This ICCV Workshop paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the accepted version;
the final published version of the proceedings is available on IEEE Xplore.

Contextual Convolutions for Scalable Forward-Only Learning on Tiny Devices

Mehdi Abbassi
Fondazione Bruno Kessler
mabbassi@fbk.eu

Alberto Ancilotto
Fondazione Bruno Kessler
aancilotto@fbk.eu

Elsabetta Farella
Fondazione Bruno Kessler
efarella@fbk.eu

Abstract

On-device training on resource-constrained hardware, such as microcontrollers with limited memory and fixed-function convolutional accelerators, remains an open challenge in embedded computer vision. Standard backpropagation is often impractical due to its high memory requirements and reliance on operations unsupported by typical inference-optimized accelerators. Recent forward-only learning methods, such as Forward-Forward and PEPITA, offer lightweight alternatives by eliminating the backward pass, enabling training on ultra-low-power devices. However, these methods tend to degrade in performance on more complex tasks involving deeper networks and larger output spaces. In this work, we introduce the Contextual Convolution Block, a novel architectural module that enhances the representational capacity of forward-only networks by injecting ground truth class information during training. This allows the network to specialize convolutional kernels for specific classes without relying on gradients or weight transport. We further present an optimized implementation of this block using an im2col-based formulation, enabling efficient training on severely constrained devices. Our method significantly improves the scalability of forward-only training approaches, achieving stronger performance on complex classification tasks while preserving compatibility with embedded hardware limitations.

1. Introduction

Training deep neural networks on-device remains a fundamental challenge in embedded computer vision. The standard backpropagation algorithm, while highly effective, is resource-intensive, particularly in terms of memory. On-device training requires storage of intermediate activations and gradients, often exceeding the memory capacity of typical microcontroller units (MCUs), which frequently provide only hundreds of kilobytes of SRAM. Furthermore, even when equipped with dedicated convolutional accelerators, these devices are generally optimized for inference and may lack support for the broader set of operations required by backpropagation, such as transposed convolutions or high-precision accumulation.

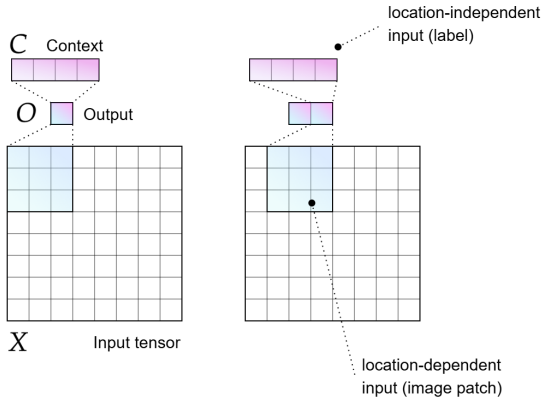


Figure 1. The proposed Contextual Convolution Block. While input image patches change during the sliding process, the context vector stays constant and is incorporated at every spatial location.

Existing approaches to on-device learning typically aim to reduce the memory demands of backpropagation through compression, activation checkpointing, or partial gradient updates. However, these strategies may still be incompatible with the limited instruction sets and constrained memory architectures of ultra-low-power MCUs. In contrast, forward-only training methods—such as Forward-Forward (FF [9]) learning and error-driven input modulation (e.g., PEPITA [3])—eliminate the backward pass entirely, significantly lowering memory requirements and computational complexity. These approaches have shown competitive performance to backpropagation on image classification benchmarks, especially for small-scale tasks and shallow networks. Yet, their performance tends to degrade on more complex tasks involving deeper architectures and a larger number of output classes.

In this work, we propose a novel architectural enhancement to forward-only networks that enables them to scale to more complex visual recognition tasks while retaining their

compatibility with severely constrained hardware. Specifically, we introduce the Contextual Convolution Block, a new module designed to improve the representational capacity of FF-trained networks by introducing a fixed context vector in the convolution operation (Figure 1), with very low parameter count but demonstrating significant performance increase in real-world applications.

To ensure the feasibility of our approach on embedded platforms, we also present an optimized implementation of the Contextual Convolution Block using the im2col transformation. This formulation allows us to express the block's operations as efficient matrix multiplications—fully compatible with the hardware acceleration paths available on many inference-optimized MCUs. The resulting training pipeline preserves the low memory footprint and fully forward nature of methods like FF and PEPITA, while extending their scalability to larger classification tasks.

The paper is structured as follows: Section 2 introduces the existing forward-only approaches for on-device learning, particularly focusing on PEPITA [3] and Forward-Forward [9]. Section 3 introduces the proposed Contextual convolution block, and Section 4 shows the proposed efficient implementation for the block based on a modified form of the traditional `im2col` algorithm. In Section 5 we analyze the performance of the proposed approach on three benchmark datasets - MNIST, Cifar10 and Cifar100. Section 6 presents an ablation study on the achieved performance, while Section 7 and Section 8 analyze the compute and memory overhead of the proposed convolutional block compared to the standard 2D convolution. Finally, Section 9 shows achieved on-device performance across different classes of devices, from microcontrollers (STM32) to microprocessors (using Raspberry Pi single board computers) to TPUs (Nvidia Jetson).

2. Previous Works

Learning on severely memory constrained devices, such as microcontrollers with 100–500 KB SRAM, poses substantial challenges. Traditional backpropagation is generally infeasible in these settings due to its high memory footprint, computational overhead, and latency introduced by the backward pass. Storing intermediate activations and gradients can increase memory usage by over 6× compared to inference alone [2, 9, 14], making on-device backpropagation prohibitively expensive. Moreover, power constraints and real-time requirements further limit the practicality of conventional training frameworks on embedded platforms [15].

To address these constraints, several alternatives to standard backpropagation have been proposed. Forward-Forward (FF) learning [9] replaces the backward pass with two forward passes: a positive pass with real data and a negative pass with corrupted inputs. Each layer learns locally to maximize a *goodness* function, thus removing the need for global error propagation and allowing training to proceed without storing intermediate states. FF achieves competitive accuracy on image classification tasks while significantly reducing memory usage, and has been extended to convolutional architectures with promising results [1, 5, 20].

PEPITA [3] (Present Error to Perturb Input To modulate Activity) takes a complementary approach by injecting the output error into the input through a fixed random feedback matrix and performing two forward passes per sample. Weight updates are derived from the differences in activations, resulting in a local Hebbian-like rule that avoids both weight transport and gradient storage. Despite its simplicity, PEPITA has demonstrated strong performance on vision benchmarks, with accuracy only marginally lower than that of backpropagation.

Other biologically inspired or forward-compatible training strategies include Feedback Alignment (FA) [13] and Direct Feedback Alignment (DFA) [18], which use random fixed weights to propagate errors, and Difference Target Propagation [12], which replaces gradients with target activations. While these relax the symmetry constraints of backpropagation, they still require backward-phase computations and intermediate state storage. More recent local learning rules such as Equilibrium Propagation [7] aim to increase biological plausibility but often rely on iterative solvers or symmetric pathways, limiting their applicability to ultra-low-power hardware.

In contrast to these methods, FF and PEPITA offer fully forward-pass training pipelines with local updates, making them attractive candidates for streaming and continual learning on highly constrained edge devices. In particular, the low memory footprint of FF positions it as a promising building block for our approach.

3. Contextual convolution formulation

Despite the promise of forward-only learning methods such as Forward-Forward (FF) and PEPITA for on-device training, their representational power is often limited by the use of standard convolutional layers. These conventional layers lack a direct mechanism for incorporating class- or context-specific information during feature extraction, potentially constraining performance on more complex visual recognition tasks.

To address this limitation, we introduce the Contextual Convolution Block. Unlike a standard convolutional layer, which applies the same set of kernels uniformly across all inputs, the Contextual Convolution Block augments the convolutional operation with a learnable, location-independent context vector $\mathbf{c} \in \mathbb{R}^d$. This vector can encode any auxiliary information relevant to the task, such as class labels or global context features, and remains fixed through-

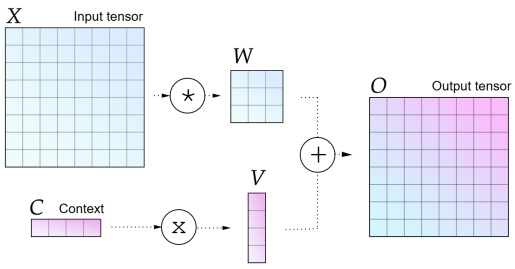


Figure 2. Illustration of the operations in the proposed Contextual Convolution Block. The input image patches change during the sliding process, while C remains constant.

out the spatial sliding process for a given input.

Formally, let $\mathbf{x} \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ denote the input feature map and $\mathbf{w} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w}$ the convolutional weights. The output at channel k and spatial location (i, j) is computed as:

$$y_{k,i,j} = \sum_{c=1}^{C_{\text{in}}} \sum_{u=1}^{K_h} \sum_{v=1}^{K_w} w_{k,c,u,v} \cdot x_{c,i+u,j+v} + \langle \mathbf{v}_k, \mathbf{c} \rangle, \quad (1)$$

where $\mathbf{v}_k \in \mathbb{R}^d$ is a learnable projection vector associated with the k-th output channel, and $\langle \cdot, \cdot \rangle$ denotes the dot product. The term $\langle \mathbf{v}_k, \mathbf{c} \rangle$ generalizes the conventional scalar bias to a context-aware, input-dependent offset.

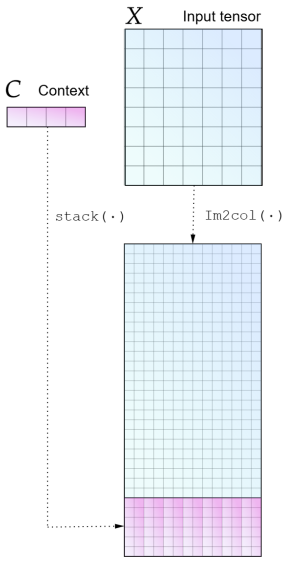
In our experiments, we instantiate $\mathbf{c}$ as a one-hot encoded label vector for positive samples and as an incorrect label for negative samples, following the FF training paradigm. This design enables the network to learn class-specialized kernels and integrate label information directly during feature extraction, overcoming a key limitation of prior forward-only architectures.

An illustration of the Contextual Convolution Block is shown in Figure 2, where the context vector is provided alongside the image input and remains constant for all spatial locations.

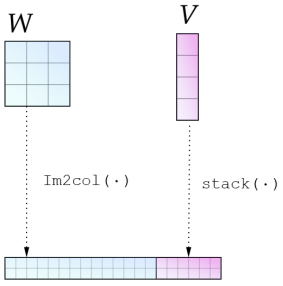
This formulation not only increases the expressive capacity of forward-only models but is also hardware-friendly. By designing the context integration as a simple, input-independent dot product, we retain compatibility with efficient matrix multiplication routines and hardware-accelerated `im2col` implementations, making the approach practical for resource-constrained devices.

4. Efficient implementation

Efficient implementation of convolutional layers is critical for embedded devices with severe memory and compute constraints. Among existing algorithms, the `im2col` (image-to-column) method is particularly attractive for



(a) Input image and context vector transformed into the modified `im2col` matrix.



(b) Filters and context weights flattened and transformed into the modified `im2col` matrix.

Figure 3. The `im2col` algorithm transforms input images and kernels into two large matrices, replacing the sliding window process with a single matrix multiplication.

resource-constrained settings due to its simplicity and compatibility with matrix multiplication accelerators. Unlike more complex algorithms such as Strassen [8] or Winograd [11], `im2col` is straightforward to implement and yields substantial performance gains for small to moderate feature sizes, which are typical in edge applications.

`im2col` reformulates the convolution operation by extracting and flattening all sliding local patches of the input tensor into column vectors. These columns are concatenated to form a large input matrix, while convolutional kernels are likewise flattened into row vectors and concatenated into a filter matrix. The convolution is then performed as a single matrix multiplication between the filter and input matrices, efficiently leveraging existing hardware-accelerated routines.

This transformation is illustrated in Figure 3. For

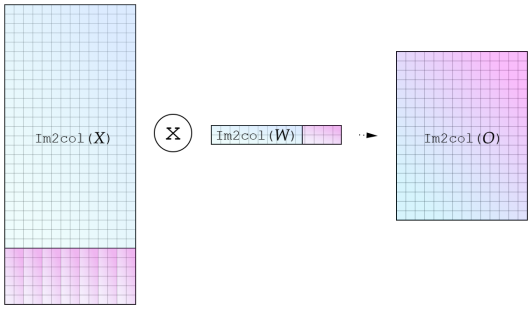


Figure 4. The GEMM operation results in the same output shape as the standard im2col operation, from which the output tensor can be computed

batched inputs, the `im2col` input matrix is extended column-wise to accommodate all samples (Figure 4). The final output is reconstructed by splitting and reshaping the matrix according to batch size and output channels.

4.1. Adaptation for Contextual Convolution

While standard `im2col` efficiently handles pure convolutional operations, our contextual convolution introduces additional context-specific information alongside the image data. We propose two key modifications to account for this context while retaining the algorithm's computational benefits:

- **Label Broadcasting and Concatenation**: The context vector (e.g., a one-hot label) is broadcast to match the number of spatial patches in the input and concatenated as additional rows to the `im2col` matrix. This increases the matrix row dimension by the length of the context vector, without affecting the number of columns.
- **Additional Weights for Context**: We introduce a set of learnable weights for the context vector, appended to the flattened kernel vectors. The filter matrix therefore grows in column dimension by the length of the context vector, while retaining the same number of output channels (rows).

Formally, if the input `im2col` matrix has shape $(C_{\text{in}} \cdot K_h \cdot K_w) \times N$, where N is the number of patches, and the context vector has length d, the augmented input matrix will have shape $[(C_{\text{in}} \cdot K_h \cdot K_w) + d] \times N$, while the augmented filter matrix will have shape $C_{\text{out}} \times [(C_{\text{in}} \cdot K_h \cdot K_w) + d]$.

4.2. Practical Example

Consider an input tensor with 3 channels and a 4×4 spatial dimension, alongside two 2×2 kernels. Furthermore, we wish to incorporate a context vector of length 5 (for instance, a one-hot class label). The adaptation process operates as follows:

1. Each local patch from the input is flattened into a column vector.
2. The context vector is broadcast and concatenated to each column vector, adding 5 additional elements at the bottom of the column.
3. The two 2×2 kernel weights are flattened into row vectors, and their respective context weights are concatenated to match the new column dimension.
4. The resulting matrix multiplication produces output values for each patch, which can then be reshaped back into their respective output formations.

This procedure is illustrated in Figure 3 and Figure 4. Importantly, these modifications retain the algorithm's compatibility with highly optimized routines for general matrix-matrix operations (GEMM), ensuring that contextual convolution can be efficiently implemented even on hardware with limited resources.

5. Experiments

We evaluate the proposed Contextual Convolution Block and training strategy on three standard vision benchmarks: MNIST, CIFAR-10, and CIFAR-100. Our experiments focus on: (1) accuracy comparisons with prior forward-only and local learning methods, (2) ablations to assess the impact of label-as-bias integration, and (3) analysis of memory usage, including peak activation buffer and parameter count.

5.1. Datasets, Model Architecture, and Training Setup

We use the conventional train/test splits for MNIST [4], CIFAR-10 [10], and CIFAR-100 [10]. Prior work such as Scodellaro et al. [20] reports results only on MNIST and does not scale to CIFAR-100; Hinton [9] also omits CIFAR-100.

Our model comprises two contextual convolutional blocks followed by a linear classifier. For MNIST and CIFAR-10, the first block has 64 filters of size 2×2 (stride 1, padding 1) and is followed by 2×2 max pooling; the second block has 256 filters of size 3×3 and is followed by 3×3 max pooling. For CIFAR-100, the blocks use 128 and 512 filters, respectively. The output of the final pooling layer is passed to a fully connected classifier. All convolutional layers use the contextual convolution described in Section 3.

To improve generalization—especially important in the absence of top-down feedback—we apply data augmentation. For MNIST, only padding and cropping are used. For CIFAR-10 and CIFAR-100, we apply:

- Random padding and cropping (up to 2 pixels)
- Brightness variation (± 0.1)
- Saturation variation (± 0.1)
- Contrast variation (± 0.1)

- Hue variation (± 0.02)
- Horizontal flipping (probability 0.5)

We train using a batch size of 256. The two convolutional blocks use learning rates of 3×10^{-7} and 2×10^{-7}, respectively. The classifier uses a learning rate of 6×10^{-3} for MNIST and 3×10^{-3} for CIFAR-10/100, with a weight decay of 5×10^{-5}. A custom scheduler adjusts the classifier's learning rate by multiplying it by 1.95 every 10 epochs and halving it in the subsequent epoch. Convolutional layers are initialized with Kaiming normal initialization, and the classifier with Kaiming uniform. The convolutional blocks are trained for 5 epochs, followed by 100 epochs of classifier training.

All results are reported as the mean and standard deviation over 10 runs with random seeds from 0 to 9. Hyperparameters were tuned using 10% of the training set; final results are reported on the test set, which remained completely isolated during training.

5.2. Accuracy Comparison

Tables 1, 2, and 3 summarize test accuracy, model size, and peak activation buffer for each method. We compare the following baselines and approaches:

- **PEPITA**: Fully connected and convolutional architectures trained with the PEPITA algorithm as reported in [3].
- **Forward-Forward (FF)**: Unsupervised and supervised variants, with fully connected, local receptive field (LRF) [9], and convolutional architectures [20]. Label integration strategies include overlaid [9] and spatially expanded [20].
- **Ours**: Forward-Forward with the proposed Contextual Convolution Block, where label information is injected as a bias in each convolutional layer.

We exclude label scoring approaches from our comparisons, as these scale poorly with increasing label counts and are not practical for deployment on highly resource-constrained devices.

5.2.1. MNIST

On MNIST, our approach matches prior state-of-the-art accuracy among forward-only methods while maintaining a compact parameter count and moderate peak activation usage. Notably, using labels as bias in convolutional architectures achieves highly competitive performance with a parameter count nearly an order of magnitude smaller than fully connected FF models. Full results are in Table 1.

5.2.2. CIFAR-10

On CIFAR-10, our approach outperforms both PEPITA and previously published FF architectures with similar model size, confirming the benefit of contextual convolution in more complex tasks. The use of label-as-bias enables our network to close the gap with significantly larger FF models using label scoring or overlaid labels. See Table 2 for details.

Method	Sup.	Arch.	Accuracy (%)	Num. Param.
PEPITA [3]	✓	FC	98.01±0.09	821K
PEPITA [3]	✓	ConvNet	98.29±0.13	193K
FF [9]	✗	FC	98.63	13.6M
FF [9]	✗	LRF	98.84	**226K**
FF (Overlaid) [9]	✓	FC	98.64	13.6M
FF (Spatial) [20]	✓	ConvNet	99.00	3.4M
FF [9]	✗	LRF	**99.36**	**226K**
Ours	✓	ConvNet	98.44±0.16	284K

Table 1. Comparison of test accuracy and model size for MNIST across different methods. Note that PEPITA includes the F matrix along with other parameters.

Method	Arch.	Accuracy (%)	Num. Param.
PEPITA [3]	FC	52.57±0.36	3.19M
PEPITA [3]	ConvNet	56.33±1.25	361K
FF (Top-down) [9]	MRNN+LRF	56	338K
Ours	ConvNet	**66.59±0.85**	**215K**

Table 2. Comparison of test accuracy for CIFAR-10 across different methods.

5.2.3. CIFAR-100

On CIFAR-100, where label space and task complexity are much higher, our method again yields the highest accuracy among forward-only and local learning approaches, despite the resource constraints. Compared to PEPITA, the Contextual Convolution Block achieves both higher accuracy and a lower parameter count, highlighting the scalability of our design. Results are reported in Table 3. Due to the algorithm formulation, the large number of classes in CIFAR-100 makes FF not directly applicable. We instead report the values from [6], noting, however, that this training was achieved with *overlapping local updates* requiring backpropagation through one network operation, and as such, the final accuracy is marked with [*].

Method	Arch.	Accuracy (%)	Num. Param.
PEPITA [3]	FC	24.91±0.22	3.56M
PEPITA [3]	ConvNet	27.56±0.60	3.59M
FF (TRIFECTA) [6]	ConvNet	35.79*±0.17	3.6M*
Ours	ConvNet	**38.84±0.57**	**2.5M**

Table 3. Comparison of Test Accuracy for CIFAR-100 Across Different Methods

Training	Dataset	Accuracy (%)	Drop
Frozen ConvNet	CIFAR-10	**57.95 ± 2.18**	**-8.64**
Frozen ConvNet	CIFAR-100	**28.86 ± 1.08**	**-9.98**

Table 4. Ablation on CIFAR-10 and CIFAR-100: In both cases we witness a significant drop in test accuracy.

6. Ablation Study

Our classification models consist of a feature extractor and a classifier head. The convolutional and pooling layers serve as the feature extractor and are trained using the proposed algorithm, while the classifier head is trained via direct supervision using cross-entropy loss between predicted logits and ground-truth labels.

To evaluate the effectiveness of the proposed algorithm in learning useful representations, we conduct an ablation study in which the classifier head is trained with features from an untrained ConvNet, keeping all other settings identical. This isolates the contribution of features learned by the Contextual Convolution blocks to the final classification performance.

7. Computational Overhead

While incorporating Contextual Convolution operations within a network strongly improves its accuracy in forward-only training, these inevitably result in computational overhead. In a typical architecture, the number of operations required to compute the dot product between v_k and c is typically small compared to the number of operations to compute the convolution result itself. The same observation also applies to the number of parameters. Table 6 shows the effect of replacing standard convolutions with Contextual Convolutions in the architectures used for classifying the datasets presented. As can be seen, the overhead in terms of MACs and parameters is less than 1% even for Contextual Convolutions with a large number of input labels (e.g., Cifar100). This demonstrates the ability of this approach to scale to high numbers of classes without too much computational overhead, which is impossible using alternative approaches.

Method	Memory Peak
PEPITA (FC)	825K
PEPITA (ConvNet)	397K
FF (MRNN with LRF)	—
Ours (Contextual ConvNet)	**366K**

Table 5. Peak memory usage for different methods on CIFAR-10. Memory usage for FF (Multi-layer Recurrent Neural Network with Local Receptive Field) is not reported due to the nature of its recurrent architecture, which results in significantly higher peak memory.

8. Memory Overhead

We analyze the impact of our proposed Contextual Convolutional Block on peak memory during training. Although the use of context vectors introduces some additional overhead, its effect on the overall memory footprint is minimal. As shown in Table 5, our method consumes less memory than prior forward-only approaches such as FF and PEPITA.

The original Forward-Forward method cannot leverage the weight-sharing efficiency of convolutional layers. PEPITA, on the other hand, requires a class-dependent projection matrix (F) to project errors into the input space, which scales poorly with the number of classes. Standard backpropagation methods demand significantly more memory due to the storage of intermediate activations for gradient computation.

In contrast, Contextual Convolutions offer context-aware feature extraction with minimal memory overhead, making them an efficient alternative for resource-constrained training environments.

9. On-device tests

To verify the effectiveness of the modified im2col algorithm implementation, we evaluated the effective latency of the Contextual Convolutions-based network against the reference network based on the standard Conv2D. For latency tests, we use four different devices to represent different types of embedded devices:

- A Raspberry Pi 5 [16] and a Raspberry Pi 4 [17], to represent performance on CPU-only microprocessor-based devices (MPUs). In these devices, storage and RAM are generally not stringent constraints, as we have GBs available for both.

Dataset	Metric	Standard Conv	Contextual Conv	Overhead (%)
MNIST	MACs	41.44M	41.51M	+0.07M (0.17%)
	RAM	0.58M	0.65M	+0.07M (12.17%)
	Parameters	254.14KB	254.94KB	+0.80KB (0.31%)
CIFAR-10	MACs	54.86M	54.95M	+0.09M (0.16%)
	RAM	0.71M	0.79M	+0.088M (12.37%)
	Parameters	280.70KB	281.50KB	+0.80KB (0.29%)
CIFAR-100	MACs	55.51M	55.60M	+0.09M (0.16%)
	RAM	1.35M	1.45M	+0.096M (7.20%)
	Parameters	928.79KB	936.79KB	+8.00KB (0.86%)

Table 6. Computational Overhead of the Contextual Convolution-based network vs Standard Convolution one. Note that real-life RAM usage is higher than the memory peak in Table 5 due to the inference library overhead.

Device	MNIST net latency [ms]			CIFAR-10 net latency [ms]			CIFAR-100 net latency [ms]		
	C2D	Proposed	Δ	C2D	Proposed	Δ	C2D	Proposed	Δ
Raspberry Pi 5	1.01	1.02	+1.0%	1.35	1.38	+2.2%	1.95	1.95	0%
Raspberry Pi 4	4.59	4.61	+0.44%	6.04	6.19	+2.5%	6.77	6.79	+0.30%
Jetson Xavier CPU	3.04	3.62	+19%	4.77	5.46	+14%	3.87	4.82	+25%
Jetson Xavier CUDA	1.56	1.59	+1.9%	1.90	1.94	+2.1%	1.81	1.82	+0.55%
STM32H7	298.07	298.21	+0.05%	424.73	424.82	+0.021%	428.80	429.34	+0.13%

Table 7. On-device latency overhead (in ms) of contextual convolution compared to standard convolution across MNIST, CIFAR-10, and CIFAR-100.

- An Nvidia Jetson Xavier NX [19], to represent larger, TPU/GPU-equipped embedded devices. We test our approach on GPU using CUDA and on CPU only.
- An STM32H7 [21] microcontroller to represent the more resource-constrained category of embedded devices. The MCU used has 2MB of Flash and 1MB of RAM.

As shown in Table 7, the latency overhead introduced by our optimized im2col implementation is minimal. Notably, even networks with a large number of classes, such as Cifar100, exhibit only a small overhead while benefitting from a significant performance boost. The overhead is particularly negligible on the STM32 microcontroller, where the simpler memory architecture and limited vectorization capabilities diminish the potential acceleration of standard convolution. In contrast, on more capable platforms with complex memory hierarchies and strong vectorization, such as the Jetson CPU, the overhead becomes larger — yet still well within reasonable bounds.

10. Conclusion

In this work, we introduced Contextual Convolution blocks, a lightweight mechanism designed to integrate context information directly into the feature extraction process. Unlike existing forward-only approaches, Contextual Convolutions allow scaling network training to more complex problems and a higher number of classes, while keeping the computational overhead negligible.

We conducted a thorough evaluation across three standard image classification benchmarks—MNIST, CIFAR-10, and CIFAR-100—comparing our approach against forward-only baselines (FF, PEPITA). Results demonstrate how the proposed approach outperforms existing forward-only techniques in accuracy while maintaining low parameter counts and competitive memory footprints. Notably, on CIFAR-100, Contextual Convolutions exhibit superior scalability and generalization despite operating under strict memory constraints.

We also presented an optimized mechanism to perform the proposed blocking based on a modified version of im2col. This was evaluated on-device on several embedded devices to verify its actual applicability, resulting in negligible latency overheads against significant performance gains, especially in more complex tasks.

Acknowledgments

This work was supported by Ministero delle Imprese e del Made in Italy (IPCEI Cloud DM 27 giugno 2022 – IPCEI-CL-0000007) and European Union (Next Generation EU).

References

[1] Ege Aktemur, Ege Zorlutuna, Kaan Bilgili, Tacettin Emre Bok, Berrin Yanikoglu, and Suha Orhun Mutluergil. Going forward-forward in distributed deep learning, 2024. 2

[2] Ayan Chakrabarti and Benjamin Moseley. *Backprop with approximate activations for memory-efficient network training*. Curran Associates Inc., Red Hook, NY, USA, 2019. 2

[3] Giorgia Dellaferrera and Gabriel Kreiman. Error-driven input modulation: Solving the credit assignment problem without a backward pass, 2022. 1, 2, 5, 6

[4] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. 4

[5] Qikang Deng, Guang Jin, Yohwan Noh, Subin Yu, and Dohoon Lee. Dff: Distributed forward-forward algorithm for large-scale model in low-performance devices. *2023 IEEE 6th International Conference on Pattern Recognition and Artificial Intelligence (PRAI)*, pages 1218–1223, 2023. 2

[6] Thomas Dooms, Ing Jyh Tsang, and Jose Oramas. The trifecta: Three simple techniques for training deeper forward-forward networks, 2023. 5, 6

[7] Maxence Ernoult, Julie Grollier, Damien Querlioz, Yoshua Bengio, and Benjamin Scellier. Equilibrium propagation with continual weight updates, 2020. 2

[8] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 610 (7930):47–53, 2022. 3

[9] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations, 2022. 1, 2, 4, 5

[10] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. 4

[11] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *CVPR*, pages 4013–4021. IEEE Computer Society, 2016. 3

[12] Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases*, pages 498–515, Cham, 2015. Springer International Publishing. 2

[13] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014. 2

[14] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, and Song Han. Tiny machine learning: Progress and futures [feature]. *IEEE Circuits and Systems Magazine*, 23(3):8–34, 2023. 2

[15] Nil Llisterri Giménez, Marc Monfort Grau, Roger Pueyo Centelles, and Felix Freitag. On-device training of machine learning models on microcontrollers with federated learning. *Electronics*, 11(4), 2022. 2

[16] Raspberry Pi Ltd. Raspberry pi5 product brief. Datasheet (Product Brief), 2025. 64-bit quad-core Cortex-A76 CPU @2.4GHz; VideoCore VII GPU. 6

[17] Raspberry Pi (Trading) Ltd. Raspberry pi4 model b datasheet. Datasheet (Release1.1), 2024. Broadcom BCM2711 SoC, up to 8GB LPDDR4 SDRAM. 6

[18] Arild Nokland. Direct feedback alignment provides learning in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, page 1045–1053, Red Hook, NY, USA, 2016. Curran Associates Inc. 2

[19] NVIDIA Corporation. Nvidia jetson xavier nx system-on-module datasheet. Datasheet DA-09366-001, preliminary version, 2019. Volta GPU + Carmel ARM, up to 21TOPS INT8 performance. 7

[20] Riccardo Scodellaro, Ajinkya Kulkarni, Frauke Alves, and Matthias Schröter. Training convolutional neural networks with the forward-forward algorithm, 2023. 2, 4, 5

[21] STMicroelectronics. Stm32h7 series 32-bit arm cortex-m7 microcontrollers datasheet. Datasheet (e.g., STM32H743xI/G family), 2022. Up to 2MB Flash, 1MB RAM, Cortex-M7 core. 7