Hardware and Software Optimizations for Capsule Networks

Alberto Marchisio, Beatrice Bussolino, Alessio Colucci, Vojtech Mrazek, Muhammad Abdullah Hanif, Maurizio Martina, Guido Masera, and Muhammad Shafique

Alberto Marchisio
Technische Universität Wien (TU Wien), Vienna, Austria
Mailing address: Treitlstrasse 3, 1040 Wien, Austria
e-mail: `alberto.marchisio@tuwien.ac.at`
Telephone number: +43 (1) 58801-18203

Beatrice Bussolino
Politecnico di Torino, Turin, Italy
Mailing address: Corso Castelfidardo, 39, 10129 Torino, Italy
e-mail: `beatrice.bussolino@polito.it`
Telephone number: +39 0110904205

Alessio Colucci
Technische Universität Wien (TU Wien), Vienna, Austria
Mailing address: Treitlstrasse 3, 1040 Wien, Austria
e-mail: `alessio.colucci@tuwien.ac.at`
Telephone number: +43 (1) 58801-18203

Vojtech Mrazek
Brno University of Technology, Brno, Czechia
Mailing address: Božetěchova 2/1, 612 00 Brno, Czechia
e-mail: `mrazek@fit.vutbr.cz`
Telephone number: +420 54114 1348

Muhammad Abdullah Hanif
eBrain Lab, Division of Engineering, New York University Abu Dhabi, UAE
Mailing address: A1-173, Division of Engineering, New York University Abu Dhabi, Saadiyat Island, Abu Dhabi, UAE
e-mail: `mh6117@nyu.edu`
Telephone number: +971-56-5262839

Maurizio Martina
Politecnico di Torino, Turin, Italy
Mailing address: Corso Castelfidardo, 39, 10129 Torino, Italy
e-mail: `maurizio.martina@polito.it`
Telephone number: +39 0110904205

Guido Masera
Politecnico di Torino, Turin, Italy
Mailing address: Corso Castelfidardo, 39, 10129 Torino, Italy
e-mail: `guido.masera@polito.it`
Telephone number: +39 0110904102

Muhammad Shafique
eBrain Lab, Division of Engineering, New York University Abu Dhabi, UAE
Mailing address: A1-173, Division of Engineering, New York University Abu Dhabi, Saadiyat Island, Abu Dhabi, UAE

Abstract
Among advanced Deep Neural Network models, Capsule Networks (CapsNets) have shown high learning and generalization capabilities for advanced tasks. Their capability to learn hierarchical information of features makes them appealing in many applications. However, their compute-intensive nature poses several challenges for their deployment on resource-constrained devices. This chapter provides an optimization flow at the software and at the hardware level for improving the energy-efficiency of the CapsNets' execution.

Keywords
Capsule Networks, Deep Learning, Hardware Accelerator, Software Optimizations, Energy Efficiency

1 Introduction

In recent years, Capsule Networks (CapsNets) have become popular among advanced Machine Learning (ML) models [3], due to their high learning capabilities and improved generalization ability, compared to the traditional Deep Neural Networks (DNNs). The ability to learn hierarchical information of different features (position, orientation, and scaling) in a single capsule allows to achieve high accuracy in machine learning vision applications, e.g., MNIST [15] and Fashion-MNIST [40] classification, as well as effective applicability to other ML application domains, such as speech recognition [39], natural language processing [41], and healthcare [30]. Indeed, CapsNets are able to encapsulate the hierarchical and spatial information of the input features in a closer way to our current understanding of the human brain's functionality. It is shown by recent analyses about the CapsNets' robustness against affine transformations and adversarial attacks [7][27][29], showing that CapsNets are more resilient against such vulnerability threats than traditional DNNs which have similar classification accuracy.

However, the presence of capsules in the layers introduces an additional dimension compared to the matrices of the convolutional and fully-connected layers of the traditional DNNs, which significantly increase the computations and communication workload of the underlying hardware. Therefore, the main challenge in deploying CapsNets is their extremely high complexity. They require intense computations due to the multiplications in the matrices of capsules and the iterative dynamic routing-by-agreement algorithm for learning the cross-coupling between capsules. Figure 1 compares the CapsNet [36] with the LeNet [15] and the AlexNet [14], in terms of their memory footprints and total number of multiply-and-accumulate (MAC) operations needed to perform an inference pass. The MACs/memory ratio is a good

e-mail: muhammad.shafique@nyu.edu
Telephone number: +971-56-5262839

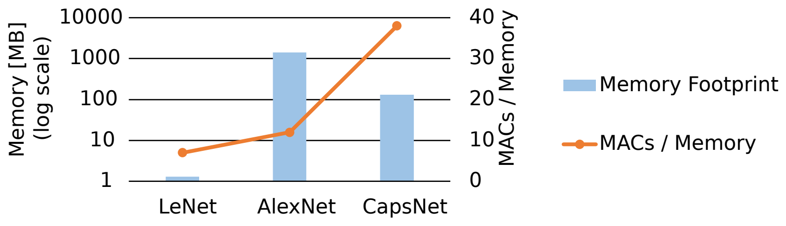

Fig. 1 Comparison of Memory footprint and (Multiply-and-Accumulate operations vs. memory) ratio (MACs/Memory) between the LeNet [15], AlexNet [14], and CapsNet [36] (based on the data presented in [20]).

metric to show the computational complexity of the models, thus demonstrating the higher compute-intensive nature of CapsNets, compared to traditional DNNs.

In this chapter, after discussing the differences between traditional DNNs and CapsNets, and their advanced model architectures, we present the state-of-the-art methodologies and optimization techniques for their efficient execution. An overview of the chapter's content is shown in Figure 2.

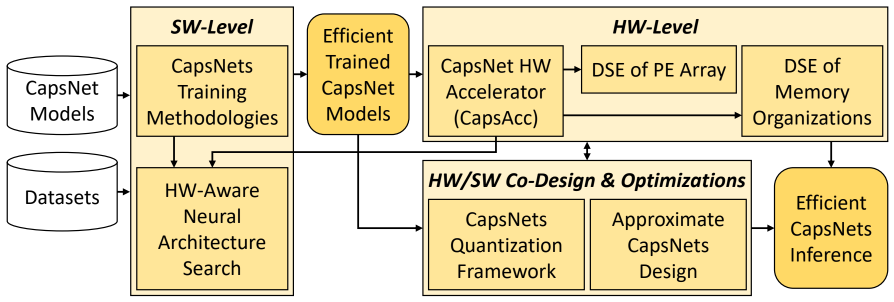
SW-Level
CapsNet Models
Datasets
CapsNets Training Methodologies
HW-Aware Neural Architecture Search
Efficient Trained CapsNet Models
HW-Level
CapsNet HW Accelerator (CapsAcc)
DSE of PE Array
DSE of Memory Organizations
HW/SW Co-Design & Optimizations
CapsNets Quantization Framework
Approximate CapsNets Design
Efficient CapsNets Inference

Fig. 2 Overview of the chapter's content.

2 Traditional DNNs vs. CapsNets

As discussed in [11], among the major drawbacks of traditional DNNs, which are based on convolutional operations, (i) they have too few structural levels, thus they cannot handle different viewpoints of the same object, and (ii) pooling layers are too naive forms of information encoding, since they make DNNs translation-invariant, rather than equivariant. To overcome these problems, the architecture of CapsNets is proposed. The key differences w.r.t. traditional DNNs are summarized in Table 1.

Inspired by the concept of inverse graphics, in [11] the neurons are grouped together into vectors to form the so-called *capsules*. A capsule encodes both the

Table 1 Key differences between traditional DNNs and CapsNets.

	Traditional DNNs	**CapsNets**
Basic Block	Neuron (scalar value)	Capsule (vector)
Activation Function	Rectified Linear Unit (ReLU)	Squash
Inter-Layer Connections	Pooling	Dynamic Routing
Detection Property	Feature Detection	Entity Detection

instantiation parameters (i.e., *pose*, like width, skew, rotation, and other spatial information), and its length (i.e., its Euclidean Norm) is associated with the instantiation probability of the entity. In this way, the CapsNets from the image pixels encode the pose of low-level features, and from the pose of the "parts", it is possible to understand the pose of the "whole", i.e., the high-level entities, to make a better prediction. As activation function, the CapsNets use the *Squash*, which is a multidimensional non-linear function that efficiently fits the prediction vector that forms the capsule.

Moreover, to overcome the problem that DNNs are not invariant to translation, the concept of routing is introduced. The (Max) Pooling operation consists of collecting a group of adjacent neurons and selecting the one with the highest activity, thus discarding the spatial information provided by this group of neurons. For this reason, the pooling layers are responsible for the so-called Picasso problem, in which DNNs classify an image having a nose below the mouth and an eye below the nose as a face, since they lose spatial relationships between features. To replace the pooling layers, an iterative routing procedure to determine the values of the coupling coefficients between a low-level capsule to higher-level capsules is proposed in [36]. It is an iterative process in which the agreements between the capsules of two consecutive layers are measured and updated for a certain number of iterations at runtime during the inference.

3 CapsNet Models and Applications

Hinton et al. [11] first showed the applicability of CapsNets, which adopt the *capsules* as basic blocks and can learn the features of an image in addition to its deformations and viewing conditions. A more detailed explanation of how poses and probabilities are represented and computed to form a CapsNet is described in [36]. A capsule is a vector of neurons, each representing an instantiation parameter of the entity, and the instantiation probability is measured by the length of the vector. To represent such probability in the range $\{0, 1\}$, the *Squash* function is employed. The iterative procedure for computing the coupling coefficients c_{ij} constitutes the Dynamic Routing-by-Agreement in Algorithm 1. The coupling coefficient determines in which amount the lower-level capsule i sends its activation to all the higher-level capsules.

Algorithm 1: Dynamic Routing-by-Agreement in CapsNets.

```
Input: Prediction Votes $\hat{u}_{i|j}$; Number of Iterations $r$; Layer $l$
Output: Activation Vectors $v_j$
for Capsule $i$ in Layer $l$ do
    for Capsule $j$ in Layer $(l+1)$ do
        Logits Initialization: $b_{ij} \leftarrow 0$;
    end
end
for $r$ Iterations do
    for Capsule $i$ in Layer $l$ do
        Softmax: $c_{ij} \leftarrow \texttt{softmax}\ (b_{ij}) = \frac{e^{b_{ij}}}{\sum_k e^{b_{ik}}}$;
    end
    for Capsule $j$ in Layer $(l+1)$ do
        Sum: $s_j \leftarrow \sum_i c_{ij} \cdot \hat{u}_{i|j}$;
    end
    for Capsule $j$ in Layer $(l+1)$ do
        Squash: $v_j \leftarrow \texttt{squash}\ (s_j) = \frac{||s_j||^2}{1+||s_j||^2} \frac{s_j}{||s_j||}$;
    end
    for Capsule $i$ in Layer $l$ do
        for Capsule $j$ in Layer $(l+1)$ do
            Update: $b_{ij} \leftarrow b_{ij} + \hat{u}_{i|j} \cdot v_j$;
        end
    end
end
```

In other words, c_{ij} represents the prior probability that an entity detected by a lower-level capsule i belongs to the higher-level entity of capsule j. To satisfy the property that the sum of these coefficients must be unitary, the *Softmax* function is applied (see line 8 of Algorithm 1). The activation v_j of the capsule j is obtained by applying the *Squash* function to the pre-activation s_j (line 14). The last step consists of updating the logits b_{ij} to be used in the following iteration by computing the agreement through the scalar product between the input prediction votes $\hat{u}_{i|j}$ and the activation v_j (line 18).

The first CapsNet model [36] using the vector capsules and the dynamic routing is shown in Figure 3. A convolutional layer with kernel 9×9, stride 1 and 256 output channels is followed by the PrimaryCaps layer, in which the neurons are grouped into $8D$ vectors, organized in 32 output channels, and form a convolutional capsule layer of kernel size 9×9 and stride 2, using the *Squash* activation function. In the last ClassCaps layer, each of the 10 capsules is dedicated to recognizing the output classes. The *Dynamic Routing* analyzes the features encoded by the 1152 $8D$ capsules of the PrimaryCaps layer to generate the 10 $16D$ activations of the ClassCaps layer. For training purposes, a decoder network (i.e., a cascade of three fully-connected layers) is built for obtaining the image reconstruction, and then employing the *reconstruction loss* along with the *margin loss* (i.e., computed from the instantiation probabilities of the output activations) to form the loss function. Despite being

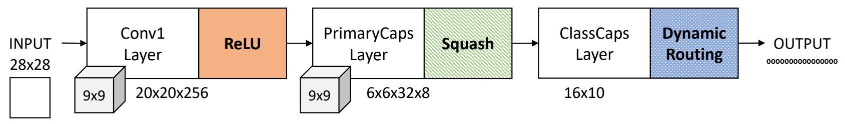

Fig. 3 Architectural model of the vanilla CapsNet [36].

applied mainly to relatively simple tasks, like MNIST [15] and Fashion-MNIST [40] classification, this architecture has been extensively analyzed and studied by the community. Hence, in the following, we consider it as vanilla CapsNet, or simply CapsNet.

A major limitation of this CapsNet is that it is extremely compute-intense and requires many parameters to reach similar performances as traditional DNNs for complex tasks. To overcome these issues, the DeepCaps architecture [35] has been proposed. As shown in Figure 4, besides increasing the depth, the DeepCaps exploits 3D convolutional capsule layers and 3D dynamic routing, thus significantly reducing the number of parameters. Moreover, the decoder employs deconvolutional layers that capture more spatial relationships than the fully-connected layers.

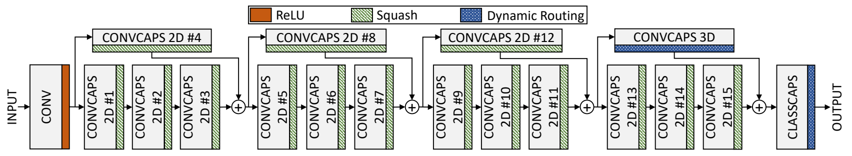

Fig. 4 Architectural model of the DeepCaps [35].

Concurrently, other modifications of the vanilla CapsNets have been proposed. Instead of using vector capsules, Hinton et al. [12] proposed the representation of capsules' inputs and outputs as matrices and replaced the dynamic routing-by-agreement with the expectation-maximization (EM) algorithm. The EM routing is a clustering process based on the Gaussian mixture model. Compared to the dynamic routing it improves the sensitivity of small coupling differences for values close to 1, but implies higher computational time and complexity. Inspired by recent research advancements on transformers, Choi et al. [4] proposed the attention routing and capsule activation, while Hahn et al. [8] proposed self-routing CapsNets, in which the values of the coupling coefficients are learned during training. Other promising CapsNet architectures introduced different variants of the routing algorithm, including the inverted dot-product (IDP) attention routing [38], the self-attention routing [28], and the straight-through (ST) attentive routing [2]. The key features of the different routing methods are summarized in Table 2.

Table 2 An overview of the most common versions of the routing algorithm for CapsNets.

Routing Method	Reference	Short Description	Benefits	Potential Limitations
Dynamic Routing	[36]	Coupling agreement computed based on cosine similarity, normalized by squash	Dynamic update with multiple iterations	Low sensitivity for values close to 1
EM Routing	[12]	Clustering-based algorithm where the agreements follow a Gaussian distribution	Overcomes the dynamic routing limitation	High computational expensive
Attention Routing	[4]	Coupling coefficients computed with an attention module	Only forward computations	Use high-complex capsule activations
Self-Routing	[8]	Each capsule is routed independently by its subordinate routing network (non-iterative procedure)	Competitive robustness performance and viewpoint generalization	Scalability issues and high computational cost
IDP Attention Routing	[38]	Coefficients computed through an IDP mechanism between high-level capsule states and low-level capsule votes	Fast computations using a concurrent iterative procedure	Memory-intense and complex backbone needed
Self-Attention Routing	[28]	Capsules between subsequent layers are routed with a self-attention mechanism	Competitive performance with few parameters	Low accuracy for complex tasks
ST Attentive Routing	[2]	Connection between high-level and low-level capsules based on binary decision with a straight-through estimator	Differentiability of computations and high accuracy on ImageNet	High number of parameters and high training time

4 Efficient CapsNets Training Methodologies

State-of-the-art learning policies for traditional DNNs are designed to tune the learning rate and batch size values during different training epochs to achieve high accuracy and fast training time. Compared to the baseline policy in which the learning rate is exponentially decreasing and the batch size is kept constant during training, the most popular learning policies include:

- *One-Cycle Policy [37]:* This technique applies a single cycle of learning rate variation. The training is conducted in three phases. In phase-1 (for the first 45% of training epochs), the learning rate is increased from a minimum to a maximum value. In phase-2 (for other 45% of epochs), the learning rate is decreased in a symmetric way. In phase-3 (for the last 10% of epochs, the learning rate is further decreased.
- *Warm Restarts [17]:* The learning rate is initialized to a maximum value and cyclically decreased with cosine annealing to a minimum value and then reset to its maximum with a step function. The cyclic repetition of this process emulates a warm restart that allows the model to traverse several saddle points and local minima to obtain fast convergence.
- *Adaptive Batch Size (AdaBatch) [6]:* Since small batch sizes typically imply the convergence in few training epochs, while large batch sizes guarantee high computational efficiency due to high parallel processing in GPU clusters, a good trade-off consists of adaptively increasing the batch size during training. Starting with a small batch size allows fast convergence in early epochs, and progressively increasing the batch size at selected epochs improves the performance due to the larger workload available per processor in later epochs.

The *FasTrCaps* framework [18] combines the above-discussed techniques and other optimization strategies for efficiently training the CapsNets. As shown in Figure 5, the methodology is composed of three steps. First, the learning rate policies are tailored and applied to CapsNets. Then, the adaptive batch size is selected. Among the explored learning rate policies, the *Warm Restarts* guarantees the most promising results in terms of accuracy, while the *AdaBatch* provides a good trade-off to obtain fast convergence. Combining these two techniques, a novel learning policy called **WarmAdaBatch** is designed.

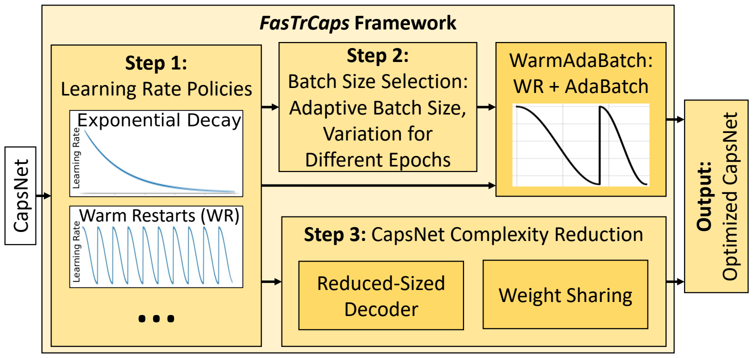
FasTrCaps Framework
CapsNet
Step 1: Learning Rate Policies
Exponential Decay
Learning Rate
Warm Restarts (WR)
Learning Rate
• • •
Step 2: Batch Size Selection: Adaptive Batch Size, Variation for Different Epochs
WarmAdaBatch: WR + AdaBatch
Step 3: CapsNet Complexity Reduction
Reduced-Sized Decoder
Weight Sharing
Output: Optimized CapsNet

Fig. 5 Overview of the *FasTrCaps* framework's functionality [18].

The *WarmAdaBatch* method, shown in Algorithm 2, is a hybrid learning policy that combines the variations of the learning rate and the batch size. For the first three epochs, the batch size is set to 1, then it is increased to 16 for the remaining training epochs. The first cycle of the warm restarts policy is done during the first three epochs, and the second one during the remaining training epochs. For completeness, the procedures describing the *AdaBatch* and *Warm Restarts* methods are shown in lines 15 to 25 and 26 to 34 of Algorithm 2, respectively.

In the third step, the computational complexity is reduced by performing two optimizations. (i) The size of the CapsNet decoder is reduced by around 5%, maintaining only the 1×16 inputs linked to the capsule that outputs the highest probability. (ii) The weights between the PrimaryCaps and ClassCaps layers are shared by having a single weight tensor associated with all the 8D vectors inside each 6×6 capsule, achieving more than 15% reduction in the total number of parameters.

The evaluation is conducted on the CapsNet [36] for the MNIST [15] and Fashion-MNIST [40] datasets. Table 3 shows the key results employing the *WarmAdaBatch* and *Weight Sharing*. The accuracy values are computed by averaging 5 training runs, each of them lasting for 30 epochs. For the MNIST dataset, the accuracy drop caused by the *Weight Sharing* is compensated by the *WarmAdaBatch*. The combination of both techniques results in a slightly higher accuracy (99.38% vs. 99.37% of the baseline), with fewer training epochs. For the Fashion-MNIST dataset,

Algorithm 2: WarmAdaBatch training method for CapsNets.

```
Procedure WarmAdaBatch($lr_{min}$, $lr_{max}$, $MaxEpoch$, $MaxStep$)
    $T_{curr} \leftarrow 0$;
    for $Epoch \in \{1, ..., MaxEpoch\}$ do // Batch size update
        AdaBatch(4,Epoch);
        if $Epoch \leq 3$ then
            $T_i \leftarrow 3 * 60,000$; // Steps in 3 epochs with batch size 1
        else
            $T_i \leftarrow 27 * 3,750$; // Steps in 27 epochs with batch size 16
        end
        for $Step \in \{1, ..., MaxStep\}$ do // Learning Rate update
            $T_{curr} \leftarrow WR(lr_{min}, lr_{max}, T_{curr}, T_i)$;
        end
    end
end
Procedure AdaBatch($P$, $CurrentEpoch$)
    if $CurrentEpoch \leq 3$ then
        $BatchSize \leftarrow 1$;
    else if $4 \leq CurrentEpoch \leq 8$ then
        $BatchSize \leftarrow 2^P$;
    else if $9 \leq CurrentEpoch \leq 13$ then
        $BatchSize \leftarrow 2^{P+1}$;
    else
        $BatchSize \leftarrow 2^{P+2}$;
    end
end
Procedure WarmRestarts($lr_{min}$, $lr_{max}$, $T_{curr}$, $T_i$)
    $lr \leftarrow lr_{min} + \frac{1}{2}(lr_{max} - lr_{min})\left(1 + \cos \pi \frac{T_{curr}}{T_i}\right)$; // Learning rate update
    if $T_{curr} = T_i$ then // Warm Restart after $T_i$ training steps
        $T_{curr} \leftarrow 0$;
    else // Current step update
        $T_{curr} \leftarrow T_{curr} + 1$;
    end
    return $T_{curr}$;
end
```

despite requiring slightly more training epochs than the baseline, the combination of *WarmAdaBatch* and *Weight Sharing* shows comparable accuracy w.r.t. the baseline.

5 Hardware Architectures for Accelerating CapsNets' Inference

For deploying CapsNets-based systems at the edge, it is crucial to minimize the power/energy consumption and maximize the performance. The unique operations involving capsules, Squash, and the dynamic routing make the existing architectures for accelerating traditional DNNs unsuitable or inefficient. Therefore, specialized architectures and dataflows need to be designed and tailored for CapsNets. A

Table 3 Accuracy results obtained with CapsNet for the MNIST and Fashion-MNIST datasets, applying different solutions using the *FasTrCaps* framework [18].

Accuracy		**Epochs to reach max accuracy**		**Learning Rate and Batch Size**	**Weight Sharing**
FashionMNIST	*MNIST*	*FashionMNIST*	*MNIST*		
90.99%	99.37%	17	29	Baseline	No
91.47%	99.45%	27	8	WarmAdaBatch	No
90.47%	99.26%	17	26	Baseline	Yes
90.67%	99.38%	20	11	WarmAdaBatch	Yes

performance analysis of the CapsNets execution is beneficial for identifying the bottlenecks. The rest of the section discusses CapsAcc [22], which is the first accelerator architecture for CapsNets, the FEECA methodology [26] for exploring the design space of the computational units of CapsNets accelerators, and the DESCNet methodology [25] for conducting design-space exploration of CapsNets memory units.

5.1 CapsNets Execution on GPUs and their Bottlenecks

To understand how the CapsNets' inference is performed, we perform a comprehensive analysis to measure the performance of the PyTorch-based CapsNet [36] implementation for the MNIST dataset on the NVIDIA GeForce RTX 2080 Ti GPU. Figure 6a shows the execution time breakdown for each layer. The ClassCaps layer is the bottleneck since it is around $10 - 20\times$ slower than the previous layers, despite counting fewer parameters than the PrimaryCaps layer. To obtain more detailed results, the execution time for each operation of the dynamic routing in the ClassCaps layer is analyzed and reported in Figure 6b. From the results, it is clear that the most compute-intensive operation is the *Squash* inside the ClassCaps layer. Hence, these analyses motivate the design of the hardware architecture and dataflow that efficiently computes the *Squash* and dynamic routing.

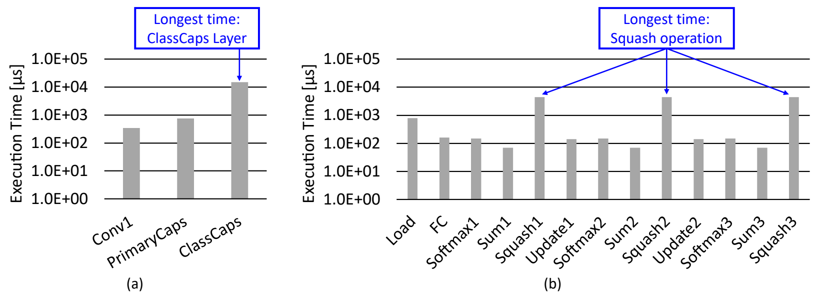

Fig. 6 Execution time breakdown of the CapsNet [18] on the Nvidia GeForce RTX 2080 Ti GPU. (a) Layer-wise breakdown. (b) Operations in the dynamic routing.

5.2 The CapsAcc Accelerator

The top-level architecture of the CapsAcc accelerator is shown in Figure 7a. The core of the computations is conducted in the Processing Element (PE) Array for efficiently computing the operations between matrices. The inputs are propagated towards the output of the PE array both horizontally (data) and vertically (weight and partial sum). Each PE, shown in Figure 7b, consists of a Multiply-and-Accumulate (MAC) unit, and four registers to synchronize the weight and data transfers, and partial sum results at each clock cycle. The *Weight$_2$ Register* allows to store and use the same weight on different data for convolutional layer operations, while for fully-connected operations, only one cycle latency overhead is introduced without affecting the throughput. The 8-bit data is multiplied with the 8-bit weight to form a 16-bit product, which is accumulated with the previous partial sum using 25 bits to avoid overflow.

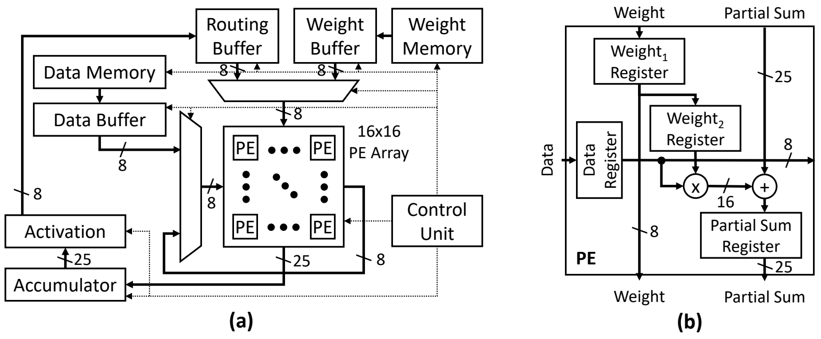

Fig. 7 Hardware architecture of the CapsAcc accelerator [22]. (a) Complete accelerator architecture. (b) Architecture of a Processing Element (PE).

The resulting partial sums coming from the PE array are stored in an accumulator, followed by the activation unit, which can selectively perform the ReLU, Squash, normalization, or Softmax. More details on the implementations of these units are discussed in [22]. At each stage of the inference process, the control unit generates the control signals for all the components of the accelerator architecture.

The memory is organized such that all the weights for each operation are stored in the on-chip weight memory, while the input data, which correspond to the pixel intensities of the image, are stored in the on-chip data memory. The data buffer and weight buffer work as cushions for the interface with the PE array at a high access rate. Moreover, the accumulator unit contains a buffer for storing the output partial sums, and the routing buffer stores the coefficients of the dynamic routing. The multiplexers at the input of the PE array are used to handle the different dataflows for each operation. To maximize the data reuse, the routing buffer stores the values of the coupling coefficients across different iterations of the dynamic routing.

The complete CapsAcc architecture has been implemented in RTL (VHDL) and synthesized in a 45-nm CMOS technology node using the ASIC design flow with the Synopsys Design Compiler (see the parameters in Table 4). The gate-level simulations conducted using Mentor ModelSim are conducted to obtain precise area, power, and performance of our design. Table 5 reports the detailed area and power breakdown, showing that the contributions are dominated by the buffers.

Table 4 Parameters of the synthesized CapsAcc.

Tech. node [nm]	45
Voltage [V]	1
Area [mm^2]	2.60
Power [mW]	427.44
Clk Freq. [MHz]	250
Bit width	8
On-Chip Mem. [MB]	8
Area [mm^2]	2.60
Power [mW]	427.44

Table 5 Area and power, for the different components of the CapsAcc architecture.

Component	**Area [µm^2]**	**Power [mW]**
PE Array	42 867	112.31
Accumulator	32 641	47.57
Activation	29 027	2.21
Data Buffer	136 222	199.31
Routing Buffer	32 598	47.56
Weight Buffer	11 961	17.46
Other	4 330	1.10

5.3 FEECA Methodology

The CapsAcc architecture represents a specific design solution for accelerating the CapsNets' inference. For systematically exploring the design space of CapsNets accelerators, the FEECA methodology [26] can be employed. As shown in Figure 8, its goal is to find Pareto-optimal sets of architectural parameters of the CapsNet accelerator to achieve good trade-offs between the design objectives, which are area, energy, and performance in terms of inference delay. Given a set of configurable parameters, the area, power, and energy of the PE arrays and the memories are computed through Synopsys Design Compiler and CACTI [16], respectively. The evaluation of each candidate solution is based on the model extraction of the CapsAcc accelerator [22] for the CapsNet [36]. For searching in the space of the solutions, the straightforward approach is to use brute force, but for reducing the exploration time, a heuristic multi-objective optimization approach based on the principles of the NSGA-II genetic algorithm [5] is used. It is an iterative algorithm that combines crossover and mutation to explore the solutions, which are progressively selected based on the Pareto frontiers.

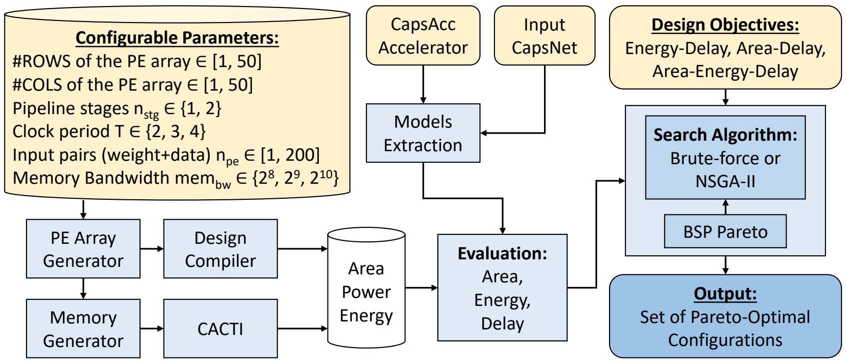
Configurable Parameters:
#ROWS of the PE array ∈ [1, 50]
#COLS of the PE array ∈ [1, 50]
Pipeline stages n_stg ∈ {1, 2}
Clock period T ∈ {2, 3, 4}
Input pairs (weight+data) n_pe ∈ [1, 200]
Memory Bandwidth mem_bw ∈ {2^8, 2^9, 2^10}
CapsAcc Accelerator
Input CapsNet
Design Objectives: Energy-Delay, Area-Delay, Area-Energy-Delay
Models Extraction
Search Algorithm: Brute-force or NSGA-II
BSP Pareto
PE Array Generator
Design Compiler
Memory Generator
CACTI
Area Power Energy
Evaluation: Area, Energy, Delay
Output: Set of Pareto-Optimal Configurations

Fig. 8 Overview of the FEECA methodology [26] for the design-space exploration of CapsNets accelerators.

Figure 9 shows the set of Pareto-optimal solutions that form the output of the FEECA methodology. For visualization purposes, the results are visualized in 2D plots, where each couple of two objectives is combined into products, which are energy × delay (EDP), area × delay (ADP), and energy × area (EAP), respectively. By reducing the dimension of the space, only a smaller number of solutions remain in the Pareto-frontiers, which are represented by the gray lines. The highlighted lowest-delay solution (i.e., the fastest architecture) lays on the Pareto-frontier only in the last two plots, i.e., the ADP vs. energy and EAP vs. delay trade-offs, while it is not Pareto-optimal for the other case of EDP vs. area.

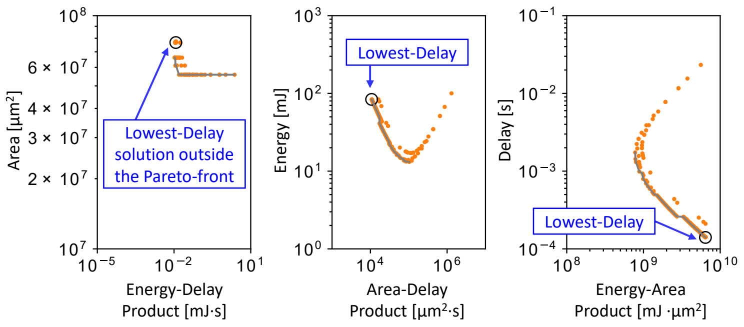

Fig. 9 Pareto-optimal set of configurations of CapsNet accelerator generated with the FEECA methodology [26].

5.4 Memory Design and Management for CapsNets

Motivated by the results previously discussed in Table 5 that shows that the on-chip area and power consumption of the complete CapsNet accelerator are dominated by the memory buffers, a specialized scratchpad memory organization (DESCNet [25]) is designed. The architectural view in Figure 10 shows that the DESCNet memory is connected to the off-chip memory and the CapsNet accelerator through dedicated bus lines. The scratchpad memory is partitioned into banks, where each bank consists of equally sized sectors. All sectors with the same index across different banks are connected through a power-gating circuitry implemented with sleep transistors to support an efficient sector-level power-management control at the cost of a certain area overhead. The application-driven memory power management unit determines the appropriate control signals for the sleep transistors.

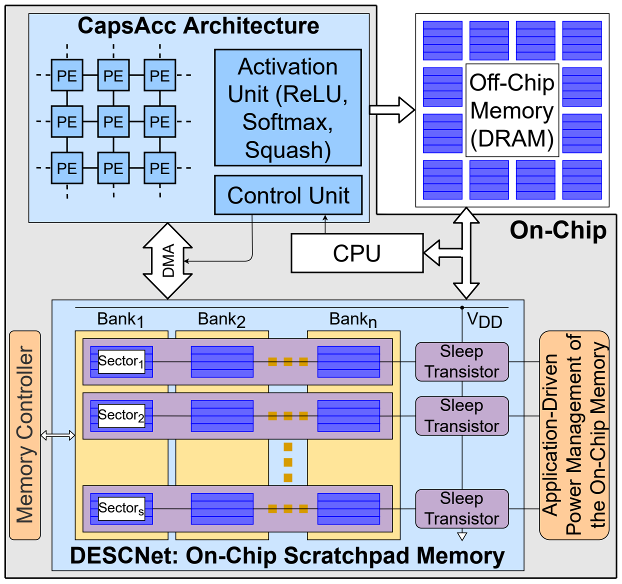

Fig. 10 Architectural view of the complete accelerator for CapsNets' inference, with a focus on the DESCNet scratchpad memory [25].

This memory model can be generalized for different memory organizations supporting different sizes and levels of parallelism, including multiport memories. Toward this, the following design options are analyzed:

1. *Shared Multiport Memory (SMP):* A shared on-chip memory with three ports for parallelized access to the weights, input data, and the accumulator's storage.
2. *Separated Memory (SEP):* Weights, input data, and partial sums are stored in three separate on-chip memories.

3. *Hybrid Memory (HY):* A combination of the other two design options, i.e., an SMP coupled with a SEP memory.

Given the different memory organizations, sizes, number of banks, and sectors per bank, a design space exploration is conducted. The flow of the DESCNet design space exploration methodology is shown in Figure 11. For each design option, the values of memory organization (i.e., the size and number of banks and sectors), the energy consumption, and area are generated. Given as input the CapsNet model and hardware accelerator, the memory usage and memory accesses for each operation of the CapsNet inference are extracted. Then, the analyzer collects the statistics for each configuration, and the design space is explored. The memory area and energy consumption estimations, with and without the power-gating option, are conducted through the CACTI-P tool [16].

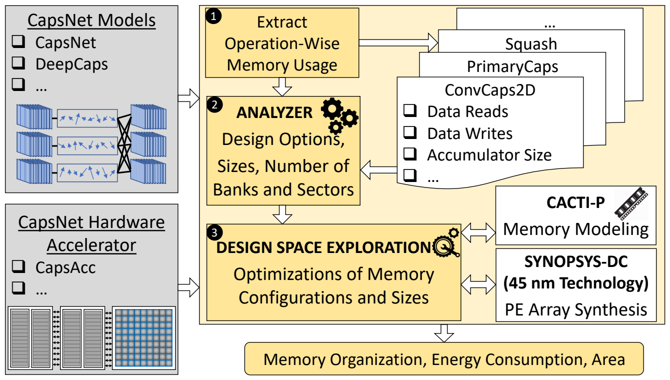
CapsNet Models
CapsNet
DeepCaps
...
CapsNet Hardware Accelerator
CapsAcc
...
1 Extract Operation-Wise Memory Usage
2 ANALYZER Design Options, Sizes, Number of Banks and Sectors
3 DESIGN SPACE EXPLORATION Optimizations of Memory Configurations and Sizes
...
Squash
PrimaryCaps
ConvCaps2D
Data Reads
Data Writes
Accumulator Size
...
CACTI-P Memory Modeling
SYNOPSYS-DC (45 nm Technology) PE Array Synthesis
Memory Organization, Energy Consumption, Area

Fig. 11 DESCNet [25] methodology and toolflow for conducting the design space exploration.

The different memory architectural options has been evaluated for area and energy consumption. Figure 12a shows the 15 233 different DESCNet architectural configurations for the CapsNet [36] on the MNIST dataset [15], while Figure 12b shows the 215 693 configurations for the DeepCaps [35] on the CIFAR10 dataset [13].

For each design option (SMP, SEP, and HY) and its corresponding version with power-gating (with the suffix -PG), the Pareto-optimal solutions with the lowest energy are highlighted. Note that while SEP, SEP-PG, and HY-PG belong to the Pareto-frontier, HY, SMP, and SMP-PG are dominated by other memory configurations. Using these optimizations, it is possible to achieve up to 80% energy saving and up to 47% area saving compared to the memory organization of CapsAcc [22].

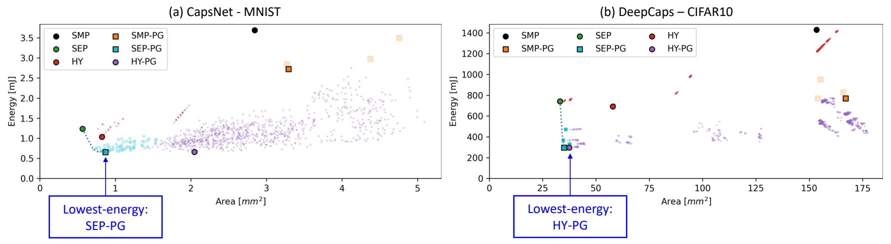

Fig. 12 Design space exploration of the DESCNet memory configurations [25]. (a) Results for the CapsNet on the MNIST dataset. (b) Results for the DeepCaps on the CIFAR10 dataset.

6 Lightweight Optimizations for CapsNets Inference

To further ease the deployment of CapsNets at the edge, other lightweight optimizations can be conducted. A reduction of the wordlength of the weights and activations of a CapsNet for computing the inference not only lightens the memory storage requirements, but might also have a significant impact on the energy consumption of the computational units. Moreover, the current trends in approximate computing can be leveraged to approximate the most compute-intensive operations, such as the multiplications, achieve energy-efficient CapsNet hardware architectures, and enable design-/run-time energy-quality trade-offs.

6.1 Quantization of CapsNets with the Q-CapsNets Framework

Despite the considerable energy savings, having a too short wordlength implies lowering the accuracy of the CapsNets, which is typically an undesired outcome from the end-user perspective. To find efficient trade-offs between the memory footprint, the energy consumption, and the classification accuracy, The Q-CapsNets framewok [19] is applied. As shown in Figure 13, it explores different layer-wise and operation-wise arithmetic precisions for obtaining the quantized version of a given CapsNet. It tackles in particular the dynamic routing, which is a peculiar feature of the CapsNets involving complex and computationally expensive operations performed iteratively, with a significant impact on the energy consumption. Given the CapsNet model architecture, together with the training and test dataset, and a set of user constraints such as the accuracy tolerance, the memory budget, and the rounding schemes, the Q-CapsNets framework progressively reduces the numerical precision of the data (e.g., weights and activations) in the CapsNet inference, aiming at satisfying both requirements on accuracy and memory usage.

A step-by-step description of the framework is the following:

1) **Layer-Uniform Quantization (weights + activations)**: All the weights and activations are converted to fixed-point arithmetic, with 1-bit integer part, and Q_w-

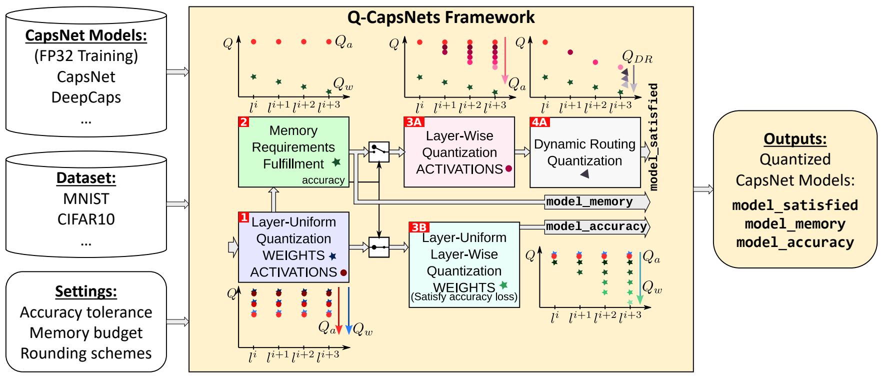
CapsNet Models:
(FP32 Training)
CapsNet
DeepCaps
...
Dataset:
MNIST
CIFAR10
...
Settings:
Accuracy tolerance
Memory budget
Rounding schemes
Q-CapsNets Framework
1 Layer-Uniform Quantization WEIGHTS ACTIVATIONS
2 Memory Requirements Fulfillment accuracy
3A Layer-Wise Quantization ACTIVATIONS
3B Layer-Uniform Layer-Wise Quantization WEIGHTS (Satisfy accuracy loss)
4A Dynamic Routing Quantization
model_memory
model_accuracy
model_satisfied
Outputs:
Quantized CapsNet Models:
model_satisfied
model_memory
model_accuracy

Fig. 13 Flow of the Q-CapsNets framework [19].

bit and Q_a-bit fractional part, respectively. Afterward, their precision is further reduced in a uniform way (e.g., $Q_w = Q_a$).

2) **Memory Requirements Fulfillment**: In this stage, only the CapsNet weights are quantized. Since the perturbations to weights in the final layers can be more costly than perturbations in the earlier layers, for each layer l its respective Q_w is set such that $(Q_w)_{l+1} = (Q_w)_l - 1$. Having defined these conditions, the `model_memory` is obtained, having the correct Q_w computed as the maximum integer value such that the sum of the weight memory occupied by each layer is lower than the memory budget. Afterward, if the accuracy of the `model_memory` is higher than the target accuracy, it continues to (3A) for further quantization steps. Otherwise, it jumps to (3B).

3A) **Layer-Wise quantization of activations**: The activations are quantized in a layer-wise fashion. Progressively, each layer of the CapsNet (except the first one) is selected, and Q_a of the current layer is lowered until the minimum value for which the accuracy remains higher than the target accuracy. This step repeats iteratively until the Q_a for the last layer is set. Afterward, it continues to (4A).

3B) **Layer-Uniform ad Layer-Wise Quantization of Weights**: Starting from the outcome of step (1), only the weights are quantized, first in a uniform and then in a layer-wise manner (as in step 3A) until reaching the target accuracy. The resulting CapsNet model (`model_accuracy`) is returned as another output of the framework.

4A) **Dynamic Routing Quantization**: Due to the computationally expensive operations, such as *Squash* and *Softmax*, the wordlength of the dynamic routing tensors may be different as compared to other layers of the CapsNet. Therefore, a specialized quantization process is performed in this step, in which the operators of the dynamic routing can be quantized more than the other activations (i.e., with a wordlength lower than Q_a, which we call Q_{DR}). The quantized CapsNet model generated at the end of this step is denoted as `model_satisfied`.

Some key results of the Q-CapsNets framework implemented in PyTorch [33] and tested on the CapsNet [36] and DeepCaps [35] models for MNIST [15], Fashion-MNIST [40] and CIFAR10 [13] datasets are shown in Table 6. An efficient `model_satisfied` for the CapsNet on the MNIST dataset achieves 4.87× weight memory reduction with only 0.09% accuracy loss compared to the baseline. Even larger memory reductions (up to 7.51× weight and 6.45× activation memory reduction) can be obtained for the DeepCaps, with less than 0.15% accuracy loss. Note that the wordlength for the dynamic routing operations can be reduced up to 3 or 4 bits with very little accuracy loss compared to the full-precision model. Such an outcome is attributed to the fact that the operations of the involved coefficients (along with Squash and Softmax) are updated dynamically, thereby tolerating a more aggressive quantization compared to previous layers.

Table 6 Q-CapsNet's accuracy results, weight (W) memory and activation (A) memory reduction for the CapsNets and for the DeepCaps on the MNIST, Fashion-MNIST and CIFAR10 datasets [19].

Model	Dataset	Accuracy	W mem reduction	A mem reduction
CapsNet	MNIST	99.58%	4.87×	2.67×
CapsNet	MNIST	99.49%	2.02×	2.74×
CapsNet	FMNIST	92.76%	4.11×	2.49×
CapsNet	FMNIST	78.26%	6.69×	2.46×
DeepCaps	MNIST	99.55%	7.51×	4.00×
DeepCaps	MNIST	99.60%	4.59×	6.45×
DeepCaps	FMNIST	94.93%	6.40×	3.20×
DeepCaps	FMNIST	94.92%	4.59×	4.57×
DeepCaps	CIFAR10	91.11%	6.15×	2.50×
DeepCaps	CIFAR10	91.18%	3.71×	3.34×

6.2 Approximations for energy-Efficient CapsNets with the ReD-CaNe Methodology

Approximation errors in hardware have been extensively employed to trade off accuracy for efficiency. Several recent works [9][10][21][32] have studied the resiliency of traditional DNNs to approximations, showing the possibility to achieve high energy savings with minimal accuracy loss. For CapsNets, the resiliency analysis is conducted through the ReD-CaNe methodology [24]. Since the estimated energy consumption of the multipliers counts for 96% of the total energy share of the computational path of the DeepCaps inference, the approximate multipliers are targeted. The ReD-CaNe methodology, shown in Figure 14, provides useful strategies

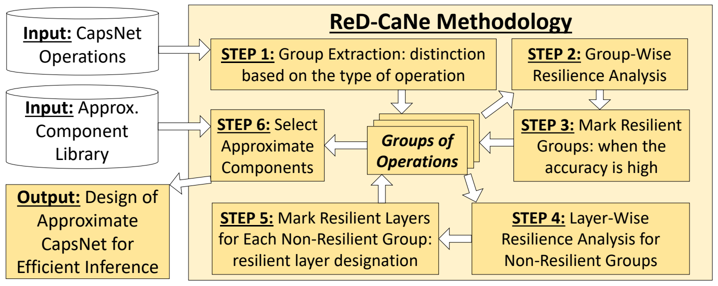
Input: CapsNet Operations
Input: Approx. Component Library
Output: Design of Approximate CapsNet for Efficient Inference
ReD-CaNe Methodology
STEP 1: Group Extraction: distinction based on the type of operation
STEP 2: Group-Wise Resilience Analysis
STEP 3: Mark Resilient Groups: when the accuracy is high
STEP 4: Layer-Wise Resilience Analysis for Non-Resilient Groups
STEP 5: Mark Resilient Layers for Each Non-Resilient Group: resilient layer designation
STEP 6: Select Approximate Components
Groups of Operations

Fig. 14 Flow of the ReD-CaNe methodology [24].

for deploying energy-efficient inference, showing the potential to design and apply approximations to specific CapsNets layers and operations (i.e., the more resilient ones) without sacrificing the accuracy much.

Since the profiling of the error distributions produced by approximate multipliers of the EvoApproxSb library [31] shows that the majority of the components have a Gaussian-like distribution of the arithmetic error, the approximations can be modeled as a noise injection of a certain magnitude and average. A step-by-step description of the ReD-CaNe methodology is the following:

1. **Group Extraction:** The operations of the CapsNet inference are divided into groups based on the type of operation involved (e.g., MAC, activation function, Softmax, or logits update). This step generates the *Groups*.
2. **Group-Wise Resilience Analysis:** The test accuracy drop is monitored by injecting noise into different groups.
3. **Mark Resilient Groups:** Based on the results of the analysis performed in Step 2, the more resilient groups are marked. After this step, there are two categories of Groups, the *Resilient* and *Non-Resilient* ones.
4. **Layer-Wise Resilience Analysis for Non-Resilient Groups:** For each non-resilient group, the test accuracy drop is monitored by injecting noise at each layer.
5. **Mark Resilient Layers for Each Non-Resilient Group:** Based on the results of the analysis performed in Step 4, the more resilient layers are marked.
6. **Select Approximate Components:** For each operation, the approximate components from a given library are selected based on the resilience measured as the noise magnitude.

As a case study, the detailed results applied to the DeepCaps for the CIFAR10 dataset are shown in Figure 15. In the experiments for Step 2, the same noise is injected into every operation within a group, while maintaining the other groups accurate. As shown in Figure 15a, the Softmax and the logits update groups are more resilient than MAC outputs and activations, because the DeepCaps accuracy starts to decrease with a correspondent lower noise magnitude. Note that, for low noise magnitude, the accuracy slightly increases due to regularization, with a similar effect as the dropout. Figure 15b shows the resiliency analysis of each layer of the

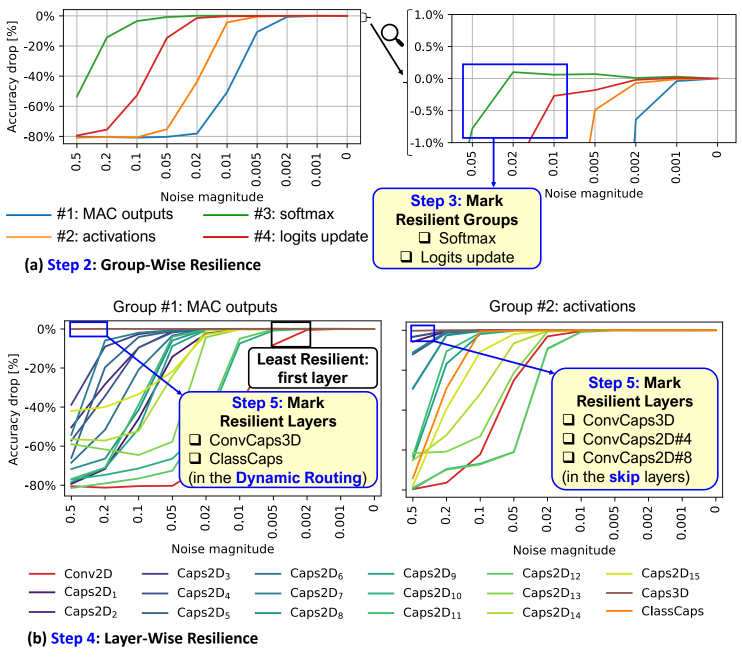

Fig. 15 ReD-CaNe methodology applied to the DeepCaps for the CIFAR10 dataset [24].

non-resilient groups (i.e., MAC outputs and activations). While the first convolutional layer is the least resilient, the ClassCaps layer and ConvCaps3D layer are the most resilient ones. Since the latter is the only convolutional layer that employs the dynamic routing algorithm, the higher resilience is attributed to the dynamic routing iterations, in which the coefficients are updated dynamically at runtime, thus they can adapt to the approximation errors. Moreover, among the activations, the most resilient layers are the ConvCaps3D, ConvCaps2D#4, and ConvCaps2D#8, which are the layers in the skip connection path of the DeepCaps.

7 HW-Aware Neural Architecture Search for DNNs and CapsNets

Manually designing CapsNets is a tedious job and incurs challenging efforts. Neural Architecture Search (NAS) methods automatically select the best model for a given application task. Moreover, hardware-aware NAS methodologies are employed to find efficient trade-offs between the accuracy of the models and the efficiency of their execution on specialized hardware accelerators in IoT-Edge/CPS devices. Toward this, the NASCaps framework [23] jointly optimizes the network accuracy and its corresponding hardware efficiency, expressed as energy, memory, and latency of a given hardware accelerator. It supports both the traditional DNN layers and the CapsNets layers and operations, including the dynamic routing.

The overall functionality and workflow of the NASCaps framework are shown in Figure 16. As input, the framework receives the configuration of the underlying hardware accelerator and a given dataset used for training, as well as a collection of the possible types of layers that can be used to form different candidate DNNs and CapsNets. The evolutionary search is based on the principles of the multi-objective iterative genetic NSGA-II algorithm [5]. Analytical models of the execution of different types of layers and operations in the hardware architecture are developed to estimate the hardware metrics during the design space exploration quickly. To further reduce the exploration time, the accuracy of each candidate DNN/CapsNet is evaluated after a limited number of training epochs, in which the number of epochs is selected based on the Pearson correlation coefficient [34] w.r.t. the fully-trained networks. Afterward, the Pareto-frontiers relative to accuracy, energy consumption, latency, and memory footprint are generated to proceed to the next iteration. At the end of this selection procedure, the Pareto-optimal DNN solutions are fully-trained to make an exact accuracy evaluation.

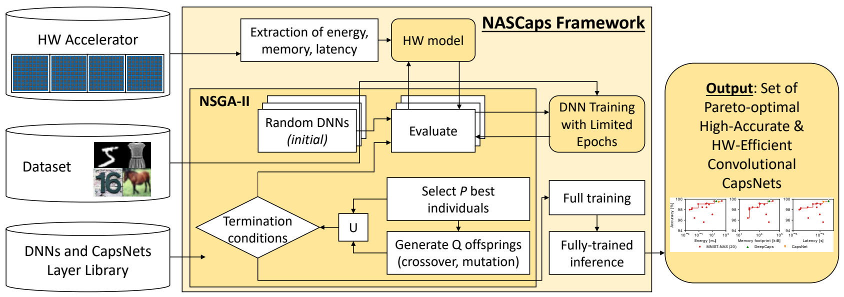
HW Accelerator
Dataset
DNNs and CapsNets Layer Library
NASCaps Framework
Extraction of energy, memory, latency
HW model
NSGA-II
Random DNNs (initial)
Evaluate
DNN Training with Limited Epochs
Select P best individuals
Termination conditions
U
Generate Q offsprings (crossover, mutation)
Full training
Fully-trained inference
Output: Set of Pareto-optimal High-Accurate & HW-Efficient Convolutional CapsNets

Fig. 16 Overview of the NASCaps framework's functionality [23].

The NASCaps framework [23] has been implemented with the TensorFlow library [1] running on GPU-HPC computing nodes equipped with four NVIDIA Tesla V100-SXM2 GPUs. A set of case-study experiments for the CIFAR10 dataset [13] running on the CapsAcc architecture [22] is shown in Figure 17. The maximum number of generations for the genetic algorithm is set to 20, but a maximum time-out of 24 hours has been imposed, thus stopping the algorithm at the 14$^{\text{th}}$ iteration. The candidate solutions in the earlier generations in Figure 17a are quite inefficient, while the networks found in the latest generations outperform the manually-designed CapsNet and DeepCaps. Note that at this stage of partially-trained networks (i.e., after 10 epochs), some solutions exhibit > 20% accuracy improvements compared to the DeepCaps. The Pareto-optimal solutions have been fully-trained for 300 epochs, and the results are shown in Figure 17b. The highlighted solution achieves an accuracy of 85.99% (about 1% accuracy drop), while reducing the energy consumption by

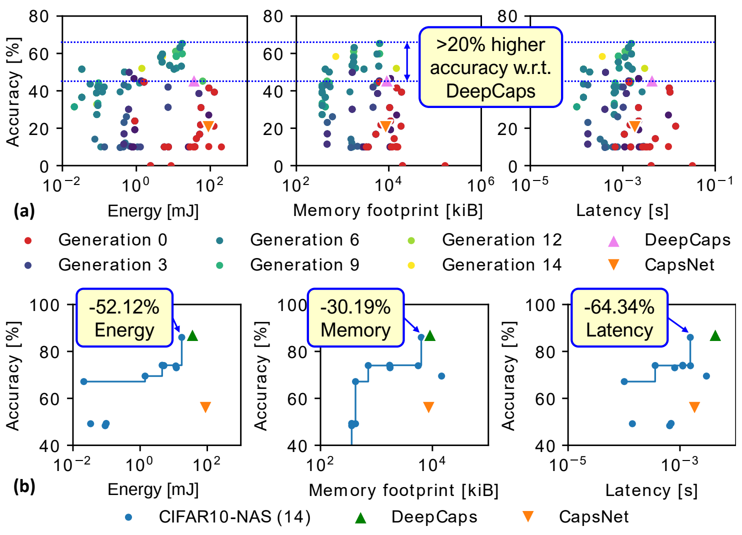

Fig. 17 NASCaps framework applied to the CIFAR10 dataset, showing the trade-offs between accuracy, energy, memory footprint, and latency [23]. (a) Partially-trained results. (b) Fully-trained results.

52.12%, the memory footprint by 30.19%, and the latency by 64.34%, compared to the DeepCaps inference run on the CapsAcc accelerator.

8 Conclusion

Capsule Networks offer high learning capabilities, which results in high accuracy in several applications and high robustness against the vulnerability threats which involve spatial transformations. However, compared to traditional DNNs, the capsule layers introduce an additional dimension, and the iterative dynamic routing makes CapsNets high compute-intensive. In this chapter, several optimization techniques and frameworks tailored for CapsNets have been proposed. The FasTrCaps framework employs state-of-the-art learning policies and reduces the complexity of CapsNets for efficient training. CapsNets hardware architectures based on the CapsAcc inference accelerator are explored with the FEECA methodology, while the DESCNet methodology optimizes the memory organizations based on the CapsNets' workload. The Q-CapsNets framework produces lightweight quantized CapsNets, and the ReD-CaNe methodology further reduces the energy consumption by employing approximate multipliers. Moreover, the NASCaps framework enables hardware-aware capsule-based neural architecture search for jointly optimizing accuracy, memory, energy, and latency, thus enabling CapsNets deployment in resource-constrained edge devices.

Acknowledgements This work has been supported in part by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P.A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: K. Keeton, T. Roscoe (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, pp. 265–283. USENIX Association (2016). URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`
2. Ahmed, K., Torresani, L.: Star-caps: Capsule networks with straight-through attentive routing. In: H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E.B. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 9098–9107 (2019). URL `https://proceedings.neurips.cc/paper/2019/hash/cf040fc71060367913e81ac1eb050aea-Abstract.html`
3. Capra, M., Bussolino, B., Marchisio, A., Shafique, M., Masera, G., Martina, M.: An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. Future Internet **12**(7), 113 (2020). DOI 10.3390/fi12070113. URL `https://doi.org/10.3390/fi12070113`
4. Choi, J., Seo, H., Im, S., Kang, M.: Attention routing between capsules. In: 2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019, pp. 1981–1989. IEEE (2019). DOI 10.1109/ICCVW.2019.00247. URL `https://doi.org/10.1109/ICCVW.2019.00247`
5. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002). DOI 10.1109/4235.996017. URL `https://doi.org/10.1109/4235.996017`
6. Devarakonda, A., Naumov, M., Garland, M.: Adabatch: Adaptive batch sizes for training deep neural networks. CoRR **abs/1712.02029** (2017). URL `http://arxiv.org/abs/1712.02029`
7. Gu, J., Tresp, V.: Improving the robustness of capsule networks to image affine transformations. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020, pp. 7283–7291. Computer Vision Foundation / IEEE (2020). DOI 10.1109/CVPR42600.2020.00731. URL `https://openaccess.thecvf.com/content\_CVPR\_2020/html/Gu\_Improving\_the\_Robustness\_of\_Capsule\_Networks\_to\_Image\_Affine\_Transformations\_CVPR\_2020\_paper.html`
8. Hahn, T., Pyeon, M., Kim, G.: Self-routing capsule networks. In: H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E.B. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 7656–7665 (2019). URL `https://proceedings.neurips.cc/paper/2019/hash/e46bc064f8e92ac2c404b9871b2a4ef2-Abstract.html`
9. Hanif, M.A., Hafiz, R., Shafique, M.: Error resilience analysis for systematically employing approximate computing in convolutional neural networks. In: J. Madsen, A.K. Coskun (eds.) 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, pp. 913–916. IEEE (2018). DOI 10.23919/DATE.2018.8342139. URL `https://doi.org/10.23919/DATE.2018.8342139`
10. Hanif, M.A., Marchisio, A., Arif, T., Hafiz, R., Rehman, S., Shafique, M.: X-dnns: Systematic cross-layer approximations for energy-efficient deep neural networks. J. Low Power Electron.

14(4), 520–534 (2018). DOI 10.1166/jolpe.2018.1575. URL https://doi.org/10.1166/jolpe.2018.1575

11. Hinton, G.E., Krizhevsky, A., Wang, S.D.: Transforming auto-encoders. In: T. Honkela, W. Duch, M.A. Girolami, S. Kaski (eds.) Artificial Neural Networks and Machine Learning - ICANN 2011 - 21st International Conference on Artificial Neural Networks, Espoo, Finland, June 14-17, 2011, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 6791, pp. 44–51. Springer (2011). DOI 10.1007/978-3-642-21735-7_6. URL https://doi.org/10.1007/978-3-642-21735-7_6
12. Hinton, G.E., Sabour, S., Frosst, N.: Matrix capsules with EM routing. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net (2018). URL https://openreview.net/forum?id=HJWLfGWRb
13. Krizhevsky, A.: Learning multiple layers of features from tiny images. University of Toronto (2012)
14. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: P.L. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (eds.) Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pp. 1106–1114 (2012). URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html
15. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998). DOI 10.1109/5.726791. URL https://doi.org/10.1109/5.726791
16. Li, S., Chen, K., Ahn, J.H., Brockman, J.B., Jouppi, N.P.: CACTI-P: architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In: J.R. Phillips, A.J. Hu, H. Graeb (eds.) 2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011, pp. 694–701. IEEE Computer Society (2011). DOI 10.1109/ICCAD.2011.6105405. URL https://doi.org/10.1109/ICCAD.2011.6105405
17. Loshchilov, I., Hutter, F.: SGDR: stochastic gradient descent with warm restarts. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net (2017). URL https://openreview.net/forum?id=Skq89Scxx
18. Marchisio, A., Bussolino, B., Colucci, A., Hanif, M.A., Martina, M., Masera, G., Shafique, M.: Fastrcaps: An integrated framework for fast yet accurate training of capsule networks. In: 2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020, pp. 1–8. IEEE (2020). DOI 10.1109/IJCNN48605.2020.9207533. URL https://doi.org/10.1109/IJCNN48605.2020.9207533
19. Marchisio, A., Bussolino, B., Colucci, A., Martina, M., Masera, G., Shafique, M.: Q-capsnets: A specialized framework for quantizing capsule networks. In: 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020, pp. 1–6. IEEE (2020). DOI 10.1109/DAC18072.2020.9218746. URL https://doi.org/10.1109/DAC18072.2020.9218746
20. Marchisio, A., Bussolino, B., Salvati, E., Martina, M., Masera, G., Shafique, M.: Enabling capsule networks at the edge through approximate softmax and squash operations. In: 2022 IEEE/ACM International Symposium on Low Power Electronics and Design, ISLPED 2022, Boston, MA, USA, August 1-3, 2022, pp. 1–6. IEEE (2022)
21. Marchisio, A., Hanif, M.A., Khalid, F., Plastiras, G., Kyrkou, C., Theocharides, T., Shafique, M.: Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges. In: 2019 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2019, Miami, FL, USA, July 15-17, 2019, pp. 553–559. IEEE (2019). DOI 10.1109/ISVLSI.2019.00105. URL https://doi.org/10.1109/ISVLSI.2019.00105
22. Marchisio, A., Hanif, M.A., Shafique, M.: Capsacc: An efficient hardware accelerator for capsulenets with data reuse. In: J. Teich, F. Fummi (eds.) Design, Automation & Test in Europe

Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019, pp. 964–967. IEEE (2019). DOI 10.23919/DATE.2019.8714922. URL `https://doi.org/10.23919/DATE.2019.8714922`

23. Marchisio, A., Massa, A., Mrazek, V., Bussolino, B., Martina, M., Shafique, M.: Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020, pp. 114:1–114:9. IEEE (2020). DOI 10.1145/3400302.3415731. URL `https://doi.org/10.1145/3400302.3415731`
24. Marchisio, A., Mrazek, V., Hanif, M.A., Shafique, M.: Red-cane: A systematic methodology for resilience analysis and design of capsule networks under approximations. In: 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020, pp. 1205–1210. IEEE (2020). DOI 10.23919/DATE48585.2020.9116393. URL `https://doi.org/10.23919/DATE48585.2020.9116393`
25. Marchisio, A., Mrazek, V., Hanif, M.A., Shafique, M.: Descnet: Developing efficient scratchpad memories for capsule network hardware. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(9), 1768–1781 (2021). DOI 10.1109/TCAD.2020.3030610. URL `https://doi.org/10.1109/TCAD.2020.3030610`
26. Marchisio, A., Mrazek, V., Hanif, M.A., Shafique, M.: FEECA: design space exploration for low-latency and energy-efficient capsule network accelerators. IEEE Trans. Very Large Scale Integr. Syst. **29**(4), 716–729 (2021). DOI 10.1109/TVLSI.2021.3059518. URL `https://doi.org/10.1109/TVLSI.2021.3059518`
27. Marchisio, A., Nanfa, G., Khalid, F., Hanif, M.A., Martina, M., Shafique, M.: Capsattacks: Robust and imperceptible adversarial attacks on capsule networks. CoRR **abs/1901.09878** (2019). URL `http://arxiv.org/abs/1901.09878`
28. Mazzia, V., Salvetti, F., Chiaberge, M.: Efficient-capsnet: Capsule network with self-attention routing. CoRR **abs/2101.12491** (2021). URL `https://arxiv.org/abs/2101.12491`
29. Michels, F., Uelwer, T., Upschulte, E., Harmeling, S.: On the vulnerability of capsule networks to adversarial attacks. CoRR **abs/1906.03612** (2019). URL `http://arxiv.org/abs/1906.03612`
30. Monday, H.N., Li, J., Nneji, G.U., Nahar, S., Hossin, M.A., Jackson, J.: Covid-19 pneumonia classification based on neurowavelet capsule network. Healthcare **10**(3) (2022). DOI 10.3390/healthcare10030422. URL `https://www.mdpi.com/2227-9032/10/3/422`
31. Mrazek, V., Hrbacek, R., Vasícek, Z., Sekanina, L.: Evoapproxsb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: D. Atienza, G.D. Natale (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017, pp. 258–261. IEEE (2017). DOI 10.23919/DATE.2017.7926993. URL `https://doi.org/10.23919/DATE.2017.7926993`
32. Mrazek, V., Vasícek, Z., Sekanina, L., Hanif, M.A., Shafique, M.: ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining. In: D.Z. Pan (ed.) Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019, pp. 1–8. ACM (2019). DOI 10.1109/ICCAD45719.2019.8942068. URL `https://doi.org/10.1109/ICCAD45719.2019.8942068`
33. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E.Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E.B. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 8024–8035 (2019). URL `https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html`

34. Pearson, K., for National Eugenics, G.L.: "Note on Regression and Inheritance in the Case of Two Parents". Proceedings of the Royal Society. Royal Society (1895). URL `https://books.google.it/books?id=xst6GwAACAAJ`
35. Rajasegaran, J., Jayasundara, V., Jayasekara, S., Jayasekara, H., Seneviratne, S., Rodrigo, R.: Deepcaps: Going deeper with capsule networks. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019, pp. 10725–10733. Computer Vision Foundation / IEEE (2019). DOI 10.1109/CVPR.2019.01098. URL `http://openaccess.thecvf.com/content\_CVPR\_2019/html/Rajasegaran\_DeepCaps\_Going\_Deeper\_With\_Capsule\_Networks\_CVPR\_2019\_paper.html`
36. Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: I. Guyon, U. von Luxburg, S. Bengio, H.M. Wallach, R. Fergus, S.V.N. Vishwanathan, R. Garnett (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp. 3856–3866 (2017). URL `https://proceedings.neurips.cc/paper/2017/hash/2cad8fa47bbef282badbb8de5374b894-Abstract.html`
37. Smith, L.N., Topin, N.: Super-convergence: Very fast training of residual networks using large learning rates. CoRR **abs/1708.07120** (2017). URL `http://arxiv.org/abs/1708.07120`
38. Tsai, Y.H., Srivastava, N., Goh, H., Salakhutdinov, R.: Capsules with inverted dot-product attention routing. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020). URL `https://openreview.net/forum?id=HJe6uANtwH`
39. Wu, X., Cao, Y., Lu, H., Liu, S., Wang, D., Wu, Z., Liu, X., Meng, H.: Speech emotion recognition using sequential capsule networks. IEEE ACM Trans. Audio Speech Lang. Process. **29**, 3280–3291 (2021). DOI 10.1109/TASLP.2021.3120586. URL `https://doi.org/10.1109/TASLP.2021.3120586`
40. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. CoRR **abs/1708.07747** (2017). URL `http://arxiv.org/abs/1708.07747`
41. Zhao, W., Peng, H., Eger, S., Cambria, E., Yang, M.: Towards scalable and reliable capsule networks for challenging NLP applications. In: A. Korhonen, D.R. Traum, L. Màrquez (eds.) Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, pp. 1549–1559. Association for Computational Linguistics (2019). DOI 10.18653/v1/p19-1150. URL `https://doi.org/10.18653/v1/p19-1150`