Future Generation Computer Systems 113 (2020) 597–606

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Improving approximate neural networks for perception tasks through specialized optimization

Cecilia De la Parra [a,*], Andre Guntoro [a], Akash Kumar [b]

[a] Robert Bosch GmbH, 71272 Renningen, Germany
[b] Chair of Processor Design TU Dresden, 01069 Dresden, Germany

ARTICLE INFO

Article history:
Received 13 January 2020
Received in revised form 11 June 2020
Accepted 13 July 2020
Available online 22 July 2020

Keywords:
Approximate neural networks
Approximate computing
Approximate multipliers
Neural network optimization

ABSTRACT

Approximate Computing has been proven successful in reducing the energy consumption of Deep Neural Networks (DNNs) implemented in embedded systems. For efficient DNN approximation at software and hardware levels, a specialized simulation environment and optimization methodology are required, to reduce execution and optimization times, as well as to maximize energy savings. Traditional frameworks for cross-layer approximate computation of DNNs are generally built only for simulation of convolutional and fully-connected layers, limiting the DNN types to be optimized through approximations. In this work, we present a specialized simulation environment for approximate DNNs, which allows for optimization of several DNN architectures built with more complex DNN layers such as depthwise convolutions and Recurrent Neural Units (RNNs) for time series processing. Low execution time overhead is achieved hereby through efficient GPU acceleration. Additionally, we deliver an analysis of approximate DNN and RNN robustness against quantization noise and different approximation levels. Finally, through specialized approximate retraining, we achieve promising energy savings and negligible accuracy losses with highly complex DNNs for image classification with ImageNet, such as MobileNet, and RNNs for keyword spotting with the Speech Commands Dataset.

1. Introduction

In recent years, several paradigms to optimize the hardware resources of deep learning applications have been proposed. One promising approach is *cross-layer approximate computing*, which involves the combination of approximation techniques at software and hardware level to reduce computational resources in embedded applications.

Approximations at a software level, such as pruning or quantization, have been thoroughly investigated even for highly complex DNNs dedicated to real-life tasks, such as large-scale image recognition or semantic segmentation for automated driving [1]. This has been possible thanks to specialized open source machine-learning frameworks such as Tensorflow Lite [2], or Ristretto [3], which allow a fast exploration of software-oriented approximation techniques.

On the other hand, approximations at hardware level are more difficult to explore, evaluate and optimize, mainly because the simulation of Approximate Computational Units (ACUs) largely increases the runtime in traditional frameworks for approximate computing. Furthermore, specialized cross-layer simulation frameworks for approximate DNNs are generally implemented for CPU, which also increases training and validation times, when compared to exact GPU implementations [4]. In recent works, GPU-based simulation frameworks for cross-layer approximation, such as *Concrete* [5] and *ProxSim* [6], have been proposed to accelerate the optimization of approximate DNN architectures with common neural layers such as convolutional and fully-connected layers. However, more complex operations such as depthwise convolutions [7] or building blocks for Recurrent Neural Networks (RNNs) [8] are not available, restricting the architectures that can be approximated in such specialized frameworks.

In this work, we update ProxSim, presented in [6], by implementing more complex approximate layers, including depthwise convolution, Hadamard product and Gated Recurrent Units (GRUs), as well as more specialized methods for DNN quantization. This variety of layers and quantization methodologies allows us not only to optimize a wide range of approximate DNNs for different applications such as image recognition or keyword spotting, but also to evaluate the accuracy and robustness of such DNNs against quantization noise at different levels.

In summary, we make the following contributions:

- ProxSimV2: A specialized simulation framework for cross-layer DNN approximations such as low bitwidth quantization and approximate hardware, in a variety of layers

* Corresponding author.
E-mail address: cecilia.delaparra@de.bosch.com (C. De la Parra).

https://doi.org/10.1016/j.future.2020.07.031
0167-739X/

that include depthwise convolution and RNNs composed of GRUs, besides traditional 1D/2D convolutional and fully-connected layers.
- Unprecedented exploration of cross-layer approximations in DNN architectures with specialized layers such as depthwise convolutions and GRUs, which perform complex image and speech recognition tasks. With this, we demonstrate the versatility of our proposed simulation framework. The evaluated DNNs include MobileNetV2 [9] for image classification using ImageNet [10] and two RNNs [11] for speech recognition using the Speech Commands Dataset (SCD) [12].
- Specialized approximate optimization for accuracy recovery in lightweight DNNs for image recognition and in RNNs for keyword spotting, reaching energy savings of up to 36% with an accuracy loss of less than 5% compared to the 32 bit Floating Point (FP) accuracy.

2. Related work

2.1. Cross-layer approximations in DNNs

In this subsection, we report a short survey of software and hardware-related approximation methods for DNNs.

- Low bitwidth quantization of DNNs. Also known as *precision scaling* of DNN weights and inputs, quantization leads to a reduction in memory and computational logic. Through different approaches, DNN parameters and activations can be generally scaled to 8 bit integers without accuracy loss [1]. A popular quantization function is the linear quantization [13], performed as follows:

$$q(x) = \text{clip}\left(\text{nint}\left[\left(\frac{X}{\Delta_X}\right), \{-2^{b-1}, 2^{b-1}-1\}\right]\right)\Delta_X = X_q, \tag{1}$$

  where nint denotes the nearest integer function, and the quantization step Δ_X is computed through traditional methods, e.g. maximum absolute value:

$$\Delta_X = \frac{max(|X|)}{2^{\text{bitwidth}-1}} - 1 \tag{2}$$

  For very large and redundant DNNs such as VGG16 [14], linear quantization delivers acceptable results. However, for more challenging DNNs with much less parameters and more complex architectures e.g. MobileNet [7], linear quantization results in very large accuracy degradation. More effective quantization methods, for instance quantized threshold retraining [15] or parameter equalization and error bias correction [16] deliver better results with negligible or no accuracy loss, even when applied to modern, complex DNNs such as MobileNet and MobileNetV2 [9].
- Filter pruning. To reduce the number of DNN kernels without accuracy loss, several methods for kernel or filter pruning have been proposed in the literature, based on different kernel properties. The most popular pruning methods are presented in [17,18]. In [17], authors propose to prune least significant filters based on the weight sum of each kernel. In [18], it is proposed to prune filters that are similar to more significant ones, selected by k-means clustering, until the accuracy decreases beyond a pre-determined threshold.
- ACUs in DNN layers. Combined with low bitwidth quantization, the use of ACUs in the DNN computation leads to higher energy savings when compared to its accurate counterpart. *Partial approximation* methods have been proven effective for reducing the energy consumption in complex DNNs for image classification. Examples of partial DNN approximation are presented in [19] and [20], where approximate multipliers are used in error-resilient neurons selected either by analysis of the error back-propagation or by computing the derivative of the approximation error. Another partial approximation method based on genetic search [4], has been proposed for efficient partial DNN approximation without retraining. Nonetheless, better energy savings can be achieved by *full approximation*, which involves the use of approximate hardware on all DNN neurons. Through adequate optimization techniques, full DNN approximation can also lead to smaller accuracy losses. The viability of applying full approximation to DNNs of different complexity was proved recently in [6,21], and is our focus in this research work. An overview of the characteristics and a graphical representation of partial and full approximation techniques is presented in Fig. 1.

2.2. Simulation frameworks for cross-layer DNN approximation

The recent exploration of cross-layer approximation techniques in DNNs has lead to the introduction of various simulation frameworks for approximate DNN computation. Authors in [22] introduced *AxDNN*, a novel pre-RTL simulation framework for comparison of different approximate strategies such as precision and voltage scaling, approximate multipliers and activation pruning. Specialized frameworks oriented towards optimization of approximate DNNs, such as *ALWANN* [4], deliver efficient optimization of partial DNN approximation. However, a notable disadvantage of these frameworks is the large execution times compared to accurate DNN computation, as these are implemented only for CPU. Another recently proposed cross-layer approximation framework for GPU simulation is *Concrete* [5], based on Caffe [23], which incorporates specialized blocks for simulation of ACUs, such as approximate multipliers, in common DNN layers (convolutional and fully-connected) with small time overhead compared to the accurate counterpart. Lastly, *ProxSim* [6], based on Tensorflow [2], is oriented not only towards GPU-accelerated ACU simulation for partial and full DNN approximation, but also towards efficient approximate DNN optimization to achieve higher energy savings with negligible accuracy losses.

3. Preliminaries

3.1. DNN quantization

3.1.1. Quantization step optimization

In this work, we quantize activations, kernels and biases to 8 bit integers, as in (1). For this, we compute the quantization step Δ using two methods:

- Maximum Absolute Value (MAV), as in (2).
- Minimization of the Propagated Quantization Error (MinPropQE), as presented in [1]:

$$\Delta_X = \underset{\Delta_X}{\text{argmin}} \|\widehat{y_k} - y_k\|_2 \quad , \tag{3}$$

  where $\widehat{y_k}$ represents the accurate output of the kth layer, and y_k refers to the approximate output of the same DNN layer. This method has a larger computational overhead compared to MAV. However, it results in lower DNN accuracy degradation [1].

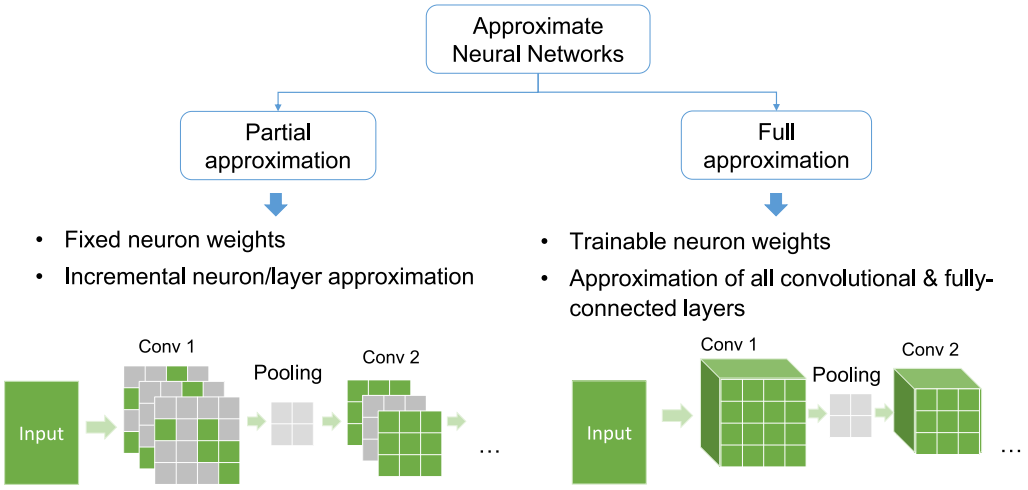


Fig. 1. Paradigm of approximate computing for DNNs. Approximated elements are marked with green, accurate operations are marked with gray. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

3.1.2. Optimization of quantized DNN weights

The use of MAV or MinPropQE for quantization of highly complex DNNs results in large DNN accuracy degradation. Therefore, in this work, additional DNN parameter retraining is performed. We only train the quantized DNN weights and biases; quantization steps are not optimized. Although efficient retraining of the quantization step leads to no accuracy losses [15], we find that our approach is sufficient when combined with approximations at hardware level, e.g. approximate multipliers: As additional DNN retraining is required after incorporating hardware approximations into the DNN computation, the quantization error is further compensated in this stage.

3.2. Approximate multipliers

For exploring the use of ACUs in the DNN computation, we implement 18 different 8 bit approximate multipliers from the following open sources:

- SMApprox [24]. Library with more than 200 approximate multipliers optimized for FPGA.
- EvoApprox [25]. Library with 471 approximate multipliers designed through cartesian genetic programming.

All implemented multipliers were randomly selected from the Pareto front of MRE and energy savings of their corresponding libraries. We select the MRE as primary metric because it is directly proportional to the final DNN accuracy degradation, as we will demonstrate in our experiments.

In Table 1, we report the relative energy savings of all selected multipliers in their corresponding platforms, as reported in [24,25], as well as their Mean Relative Error (MRE), formally computed for every possible output as follows:

$$MRE = \sum_{i=0}^{n}\sum_{j=0}^{n}\frac{|g(i,j)-\widehat{g}(i,j)|}{\mathbf{max}(1,g(i,j))}, \quad (4)$$

where $g(i,j)$ is the accurate multiplication of i and j, $\widehat{g}$ is the approximate counterpart, $n = 2^{\text{bitwidth}} - 1$ and the *max* function is applied to avoid divisions by 0.

4. Simulation of application-specific layers

In ProxSimV2, each layer can be computed either accurately or using ACUs. In this section, we present our analytical approach for the implementation of each approximate DNN layer for GPU acceleration.

Table 1
Approximate multipliers used in this work.

Library	Multiplier	Energy savings [%]	MRE [%]
SMApprox	1000	35.47	0.50
	1100	35.76	2.08
	2200	36.05	3.78
	3300	36.05	3.63
	1110	36.10	3.96
	2220	36.34	6.66
	3330	36.34	6.39
	1111	36.63	9.00
	2222	37.21	16.56
	3333	36.92	13.63
EvoApprox	470	0.90	0.29
	365	6.36	0.83
	42	12.07	1.87
	305	15.81	2.44
	231	22.14	4.94
	10	26.83	5.08
	467	32.62	6.06
	63	42.13	8.59

4.1. Implementation of approximate computational units

In ProxSimV2, two possible implementations of Approximate Computational Units (ACUs) are allowed:

- As Look-up Table (LUT). The behavioral simulation of approximate multipliers (8 bit or smaller) can be loaded as a 128 KB LUT. To avoid shared memory overflow (generally the GPU shared memory has a size of 48 KB), the framework copies this LUT to the GPU global memory. This is then accessed by the shared memory of each GPU block that will compute the Multiply-and-Accumulate (MAC) operations in the corresponding layer, as in Fig. 2, for efficient memory allocation. The advantage of this implementation is that any new ACU of 8 bits or smaller can be directly utilized in the DNN computation without having to re-compile the approximate layers, and moreover, the computation is much faster compared to directly implementing the behavioral code in the DNN layers. Therefore, in this work we focus on this ACU implementation.
- As behavioral code. For simulation of approximate multipliers or adders with operands larger than 8 bits, the behavioral code of such ACUs can be directly integrated into the approximate layers, which should be then re-compiled,

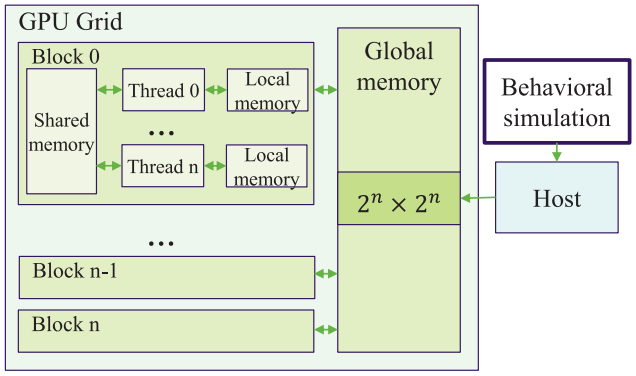


Fig. 2. Loading the behavioral simulation of an ACU in ProxSimV2.

with a performance decrease in the execution time, dependent on the complexity of the corresponding behavioral code.

4.2. Convolutional and fully connected layers

A 2D Convolution takes as input a tensor X of shape (H_{in}, W_{in}, C_{in}), where H is the height, W the width and C the number of channels, and outputs a tensor Y of shape $(H_{out}, W_{out}, C_{out})$. The convolution is parameterized by a tensor K or *kernel*, which contains trainable weights. This kernel has a shape of $(H_K, W_K, C_{in}, C_{out})$. The tensor Y is then computed as follows:

$$Y_{k,l,n} = \sum_{i,j,m} \mathrm{K}_{i,j,m,n} \cdot \mathrm{X}_{k+i-1,l+j-1,m} \quad , \tag{5}$$

where i, j, m, n correspond to the dimensions of K.

A fully connected (FC) layer takes as input a tensor X of shape (H_{in}, W_{in}) and multiplies it with a kernel K of shape (W_{in}, W_{out}) to output a tensor Y of shape (H_{in}, W_{out}), as follows:

$$Y_{i,j} = \sum_{k}^{n} X_{i,k} W_{k,j} \tag{6}$$

We simulate convolutional and FC layers through General Matrix Multiplication (GEMM). The motivation is two-fold: first, it allows for a faster execution of the convolution, and second, it facilitates the implementation of both layers and their corresponding gradient, as we only need to define the derivative of one operation, the GEMM, w.r.t. its inputs.

For computing FC layers, no transformation is needed. For convolutional layers, we transform the kernel K into tensor K' with shape $(H_K \times W_K \times C_{in}, C_{out})$. The input X is then transformed into a 2D tensor X'. The shape of this tensor is dependent not only on its initial shape but also on the stride and padding of the convolution. For example, in case of a convolution with stride of 1 and padding, the tensor X' has a shape of $(W_{in} \times H_{in}, H_K \times W_K \times C_{in})$. Then, an operation equivalent to (6) is performed. The output Y' is finally reshaped to $(H_{out}, W_{out}, C_{out})$.

4.3. Depthwise separable convolution

In modern DNN architectures, to reduce computational costs in 2D convolutional layers, 2D convolutions can be factorized into a depthwise convolution and a 1×1 convolution, as in Fig. 3. This factorization is known as *depthwise separable convolution*, and is used in state-of-the-art DNN models such as MobileNet [7] and MobileNetV2 [9]. In Fig. 3, a graphical description of our approach to implement this operation is presented. Details about the computation of a depthwise convolutional layer in ProxSimV2 are given in the following subsection.

4.3.1. Depthwise convolutions

Depthwise convolutions apply a single filter per input channel, and are computed as in (7) [7].

$$Y_{k,l,m} = \sum_{i,j} \mathrm{K}_{i,j,m} \cdot \mathrm{X}_{k+i-1,l+j-1,m} \tag{7}$$

The factorization of traditional 2D convolutions into depthwise separable convolutions is efficient in reducing the computational costs from:

$W_{in} \times H_{in} \times H_K \times W_K \times C_{in} \times C_{out}$ to:

$W_{in} \times H_{in} \times H_K \times W_K \times C_{in} + C_{in} \times C_{out} \times W_{in} \times H_{in}$ multiplications. This corresponds to a computational decrease of $\frac{1}{C_{out}} + \frac{1}{H_K W_K}$ [7].

Similarly to the implementation of depthwise convolutional layers in Tensorflow, in ProxSimV2, our implementation for approximate depthwise convolution with ACUs dynamically loads input and kernel tiles into the GPU shared memory and only a single channel of each kernel is loaded in each thread. This implementation is characterized by:

- More efficient execution compared to the GEMM implementation. Depthwise convolution is performed just with one filter per input channel, which reduces data transfer times and memory occupation.
- Better data alignment and thread synchronization per block.

4.4. Recurrent neural networks

Recurrent Neural Networks (RNNs) are a type of neural networks with internal states or *memory*, which allows to dynamically process temporal sequences. RNN architectures are therefore commonly used for time series processing tasks such as speech recognition or keyword spotting.

RNNs consist of a hidden state $\mathbf{h}$ and an output $\mathbf{y}$, and receive an input of variable length $\mathbf{x} = (x_1, \ldots, x_n)$ [8]. At time t, $\mathbf{h}_t$ is updated according to (8), where f is a non-linear function, with complexity varying from a single sigmoid operation to a Long-Short-Term Memory (LSTM) unit [26].

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t) \quad , \tag{8}$$

In RNNs, a common non-linear function f which serves as gating mechanism to store previous states is the Gated Recurrent Unit (GRU).

4.4.1. Gated recurrent units

At each time step t, GRUs receive two tensors: an input tensor x_t and a previous state h_{t-1}, and output a tensor h_t. Auxiliary tensors or *gates* are computed inside a GRU to regulate the information flow to the output. Generally, a GRU is composed of an *update* gate z, which works as regulator for input values, and a *reset* gate r, which functions as control for the previous states. As proposed in [8], the output h_t is determined according to the following steps:

Step 1. The reset gate r is computed as in (9), where σ_g is a sigmoid function, W_r, U_r are learned weights and b_r is the reset bias.

$$r = \sigma_g(W_r x + U_r h_{t-1} + b_r) \tag{9}$$

Step 2. The update gate z is computed as in (10), where W_z, U_z are learned weights and b_z the update bias.

$$z = \sigma_g(W_z x + U_z h_{t-1} + b_z) \tag{10}$$

Step 3. The final activation h_t is computed by:

$$h_t = z \odot h_{t-1} + (1 - z) \odot \widetilde{h}_t \quad , \tag{11}$$

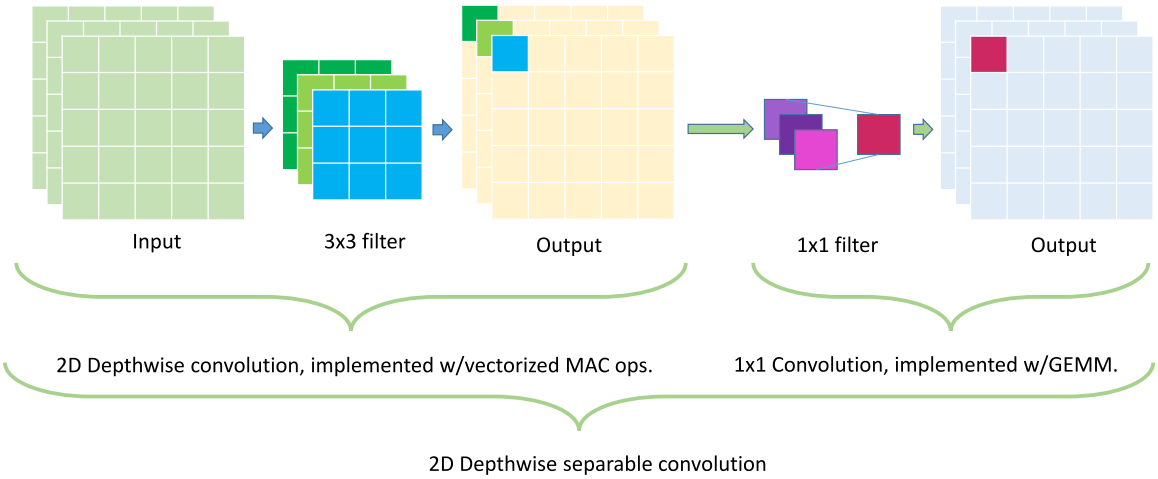


Fig. 3. Implementation of depthwise separable convolutions in ProxSimV2.

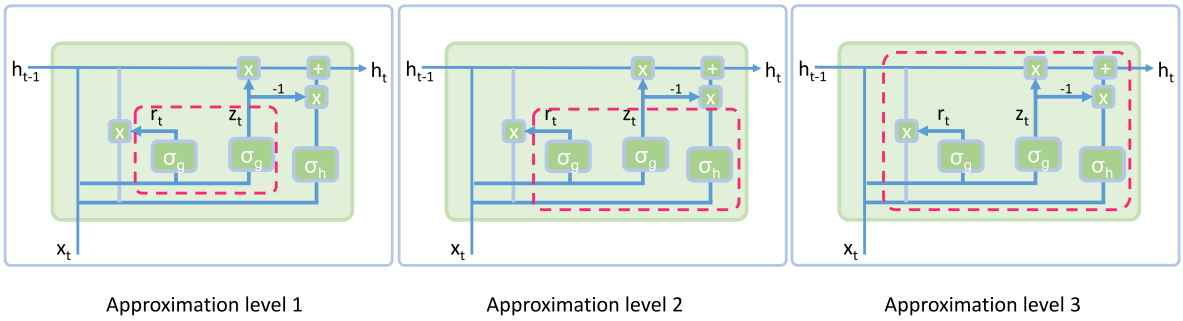


Fig. 4. Proposed approximation levels for GRUs in Recurrent Neural Networks. Elements marked inside the dashed line indicate the approximated operations.

where the hidden state $\widetilde{h}_t$ is determined by (12). In (12), σ_h denotes an activation function, typically the hyperbolic tangent, and $\odot$ denotes the Hadamard product.

$$\widetilde{h}_t = \sigma_h(W_h x + U_h(r \odot h_{t-1}) + b_h) \tag{12}$$

In ProxSimV2, r and z are computed as a single tensor v of size $[n, m]$ as in (14), where:

$$K_{r,z} = \begin{bmatrix} W_r & U_r \\ W_z & U_z \end{bmatrix}, \quad \widetilde{x} = \begin{bmatrix} x \\ h_{t-1} \end{bmatrix}, \quad b_{r,z} = \begin{bmatrix} b_r \\ b_z \end{bmatrix} \tag{13}$$

$$v = \sigma_g(K_{r,z}\widetilde{x} + b_{r,z}) \tag{14}$$

The result is then split accordingly:

$$r = (v_{i,j})_{\substack{i\in n \\ 1\le j\le \frac{m}{2}}}, \quad z = (v_{i,j})_{\substack{i\in n \\ \frac{m}{2}<j\le m}} \tag{15}$$

For the computation of $\widetilde{h}_t$, a similar grouping of variables is performed according to (16). Then, $\widetilde{h}_t$ is computed by (17).

$$K_h = \begin{bmatrix} W_h & U_h \end{bmatrix}, \quad \widehat{x} = \begin{bmatrix} x \\ r \odot h_{t-1} \end{bmatrix} \tag{16}$$

$$\widetilde{h}_t = \sigma_h(K_h \widehat{x} + b_h) \tag{17}$$

To explore the incorporation of quantization and ACUs in the RNN computation, we propose three approximation levels:

Level 1. Approximation of tensor v (14). Only the multiplication between the matrices $K_{r,z}$ and $\widetilde{x}$ is approximated. For this, only tensors x_t, h_{t-1} and $K_{r,z}$ are quantized. If a bias $b_{r,z}$ is used, it is quantized as well. The consequent operations are computed with FP accuracy.

Level 2. Approximation of v and $\widetilde{h}_t$. In this approach, matrix multiplications between $K_{r,z}$ and $\widetilde{x}$, and between K_h and $\widehat{x}$ are approximated. For this, K_h and $\widehat{x}$ are quantized as well. Biases $b_{r,z}$ and b_h are also quantized, if bias addition is used.

Level 3. Full Approximation. All multiplications involved in the computation of h_t are approximated, including Hadamard products. This requires the additional quantization of the update gate z.

The graphical representation of each approximation level in a GRU is depicted in Fig. 4.

5. Approximate DNN retraining

After applying cross-layer approximations to a DNN, the accuracy suffers certain degradation, proportional to the introduced approximation error. Approximate retraining is an effective approach for recovering such accuracy losses [6,21], and therefore we adapt this optimization method to be applied in more complex DNNs, approximated by quantization and ACUs. We perform retraining by stochastic gradient descent algorithms, following the optimization scheme presented [6], and depicted in Fig. 5. In this optimization flow, w_t are the weights at iteration t, Δw_{t+1} is the weight update policy with a learning rate η, and $C(\widetilde{y})$ is the loss function used to train the original DNN. For all DNNs implemented in this work, we use the cross-entropy loss, defined in (18) either for image or speech classification. In (18), p denotes the input label, $\widetilde{y}$ denotes the DNN output and n represents the number of possible classes. During retraining, the loss function

Table 2
Characteristics of evaluated DNNs.

DNN	Dataset	Top1 Acc.	Params. [$\times 10^3$]	MAC Ops. [$\times 10^6$]
MobileNetV2	ImageNet	71.83	3487.80	300
RNN1	Speech Commands	94.24	75.79	1.5
RNN2	Speech Commands	94.35	189.49	3.8

Table 3
Approximate MobileNet V2 — retraining times.

DNN	Time per epoch [s]	Time overhead [%]
8 bit accurate	1250	–
8 bit approximate	3930	314

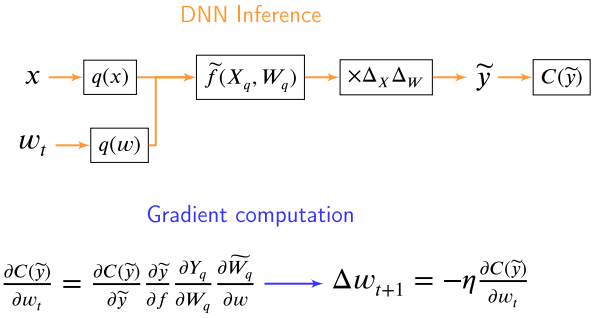


Fig. 5. Approximate DNN inference (forward pass) and gradient computation (backward pass) during retraining in ProxSimV2.

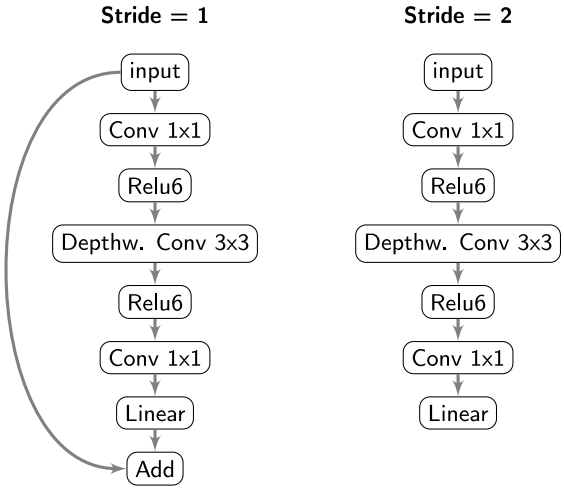


Fig. 6. MobileNetV2: Bottleneck residual blocks.

$C(\widetilde{y})$ is minimized at each iteration by gradient back-propagation.

$$C(x) = -\sum_{i=1}^{n} p_k \log \widetilde{y}_k \tag{18}$$

All approximate layers, as well as some quantization functions such as the nearest integer function, have undefined gradients because of their non-differentiable nature. During back-propagation, to deal with the undefined gradients of these operations, a Straight-Through Estimator (STE) [27] is adapted and implemented for **all** approximate layers as follows:

$$STE \rightarrow \frac{\partial \widetilde{f}}{\partial W_q} \approx \frac{\partial f}{\partial W_q} \quad , \tag{19}$$

where f is the accurate layer and $\widetilde{f}$ is the approximate layer computed with ACUs [6].

6. Evaluation

In this section we present the evaluation results of ProxSimV2 using three different DNNs: MobileNetV2 for image classification and two RNNs for speech recognition. The characteristics of each DNN are given in Table 2, and each DNN is detailed in the followings subsections. For both tasks, the accuracy is computed with the validation dataset.

6.1. MobileNet V2

This is a lightweight DNN for image classification using ImageNet [10]. This network is composed by several bottleneck residual blocks as depicted in Fig. 6. Note that when the convolutional stride is equal to 1, a *shortcut* from input to output is added to improve the gradient propagation across multiple layers [9]. Thanks to this residual shortcut, further optimization techniques for avoiding vanishing gradients, such as alpha regularization [6], are not necessary.

6.1.1. Quantization

To quantize all parameters of MobileNetV2 to 8 bits, we first estimate the corresponding step sizes Δ using MinPropQE. We then obtain an initial 8 bit accuracy of 61.48%. To recover from this accuracy drop of more than 10%, we apply the training scheme presented in Section 5. We retrain the quantized DNN for one epoch with stochastic gradient descent, a batch size of 128 and a learning rate of 1e-4, using 20% of the training dataset. We reach a new 8 bit accuracy of 70.69%. Note that we do not merge the parameters of the batch normalization layers with the ones of the precedent convolutional layers, as these improve the consequent approximate DNN retraining.

6.1.2. Optimized approximation

Due to the complexity of MobileNetV2, we propose two accuracy tolerances: for low approximation levels (multipliers with an MRE smaller than 2%), an accuracy drop of 1% w.r.t. the quantized 8 bit accuracy is allowed, and for larger approximation levels, a maximum accuracy degradation of 5% is proposed. We retrain the MobileNetV2 for 5 epochs with all 18 different approximate multipliers from Table 1, using the gradient computation method from Fig. 5. The results are presented in Fig. 7 (for multipliers with an MRE smaller than 2%), Fig. 8 (for multipliers with an MRE between 2% and 4%), and Fig. 9 (for multipliers with an MRE larger than 4%).

When using ACUs, retraining times are 3.14× longer, independently of the utilized ACU, compared to the 8 bit accurate retraining times. This is shown in Table 3, where the reported retraining times were averaged over 5 epochs using 5 different approximate multipliers. This represents an improvement compared to the first version of ProxSim, which required up to 11× longer inference and retraining times [6].

Regarding DNN accuracy optimization, we observe that after the first epoch, the largest accuracy improvement is reached. Afterwards, the DNN accuracy slowly converges to the maximum possible value, which strongly depends on the approximation error. Furthermore, we observe that our proposed accuracy degradation limits cannot be reached using approximate multipliers with an MRE larger than 4%.

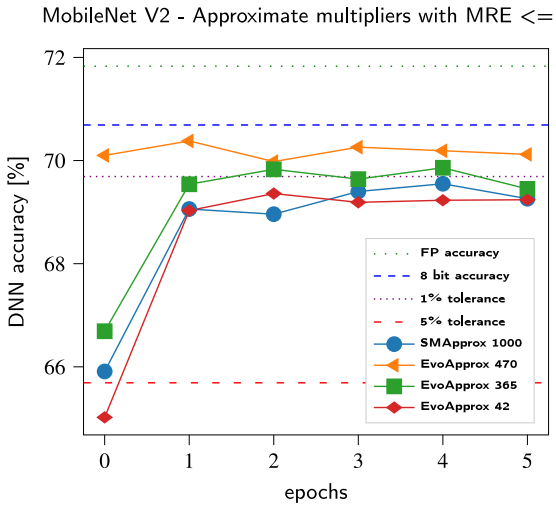


Fig. 7. MobileNetV2 accuracy with low approximation levels, over 5 retraining epochs. Accuracy tolerance of 1% is reached.

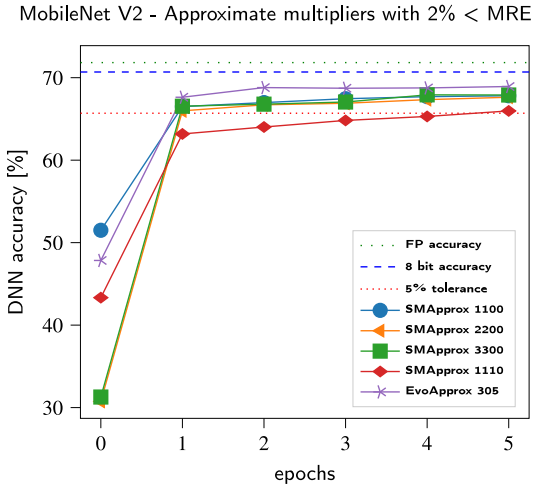


Fig. 8. MobileNetV2 accuracy with medium approximation levels, over 5 retraining epochs. Accuracy tolerance of 5% is reached.

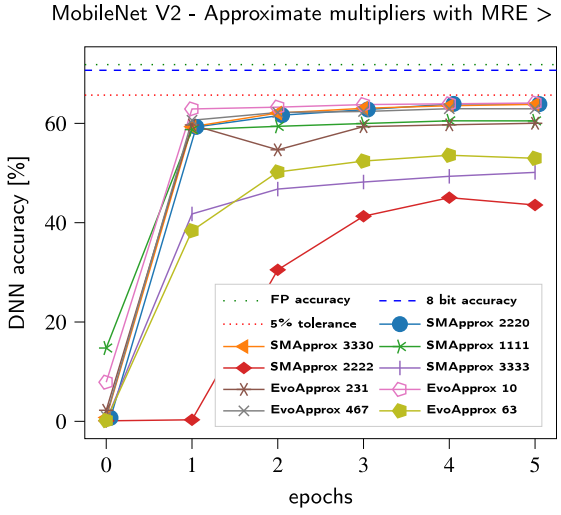


Fig. 9. MobileNetV2 with large approximation levels — Validation accuracy over 5 retraining epochs. The accuracy tolerance is not reached.

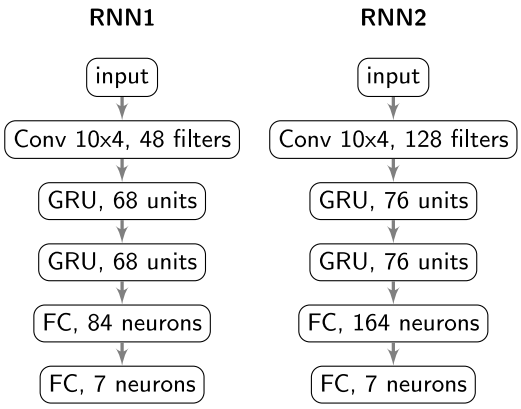


Fig. 10. RNNs for keyword spotting using the Speech Commands Dataset.

Table 4
Relative energy savings in MobileNetV2.

Library	Multiplier	Energy savings [%]	DNN Acc. [%]
SMApprox	1000	35.47	69.55
	1100	35.76	67.81
	2200	36.05	67.63
	3300	36.05	67.62
	1110	**36.10**	**65.97**
EvoApprox	470	0.90	70.38
	365	6.36	69.86
	42	12.07	69.36
	305	**15.81**	**68.93**

6.1.3. Evaluation of energy savings

For this evaluation, we estimate the relative energy savings at the multiplications performed in depthwise convolutions, in traditional convolutions and in FC layers using the energy savings of a single approximate multiplier. In Table 4, all multipliers that are within the proposed DNN accuracy tolerance of 5% are presented. We further include the final accuracy after retraining and the relative energy savings. We are able to reach relative energy savings of 15% using the EvoApprox library (multiplier 305) and of 36% using the SMApprox library (multiplier 1110).

6.2. Recurrent neural networks

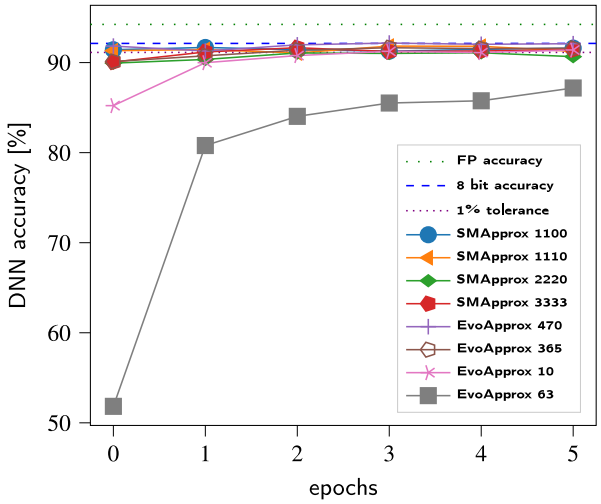
We implement two RNNs from [11] for keyword spotting using the Speech Commands Dataset [12]. These DNNs classify input keywords in 12 different classes, including one class for unknown words. The general architecture of each RNN is depicted in Fig. 10.

6.2.1. Quantization

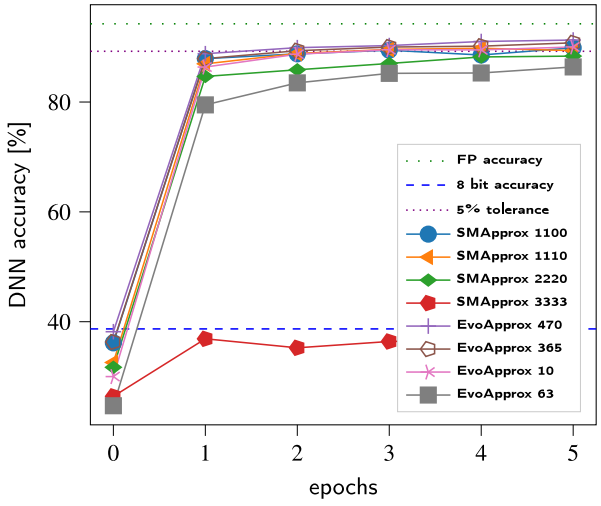
As explained in sub Section 4.4.1, we evaluate the impact of 8 bit quantization at three different approximation levels. For each level, we compute all quantization steps using MAV (2). The quantized accuracy for these three different approximation levels is given in Table 5, as well as the percentage of approximated MAC operations in the GRU blocks. We observe that an approximation level 3 leads to a large accuracy drop of more than 50% for both RNNs, even though the additional number of approximated operations is less than 1% of the total MAC operations, according to Table 5. Note that for level 1 and 2, a better quantized accuracy is reached with RNN1, despite RNN2 having better FP accuracy. This is a result of the utilized quantization method, which is straightforward and does optimize the quantization parameters.

Table 5
RNNs for keyword spotting quantized to different approximation levels.

Approximation level	RNN1		RNN2	
	Accuracy [%]	Approximated MAC ops. [%]	Accuracy [%]	Approximated MAC ops. [%]
FP - no approximation	94.24	0	94.35	0
8 bit level1	92.35	66.31	91.81	66.47
8 bit level2	92.13	99.46	91.83	99.73
8 bit level3	38.69	100	41.28	100

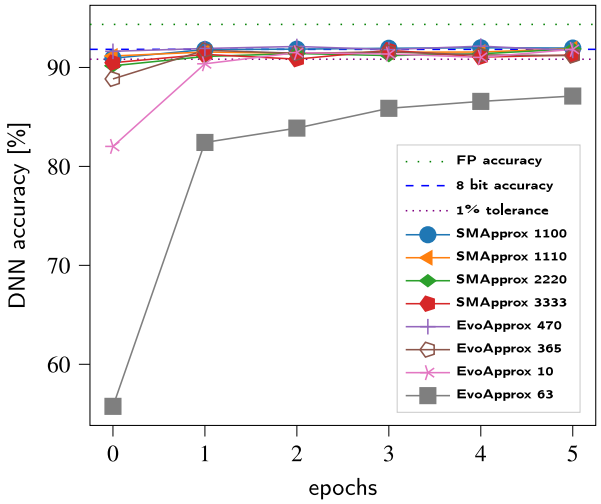


(a) RNN1 - Approximation level 2.

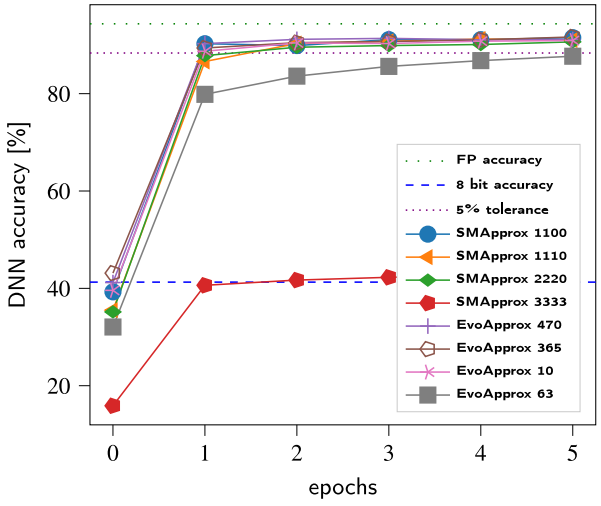


(b) RNN1 - Approximation level 3.

Fig. 11. 8 bit approximate RNN1, retrained over 5 epochs using 8 different approximate multipliers.



(a) RNN2 - Approximation level 2.



(b) RNN2 - Approximation level 3.

Fig. 12. 8 bit approximate RNN2, retrained over 5 epochs using 8 different approximate multipliers.

6.2.2. Optimized approximation

We focus on optimizing the quantized RNNs with approximation levels 2 and 3, to maximize energy savings. The retraining time per epoch for each RNN is reported in Table 6. Note that the time overhead is very small, compared to the computation of the approximate MobileNetV2. This is due to the following reasons:

1. The implemented RNNs have fewer layers, and thus the approximate computation is called in fewer occasions.
2. In ProxSimV2, the time required for building the GRU blocks and the corresponding dynamic RNN calls is more significant than the time needed for computing the approximate operations.

For level 2 approximations, we propose an accuracy tolerance of 1% w.r.t the quantized 8 bit accuracy. In the case of level 3 approximations, we do not use the 8 bit quantized accuracy as reference, because the accuracy drop in the 8 bit RNNs is very large. Instead, we propose a tolerance of 5% w.r.t. the FP accuracy. For each approximation level, we retrain the corresponding RNN with 8 different approximate multipliers from Table 1. We use Adam optimizer [28], which has a better performance for this task, compared to stochastic gradient descent. We retrain for 5 epochs with a learning rate of 1e-3, a batch size of 256 and no background noise, following the retraining methodology presented in Section 5. The results are plotted in Figs. 11 and 12. From these plots, we conclude that the initial quantization error

Table 6
Approximate RNNs — retraining times.

8 bit DNN	Time per epoch [s]	Time overhead [%]
RNN1 accurate	41.29	–
RNN1 approximate	45.05	9.11
RNN2 accurate	41.14	–
RNN2 approximate	45.30	10.11

Table 7
Relative energy savings in RNN1.

Multiplier	Approx. level 2		Approx. level 3	
	Acc. [%]	Savings [%]	Acc. [%]	Savings [%]
1100	91.68	35.55	89.88	35.76
1110	91.83	35.88	89.79	36.10
2220	91.09	36.12	88.36	36.34
3333	**91.65**	**36.70**	36.9	36.92
470	92.17	0.89	91.79	0.90
365	91.65	6.32	90.78	6.36
10	91.40	26.67	**90.03**	**26.83**
63	87.18	41.88	86.39	42.13

Table 8
Relative energy savings in RNN2.

Multiplier	Approx. level 2		Approx. level 3	
	Acc. [%]	Savings [%]	Acc. [%]	Savings [%]
1100	91.99	35.65	91.50	35.76
1110	91.9	35.99	91.23	36.10
2220	91.87	36.23	90.64	36.34
3333	**91.74**	**36.81**	43.58	36.92
470	92.15	0.90	91.36	0.90
365	91.74	6.34	91.65	6.36
10	91.83	26.75	**90.91**	**26.83**
63	87.11	42.00	87.69	42.13

can be almost fully compensated after the approximate retraining, and therefore this training step is useful to compensate not only the approximation error introduced by ACUs but also the quantization error. However, in most cases, we observe a better performance with the same approximate multiplier when using an approximation level 2.

As observed in Figs. 11 and 12, the corresponding accuracy tolerances are reached after just 5 epochs in most cases, except when computing with approximate multipliers with an MRE larger than 10% (SMApprox 3333).

6.2.3. Evaluation of energy savings

We estimate the relative energy savings at multiplications performed in GRUs, in convolutions and in FC layers, based on:

- The energy savings of a single approximate multiplier.
- Percentage of approximated MAC operations according to the approximation level (see Table 5).

In Table 7, we present the final accuracy of RNN1 after retraining, as well as the relative energy savings with each evaluated approximate multiplier. The multipliers SMApprox 3333 and EvoApprox 10 deliver the best trade-off between accuracy and energy savings while maintaining the accuracy tolerance of 5%. The same results are observed with RNN2, where multipliers SMApprox 3333 and EvoApprox 10 deliver the most optimal results regarding accuracy and energy savings, according to Table 8.

Conclusion and outlook

Approximate computing in DNN architectures for different perception tasks can lead to promising computational savings. In this work we present ProxSimV2, a specialized simulation framework for approximate DNNs with complex layers such as depthwise convolutions and recurrent units such as GRUs. Furthermore, we propose an efficient combination of cross-layer approximation techniques such as low bitwidth quantization and the use of approximate multipliers. This is achieved by implementing a variety of approximate DNN layers in our simulation framework, as well as an optimized retraining approach for approximate DNN computation. Through extensive evaluation of state-of-the-art DNNs such as MobileNetV2, and two RNNs for keyword spotting, we demonstrate the versatility of ProxSimV2, and the effectiveness of our proposed optimization method for efficient cross-layer DNN approximation, delivering minimum accuracy loss and maximized energy savings. To the best of our knowledge, we are the first to present such an extended analysis of cross-layer approximations in highly complex DNNs like MobileNetV2 for ImageNet and RNNs for keyword spotting with the Speech Commands Dataset.

CRediT authorship contribution statement

Cecilia De la Parra: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Visualization. **Andre Guntoro:** Conceptualization, Resources, Supervision. **Akash Kumar:** Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was funded by the ECSEL Joint Undertaking project TEMPO, in collaboration with the European Union's Horizon 2020 Research and Innovation Program and National Authorities, under grant agreement No. 826655. The work presented in this article is partly supported by the German Research Foundation (DFG) funded project ReAp (Project Number: 380524764).

References

[1] S. Vogel, J. Springer, A. Guntoro, G. Ascheid, Self-supervised quantization of pre-trained neural networks for multiplierless acceleration, in: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), 2019, http://dx.doi.org/10.23919/DATE.2019.8714901.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, Tensorflow: Large-scale machine learning on heterogeneous systems, 2015, URL: https://www.tensorflow.org/ Software available from tensorflow.org.

[3] P. Gysel, J. Pimentel, M. Motamedi, S. Ghiasi, Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks, IEEE Trans. Neural Netw. Learn. Syst. (2018) http://dx.doi.org/10.1109/TNNLS.2018.2808319.

[4] V. Mrazek, Z. Vasícek, L. Sekanina, M.A. Hanif, M. Shafique, ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining, 2019, CoRR abs/1907.07229. URL: http://arxiv.org/abs/1907.07229 arXiv:1907.07229.

[5] Z. Liu, G. Li, F. Qiao, Q. Wei, P. Jin, X. Liu, H. Yang, Concrete: A per-layer configurable framework for evaluating DNN with approximate operators, in: ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 1552–1556, http://dx.doi.org/10.1109/ICASSP.2019.8682883.

[6] C. De la Parra, A. Guntoro, A. Kumar, Proxsim: gpu-based simulation framework for cross-layer approximate dnn optimization, in: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), 2020, pp. 1193–1198.

[7] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017, CoRR abs/1704.04861. URL: http://arxiv.org/abs/1704.04861.
[8] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, in: Conference on Empirical Methods in Natural Language Processing, 2014, URL: http://arxiv.org/abs/1406.1078.
[9] M. Sandler, A.G. Howard, M. Zhu, A. Zhmoginov, L. Chen, Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation, in: IEEE Conference on Computer Vision and Pattern Recognition, 2018, URL: http://arxiv.org/abs/1801.04381.
[10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, Imagenet large scale visual recognition challenge, Int. J. Comput. Vis. (IJCV) 115 (3) (2015) 211–252, http://dx.doi.org/10.1007/s11263-015-0816-y.
[11] Y. Zhang, N. Suda, L. Lai, V. Chandra, Hello edge: Keyword spotting on microcontrollers, 2017, CoRR abs/1711.07128. URL: http://arxiv.org/abs/1711.07128 arXiv:1711.07128.
[12] P. Warden, Speech commands: A dataset for limited-vocabulary speech recognition, 2018, CoRR abs/1804.03209. URL: http://arxiv.org/abs/1804.03209 arXiv:1804.03209.
[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713.
[14] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015, URL: http://arxiv.org/abs/1409.1556.
[15] S.R. Jain, A. Gural, M. Wu, C. Dick, Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware, 2019, CoRR abs/1903.08066. URL: http://arxiv.org/abs/1903.08066 arXiv:1903.08066.
[16] M. Nagel, M. van Baalen, T. Blankevoort, M. Welling, Data-free quantization through weight equalization and bias correction, in: The IEEE International Conference on Computer Vision (ICCV), 2019, abs/1906.04721.
[17] H. Li, A. Kadav, I. Durdanovic, H. Samet, H.P. Graf, Pruning filters for efficient convnets, in: International Conference on Learning Representations, 2016.
[18] L. Li, Y. Xu, J. Zhu, Filter level pruning based on similar feature extraction for convolutional neural networks, IEICE Trans. Inf. Syst. E101.D (2018) 1203–1206, http://dx.doi.org/10.1587/transinf.2017EDL8248.
[19] S. Venkataramani, A. Ranjan, K. Roy, A. Raghunathan, Axnn: Energy-efficient neuromorphic systems using approximate computing, in: 2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2014, pp. 27–32.
[20] Q. Zhang, T. Wang, Y. Tian, F. Yuan, Q. Xu, Approxann: An approximate computing framework for artificial neural network, in: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), 2015, pp. 701–706.
[21] X. He, L. Ke, W. Lu, G. Yan, X. Zhang, Axtrain: Hardware-oriented neural network training for approximate inference, in: ISLPED'18: International Symposium on Low Power Electronics and Design, Association for Computing Machinery, 2018.
[22] Y. Fan, X. Wu, J. Dong, Z. Qi, Axdnn: towards the cross-layer design of approximate DNNs, in: Asia and South Pacific Design Automation Conference, 2019.
[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional architecture for fast feature embedding, 2014, arXiv preprint arXiv:1408.5093.
[24] S. Ullah, S.S. Murthy, A. Kumar, Smapproxlib: Library of FPGA-based approximate multipliers, in: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1–6, http://dx.doi.org/10.1109/DAC.2018.8465845.
[25] V. Mrazek, R. Hrbacek, Z. Vasicek, L. Sekanina, Evoapproxsb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2017, 2017, pp. 258–261, http://dx.doi.org/10.23919/DATE.2017.7926993.
[26] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (1997) 1735–1780, http://dx.doi.org/10.1162/neco.1997.9.8.1735.
[27] Y. Bengio, N. Léonard, A.C. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation, 2013, ArXiv.
[28] D. Kingma, J. Ba, Adam: A method for stochastic optimization, in: International Conference on Learning Representations, 2014.

Cecilia De la Parra is a Ph.D. candidate at Robert Bosch GmbH, in collaboration with the Chair of Processor Design at the Technological University of Dresden. Previously, she received her Master of Science (M.Sc.) degree in Electronics from the Technical University of Dortmund, Germany, and her Bachelor of Engineering (B.E.) degree in Mechatronics from the Meritorious Autonomous University of Puebla, Mexico (BUAP). Her primary areas of research interest include digital hardware for approximate computing of deep learning applications and optimization of cross-layer approximations in deep neural networks.

Andre Guntoro (M) received his first M.Sc. degree in electrical engineering specialized in Telecommunication from Hochschule Darmstadt in 2002 and his second M.Sc. degree also in electrical engineering specialized in Information and Communication from Technical University Darmstadt in 2003. In 2009 he received his Dr.-Ing. Degree in Microelectronics from Technical University Darmstadt with focus on flexible wavelet processor design and implementation. Since 2011 he joined Robert Bosch GmbH in Research Division, mainly dealing with embedded Machine Learning, where he leads the research activities in Machine Learning and Deep Learning specialized for automotive and IoT devices.

Akash Kumar (SM'13) received the joint Ph.D. degree in electrical engineering and embedded systems from the Eindhoven University of Technology, Eindhoven, The Netherlands, and the National University of Singapore (NUS), Singapore, in 2009. From 2009 to 2015, he was with NUS. He is currently a Professor with Technische Universität Dresden, Dresden, Germany, where he is directing the Chair for Processor Design. His current research interests include the design, analysis, and resource management of low-power and fault-tolerant embedded multiprocessor systems.