POLITECNICO DI TORINO
Repository ISTITUZIONALE

Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks

Original
Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks / Marchisio, Alberto; Bussolino, Beatrice; Colucci, Alessio; Martina, Maurizio; Masera, Guido; Shafique, Muhammad. - ELETTRONICO. - 1:(2020), pp. 1-6. ((Intervento presentato al convegno 2020 57th ACM/IEEE Design Automation Conference (DAC) tenutosi a San Francisco, CA, USA, USA nel 20-24 July 2020 [10.1109/DAC18072.2020.9218746].

Availability:
This version is available at: 11583/2848255 since: 2020-10-16T15:04:29Z

Publisher:
IEEE

Published
DOI:10.1109/DAC18072.2020.9218746

Terms of use:
openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

16 July 2022

To appear at the 57th Design Automation Conference (DAC), July 2020, San Francisco, CA, USA.

arXiv:2004.07116v2 [cs.LG] 17 Apr 2020

Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks

Alberto Marchisio*[1], Beatrice Bussolino*[2], Alessio Colucci[1], Maurizio Martina[2], Guido Masera[2] and Muhammad Shafique [1]
[1]*Technische Universität Wien (TU Wien), Vienna, Austria,* [2]*Politecnico di Torino (PoliTo), Turin, Italy*
Email: {alberto.marchisio, alessio.colucci, muhammad.shafique}@tuwien.ac.at
{beatrice.bussolino, maurizio.martina, guido.masera}@polito.it

***Abstract*—Capsule Networks (CapsNets), recently proposed by the Google Brain team, have superior learning capabilities in machine learning tasks, like image classification, compared to the traditional CNNs. However, CapsNets require extremely intense computations and are difficult to be deployed in their original form at the resource-constrained edge devices. This paper makes the first attempt to quantize CapsNet models, to enable their efficient edge implementations, by developing a specialized quantization framework for CapsNets. We evaluate our framework for several benchmarks. On a deep CapsNet model for the CIFAR10 dataset, the framework reduces the memory footprint by 6.2x, with only 0.15% accuracy loss. We will open-source our framework at https://git.io/JvDIF in August 2020.**

***Index Terms*—Capsule Networks, Quantization, Compression.**

I. Introduction

Since the introduction of AlexNet [12] in 2012, the interest in deep neural networks (DNNs) has grown steadily. Many models have been subsequently developed, achieving good accuracy in different tasks, such as object detection, computer vision and natural language processing. To achieve high accuracy, very deep and large DNNs models have been developed, for instance, from LeNet-5 [13] having five layers to ResNet-152 [7] having 152. Consequently, DNNs have a huge number of parameters, i.e., weights and biases, and their deployment in IoT systems is a challenge in terms of memory and computational resources. For example, the AlexNet requires 250MB memory for 60M parameters stored as 32-bits `float`. These memory and computational requirements make DNNs unsuited for mobile and embedded devices. Much effort has been dedicated towards *compressing DNN models* to address this problem. Quantization allows to significantly reduce the DNN model size, as well as enabling their applicability on different computing platforms like GPUs and FPGAs. In the literature, several quantization methods have been proposed [2][9][6][5][16][23][10][1][22].

Meanwhile, to improve the learning capabilities and accuracies of DNNs, the researchers at Google [21] introduced a novel DNN structure called *Capsule Network (CapsNet)*, where individual neurons are substituted with *capsules*, i.e., vectors of neurons. To overcome the loss of information introduced by the pooling layers, the pooling is substituted by a *dynamic routing* process between the capsules of adjacent layers. As a drawback, CapsNets are much more challenging in terms of their memory requirement, memory bandwidth and energy consumption for the computational resources, compared to the traditional DNNs. To demonstrate this fact, we compare the CapsNet architecture introduced in [21][1] with the AlexNet [12] and the LeNet [13]. For these networks, we analyze their respective memory requirements and the number of multiply-and-accumulate operations (MACs) necessary to compute an inference pass. Fig. 1 shows the results for the memory requirement (left) and the ratio between the MACs and the memory requirement (right). The latter is used as a comparative measure for the computational complexity. We noticed that the AlexNet has a larger memory requirement than the ShallowCaps, but with a lower MACs/Memory ratio. Hence, as shown, the ShallowCaps is more compute-intensive not only when compared to a simpler and smaller CNN like the LeNet, but also when compared to a deeper and heavier CNN like the AlexNet. This is attributed to the larger dimension of the constituent elements of the CapsNets and the high computational effort required to dynamically route the capsules.

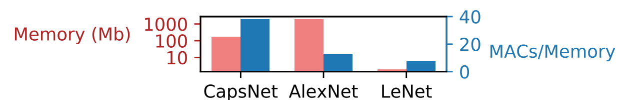

Fig. 1: *Memory requirements and (Multiply-and-Accumulate operations vs. memory) ratio (MACs/memory) for ShallowCaps [21], AlexNet [12] and LeNet [13].*

Motivational Analysis for our Target Research Problem: Our overarching goal is to make CapsNets deployable at the edge, abandoning floating-point representation and adopting a lighter fixed-point representation. 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. We perform a detailed analysis of the energy consumption and area footprint of a MAC unit, which is the basic block of specialized CapsNet accelerators like [17], and of hardware blocks which perform computationally complex operations, i.e., *squash* and *softmax*, which are required during the CapsNets inference. We design different versions of a MAC unit, a squash module, and a softmax module, varying their wordlength, and we synthesize them in a UMC 65nm CMOS technology with the Synopsys Design Compiler tool to measure their area and energy consumptions. Fig. 2 shows that *the area and energy consumption of MAC units decrease quadratically w.r.t. the wordlength*. Such analysis motivates us to focus on minimizing the wordlength to reduce the energy consumption. The results shown in Fig. 3 are obtained varying the number of fractional bits and keeping a single bit for the integer part. As expected, *the squash and the softmax functions require more energy and area than a simple MAC operation*. The dependence of the energy consumption and of the area footprint is related quadratically to the number of fractional bits. This further motivates us to reduce the number of bits employed to perform the operations in the various layers of the CapsNets architectures.

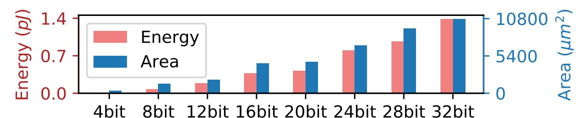

Fig. 2: *Energy consumption and area footprint for a fixed-point Multiply-and-Accumulate unit (MAC) with different wordlengths.*

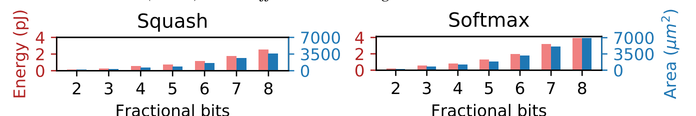

Fig. 3: *Energy consumption and area footprint for fixed-point modules performing (**left**) the squash and (**right**) the softmax with different wordlengths.*

*These authors contributed equally

[1]We will refer to the CapsNet architecture introduced in [21] as to ShallowCaps, to distinguish it from the DeepCaps [20] architecture

Associated Research Challenges: 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 an efficient trade-off between the memory footprint, the energy consumption and the classification accuracy, we propose a novel framework Q-CapsNets (see Fig. 4), which explores different layer-wise and operation-wise arithmetic precisions for obtaining the quantized version of a given CapsNet, with a maximum accuracy tolerance and a memory budget specified as constraints to the framework. Our approach tackles in particular the dynamic routing, which is a peculiar feature of the CapsNets and, as demonstrated in the previous paragraphs, involves complex and computationally expensive operations performed iteratively, with a significant impact on the energy consumption.

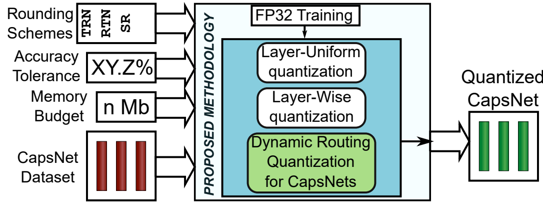

Fig. 4: *An overview of our quantization framework.*

In a nutshell, our novel contributions are:

- We propose a specialized framework for systematically quantizing CapsNets, given a certain accuracy tolerance (w.r.t. the full-precision CapsNet) and a certain memory budget for storing the weights. (**Section III**)
- Since an expensive part of CapsNets is the dynamic routing process, we further specialize the search of the numerical precision for the operations of the dynamic routing. *A key advantage of using our framework*, compared to traditional DNN quantization methods, is that, as we will demonstrate in our experiments, *the number of bits to route capsules can be further reduced* compared to the activations of the other layers. (**Section III, Step 4A**)
- We test our framework on the CapsNet model [21] on the MNIST [14] and Fashion-MNIST [24] datasets, and on the Deep-Caps model [20] on the MNIST, FashionMNIST and CIFAR10 [11] datasets[2]. As a key result for the latter dataset, we reduce the memory footprint by 6.2× with an accuracy loss of 0.15%. (**Section IV**)
- **Open-Source Contribution:** for reproducible research, we will release the complete source code of our framework, including the quantized CapsNet models, at https://git.io/JvDIF (Aug. 2020).

In the following **Section II**, we first discuss the CapsNets and the rounding schemes, to a level of details that is necessary to understand the rest of the paper.

II. Background and Related Work

A. Capsules Networks

CapsNets were introduced by Hinton et al. [8]. A capsule is a group of neurons that are organized in the form of a vector, where its length (i.e., the Euclidean Norm) is the instantiation probability of a certain feature, while the individual elements of the vector encode different spatial information, like width, skew, and rotation. The main advantage of capsules is that they preserve spatial information of detected features, an important quality when performing different recognition tasks.

The architecture[3] of the CapsNet proposed by Google [21] is reported in Fig. 5. It is composed of the following three layers:

1) **(L1) Conv Layer:** 9x9 convolutional with 256 output channels;
2) **(L2) PrimaryCaps:** convolutional with 256 output channels. These channels are divided into 32 8-dimensional (8-D) capsules (32 8-D vectors of neurons). The *squash* nonlinear function forces the length of the capsule's vector to be in the range of [0:1].
3) **(L3) DigitCaps:** fully-connected with 16-D capsules. The number of capsules depends on the number of classes of the dataset (e.g., 10 for MNIST and FashionMNIST). Between PrimaryCaps and DigitCaps, the so-called *dynamic routing* algorithm is used, as shown in Fig. 6.

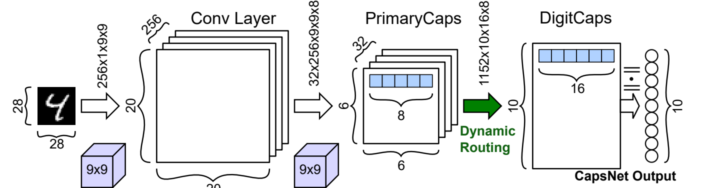

Fig. 5: *CapsNet architecture for MNIST/Fashion-MNIST dataset.*

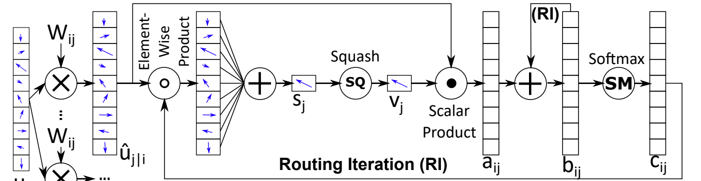

Fig. 6: *The operations to be computed for the dynamic routing.*

Recently, a novel deep CapsNet architecture, *DeepCaps* [20], has been proposed (see Fig. 7). It introduces Convolutional layers of capsules (*ConvCaps*). After the first convolutional layer with ReLU activation function, the network features 12 ConvCaps layers. Every three sequential ConvCaps layers have an additional ConvCaps layer that operates in parallel. The last parallel ConvCaps layer performs dynamic routing, while the other ConvCaps layers perform the squash function. The output layer of the DeepCaps architecture is a fully-connected capsule layer with dynamic routing.

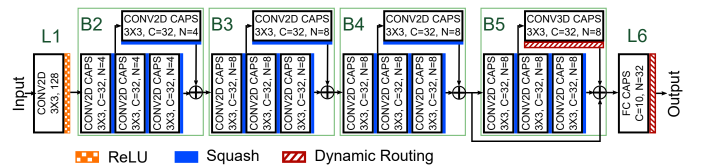

Fig. 7: *DeepCaps architecture as in [20].*

The dynamic routing (see Fig. 6) is an iterative algorithm that measures the agreement between capsules in a lower layer. Each capsule is assigned to a routing coefficient. If many capsules point in the same direction with high intensity (length), they all get a high coefficient. Hence, a capsule j in a higher layer is connected to all the capsules i in the lower layer that mostly agree with each other. The computations are the following:

1) Votes $\hat{u}_{j|i} = W_{ij} \times u_i$
2) Logits initialization $b_{ij} = 0$
3) Coupling coefficients $$c_{ij} = \text{softmax}(b_{ij}) = \frac{e^{b_{ij}}}{\sum_k e^{b_{ik}}} \quad (1)$$
4) Preactivation $s_j = \sum_i c_{ij} \hat{u}_{j|i}$
5) Activation $$v_j = \text{squash}(s_j) = \frac{||s_j||^2}{1+||s_j||^2} \frac{s_j}{||s_j||} \quad (2)$$
6) Agreement $a_{ij} = v_j \cdot \hat{u}_{j|i}$
7) Logits update $b_{ij} = b_{ij} + a_{ij}$

The dynamic routing consists of iterating the steps 3-7 for a defined number of times (e.g., 3 iterations in [21]). *From a hardware*

[2] To the best of our knowledge, they are the best available CapsNet models, and there is no related work able to train CapsNet models on the ImageNet [3] dataset.

[3] Since we focus on the CapsNet inference, we do not discuss the layers and the algorithms that are *only* involved in the training process (e.g., decoder and reconstruction loss).

perspective, such iterative computations are challenging, because they are difficult to be parallelized at a large scale.

B. Rounding Schemes

A fixed-point number [4] has an integer part QI and a fractional part QF, and thus can be written as $\langle QI.QF \rangle$. The total number of bits, i.e., the wordlength N, is computed as the sum $NI + NF$, where NI and NF are the bits of the integer part and the fractional part, respectively. The precision of a fixed-point representation is $\epsilon = 2^{-NF}$, and its corresponding range of representable numbers, in a two's complement format, is $[-2^{NI-1}, 2^{NI-1} - 2^{-NF}]$.

The *rounding operation* converts a floating-point or a large-sized fixed-point number into a "fixed-point number with shorter wordlength". Next, we discuss the most common rounding schemes.

Truncation (TRN) simply removes all the extra digits from the fractional part, i.e., $x_q = \lfloor x \rfloor$. If we assume uniformly distributed numbers, the truncation introduces a negative average error (bias), where such error is defined as $x_q - x$.

Round-to-Nearest (RTN) sets a rule for approximating those values which fall exactly half-way between the two representable numbers. In particular, rounding half-up consists of rounding up these values. Considering uniformly distributed numbers, rounding-up half-way values introduces a negative average error, which is lower than the one introduced by a simple truncation.

$$x_q = \lfloor x + \frac{\epsilon}{2} \rfloor \tag{3}$$

Stochastic Rounding (SR) is defined as:

$$\begin{cases} \lfloor x \rfloor & \text{if } P \geq \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x + \frac{\epsilon}{2} \rfloor & \text{if } P < \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases} \tag{4}$$

Here, $P \in [0, 1)$ is a random number with uniform distribution. The SR is an unbiased rounding scheme, but it is the most demanding one from the hardware perspective because its implementation requires the generation of random numbers.

C. Quantization of Traditional DNNs

Given the memory and computational requirements of DNNs, model compression is a widely studied subject where various techniques have been proposed. Han et. al [6] proposed Deep Compression, a three-stage pipeline to compress DNN models that combines pruning, quantization and Huffman coding, thus achieving outstanding memory reduction for different architectures.

Focusing only on quantization, Courbariaux et al. [2] introduced BinaryConnect, constraining all the weights of a network to the two values {-1, +1}, while Hubara et al. [9] binarized both the weights and the activations. Both approaches required to train the network with binary weights. Gysel et al. [5] proposed the Ristretto framework, where the weights and the activations of DNN models are quantized using fixed-points, starting from a model trained in full-precision. The required numerical resolution is found with a statistical analysis of the parameters and the model is fine-tuned by retraining after the quantization. Similarly, Lin et al. [16] determined the fixed-point format of the weights and activations collecting the statistics of the data and minimizing the signal-to-quantization-noise-ratio (SQNR).

Targeting the development of efficient hardware accelerators for DNN inference, the works in [23] and [10] tested the effect of 8-bits fixed-point quantization of the weights and the activations of different architectures, obtaining significant speed-ups at the cost of low or no accuracy reduction. Contrarily to [23] and [10], the works in [1] and [22] proposed a layer-wise optimization of the fixed-point representation adopted for the weights and the activations of each layer of the network. The work in [22] demonstrated that the precision required by the weights lowers for layers closer to the output, while the precision required by the activations is more constant across the layers of the network.

In our work, we introduce a novel method for quantizing the CapsNets architectures in a layer-wise fashion, tackling specifically the dynamic routing, which is peculiar for these networks. Moreover, we do not restrict the space to a single rounding scheme or to a particular data domain (weights or activation); rather our framework chooses an efficient solution to quantize different layers in a hybrid manner, thereby providing better trade-offs between the model complexity and the resulting accuracy loss.

III. Our Q-CapsNet framework

Our framework is able to progressively reduce the numerical precision of the data (e.g., weights and activations) in the CapsNet inference. During the first stage, we start with adapting/customizing the techniques for CapsNets, which are also applicable to traditional DNNs. Afterwards, we employ a specialized technique for CapsNets, which is tailored for the loops of the dynamic routing. The inputs of our framework are:

- *A CapsNet architecture*, together with the training and test dataset, and its associated architecture-specific hyperparameters.
- *A library of rounding schemes* to choose from when quantizing the data, with the option of adopting a single rounding scheme, based on the application demand. In the first case, the framework is free to choose any rounding scheme from the library. Otherwise, it is fixed. The process of selecting an appropriate rounding scheme will be discussed in Sec. III-B.
- As will be explained in Sec. III-A, lowering numerical precision reduces the accuracy reached by the model. Therefore, a tolerance acc_{TOL} on the loss of accuracy must be set to have a margin for quantizing the network. The target accuracy acc_{target} is computed in Equation 5.

$$acc_{target} = acc_{FP32} \cdot (1 - acc_{TOL}) \tag{5}$$

- *Maximum memory budget* that can be occupied for the storage of the quantized weights and biases.

Our Q-CapsNet framework aims at satisfying both requirements on accuracy and memory usage. An effective way to reduce the model's memory usage is through aggressively quantizing the weights. We perform this operation in the steps (1) and (2) of the proposed framework. Once the memory budget is satisfied, if there is still some margin on the tolerable accuracy loss, we reduce the numerical precision of the weights and activations, to reduce the energy consumed during the CapsNet inference computations, and the framework returns the `model_satisfied`. Otherwise, if a solution which satisfies both the requirements on the accuracy and the memory usage cannot be found, our framework returns two sub-optimal solutions as the followup:

I `model_accuracy`: A quantized CapsNet with the target accuracy and the minimum possible memory footprint (which can be slightly higher than the budget);

II `model_memory`: A quantized CapsNet that satisfies the memory requirements, and achieving the maximum possible accuracy (which can be slightly lower than the target).

A. Step-by-Step Description of our Framework

As a preliminary stage, a given input CapsNet is trained in full-precision (32-bits floating-point), whose accuracy is denoted as acc_{FP32}. From acc_{FP32} and the accuracy tolerance (acc_{TOL}, input

of the framework), we compute the target accuracy (acc_{target}) as in Equation 5. The procedure followed for quantizing the given CapsNet (see Figure 8 and Algorithm 1) is composed of the following steps:

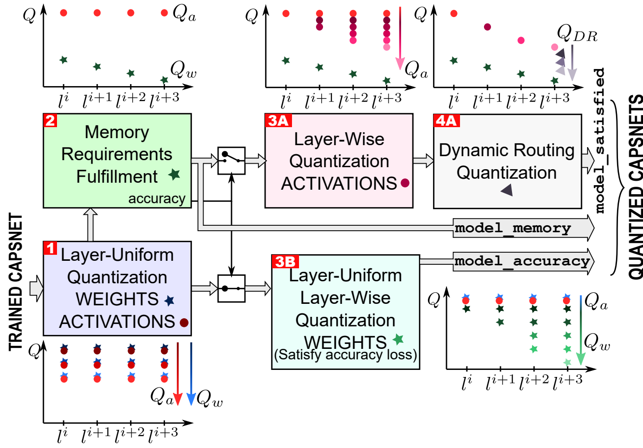

Fig. 8: *Flow of Our Framework for Quantizing CapsNets.*

Algorithm 1 Pseudo-Code of Our Framework. (see Fig. 8 for its Flow)

1: **procedure** Q-CAPSNET(CapsNet, acc_{TOL}, memory_budget)
2: ▷ Full Precision training
3: model, $acc_{FP32} \leftarrow$ train(CapsNet)
4: $acc_{target} = acc_{FP32}(1 - acc_{TOL})$
5: ▷ *Step 1)*
6: $acc_{step1} = acc_{FP32}(1 - acc_{TOL} \cdot 0.05)$
7: model, Q $\leftarrow$ BinarySearch(model, (weights, act), $Q_{init} = 32$, $acc_{min} = acc_{step1}$)
8: $(Q_{w,s1})_l = Q$; $(Q_{a,s1})_l = Q\ \forall l$
9: ▷ *Step 2)*
10: $[(Q_{w,mm})_0, ..., (Q_{w,mm})_L] \leftarrow$ Eq.6(params P, memory_budget)
11: model_memory, $acc_{mm} \leftarrow$ test(quant(model, weights $\leftarrow Q_{w,mm}$, act $\leftarrow Q_{a,s1}$))
12: **if** $acc_{mm} > acc_{target}$ **then**
13: ▷ *Step 3A)*
14: model, $Q_a \leftarrow$ LayerWise(model, act, $Q_{init} = Q_{a,s1}$, $acc_{min} = acc_{target} + 0.5(acc_{mm} - acc_{target})$)
15: ▷ *Step4A)*
16: **for** each layer l with dynamic routing **do**
17: model, $(Q_a)_l \leftarrow$ DRquant(model, model.DRact$_l$, $Q_{init} = (Q_a)_l$, $acc_{min} = acc_{target}$)
18: **end for**
19: **return** model_satisfied
20: **else**
21: ▷ *Step3B)*
22: model, $Q_w \leftarrow$ BinarySearch(model, weights, $Q_{init} = Q_{w,1}$, $acc_{min} = acc_{target}$)
23: model_accuracy, $Q_w \leftarrow$ LayerWise(model, weights, $Q_{init} = Q_w$, $acc_{min} = acc_{target}$
24: **return** model_memory, model_accuracy
25: **end if**

1) **Layer-Uniform Quantization (weights + activations)**: We convert all weights and activations to a fixed-point arithmetic, with 1-bit integer part, and Q_w-bit and Q_a-bit fractional part, respectively. Afterwards, we further reduce their precision in a uniform way (e.g., $Q_w = Q_a$). In this stage, only 5% of the acc_{TOL} is consumed. To find the correct wordlength of Q_w and Q_a, we use a binary search algorithm [15].

2) **Memory Requirements Fulfillment**: In this stage, we quantize only the CapsNet weights. Following the idea of Raghu et al. [19] that perturbations to weights in final layers can be more costly than perturbations in the earlier layers, we set for each layer l its respective Q_w such that $(Q_w)_{l+1} = (Q_w)_l - 1$. Having set these conditions, we can compute the correct Q_w as the maximum integer value that satisfies the Equation 6, where L is the total number of layers, M is the memory budget, and P^l is the number of parameter (weights) in the layer l.

$$\sum_{l=0}^{L-1} \left(P^l \cdot ((Q_w)_0 - l) \right) \leq M \tag{6}$$

With this rule, we obtain a quantized CapsNet model, denoted as `model_memory`, which fulfills the memory requirements. Afterwards, we test the accuracy of the `model_memory`, denoted as acc_{mm} and compare it to acc_{target}. Based on its results, the next step can take two directions. If acc_{mm} is higher, we continue to (3A) for further quantization steps. Otherwise, it jumps to (3B).

3A) **Layer-Wise quantization of activations**: To quantize the activations, we start from the initial Q_a, as computed during the step (1). As shown in Algorithm 2, we proceed in a layer-wise fashion. During the first step, each layer of the CapsNet (except the first one) is selected, and Q_a is lowered until the minimum value for which the accuracy remains higher than acc_{target}. Afterwards, the wordlength of the first two layers is fixed, while we further reduce Q_a for all but the first layers. We repeat this step iteratively until the Q_a for the last layer is set.

Algorithm 2 Algorithm for Layer-wise Quantization

1: Given: Q_{init} initial number of quantization bits to start the algorithm, acc_{min} minimum value of accuracy that can be reached.
2: **procedure** LAYERWISE(model, params, Q_{init}, acc_{min})
3: Q = [$(Q)_0$, $(Q)_1$, ..., $(Q)_L$], $(Q)_l = Q_{init}$
4: $StartL = 1$
5: **while** $StartL < L$ **do**
6: $acc = 100$
7: **while** $acc \geq acc_{min}$ **do**
8: $(Q)_l \leftarrow (Q)_l - 1$, $l \in [StartL, ..., L]$
9: model, acc = test(quant(model, params $\leftarrow Q$))
10: **end while**
11: $(Q)_l \leftarrow (Q)_l + 1$, $l \in [StartL, ..., L]$
12: $StartL \leftarrow StartL + 1$
13: **end while**
14: **return** quant(model, params $\leftarrow Q$), Q

4A) **Dynamic Routing Quantization**: The dynamic routing is computationally expensive due to the complex operations, such as *squash* (Eq. 2) and *softmax* (Eq. 1), and the operations are performed iteratively. Hence, the wordlength of its arrays may be different as compared to other layers of the CapsNet. This step operates only on the data involving the *squash* and *softmax* operations. A specialized quantization process is performed in this step, as shown in Fig. 9 and Algorithm 3. As we will demonstrate in our experiments, 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 that is generated at the end of this step is denoted as `model_satisfied`.

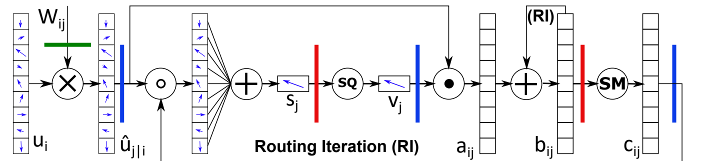

Fig. 9: *Quantization of a capsule layer with dynamic routing. Colored bars show the arrays that are rounded and quantized. In green, the weights are quantized with Q_w bits. In blue, the activations are quantized with Q_a bits. In red, data are quantized more aggressively with Q_{DR} bits. The precision is lowered before complex and compute-intensive functions (squash, softmax).*

3B) **Layer-Uniform ad Layer-Wise Quantization of Weights**: Starting from the outcome of step (1), we quantize the weights only, first in a uniform and then in a layer-wise manner (as in step 3A) until reaching acc_{target}. The resulting CapsNet model

Algorithm 3 Algorithm for Dynamic Routing Quantization

1: Given: Q_{init} initial number of quantization bits to start the algorithm, acc_{min} minimum value of accuracy that can be reached.
2: **procedure** DRQUANT(model, params, Q_{init}, acc_{min})
3: $Q = Q_{init}$
4: $acc = 100$
5: **while** $acc \geq acc_{min}$ **do**
6: $Q \leftarrow Q - 1$
7: model, acc = test(quant(model, params $\leftarrow Q$))
8: **end while**
9: $Q \leftarrow Q + 1$
10: **return** quant(model, params $\leftarrow Q$), Q

(`model_accuracy`) is returned as the output of the framework, together with `model_memory`, as generated in step (2).

B. Rounding Scheme Selection

For each rounding scheme from the given library, its corresponding quantized model is generated. Hence, *our framework executes the Algorithm 1 for each rounding scheme in parallel*. Note, due to different rounding errors, it is possible that for one rounding scheme our framework executes the Path A, while for another schemes it executes the Path B. At the end of the execution of all branches, the best rounding scheme within the library is selected with the following criteria, depending on whether the algorithm has followed Path A or not.

A) **There are some models generated from Path A:**

1) Models from Path B are discarded.
2) The model with lower memory is selected.
3) With the same memory, the model with fewer bits used to represent activations is selected.
4) With the same memory and bits for the activations, the model with the simplest rounding scheme is selected, e.g., with our examples, in order, truncation, round-to-nearest-even, and stochastic rounding. Note, while the first one simply requires the deletion of the LSBs, the last one requires more complex operations to decide the orientation of the rounding.

B) **There are models only from Path B:**

1) In this case, two models are returned. Selecting from `memory_model`, the model with the highest-possible accuracy is returned.
2) Selecting from `accuracy_model`, the model with the lowest-possible memory is returned.
3) If more than one model have the same highest accuracy and the lowest memory, the simplest rounding scheme is preferred to break the tie.

IV. Results

A. Experimental setup

We implement the Q-CapsNet framework (see Fig. 10) in PyTorch [18], and we run it on two Nvidia GTX 1080 Ti GPUs. We test it on the CapsNet model proposed by Google [21], i.e., ShallowCaps, also previously described in Sec. II-A, for MNIST [14] and Fashion-MNIST [24] datasets, and on the DeepCaps model for the MNIST, FashionMNIST and CIFAR10 [11] datasets. The MNIST database is a collection of 28x28 grayscale handwritten digits, from 1 to 10, composed of 60,000 training samples and 10,000 testing samples. The FashionMNIST is a collection of 28x28 grayscale images, representing Zalando's articles associated to 10 different classes. It is composed of 60,000 training samples and 10,000 testing samples. The CIFAR10 is a collection of 32x32 color images organized in 10 different classes, with the training set composed of 50,000 samples and the testing set of 10,000 samples. For full precision training, data augmentation is achieved as follows:

- MNIST: images are randomly shifted by maximum two pixels and rotated of 2 degrees;
- FashionMNIST: images are randomly shifted of 2 pixels and horizontally flipped with a probability of 0.2;
- CIFAR10: images are resized to 64x64[4], randomly shifted of 5 pixels, rotated of 2 degrees and horizontally flipped with a probability of 0.5.

No data augmentation is done on the images for testing.

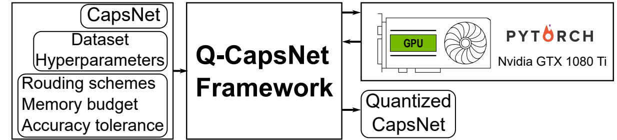

Fig. 10: *Experimental setup to test our Q-CapsNet framework.*

B. Quantized Architectures

ShallowCaps for the MNIST Dataset

The ShallowCaps architecture [21] is trained in full precision (FP32) on the MNIST dataset, for 100 epochs and with batch size equal to 100. We use an exponential decay learning policy, with an initial learning rate of 0.001, 2000 decay steps and 0.96 decay rate. Its achieved test accuracy is 99.67%.

Afterwards, the framework proceeds as described in Sec. III-A, with the aim of concurrently satisfying the memory and accuracy requirements. Since the algorithm has a conditional path, for the sake of clarity, we present two examples, which correspond to the execution of the different branches of the algorithm.

Test of the Path A: For the first set of experiments, we test the Path A of the framework, i.e., when both the memory and accuracy constraints are satisfied. Since the memory requirement at FP32 is 217Mbit, we set the memory budget equal to 45Mbit, with an accuracy tolerance of 0.2%. The results in Fig. 11 [Q1] show that the `model_satisfied` reduces the memory footprint of the weights by 4.11×, as compared to the FP32 model, with an accuracy equal to 99.52%. Along with the reduction of the memory occupied by the weights (W mem), we report the reduction of the memory required to store the activations (A mem). For `model_satisfied`, this memory footprint is reduced of 2.72×.

Test of the Path B: Since our framework executes the Path B if it cannot find a solution which satisfies both requirements, for its testing purpose, we specify very low memory budgets as the input. The results of our experiments, shown in Fig 11, indicate that to satisfy the memory requirements, weights of `model_memory` [Q3] are set to very low wordlengths, causing an extreme reduction of accuracy. To satisfy the accuracy requirements in `memory_accuracy` [Q2], weights are reduced to the minimum possible wordlength.

ShallowCaps for the FashionMNIST Dataset

Similar sequences of tests, with a set of memory budget and accuracy tolerance specifications, are performed on the same ShallowCaps architecture for the FashionMNIST dataset. The results from our experiments are reported in Table I.

DeepCaps for MNIST, FashionMNIST and CIFAR10 datasets

Several tests are carried out on the DeepCaps architecture. We mainly discuss the results obtained with the SR scheme, which outperforms the other (simpler) rounding schemes. The DeepCaps architecture trained in full-precision on the MNIST dataset achieves a 99.75%

[4]The original images of size 32x32 are resized to 64x64 by bilinear interpolation, to allow deeper networks, as reported in the original paper [20].

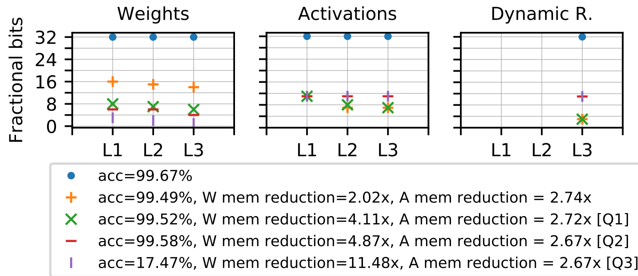

Fig. 11: *Q-CapsNet results of the ShallowCaps [21] for the MNIST [14] dataset.*

TABLE I: Q-CapsNet's accuracy results, weight (W) memory and activation (A) memory reduction for the ShallowCaps [21] and for the DeepCaps [20] on MNIST [14], Fashion-MNIST [24] and CIFAR10 [11] datasets.

Model	Dataset	Accuracy	W mem reduction	A mem reduction
ShallowCaps	MNIST	99.58%	4.87x	2.67x
ShallowCaps	MNIST	99.49%	2.02x	2.74x
ShallowCaps	FMNIST	92.76%	4.11x	2.49x
ShallowCaps	FMNIST	78.26%	6.69x	2.46x
DeepCaps	MNIST	99.55%	7.51x	4.00x
DeepCaps	MNIST	99.60%	4.59x	6.45x
DeepCaps	FMNIST	94.93%	6.4x	3.20x
DeepCaps	FMNIST	94.92%	4.59x	4.57x
DeepCaps	CIFAR10	91.11%	6.15x	2.50x
DeepCaps	CIFAR10	91.18%	3.71x	3.34x

accuracy, on par with the accuracy obtained in [21], while on the FashionMNIST it achieves a 95.08% accuracy. Table I reports some key results obtained with the Q-CapsNet framework on these two datasets. Fig. 12 reports graphically some key results obtained with the Q-CapsNet framework on the DeepCaps for the CIFAR10 dataset.

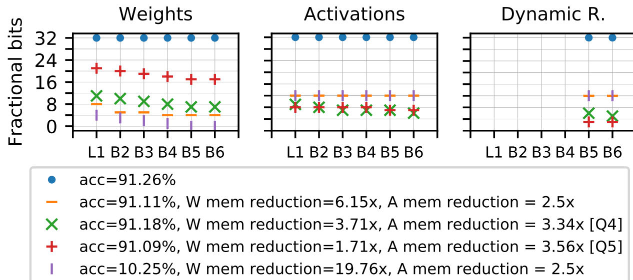

Fig. 12: *Q-CapsNet results of the DeepCaps [20] for CIFAR10 [11] dataset.*

C. Comparison between Different Rounding Schemes

Experiments performed for different inputs to the framework show that truncation and round-to-nearest schemes return identical results. This is due to the fact that these schemes differ from each other only for a very small set of continuous values, i.e., those falling half-way between two discrete values, and therefore the influence on the final results of the network is negligible.

Fig. 13 shows the accuracy reached by the ShallowCaps when different rounding schemes are applied, with the same memory usage. For both the MNIST and FashionMNIST datasets, stochastic rounding outperforms simpler methods, e.g., when a lower memory footprint is required. Indeed, the stochastic rounding presents the advantage of randomizing the quantization noise. Small values close to zero have a non-null probability of being rounded up rather than always being forced to zero. This solution avoids an excessive loss of information when iteratively performing computations and quantizations.

D. Further Discussion on the Results

By considering the occupied weight memory and the accuracy as the evaluation metrics, we noticed that, usually, the `model_satisfied` seems to be Pareto-dominated by the `model_accuracy`, like in the case of Q1 and Q2 in Fig. 11, and of Q4 and Q5 in Fig. 12. However, since Q1 and Q5 have lower wordlengths for the activations and the dynamic routing, compared respectively, to Q2 and Q4, the potential energy-efficiency gains for its computations using MAC operators, *squash* and *softmax* (recall Figures 2 and 3) are huge, even with a small change in the activation memory. Note, the wordlength for the dynamic routing operations can be reduced up to 3 or 4 bits with very limited accuracy loss compared to the full-precision model. Such an outcome is attributed to a common feature of the dynamic routing. The operations of the involved coefficients (along with *squash* and *softmax*, see Fig. 6) are updated dynamically, thereby adapting to the quantization more easily than previous layers like Conv Layer and PrimaryCaps. Hence, these computations can tolerate a more aggressive quantization.

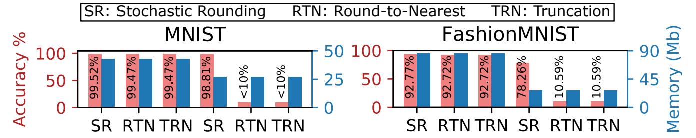

Fig. 13: *Accuracy and memory comparison for Q-CapNet models of the ShallowCaps architecture, obtained using different rounding schemes.* ***(left)*** *Results for MNIST.* ***(right)*** *Results for Fashion-MNIST.*

V. Conclusion

We proposed a specialized framework for quantizing CapsNets, called Q-CapsNets. We exploited the peculiar features of CapsNets, occurring during the dynamic routing, for designing a quantization methodology that enables further precision reduction of the wordlength while a certain accuracy loss is tolerated. Our Q-CapsNets framework produces compact yet accurate quantized CapsNet models. Hence, it represents the first step towards designing energy-efficient CapsNets, and could potentially open new avenues towards the large-scale adoption of CapsNets for inference in a resource-constrained scenario.

Acknowledgments

This work has been partially supported by the Doctoral College Resilient Embedded Systems which is run jointly by TU Wien's Faculty of Informatics and FH-Technikum Wien.

References

[1] S. Anwar, K. Hwang, and W. Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *ICASSP*, 2015.
[2] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*. 2015.
[3] J. Deng et al. Imagenet: A large-scale hierarchical image database. *CVPR*, 2009.
[4] A. Granas and J. Dugundji. *Fixed Point Theory*. Springer, 2003.
[5] P. Gysel, M. Motamedi, and S. Ghiasi. Hardware-oriented approximation of convolutional neural networks. 2016.
[6] S. Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.
[7] K. He et al. Deep residual learning for image recognition. *CVPR*, 2015.
[8] G. E. Hinton et al. Transforming auto-encoders. In *ICANN*, 2011.
[9] I. Hubara et al. Binarized neural networks. In *NIPS*. 2016.
[10] B. Jacob et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CVPR*, 2018.
[11] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.
[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*. 2012.
[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.
[14] Y. LeCun et al. The MNIST database of handwritten digits, 1998.
[15] G. N. Lewis, N. J. Boynton, and F. W. Burton. Expected complexity of fast search with uniformly distributed data. *Inform. Proc. Let.*, 1981.
[16] D. D. Lin, S. S. Talathi, and V. S. Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, 2016.
[17] A. Marchisio, M. A. Hanif, and M. Shafique. Capsacc: An efficient hardware accelerator for capsulenets with data reuse. In *DATE*, 2019.
[18] A. Paszke et al. Automatic differentiation in pytorch. 2017.
[19] M. Raghu et al. On the expressive power of deep neural networks. In *ICML*, 2017.
[20] J. Rajasegaran et al. Deepcaps: Going deeper with capsule networks. *CVPR*, 2019.
[21] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *NIPS*, 2017.
[22] C. Sakr and N. Shanbhag. Per-tensor fixed-point quantization of the back-propagation algorithm. In *ICLR*, 2019.
[23] V. Vanhoucke et al. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.
[24] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. 2017.