, cheap, and energy efficient [6]. Unlike multicore CPUs and GPUs, that can perform multiple computations simultaneously [8], MCUs typically have single-core processors and sequential execution. Such an execution mode requires less memory than parallel execution because only one block of the network can be executed per operation. In addition, operations in a CNN convolution layer are local, as they only depend on the input of that layer. Once the input of the layer has been processed the memory can be replaced. Thus, the memory is reusable among layers.

In this work, we propose a memory replacement strategy that reuses the memory assigned to convolutional layers in order to reduce the overall memory usage of the CNN. The proposed strategy relies on the use of a single input/output buffer which is shared among all the layers in the inference phase of a CNN, exploiting the sequential execution paradigm of single-core processors.

The remainder of this article is organized as follows. Sections II and III present the related work and the background, respectively. Section IV illustrates the proposed methodology, while section V shows the experimental analysis of a case study. Section VI provides the conclusions on the work.

II. Related work

The use of neural networks on microcontrollers is an active topic in deep learning research, the main challenge being the design of effective models for inference, with a small memory footprint. To obtain a compact neural network without significantly lowering performance, some compression techniques have been proposed, such as pruning and quantization [9]. Memory replacement strategies have been investigated as well. Gural and Murmann [10] present Memory-Optimal Direct Convolutions (MODC) that performs existing convolution operators in place targeting KB-size devices at the expense of extra computation and slower performance. A comprehensive analysis [11] shows that the MODC presented in [10] achieved 65.7% accuracy on the CIFAR-10 dataset with less than 60 KB model memory consumption, but no experimental analysis is available on embedded devices with the achieved inference time. Unlu [12] proposes two optimizations that provide memory savings for 2D convolutions and fully connected layers; the first is in-

978-1-6654-2574-2/21/$31.00 ©2021 IEEE
DOI 10.1109/FiCloud49777.2021.00050

Authorized licensed use limited to: Universita degli Studi di Genova. Downloaded on February 11,2022 at 18:02:18 UTC from IEEE Xplore. Restrictions apply.

place max-pooling, the second is the use of ping-pong buffers between layers. The work focuses on memory optimization, achieving a 74% RAM utilization reduction than the reference CMSIS-NN implementation [13], but at the expense of a significant reduction in execution performance (e.g., 0.26 FPS on the gray scale MNIST dataset at 352 MHz core clock compared to 10.1 FPS on the more complex RGB CIFAR-10 dataset at 216 MHz) due to the latency caused by fetching data from flash and the lack of SIMD (single instruction multiple data) instructions. CMSIS-NN [13], that we use as the term of comparison also for our work, is a set of efficient neural network kernels designed to maximize the performance and minimize the memory requirements of neural network applications on ARM Cortex-M processors. The CMSIS-NN implementation uses a total buffer size for the feature maps corresponding to the sum of the output size of all the convolutional layers. Exploiting memory replacement in sequential execution, our approach allocates memory bounded by an estimation of the maximum needed without affecting the performance nor accuracy.

III. BACKGROUND

In this section, we briefly introduce the convolutional neural network class of deep learning, the software kernels utilized, and the deep learning framework used for training.

A. *Convolutional Neural Network*

A Convolutional Neural Network (CNN) is a deep learning algorithm which is widely used in pattern recognition and computer vision tasks. It consists of several building blocks such as convolutional layers, pooling layers and fully connected layers. A CNN is designed to learn spatial hierarchies of features through a backpropagation algorithm [14]. This model is particularly suited to image classification as it extracts useful features from the image by observing patterns in the dataset. Other deep learning models are much less efficient in this task, requiring a much higher number of trainable parameters.

B. *CMSIS-NN*

Arm's CMSIS-NN is a state of the art open-source library of optimized neural network functions for Cortex-M microcontrollers that enables the integration of NNs into the edge nodes of Internet-of-Things (IoT) applications. These optimized kernels are divided into several functions, each covering one category: Convolution, Pooling, Activation, Fully Connected, Softmax, and Optimized Basic Math [13]. This tool supports models trained with popular frameworks such as Caffe [15] or TensorFlow [16]. The model parameters (weights and biases) are first quantized into 8-bit or 16-bit integers and then deployed to the microcontroller. **Fig. 1** shows the structure of the CMSIS-NN kernels. Neural networks generated with CMSIS-NN achieve about 4.6X improvement in performance and 4.9X in energy efficiency compared to a baseline version using CMSIS-DSP functions [13]. Our approach focused on optimizing another key aspect, such as memory footprint, as shown in Section **IV**.

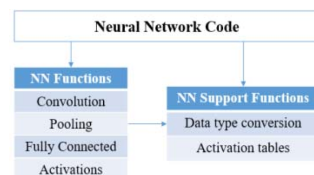

Fig. 1. CMSIS-NN structure.

C. *CAFFE*

Convolutional Architecture for Fast Feature Embedding or CAFFE [15] is a deep learning framework developed at the University of California, Berkeley. It is an open-source framework written in C++ and released under the BSD 2-Clause license. Caffe supports different types of deep learning architectures (CNN, RCNN, LSTM, and fully connected neural networks) focused on image segmentation and image classification. It also supports both GPU and CPU-based acceleration computational kernel libraries (NVIDIA, cuDNN, Intel MKL). In our implementation, we used the same model as CMSIS-NN for comparison purposes. The model is based on a built-in example provided by the Caffe framework.

IV. OPTIMIZATION

Our goal is to optimize memory efficiency in convolutional layer computation in CNNs through a replacement strategy. State of the art frameworks, such as CMSIS-NN, already do *in-place* (or *in-situ*) computation in the pooling layer, but they allocate memory space for each output feature map for each layer. The resulting memory footprint is the sum of the output sizes of the feature maps in all the convolutional layers. The deeper a network, the higher the sum. Our idea is to allocate a single buffer (we call it *total buffer*) sized as the maximum needed capacity.

The idea stems from the fact that convolutions are local operations; the output feature maps of a convolutional layer depend only on the input features of that layer. Therefore, after a layer has computed its operations, its memory can be reused to store output feature maps of subsequent layers. This requires that there is no out of order instruction execution, which guarantees that only the feature maps in the currently active layer are used in each layer operation. In order execution is the case of single-core processors, that cover the vast majority of microcontrollers.

The maximum needed capacity is computed as the maximum, among all the couples of adjacent layers, of the needed buffer size for each couple of adjacent layers (Eq. 1). **TABLE I.** reports the needed buffer size for the two most common types of layers of CNNs.

$$\text{total buffer size} = \max_{i\in 1,\, N-1} (\text{needed_buff_size}\ (L_i, L_{i+1})) \quad (1)$$

A pooling layer can be filled completely in place, since each pooling operation produces a single value and destroys at least one value. Convolutions, on the other hand, process the same input area one time for each filter, thus results cannot replace the values in the current layer, but should be put in a new buffer. Consequently, the required size for a couple of layers of which the second one is a convolutional one is given by the sum of their respective sizes.

TABLE I. NEEDED BUFFER SIZE FOR A COUPLE OF ADJACENT LAYERS

Next layer	
Convolutional	**Pooling**
Sum of the sizes of current and next layer's feature maps	Size of current layer's feature maps

Fig. 2 shows a typical application case of our optimization. 3-D tensors are used to represent data and weights, as it is common in machine learning frameworks. Please note that,

Authorized licensed use limited to: Universita degli Studi di Genova. Downloaded on February 11,2022 at 18:02:18 UTC from IEEE Xplore. Restrictions apply.

targeting an embedded environment, the CMSIS-NN implementation (atop of which we built), differs from this representation. Particularly, the weight tensors are transposed (reordered) and flattened into a 1-D array with {num. of input channels x filter height x filter width x num. of output channels} elements. Similarly, each layer's output is stored in a 1-D array with {num. of output channels x feature map height x feature map width} elements.

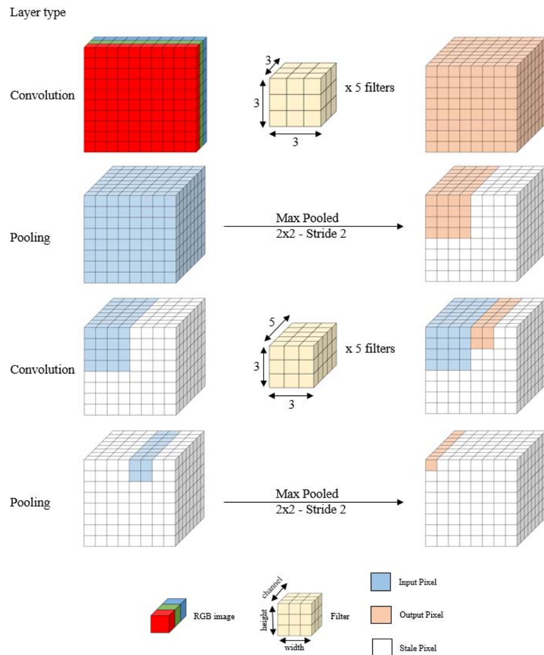

Fig. 2. Example of pixels arrangement in memory with the proposed replacement strategy.

The example uses two convolutional layers interleaved with max-pooling on a 10x10 RGB image. Here we focus on 3D convolutions with odd square filters (3x3, 5x5, etc.), same padding, and stride of 1, which covers the most common cases for CNNs in memory-limited applications [10], [12], [13].

The first convolutional layer consists of 5 filters of shape 3x3x3 and stride of 1 (in the CMSIS-NN implementation, this input is a 1-D array of shape {3x3x3x5}). The output size is 8x8x5, which corresponds to the feature map with width = (input width - (filter width - 1)), height = (input height - (filter height - 1)), and channels = (number of input filters). From equation (1) and **TABLE I.** it appears that the *total buffer* size is equal to the one of the first convolutional layer: 8x8x5, whose feature maps thus completely occupy the *total buffer*. The first 2x2 max-pooling layer takes as input these first output feature maps and results in a 4x4x5 second output feature maps. The pooling operation is destructive on the input; thus no additional memory is needed to store the output. Therefore, the second output feature maps are saved in place of the input, resulting in many pixels being freed, which we call stale pixels (8x8x5 - 4x4x5 = 240 pixels out of 320 allocated for the *total buffer* are stale). The third layer consists of 5 filters of shape 3x3x5 and stride of 1. We should emphasize that we used the same number of filters for a better graphical representation of our methodology, while using a different number is also possible, as in our case study in Section **V**. The input pixels in this layer needs to be preserved during the convolution operation, therefore the output pixels of size 2x2x5 (third output feature maps) should be stored next to (not in place of, as it happened for the pooling operation) the input pixels, occupying part of the stale pixels. In the last layer, the third output feature maps are the only input needed for this operation. The output size of this operation is 1x1x5 (fourth output feature maps), and is stored from the starting position given the destructive nature of the pooling operation. We can observe that, for each layer, there is always enough space to store its input and output in the total buffer, thus reducing the overall memory footprint.

Two main changes were needed to upgrade the CMSIS-NN implementation with our memory replacement feature. First, in the code generated by CMSIS-NN, instead of allocating the sum of the output sizes of the convolutional layers as scratch buffers before starting the inference, just one buffer is allocated (namely, the *total buffer*), sized as the maximum of the output feature maps, is allocated. Second, a pointer is created to refer to the first free slot in this buffer (namely, the *p_buffer*). As the inference algorithm progresses, the pointer is updated to place the output of the current convolutional layer in its correct position. The implementation can be found here: https://github.com/FouadSakr/Memory-Efficent-CMSIS-NN.

V. Case Study

We tested our approach by implementing our memory optimization strategy atop CMSIS-NN kernels on a CNN trained on the CIFAR-10 dataset [17]. The latter consists of 60,000 32x32 RGB images divided into 10 classes. For the assessment, we adopted the network architecture that was used to test CMSIS-NN [13], which is based on a built-in example in Caffe and whose topology is shown in **TABLE II.** This architecture is limited in size and provided good performance on edge devices. The model, pre-trained by Caffe, is quantized to 8-bit integers (one byte/parameter) with 79.9% accuracy (like the CMSIS-NN model), and then translated into source and header files for deployment on the microcontroller.

TABLE II. LAYER PARAMETERS

Operation Layer	Input Shape	Stride	Padding	Output Shape
Convolution	3x5x5x32	1	Same	32x32x32
Pooling ReLU	-	2	-	32x16x16
Convolution ReLU	32x5x5x32	1	Same	32x16x16
Pooling	-	2	-	32x8x8
Convolution ReLU	32x5x5x64	1	Same	64x8x8
Pooling	-	2	-	64x4x4
Fully-Connected	64x4x4x10	-	-	10

As explained in the previous section, we first need to allocate the *total_buffer*. Since the activations in our example are quantized to *int8*, each output value corresponds to one byte. Thus, the size of the buffer is 32x32x32 or 32,768 bytes, given by the largest output feature maps, resulting from the first convolutional layer. Second, the *p_buffer* pointer is

Authorized licensed use limited to: Universita degli Studi di Genova. Downloaded on February 11,2022 at 18:02:18 UTC from IEEE Xplore. Restrictions apply.

initialized, in order to place the output of the convolutional layer in the correct position after each pooling layer.

Fig. 3 shows the memory replacement strategy in our case study. We are using a 1-D array buffer to store all parameters during computation (coherently with the modality in which CMSIS-NN saves the output, as said in the previous section):

1. Convolution 1: the output of this layer is equal to 32x32x32 = 32,768 bytes (largest output feature maps) and occupies all the memory allocated to the *total_buffer*.

   Free space in total_buffer: 0 bytes

2. Pooling 1: the output of this layer is equal to 32x16x16 = 8,192 bytes. The nature of the pooling operation allows the output to be stored in place of the input starting from address 0 of *total_buffer*.

   Next p_buffer offset: 8,192

   Free space in total_buffer: 32,768 - 8,192 = 24,576 bytes

3. Convolution 2: the output of this layer is equal to 32x16x16 = 8,192 bytes. The blue sector represents the previously stored 8,192 bytes resulting from the pooling operation and they are only needed in the current layer during the convolution operation. Thus, the output activations of this convolutional layer are stored from *p_buffer* offset 8,192.

   Free space in total_buffer: 32,768 – 8,192 – 8,192 = 16,384 bytes

4. Pooling 2: the output of this layer is equal to 32x8x8 = 2,048 bytes. As indicated in the previous step, the blue sector is no longer needed in this layer, so the output of the pooling operation can be stored starting from address 0 of *total_buffer*.

   Next p_buffer offset: 2,048

   Free space in total_buffer: 32,768 – 2,048 = 30,720 bytes

5. Convolution 3: the output of this layer is equal to 64x8x8 = 4,096 bytes. The blue sector represents the previously stored 2,048 bytes resulting from the pooling operation and they are only needed in the current layer during the convolution operation. Thus, the output activations of this convolutional layer are stored from *p_buffer* offset 2,048.

   Free space in total_buffer: 32,768 – 2,048 – 4,096 = 26,624 bytes

6. Pooling 3: the output of this layer is equal to 64x4x4 = 1,024 bytes. As indicated in the previous step, the blue sector is no longer needed in this layer, so the output of the pooling operation can be stored starting from address 0 of *p_buffer*.

   Next p_buffer offset (if additional layers exists): 1,024

   free space in total_buffer: 32,768 – 2,048 = 30,720 bytes

As our work focuses only on reducing the required memory for the activations, the model parameters (weights and biases) in both cases (plain and enhanced CMSIS-NN) are the same (their size is equal to the sum of the input shape column in **TABLE II.** , i.e., 89,440 bytes). On the other hand, the memory needed for the activation (feature maps) in the original CMSIS-NN case is equal to the sum of the output shape column in **TABLE II.** without adding the rows resulting from the pooling layers as pooling operations are performed *in-place* (45,056 bytes). Overall results (**TABLE III.**) show that our replacement strategy reduces the feature maps' memory footprint by more than 27%. Considering the total NN model occupancy (i.e., parameters + activations) is reduced by slightly more than 9%.

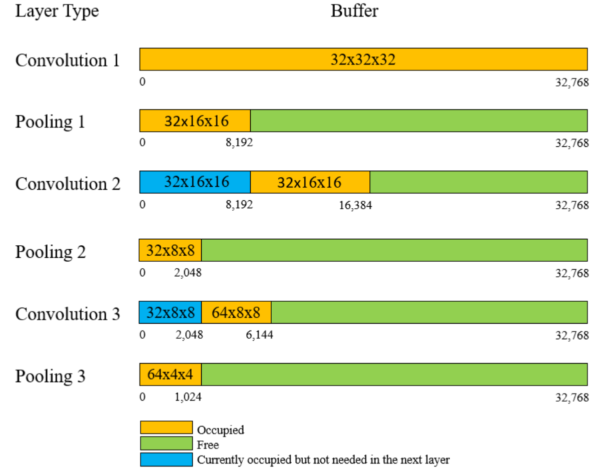

Fig. 3. Buffer utilization to save output activations throughout the layers.

TABLE III. FEATURE MAPS' MEMORY UTILIZATION COMPARISON

	CMSIS-NN (bytes)	**Proposed Memory Replacement Strategy** (bytes)	**Reduction**
Activations	45,056	32,768	27.27%

Considering the timing performance, we measured the inference time using a NUCLEO-H743ZI2 board with an Arm Cortex-M7 core running at 216 MHz (max 480 Mhz) [18], which is the processing speed used in the testing experiment on the CMSIS-NN kernels [13]. The results show that the runtime is not affected by this optimization and is stable at 10.1 FPS. This is reasonable, as our solution relies on the efficient CMSIS-NN kernels, and the memory required for the activations is allocated before the execution starts, in both cases. Thus, our proposed memory replacement strategy does not affect the execution performance.

A. Experiment with deeper networks

In this section, we are interested in exploring the performance of our proposed optimization in the case of deeper architectures, with a higher number of filters. Deeper networks are capable of learning more abstract and powerful patterns of the input and are increasingly being employed.

We thus tested two other CNN architectures, such as the following ones:

- The first architecture (Model 2) has four convolutional layers each one followed by a 2x2 max-pooling layer, and a fully connected layer:

  1- 32 3x3 kernels, stride = 1, padding = same

Authorized licensed use limited to: Universita degli Studi di Genova. Downloaded on February 11,2022 at 18:02:18 UTC from IEEE Xplore. Restrictions apply.

2- 64 3x3 kernels, stride = 1, padding = same

3- 128 3x3 kernels, stride = 1, padding = same

4- 256 3x3 kernels, stride = 1, padding = same

- Another deeper architecture (Model 3) has five convolutional layers each one followed by a max-pooling layer, and a fully connected layer:

1- 32 1x1 kernels, stride = 1, padding = same

2- 64 1x1 kernels, stride = 1, padding = same

3- 128 1x1 kernels, stride = 1, padding = same

4- 256 1x1 kernels, stride = 1, padding = same

5- 512 1x1 kernels, stride = 1, padding = same

The results using our memory optimization method are shown in the bar chart of **Fig. 4**, which also includes the first model in our case study (Model 1).

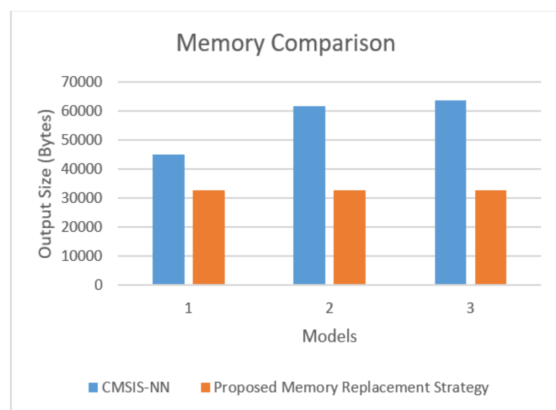

Fig. 4. Memory comparison bar chart.

Overall, the results show that using our approach significantly reduces the amount of memory needed to store feature maps during computation. Moreover, we can observe that as the architecture becomes deeper, the reduction percentage increases significantly: 46% and 48% for Model 2 and Model 3, respectively, compared to the first model (27%).

VI. CONCLUSION AND FUTURE WORK

The ever-evolving field of embedded deep learning presents several challenges and opportunities. Using microcontrollers to run machine learning applications has proven to be a viable solution, but is limited by memory constraints. To alleviate this issue, we have proposed a memory replacement strategy that minimizes memory footprint during inference, while maintaining the same performance offered by the state of the art CMSIS-NN kernels. Our experiments show that we were able to achieve a significant reduction in the required memory for feature maps compared to the CMSIS-NN implementation (27%, corresponding to a 9% reduction of the total NN model size), and the reduction is even higher for deeper networks. This is achieved at no expense in terms of accuracy nor timing performance.

As initial results are promising, our approach should be more extensively tested on other datasets and architectures. Moreover, as the analysis shows that also most of the pixels in the *total buffer* tend to stay idle, it would be interesting to verify a pipeline approach among different deep learning tasks.

REFERENCES

[1] S. Branco, A. G. Ferreira, and J. Cabral, “Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey,” Electron., vol. 8, no. 11, 2019, doi: 10.3390/electronics8111289.

[2] S. Han, H. Mao, and W. J. Dally, “DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING.”

[3] L. Hou, Q. Yao, and J. T. Kwok, “LOSS-AWARE BINARIZATION OF DEEP NETWORKS.”

[4] M. Denil, B. Shakibi, L. Dinh, M. ’ A. Ranzato, and N. De Freitas, “Predicting Parameters in Deep Learning.”

[5] G. Hinton and J. Dean, “Distilling the Knowledge in a Neural Network,” 2015.

[6] E. Liberis and N. D. Lane, “Neural networks on microcontrollers: saving memory at inference via operator reordering.”

[7] “Cortex-M – Arm Developer.” [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m. [Accessed: 22-Feb-2021].

[8] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, “An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks,” Futur. Internet, vol. 12, no. 7, p. 113, 2020, doi: 10.3390/fi12070113.

[9] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “A survey of model compression and acceleration for deep neural networks,” arXiv, pp. 1–10, 2017.

[10] A. Gural and B. Murmann, “Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications,” 36th Int. Conf. Mach. Learn. ICML 2019, vol. 2019-June, pp. 4454–4472, 2019.

[11] S. Müksch, T. Olausson, J. Wilhelm, and P. Andreadis, “Quantitative Analysis of Image Classification Techniques for Memory-Constrained Devices,” 2020.

[12] H. Unlu, “Efficient neural network deployment for microcontroller,” arXiv, 2020.

[13] L. Lai, N. Suda, and V. Chandra, “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs,” pp. 1–10, 2018.

[14] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, “Convolutional neural networks: an overview and application in radiology,” Insights into Imaging, vol. 9, no. 4. Springer Verlag, pp. 611–629, 01-Aug-2018, doi: 10.1007/s13244-018-0639-9.

[15] Y. Jia et al., “Caffe: Convolutional architecture for fast feature embedding,” MM 2014 - Proc. 2014 ACM Conf. Multimed., pp. 675–678, 2014, doi: 10.1145/2647868.2654889.

[16] “TensorFlow.” [Online]. Available: https://www.tensorflow.org/. [Accessed: 23-Feb-2021].

[17] “CIFAR-10 and CIFAR-100 datasets.” [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 23-Feb-2021].

[18] F. Sakr, F. Bellotti, R. Berta, and A. De Gloria, “Machine learning on mainstream microcontrollers†,” Sensors (Switzerland), vol. 20, no. 9, 2020, doi: 10.3390/s20092638.

Authorized licensed use limited to: Universita degli Studi di Genova. Downloaded on February 11,2022 at 18:02:18 UTC from IEEE Xplore. Restrictions apply.