Neurocomputing 370 (2019) 56–69

Contents lists available at ScienceDirect

Neurocomputing

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

Retraining-free methods for fast on-the-fly pruning of convolutional neural networks

Amir H. Ashouri[a,*], Tarek S. Abdelrahman[a], Alwyn Dos Remedios[b]

[a] University of Toronto, Canada
[b] Qualcomm Inc., Canada

ARTICLE INFO

Article history:
Received 30 March 2019
Revised 8 August 2019
Accepted 19 August 2019
Available online 2 September 2019

Communicated by Dr Jie Wang

Keywords:
Deep learning
Convolutional neural networks
Sparsity
Pruning

ABSTRACT

We explore retraining-free pruning of CNNs. We propose and evaluate three model-independent methods for sparsification of model weights. Our methods are magnitude-based, efficient, and can be applied on-the-fly during model load time, which is necessary in some deployment contexts. We evaluate the effectiveness of these methods in introducing sparsity with minimal loss of inference accuracy using five state-of-the-art pretrained CNNs. The evaluation shows that the methods reduce the number of weights by up to 73% (i.e., compression factor of 3.7 ×) without incurring more than 5% loss in Top-5 accuracy. These results also hold for quantized versions of the CNNs. We develop a classifier to determine which of the three methods is most suited for a given model. Finally, we employ additional, but impractical in our deployment context, fine-tuning and show that it gains only 8% in sparsity. This indicates that our on-the-fly methods capture much of the sparsity than can be attained without retraining, yet remain efficient and straight-forward to use.

1. Introduction

Today's Deep Neural Networks (DNNs) are successful in a wide range of application domains, ranging from speech recognition [1] to image classification [2]. One important class of DNNs is Convolutional Neural Networks (CNNs). These networks were first to demonstrate the significant potential of deep models by classifying images with high accuracy, far beyond what conventional models are able to accomplish [3]. Thus, it is no surprise that CNNs are leveraged on mobile devices for many applications, including face recognition [4], smile detection [5] and image segmentation [6].

However, the continuing success of CNNs comes at the cost of a significant growth in the number of parameters (i.e., layer weights), and the corresponding number of multiply-accumulate operations (MACs) needed to utilize a CNN model [3,7–13]. For example, the number of parameters/MACs has increased from 2.6K/341K in LeNet-5 [14] to 23.5M/3.9G in ResNet-50 [15]. Such growth makes it challenging to deploy CNNs on mobile devices, where memory capacity, computational power, and energy are constrained.

There are several techniques to *prune* or *sparsify* CNNs. That is, force some of a model's weights to 0 in order to both compress the model and to save computations during inference. At deployment, an embedded platform may then exploit these 0′s to both compress the model and to avoid the unnecessary MACs that involve these 0′s. Examples of these techniques include: iterative pruning and retraining [16–20], Huffman coding [21], exploiting granularity [22,23], structural pruning of network connections [24–27], use of sparse networks [28], and Knowledge Distillation (KD) [29].

A common theme to the aforementioned techniques is that they require a retraining of the model to fine-tune the remaining non-zero weights and thus maintain inference accuracy. Such retraining, while feasible in many contexts, is not feasible in others, particularly for mobile platforms. For example, on a mobile platform, a machine learning model is typically embedded within an app, which the user directly downloads, for the platform. The app utilizes the vendor's platform runtime support (often in the form of a library) to load and use the model. Thus, the platform vendor must sparsify the model at runtime, i.e., *on-the-fly*, within the library with no opportunity to retrain the model. Even if the opportunity exists, the vendor rarely has access to the labelled data used to train the model. While techniques such as Knowledge Distillation [29] can address this lack of access, it is not possible to apply them on-the-fly.

There is an inherent trade-off between sparsity and inference accuracy. In this paper, we seek to answer two basic questions that relate to this trade-off: (1) To what extent can a CNN be sparsified *without retraining*, while maintaining a reasonable inference

* Corresponding author.
E-mail addresses: aashouri@ece.utoronto.ca (A.H. Ashouri), tsa@ece.utoronto.ca (T.S. Abdelrahman), adosreme@qti.qualcomm.com (A. Dos Remedios).

https://doi.org/10.1016/j.neucom.2019.08.063
0925-2312/

accuracy, and (2) what are fast model-independent methods for sparsifying CNNs without retraining? To answer these questions, we develop fast retraining-free sparsification methods that can be deployed for on-the-fly sparsification of CNNs in the deployment context described above. We show that these *model-independent* methods result in large amount of sparsity with little loss to inference accuracy.

More specifically, we propose three sparsification methods: *Flat*, *Triangular*, and *Relative*. We implement these methods in TensorFlow and use the framework to evaluate the sparsification of five pretrained models: Inception-v3, MobileNet-v1, ResNet-v2, VGG, and AlexNet. Our evaluation shows that up to 73% of layer weights may be forced to 0, incurring only a 5% loss in Top-5 inference accuracy.

Our evaluation also shows that while the Relative method appears to be more effective for some models, the Triangular method is more effective for others. Thus, we also develop a predictive modeling classifier to identify, at run-time, the optimal choice of method and it hyperparameters. We also conduct additional, but impractical in the deployment context described earlier, fine-tuning and determine the extent to which further sparsity can be gained, still with no retraining. We show that only an 8% gain in sparsity is achieved. This indicates that our on-the-fly methods capture most of the sparsity than can be attained without retraining, yet remain straight-forward to use.

Thus, this paper makes the following contributions:

- We propose and evaluate three model-independent sparsification methods for CNNs. These methods introduce sparsity based on the distribution of the weights and are straightforward to apply for a given CNN.
- We develop a framework sparsifying CNNs built around TensorFlow, which does not require retraining after sparsification.
- We provide a classifier for which, at runtime, is able to decide which sparsification method is more suited for a given pretrained model.

The main goal of this work is to explore the extent to which sparsity can be introduced with no retraining of a model. The realization of the benefits of this sparsity (i.e., model compression and reduction in compute time) requires a framework capable of exploiting this sparsity. We leave the exploitation of sparsity for future work. Nonetheless, the benefits of the sparsity has been explored in the literature [21,30,31] and in general, the reduction in compute time is proportional to the sparsity. Thus, we use degree of sparsity as a proxy of the attainable benefit.

The remainder of the paper is organized as follows. Related work is presented in Section 2. Additionally, we discuss the distinct contrasts of our proposed methodology. In Section 3 we present our sparsification approach and our three sparsification methods. Section 4 describes how these methods are implemented in the TensorFlow framework. Section 5 presents our experimental evaluation, including the classifier for choosing a method, our fine-tuning experiments and how our results extend to quantized models, and finally, in Section 6 we offer some concluding remarks.

2. Related work

There exists a large body of work that deals with the sparsification and pruning of machine learning models. Generally, this body of work can be classified into four broad categories: (1) pruning and weight sparsification [16,19–22,28,30], (2) structural pruning [24–27,32], (3) hybrid pruning [33], and (4) low rank approximation [34–37]. All of this work uses retraining to fine-tune the resulting sparsified, pruned or reduced model [38,39].

Le Cun et al. [16] propose Optimal Brain Damage as the seminal work on sparsification and network pruning to reduce neural connection of pairs using the *saliency* of model parameters. Others [17,18] extend this work to use second order derivatives. More recently, Mao et al. [22,25,40] explore coarse-grain and fine-grain pruning and evaluate the trade-off between accuracy and sparsity using recent CNNs.

Huang et al. [41] propose a try-and-learn algorithm for pruning redundant filters in CNNs. They use a reward function to aggressively prune with minimal loss of accuracy; however, their method requires retraining as well as user input to achieve its results.

Sun et al. [19] propose ConvNets as a framework that can be used to iteratively learn sparsified neural connections through correlations among neural activations. The connections are dropped iteratively one layer at a time and the model is retrained to compensate the loss in accuracy. The authors achieve sparsity ratio of 88% and reported a Top-5 inference accuracy of 98.95% when trained using 300K face images of the DeepID2+ dataset.

Mathew et al. [20] propose a framework that compensates for the loss of accuracy after sparsification by retraining. They quantize their sparsified model for an embedded architecture and observe a nearly 4 × improvement in the inference speed with 80% sparsity.

Wen et al. [24] propose SSL, a sparsifying framework to exploit and regularize structural sparsity of a sample DNN. The authors evaluate their method using ResNet-v2, resulting in a model with reduced layers while improving inference accuracy by around 1.5%.

Parashar et al. [30] propose the SCNN accelerator architecture that utilizes a compressed encoding of sparse weights and activations. The authors observe up to 2.7 × energy improvement during both retraining and inference.

Our work stands in contrast to the above body of work in that it does not perform retraining of the sparsified model to fine-tune the remaining non-zero weights. Our methods [42] introduces sparsity into a model based on the magnitude of the weights. This approach results in sparse models without the overhead of retraining and without the need for training data, which is particularly important in our deployment context.

3. Sparsification methods

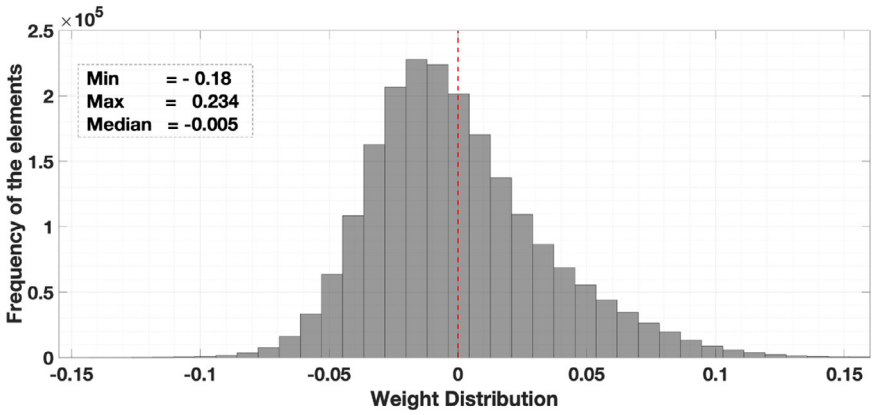
Sparsity in a CNN stems from three main sources: (1) weights within convolution (`Conv`) and fully-connected (`FC`) layers (some of these weights may be zero or may be forced to zero); (2) activations of layers, where the often-applied `ReLU` operation results in many zeros [15]; and (3) input data, which may be sparse. In this paper, we focus on the first source of sparsity, in both `Conv` and `FC` layers. This form of sparsity can be determined a priori, which alleviates the need for specialized hardware accelerators to exploit the sparsity.

The input to our framework is a CNN that has L layers, numbered $1 \ldots L$. The weights of each layer l are denoted by ω_l. In the case of `Conv` layers, these weights form a 4D tuple (tensor) $\omega_l \epsilon R^{F_l \times C_l \times H_l \times W_l}$, where F_l, C_l, H_l, and W_l are the dimensions of the lth weight tensor across the axes of filters, channels, height and width, respectively. We sparsify these weights using a sparsification function, $\mathbb{S}$, which takes as input ω_l and a threshold τ_l from a vector of thresholds T. Each weight i of ω_l is modified by $\mathbb{S}$ as follows:

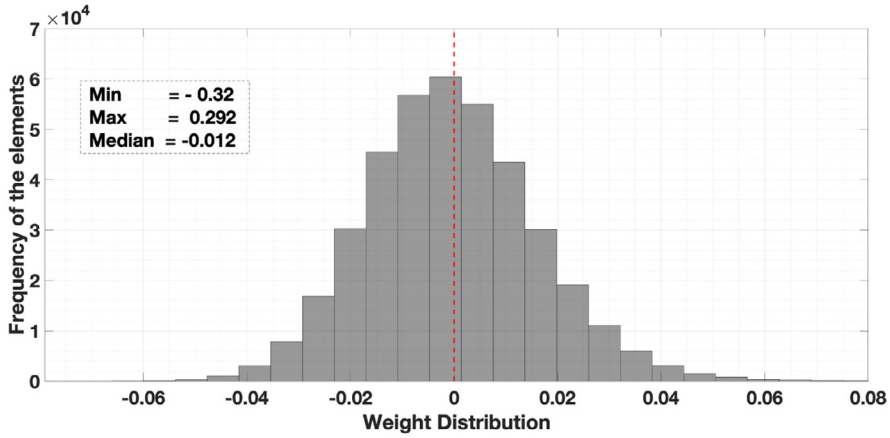
$$\mathbb{S}(\omega_l(i), \tau_l) = \begin{cases} 0 & \text{if } |\omega_l(i)| \leq \tau_l \\ \omega_l(i) & \text{otherwise} \end{cases} \quad (1)$$

where $\tau_l = T(l)$ is the threshold used for layer l. Thus, applying a threshold τ_l forces weights within $-\tau_l$ and $+\tau_l$ in value to become 0.

We define the *sparsity ratio* for each layer, denoted by s_l as the number of zero weights present after sparsification divided by the



(a) Inception-v3 Softmax



(b) Inception-v3 Last Conv

Fig. 1. Distribution of the weights in two layers of Inception-v3.

total number of weights in the layer. We similarly define the *model sparsity ratio*, denoted by s_m as the total number of zero weights in all layers of the model divided by the total number of weights in the model:

$$s_m = \frac{\sum_{l=1}^{L} \#zeros}{\#parameters} \tag{2}$$

Our use of thresholds to sparsify layers is motivated by the fact that recent CNNs' weights are distributed around the value 0. Fig. 1 shows the histograms of the weights of the last fully-connected and convolution layers of the Inception-v3 pretrained model. The figure shows that the weights are distributed in a quasi-symmetric way around 0. Thus, applying a single threshold τ_l forces weights within $-\tau_l$ and $+\tau_l$ in value to become 0.

The choice of the values of the elements of the vector T defines a *sparsification method*. These values impact the resulting sparsity and inference accuracy. We define and compare three sparsification methods. The *Flat* method defines a constant threshold τ for all layers, irrespective of the distribution of their corresponding weights. The *Triangular* method is inspired by the size variation of layers in some state-of-the-art CNNs, where the early layers have smaller number of parameters than latter layers. Thus, the method uses linearly growing thresholds from earlier to latter layers. Finally, the *Relative* method defines a unique threshold for each layer that sparsifies a certain percentage of the weights in the layer. The three methods are depicted graphically in Fig. 2.

3.1. Flat method

This method defines a constant threshold τ for all layers, irrespective of the distribution of their corresponding weights. It is graphically depicted in Fig. 2(a). The weights of the layers are profiled to determine the span $\sigma = min_{\forall lin L}(max(\omega_l) - min(\omega_l))$. This span corresponds to the layer k with the smallest range of weights within the pretrained model. This span is used as an upper-bound value for the threshold τ. Since using σ as a threshold eliminates all the weights in layer k and is likely to adversely affect the accuracy of the sparsified model, we use a fraction δ, $0 \leq \delta \leq 1$, of the span for the threshold:

$$\tau = \sigma \times \delta \tag{3}$$

where δ is varied to achieve different degrees of model sparsity. The values of σ and δ are parameters of the method that determine the resulting sparsity. We elaborate on these parameters in Section 5.2.

3.2. Triangular method

The Triangular method is inspired by the size variation of layers in state-of-the-art CNNs: early Conv layers have a smaller number of parameters than in later layers, i.e., final Conv and FC layers.

Thus, the Triangular sparsification method introduces the highest sparsity in the final layers and gradually diminishes the sparsification towards the early layers. This is done using an isosceles triangular shape for the values of the thresholds, as shown in

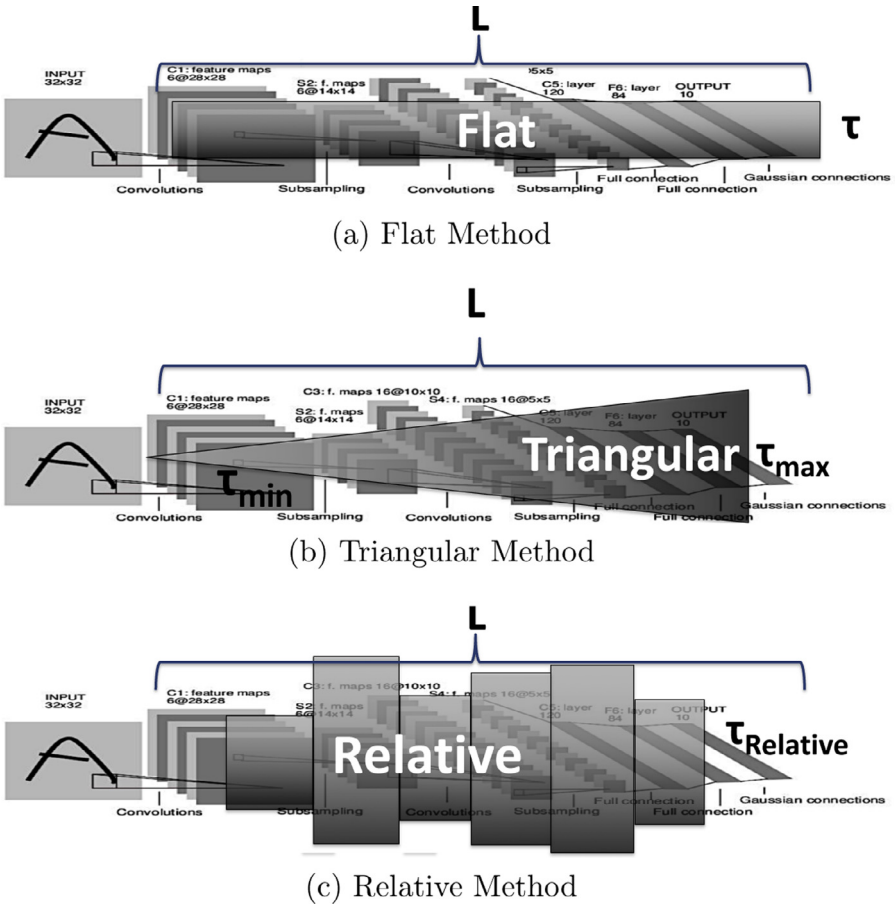


Fig. 2. Proposed Sparsification Methods: (a) Flat, (b) Triangular, and (c) Relative.

Fig. 2(b). This not only caters to the size variation of layers, it also avoids aggressive sparsification in early layers, which can adversely impact subsequent layers because pruning too many neurons in earlier layers can effectively remove many paths in layers further down.

The Triangular method is defined by two thresholds τ_{conv} and τ_{fc} for respectively the first `Conv` layer (i.e, layer 1) and the last FC layer (i.e., layer L). They represent the thresholds at the tip and the base of the triangle in Fig. 2(b). These thresholds are determined by the span of the weights in each of the two layers. Thus,

$$\begin{aligned} \tau_{conv} &= \sigma_{conv} * \delta_{conv} \\ \tau_{fc} &= \sigma_{fc} * \delta_{fc} \end{aligned} \tag{4}$$

where σ_{conv} is the span of the weights in the first `Conv` layer, defined in a similar way as for the Flat method, and it represents an upper bound on τ_{conv}, and δ_{conv} is a fraction between 0 and 1. Similarly, σ_{fc} is the span of the weights in the last FC layer and it represents an upper bound on τ_{fc}. Again, δ_{fc} is a fraction that ranges between 0 and 1. The values of σ_{conv}, σ_{FC}, δ_{conv} and δ_{FC} are parameters of the method that determine the resulting sparsity. We elaborate on these parameters in Section 5.2.

The thresholds of the remaining layers are dictated by the position of these layers in the network. Thus, the τ_l value for each layer l, $1 \le l \le L$, is given by:

$$\tau_l = \begin{cases} \tau_{conv} & \text{if} \quad l = 1 \\ \tau_{fc} & \text{if} \quad l = L \\ \dfrac{\tau_{fc} - \tau_{conv}}{L} \times (l-2) & \text{Otherwise} \end{cases} \tag{5}$$

3.3. Relative method

Several state-of-the-art CNNs, such as Inception-v3 [12] or AlexNet [3] no longer exhibit a linear expansion in the design of their layers. Rather, they exhibit a more complex structure in which bigger layers are replaced by multiple smaller layers that are grouped together. To this end, the Flat or Triangular sparsification methods may aggressively prune smaller layers but under sparsify larger layers.

The Relative method attempts to address this issue. It defines a unique threshold for each layer that is solely based on the distribution of the weights of that layer. The value of τ_l is picked such that it is the δ_lth *percentile* of the weights in layer l. That is, τ_l is a value such that approximately δ_l of the weights (in absolute value) are less than or equal to τ_l. Thus, our sparsification function (Eq. 1) results in approximately δ_l zeros for layer l. Thus, the values of δ_l, $0 \le \delta_l \le 1$, define the desired percentiles ($\delta_l \times 100\%$) of zero weights in the layers, and are thus parameters of the method.

3.4. Method parameters

The parameters of a sparsification method fall into two categories. The first is *model parameters*, which are determined by the underlying model. The second are *hyperparameters*, which must be selected to achieve the desired performance by the methods.

Model parameters depend on the distribution of the weights in the layers of a model. They are σ for the Flat methods and σ_{FC} and σ_{Conv} for the Triangular method (and none for the Relative method). These parameters are determined by the weights of the layers of a model. For instance, in the Triangular method, σ_{FC} and

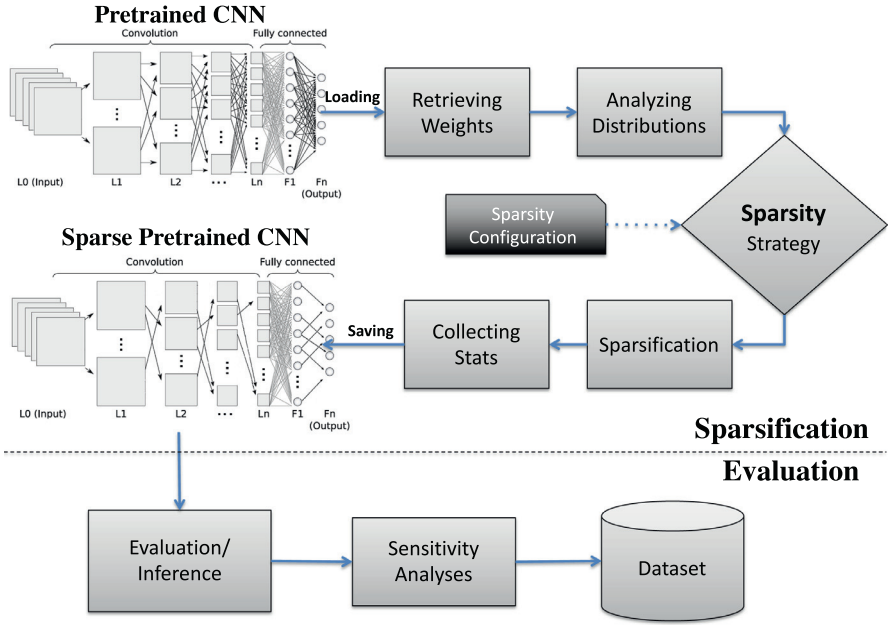


Fig. 3. Sparsification Workflow.

σ_{Conv} are determined by the weights of the last FC layer and the first Conv layer, as described earlier. Thus, model parameters vary from one model to another, but are determined by and can be extracted from the model. In Section 5.2 we give the values of the model parameters for the models we use in our evaluation.

In contrast, hyperparameters are model independent, but are method dependent. For example, the δ parameter of the Flat method is a hyperparameter that determines the degree of sparsity introduced in a model. Similarly are the δ_{conv}, δ_{fc} and δ_l values for the Triangular and Relative methods, respectively. Since our goal is to explore the trade-off between model sparsity and the resulting inference accuracy, we sweep thought the values of the hyperparameters.

4. Implementation

This section highlights the implementation of our sparsification methods. We implement the methods in TensorFlow [43]. Fig. 3 depicts the high level flow of our implementation. During sparsification, a pretrained CNN is loaded by TensorFlow, as shown on the upper left side of the figure. The weights of the model are retrieved using TensorFlow's `tf.train.saver` class. The histogram of the weights in each layer is generated, and the threshold value(s) of the sparsification methods used are determined (as described in Section 3). Once a method is applied, the sparsified weights are saved in a new sparsified version of the CNN, again using TensorFlow.

The evaluation of the sparsified model is also done using TensorFlow, as shown in the lower part of Fig. 3. The model is loaded and is then evaluated to capture its new inference accuracy. We determine the Top-5 accuracy on an NVIDIA's GeForce GTX 1060 with a host running Ubuntu 14.04, kernel v3.19 using ImageNet [2]).

5. Experimental evaluation

In this section we present our experimental evaluation of our sparsification methods. We first the evaluation metrics used. We then give the model parameters of the five models we use and also describe how the methods' hyperparameters are selected. We compare the performance of the three proposed sparsification methods, in terms of the degree of sparsity they achieve. We then present the results of our fine-tuning of the methods and describe how our results extend to quantized models. We finally compare the performance of our methods to existing work.

5.1. Metrics

We evaluate our methods using two key metrics. The first is the resulting *sparsity ratio*, as defined in Section 3. We use it as a proxy for the amount of memory and computation (i.e, number of MACs) savings possible when sparsity is exploited. This ratio represents an upper bound on the reduction in compute time because of the overhead of sparsifying. Nonetheless, existing work [21,30,31] shows that more benefit arises with more sparsity.

The second metric is the *drop in the Top-5 inference accuracy* of the resulting sparsified model. The acceptable drop in inference accuracy is highly application dependent. In our evaluation, we set a threshold for the drop in the Top-5 accuracy to an absolute 5% drop. We explore sparsification that result in the largest model sparsity within this threshold.

5.2. Method parameters

In this section, we give the method parameters for the models we use in our evaluation. We first give the model parameters and then describe the range of values of hyperparameters we sweep through. This range is determined experimentally–we find that going beyond the bounds of this range either does not sufficiently sparsify or adversely affects a model's accuracy.

Tables 1 and 2 give the model parameters and the range of hyperparameters swept through of for each model/method. The Relative method has no model parameters and thus does not appear in Table 1.

Table 3 gives the range of the resulting sparsification thresholds that result from the sweep of the hyperparameters. For the Relative method, the same value of δ_l can potentially result in a different value of τ_l for each layer. Given the sizes of the models, it is not feasible to succinctly report the resulting values of τ_l.

Table 1
Model parameters (σ).

Model	Flat	Triangular	
	σ	σ_{Conv}	σ_{FC}
Inception-v3	0.2	1.06	0.4
MobileNet-v1	6.33	29.33	6.33
AlexNet	0.22	3.04	0.26
VGG-16	0.05	1.27	0.09
ResNet-v2	0.12	1.58	0.79

Table 2
Hyperparameters (δ).

Models	Flat		Triangular				Relative	
	$\leq \delta \leq$		$\leq \delta_{Conv} \leq$		$\leq \delta_{FC} \leq$		$\leq \delta \leq$	
Inception-v3	0.2	0.8	0.004	0.009	0.25	0.88	0.1	0.8
MobileNet-v1	0.001	0.01	0.08	0.15	0.25	0.5	0.1	0.8
AlexNet	0.03	0.7	0.02	0.03	0.25	0.55	0.1	0.9
VGG-16	0.03	0.68	0.05	0.12	0.19	0.41	0.1	0.8
Resnet-v2	0.09	0.84	0.01	0.1	0.12	0.31	0.1	0.8

Table 3
Sparsification thresholds (τ).

Models	Flat		Triangular			
	$\leq \tau \leq$		$\leq \tau_{Conv} \leq$		$\leq \tau_{FC} \leq$	
Inception-v3	0.005	0.16	0.005	0.01	0.1	0.45
MobileNet-v1	0.01	0.08	0.07	0.14	0.62	1.25
AlexNet	0.06	0.15	0.06	0.09	0.06	0.14
VGG-16	0.001	0.03	0.06	0.15	0.01	0.03
Resnet-v2	0.01	0.07	0.01	0.15	0.09	0.24

Table 4
Model sparsity (S_m).

Models	Flat		Triangular		Relative	
	$\leq S_m \leq$		$\leq S_m \leq$		$\leq S_m \leq$	
Inception-v3	0.05	0.92	0.14	0.76	0.1	0.8
MobileNet-v1	0.09	0.78	0.01	0.68	0.1	0.8
AlexNet	0.05	0.92	0.05	0.88	0.1	0.9
VGG-16	0.04	0.91	0.03	0.83	0.1	0.8
Resnet-v2	0.09	0.84	0.09	0.83	0.1	0.8

Finally, Table 4 shows the range of sparsity ratios that are obtained by using the corresponding range of thresholds. The table shows that we sweep through a wide range of sparsity values, allowing us to explore the trade-off between sparsity and the resulting model inference accuracy.

5.3. Sparsification results

Fig. 4 shows the inference accuracy as a function of the introduced sparsity by each method for the models we evaluate. We do not show data points that result in more than 30% drop in the Top-5 accuracy since such a sparsified model with such large drop in accuracy is not likely useful. The figure reflects that significant sparsity can be obtained with a small reduction in inference accuracy. With less than 5% reduction in accuracy, we gain 51% sparsity (2.04× compression factor), 50% (2×), 62% (2.63×), 70% (3.33×), and 73% (3.7×) for the models. This validates our approach.

Further, the figure reflects that the Relative method outperforms the other two methods for Inception-v3, VGG and ResNet-V2, but the Triangular method performs best for MobileNet-v1 and Alexnet. This is likely due to the structure of the models. For instance, in Fig. 4a, it is evident that introducing more sparsity gradually reduces the model accuracy and that the Triangular and Flat method are more prone to adversely affect the model's accuracy.

Table 5
Best found hyperparameters.

Models	Flat (δ_l)	Triangular (δ_{conv}, δ_{FC})	Relative (δ_l)
Inception-v3	0.68	(0.009, 0.41)	0.5
AlexNet	0.43	(0.22, 0.81)	0.68
MobileNet-v1	0.007	(0.1, 0.34)	0.41
VGG-16	0.51	(0.08, 0.3)	0.68
ResNet-v2	0.41	(0.06, 0.23)	0.62

Inception-v3, has a complex `Conv` groups, each comprised of multiple 3×3 convolutions and 1×1 s instead of 5×5 ones in the earlier Inception-v1 (GoogleNet [9]).

Thus, it is reasonable that the Triangular and Flat method fall short of sparsifying the optimal range of weights compared to the Relative method. Thus, as we pass the 30% sparsity threshold, the former two methods tend to over-sparsify certain segment of the weights which leads to a drop in overall accuracy. The above observation can also be seen in Fig. 4c. In contrast, MobileNet-v1 and Alexnet have a gradual linear increase in the size of their convolution layers, making the Triangular method more effective.

5.4. Method hyperparameters

Table 5 reports the best value for each hyperparameter for each method, defined as that which results in the largest sparsity ratio with no more than a 5% loss in Top-5 accuracy. The table indicates that that best value of the hyperparameters depends on the model being sparsified. Thus suggests the need for automatic fine-tuning of the hyperparameters. This can be done, for example, through user feedback during deployment. We leave the exploration of this fine-tuning to future work.

5.5. Sparsification method selection

The above results show that our sparsification methods are effective in introducing a significant percentage of sparsity into a pretrained model. Nonetheless, the choice of the best method varies from model to model. To this end, we provide a simple classifier which can be paired with our framework to identify the right type of sparsification method given a pretrained model as input. The Relative and Triangular methods consistently outperform the Flat method. Thus, we limit the classifier to be a binary selection between these two methods.

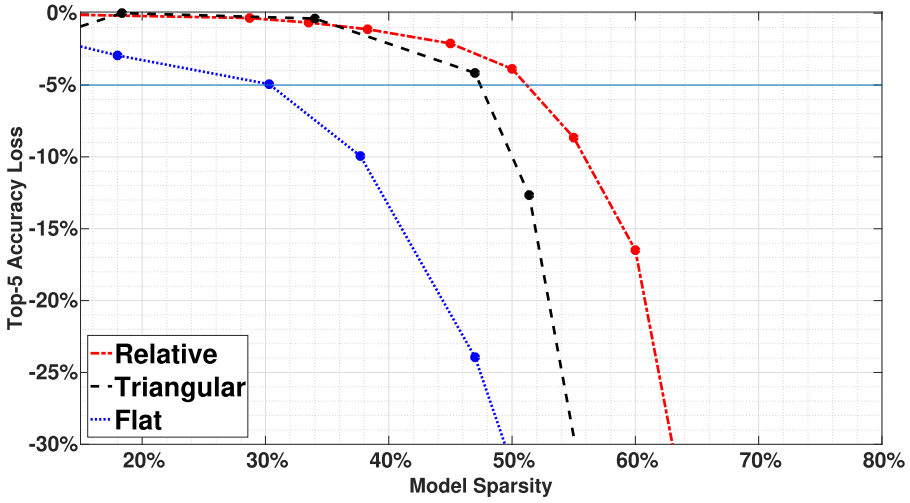
We define a classifier function named, *IsMonotonic?*, to decide whether Relative or Triangular sparsification is better for a given CNN. We elect to make the classifier as fast and straightforward as possible so it can be deployed on-the-fly when a pretrained model is to be executed on a target platform. Thus, we design the binary classifier to simply examine whether the given CNN layers exhibit monotonic increase in the number of their weights. If they do, then the Triangular method is used. Otherwise, the Relative method is used. Thus, the *IsMonotonic?* function returns true if:

$$\forall \;\; l_i, l_j \mid l_i \leq l_j \; then \; |w_{li}| \leq |w_{lj}| \tag{6}$$

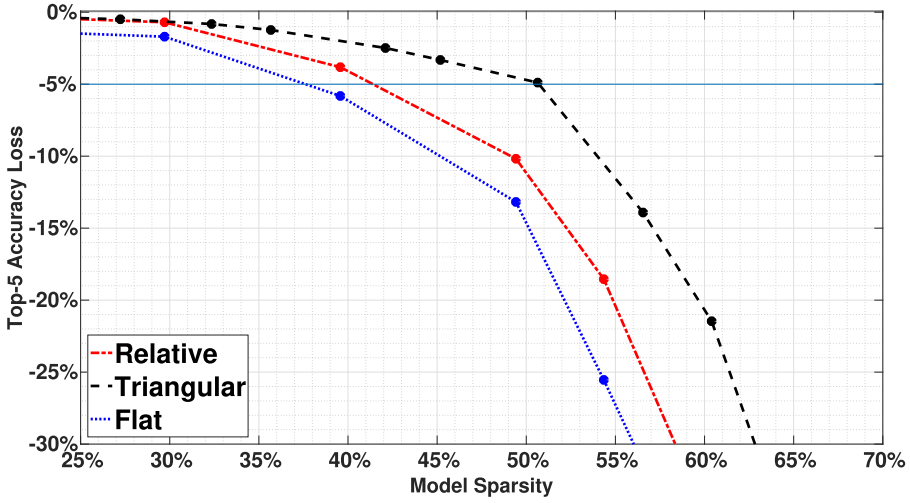
where l_i and l_j are layer numbers, ranging from the first `Conv` layer to the last FC layer and $|w_{li}|$ and $|w_{lj}|$ are the number of weights in layers l_i and l_j, respectively. We observe that the above function is a simple yet effective classifier to identify the right method of sparsification. Table 6 shows the results of applying the classifier to the models we use.

5.6. Fine-tuning the methods

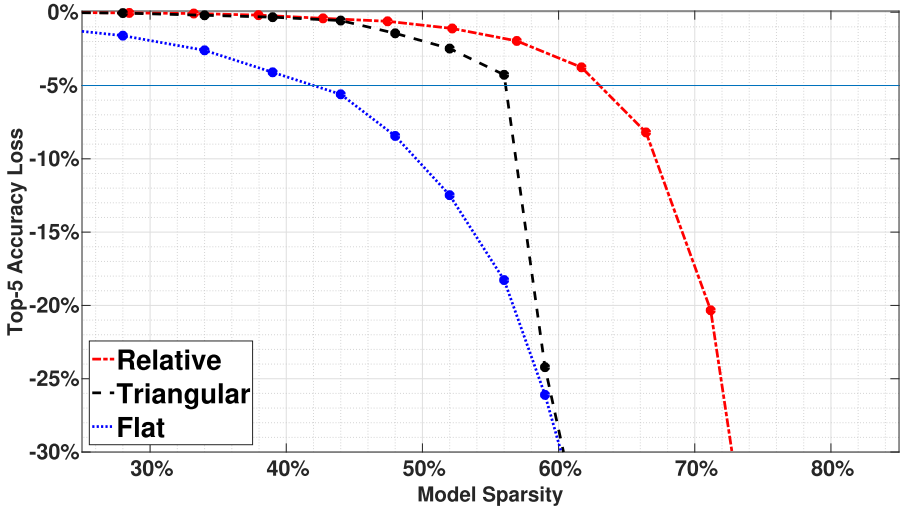
We explore what can be achieved by some fine-tuning of our methods, still with no retraining, in order to gain more sparsity.



(a) Inception-v3

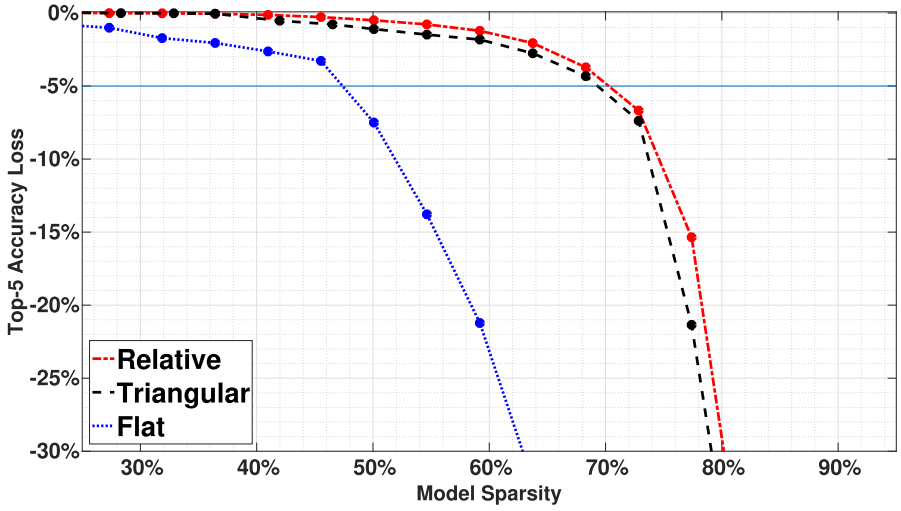


(b) MobileNet-v1

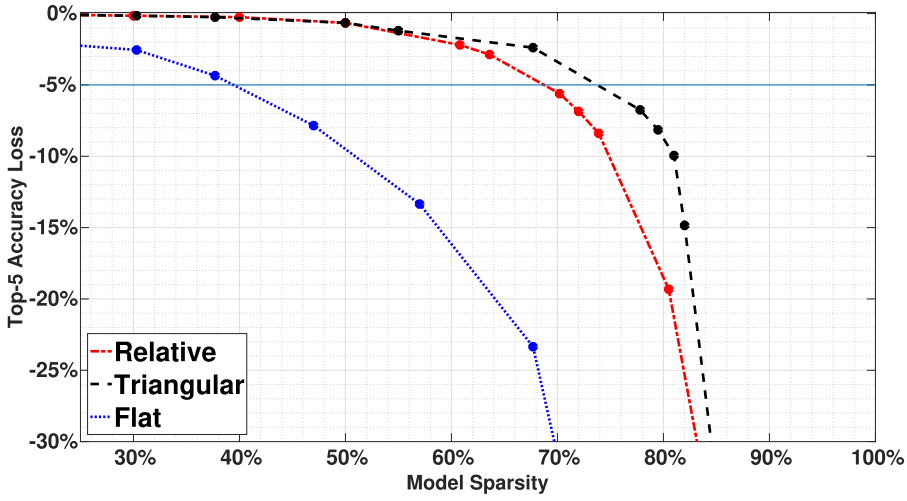


(c) ResNet-v2

Fig. 4. Sparsity-Accuracy Trade-off for Our Three Proposed Sparsification Methods.



(d) VGG-16



(e) Alexnet

Fig. 4. Continued

Table 6
Identifying the sparsification method.

CNN Name	Classifier		Sparsification method
AlexNet	IsMonotonic?	True	Triangular
MobileNet-v1			Triangular
VGG-16			Relative
Inception-v3		False	Relative
ResNet-v2			Relative

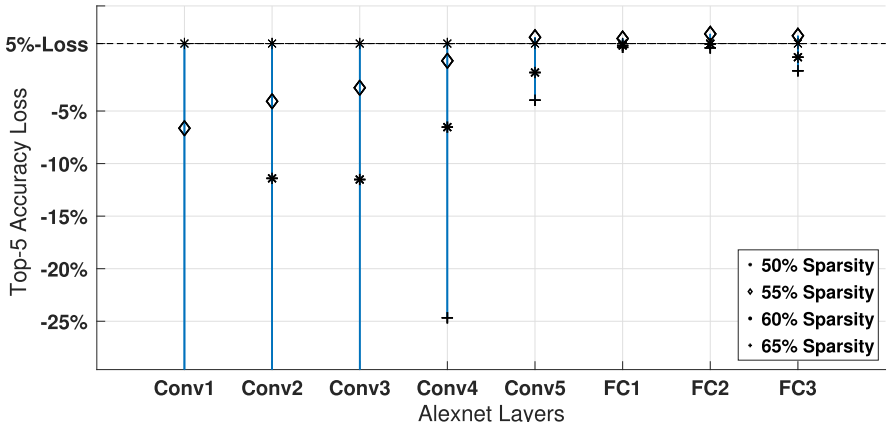
We do so to determine the effectiveness of our on-the-fly methods, since the fine-tuning is not likely feasible in out deployment scenario. Thus, the goal of the fine-tuning is to determine how much more sparsity can be achieved, still with no retraining.

We focus on the Relative method and start with a baseline sparsity. We vary the degree of sparsity of each layer, one at a time, around the base sparsity, attempting to maintain no more than 5% drop in inference accuracy. We refer to this process as *layer sensitivity analysis*.

We build a sparsity configuration vector as $SP(l)$ where each element l $(1 \leq l \leq L)$ corresponds to a tuple (O, D) that controls the sparsification and the degree of sparsification of a layer l, respectively. O is a binary parameter representing whether or not layer l is sparsified and D specifies the desired degree of sparsity the layer. We force each layer to have a number of higher sparsity ratios, while having the remaining layers at a baseline. This determines how robust a layer can be in handling different sparsity ratios compared to the other layers. The results of this analysis identify sparsity configurations for layers that can be more sparsified without further loss in inference accuracy. Our baseline is the best sparsity ratio found corresponding to a 5% loss in Top-5 accuracy from Fig. 4.

AlexNet. The sensitivity results for AlexNet are shown in Fig. 5 a. The horizontal axis represents that layers sparsified in the network. The vertical axis represents the drop in the Top-5 inference accuracy. The baseline sparsity is set to 70%. The vertical bars represent the span in the inference accuracy of the model when δ_l of the Relative method is set to 50%, 55%, 60%, and 65% (the symbols $\star, *, +,$ *and* $\Diamond$, respectively).

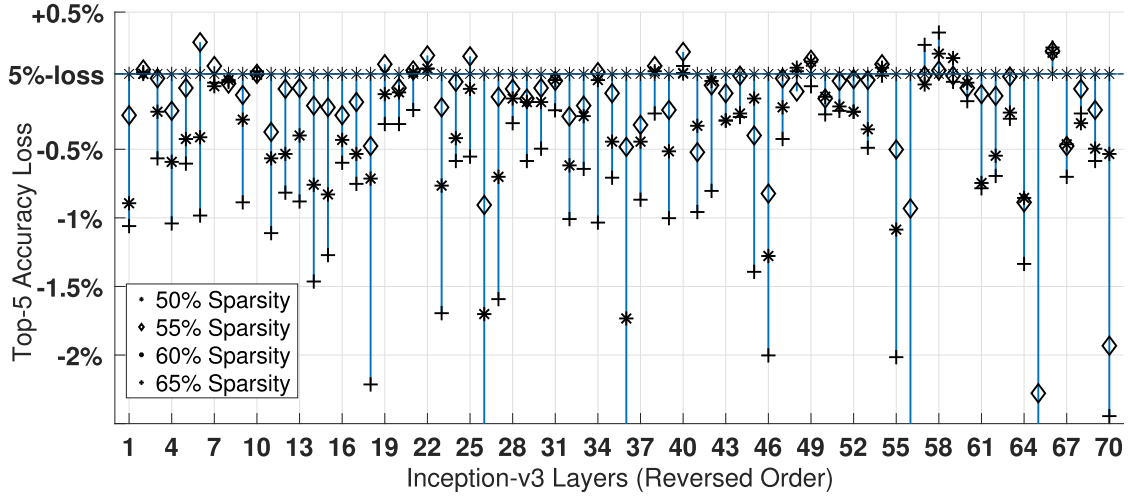
The figure shows that it is possible for some layers, particularly larger ones, to have higher sparsity, while smaller layers are more sensitive to sparsification and benefit from having lower sparsity. Nonetheless, there is a gain of 8% in overall model sparsity and the fine-tuned layers are reported in Table 5b.



(a) AlexNet Sensitivity Analysis

Layers	#Par	%Sparsified
Conv1	23 K	10%
Conv2	307K	35%
Conv3	663K	35%
Conv4	1.3M	35%
Conv5	884K	35%
FC1	26M	85%
FC2	16M	85%
FC3	4M	73%
Total	**50M**	**81.1%**

(b) AlexNet Fine-tuning

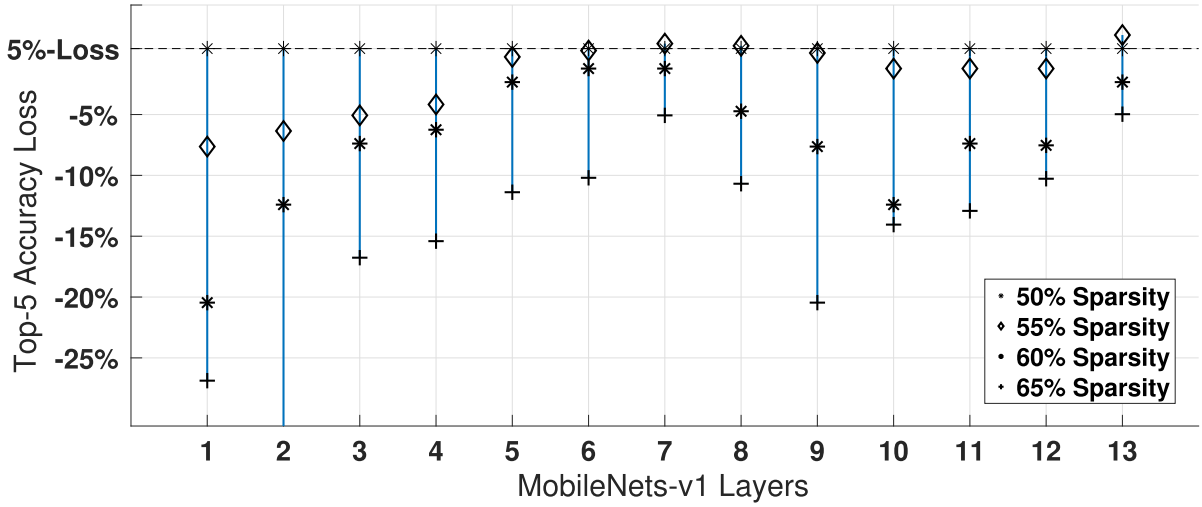


(c) Inception-v3 Sensitivity Analysis

Layers in reversed: FC=1...Conv1=95	Maximum Tolerated Sparsity
Rest of the layers	50%
6, 7, 19, 25, 34	55%
21, 38, 54, 59	60%
2, 10, 22, 40, 48, 57, 58, 66	65%

(d) Inception-v3 Fine-tuning

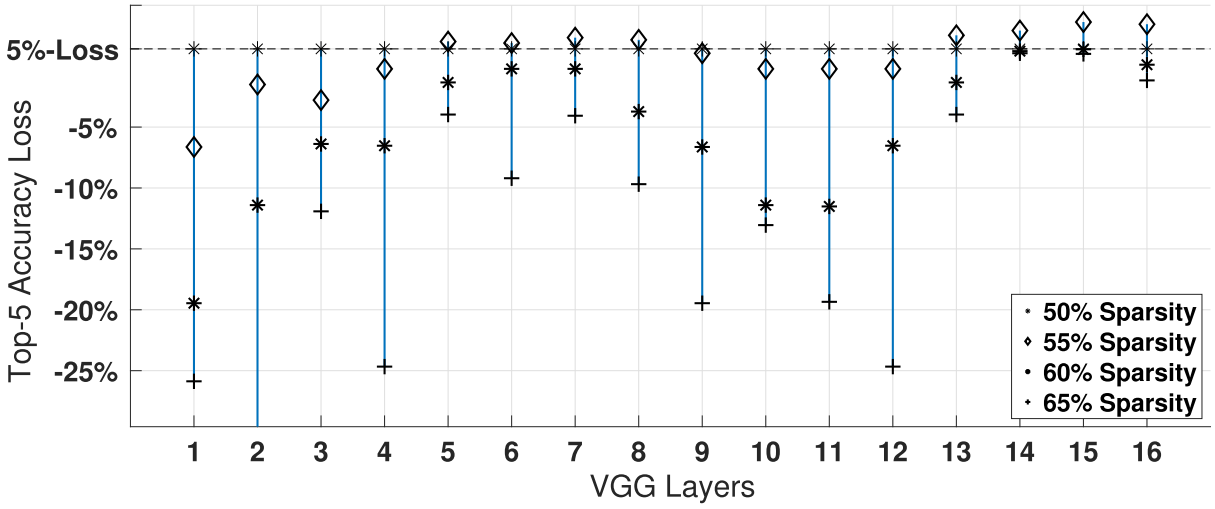
Fig. 5. Sensitivity analysis and fine-tuning results.



(e) MobileNet-v1 Sensitivity Analysis

Layers	Maximum Tolerated Sparsity
1, 2, 3, 4, 5	30%
9, 10	50%
6, 7, 8, 11	55%
12, 13	65%

(f) MobileNet-v1 Fine-tuning

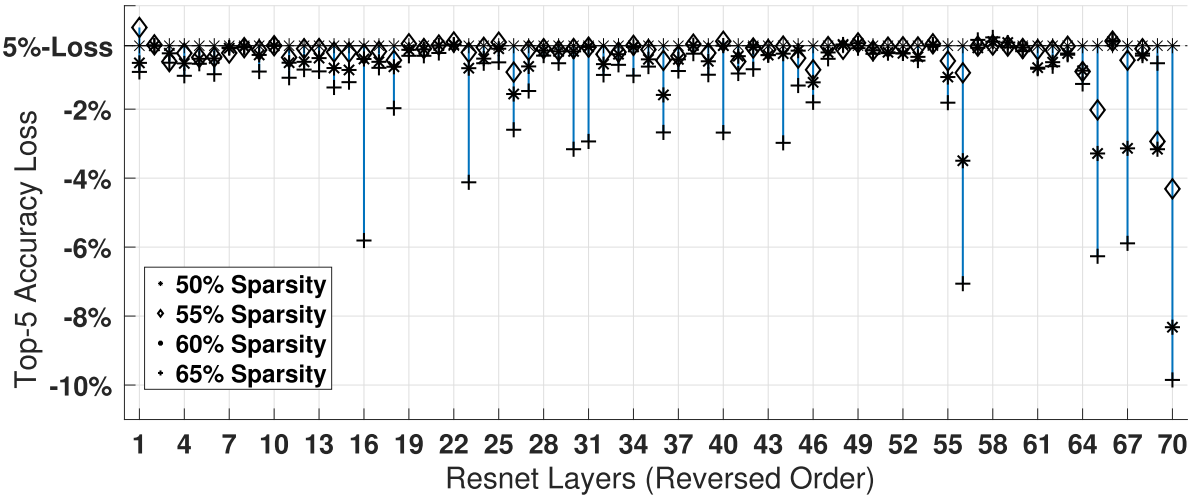


(g) VGG-16 Sensitivity Analysis

Layers	Maximum Tolerated Sparsity
1, 2, 4	30%
3, 5, 6, 8, 10	50%
7, 9, 11, 12	60%
13, 14, 15, 16	75%

(h) VGG-16 Fine-tuning

Fig. 5. Continued



(i) ResNet-v2 Sensitivity Analysis

Layers in reversed: FC=1...Conv1=152	Maximum Tolerated Sparsity
Rest of the layers	40%
16, 17, 19, 25, 34	55%
20, 54, 59	60%
1, 2, 4, 22, 40, 48, 66	65%

(j) ResNet-v2 Fine-tuning

Fig. 5. Continued

Inception-v3. Fig. 5c presents the fine-tuning results for this network. The horizontal axis represents the layer number in the model, for last 70 layers. We did not sparsify the first 25 layers, as the sparsity gain was limited and they are very sensitive to even a minimal pruning. The vertical axis is the loss in the inference accuracy. The horizontal line is the accuracy baseline of 50% model sparsity. The vertical bars represent the span in the inference accuracy of the model when the Relative sparsity is set to 50%, 55%, 60%, and 65%. We observe that, interestingly, smaller layers before the last `logit` layer tend to be more robust with more sparsification. For instance, Layers 2, 10, 22, 40, 58, and 66 can handle up to 65% Relative sparsity with the inference accuracy remaining well within our baseline. Moreover, as many layers can be further sparsified with no loss of accuracy, we attempt to fine-tune the sparsity of various layers in Table 5d. Thus, A total sparsity ratio of 4% is achieved, with equally good normalized accuracy Relative to the baseline.

MobileNet-v1. Fig. 5e presents the fine-tuning results for this network. The horizontal axis represents the layer number of the model for all 13 convolutional layers. As expected, the early smaller layers are more sensitive to pruning and thus, the fine-tuned ratio is the lowest. The majority of the parameters come at the last three layers and they are more robust in handling higher sparsity ratio. Overall, fine-tuning gain 3%. Table 5f shows the final additional sparisty gained for various layers.

VGG-16. This network has 16 layers from which the last three are FC layers. These layers tend to be more tolerant to sparsifying. The horizontal line is the accuracy baseline corresponding to a 5%-loss accuracy. Fig. 5g and Table 5h indicate that the final 4 layers can be sparsified up to 75% with no further loss in accuracy, while the earlier layers are tolerant to only 30%.

ResNet-v2. ResNet-v2 consists of many deep stacked residual units in a sequence of the weights, a Batch-norm (BN) [44], a ReLU, and again weight plus BN. This structure is beneficial in reaching higher prediction accuracy and faster convergence. Overall, ResNet-v2 has 152 layers and the majority of the layers are a 3×3 convolutions. Our sensitivity analyses gained only 2% further sparsity by fine-tuning the model.

In summary, the sensitivity analysis results show that it is possible to gain more sparsity by some fine-tuning. Specifically, additional sparsity of 8%, 4%, 3%, 2%, and 5% can be achieved for Alexnet, Inception-v3, Mobilenet-v1, ResNet-v2, and VGG, respectively. On the one hand, these gains come at the expense of an exploration of different sparsity ratios for the layers and thus more computations. This makes such fine-tuning not feasible in the deployment context we explore. On the other hand, the gains are not significant, indicating that our on-the-fly methods likely achieve much of the sparsity that can be realized without retraining, yet remain computationally efficient for on-the-fly use.

5.7. Quantized models

There are two main motivations behind quantization of CNNs. First, using 8-bit integers instead of 32-bit floating point values shrinks the size of the model. Second, it reduces the computational needs for inference, which is especially important on mobile platforms. Several state-of-the-art CNN models have been quantized to 8-bit integer to achieve these benefits [21,45].

Thus, we also evaluate our methods for quantized models. Specifically, we reproduce our earlier results using quantized versions of our models to determine if our sparsification methods remain effective. We use TensorFlow's TF8

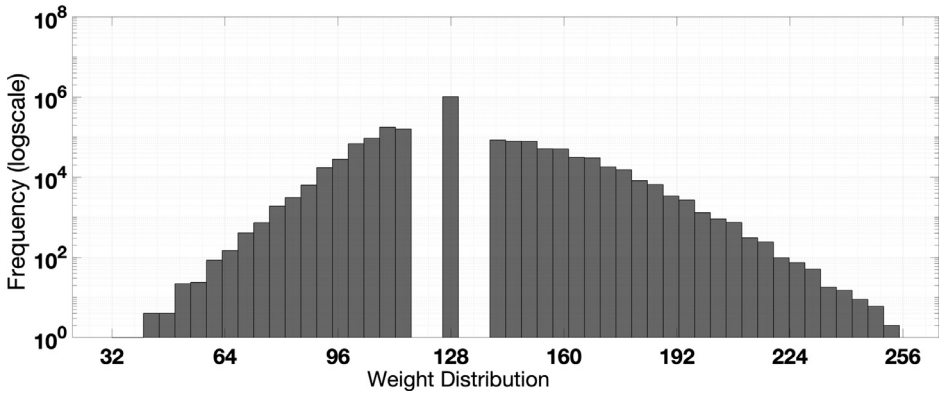


Fig. 6. Inception-v3: distribution of FC's weights. Note that *Y* axis is a log-scale.

Table 7
Quantization results for all sparsified models (Q=Quantized, S=Sparsified.

Model	Top-5 normalized accuracy
Inception-v3-SQ	0.951
MobileNet-v1-SQ	0.9506
AlexNet-SQ	0.952
VGG-16-SQ	0.9517
ResNet-v2-SQ	0.9504

quantization tool, `graph_transform:transform_graph` [46], enabling the `fold_constant`, `strip_unused_nodes` and `fold_batchnorms` optimizations to quantize our models.

The current implementation of TensorFlow's quantization tool uses unsigned 8-bit values and the representations of the zero value is neither 0 nor unique (i.e., depending on the weights being quantized, the value of zero may be anywhere in the range of 0–255). This makes it difficult to exploit the resulting sparsity. Thus, we modified TensorFlow's implementation of quantization to ensure that floating-point zeros always translate to the quantized value 128. We do so by adjusting the range of the weights (only for the purpose of quantization) so that the range is symmetric around 0, ensuring that 0 maps to 128.

As an example, Fig. 6 depicts the last fully-connected layer's weights of Inception-v3 after applying quantization followed by sparsification. The shape of the distribution, after quantization, still resembles the original distribution depicted in Fig. 1a. Further, all zero values are represented by 128.

Table 7 shows the Top-5 normalized inference accuracy for the sparsified and quantized models. The table shows that the quantization does not incur further loss in accuracy and the models are all within the 5% accepted threshold. These accuracies are similar to the best found sparsity configuration in Fig. 4.

5.8. Comparison to existing work

To the best of authors' knowledge, our proposed work is unique in that it introduces sparsity in a model without any retraining of the model. This stands in contrast to existing work that relies on retraining to fine tune the remaining non-zero weights for the introduced sparsity. This makes it difficult to make a quantitative comparison to existing work. Nonetheless, we present a comparison of our introduced sparsity and the resulting inference rates to existing work. The purpose of this comparison is not to directly compare our retraining-free methods to methods that rely on training, but rather to assess the fraction of sparsity that can be obtained without retraining relative to existing work.

Sun et al. [19] propose a correlation-based iterative pruning method for a customized VGG-like network (ConvNet) for which they: (1) prune several connections between layers, (2) initialize a new network with the modified structure, and (3) iteratively retrain the remaining parameters to compensate the loss in accuracy. A Top-5 inference accuracy of 98.95% is reported for ConvNet trained using 300K face images of the DeepID2+ dataset. The resulting sparsity ratio is 88%. This definitely demonstrates the benefit of retraining. In contrast, for our Tensorflow implementation of VGG-16, we achieve 70% sparsity with a 5% loss in accuracy without retraining.

Han et al. [21] also iteratively prune and retrain CNNs to compensate the loss of accuracy using various regularization techniques (i.e., L1 and L2). Their approach results in over 90% sparsity with almost no drop in the Top-5 accuracy for LeNet, AlexNet, and VGG-16. However, this is achieved at the cost of hours of retraining using the original training data (e.g., 175 extra hours for AlexNet [21]), which renders the approach unusable in our deployment context. In contrast, we achieve 73% sparsity for AlexNet with 5% drop in accuracy, but we do so on-the-fly. Thus, we are able to achieve a significant portion of the sparsity achieved with retraining, but with a fast method that is suitable for our deployment context.

6. Concluding remarks

We explored sparsification without retraining of CNNs and proposed three methods for doing so. This approach to sparsification is particularly important in some deployment contexts in which the access to training data is not possible and the retraining represents a significant computational overhead. In these contexts, sparsification must be done on-the-fly when inference is performed and thus, must be fast.

We experimentally evaluated our proposed methods and showed that they can result in up to 73% sparsity with less than 5% drop in inference accuracy. Our evaluation also shows that no single method works best across the models. Thus we also propose and evaluate a simple classifier that is able to determine which method to use for a given model. We further demonstrate that our approach holds the same benefit when the model is quantized.

Finally, our evaluation showed that it is possible to fine-tune the methods to further gain sparsity with no additional drop in inference accuracy. However, the gains in sparsity are minimal (less than 8%) and come at significant compute cost. This suggests that our on-the-fly methods attain much of the sparsity than can be attained with no retraining while remaining computationally efficient.

There are several key directions for future work. The first is to explore heuristics for selecting and/or fine-tuning, the hyper-parameters of the sparsification method, possibly based on the model. The second is to realize the benefit of the sparsity in the model's implementation on a mobile device, in a typical library, such as NNlib [47], which offloads neural networks operations from TensorFlow to Qualcomm's Hexagon-DSP [48].

The introduction of zeros in a model's weights can only benefit performance/energy if their presence is exploited. In particular, we also wish to address the additional challenge that arises in a quantized model, which stems from the fact that the representation of a zero is neither the value zero nor unique. Thus, another direction for future work is to explore different quantization approaches to address this shortcoming.

Declaration of Competing Interest

All authors have equally participated in the work described in the paper. The authors have no financial or non-financial conflict of interest in the work presented in this paper. This work was supported by grants from Mitacs and Qualcomm. This support does not represent a conflict of interest. For transparency, we have an acknowledgement section at the end of the paper to acknowledge the support received.

Acknowledgments

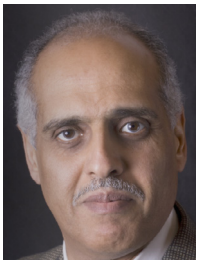
This work was supported by grants from Qualcomm, Inc. Canada and Mitacs Canada, Grant number IT09525.

References

[1] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: a large-scale hierarchical image database, in: Proceedings of Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255.
[3] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.
[4] O.M. Parkhi, A. Vedaldi, A. Zisserman, et al., Deep face recognition, in: British Machine Vision Conference (BMVC), 2015, p. 6.
[5] K. Zhang, L. Tan, Z. Li, Y. Qiao, Gender and smile classification using deep convolutional neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2016, pp. 34–38.
[6] A. Işın, C. Direkoğlu, M. Şah, Review of mri-based brain tumor image segmentation using deep learning methods, Proc. Comput. Sci. 102 (2016) 317–324.
[7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[8] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv:1409.1556 (2014).
[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[10] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[11] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, Squeezenet: alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size, arXiv:1602.07360 (2016).
[12] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2818–2826.
[13] C. Szegedy, S. Ioffe, V. Vanhoucke, A.A. Alemi, Inception-v4, inception-resnet and the impact of residual connections on learning, in: Thirty-First AAAI Conference on Artificial Intelligence, 2017.
[14] Y. LeCun, et al., Lenet-5, convolutional neural networks, URL: http://yann.lecun.com/exdb/lenet 20 (2015).
[15] V. Sze, Y.-H. Chen, T.-J. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329.
[16] Y. LeCun, J.S. Denker, S.A. Solla, Optimal brain damage, in: Advances in Neural Information Processing Systems, 1990, pp. 598–605.
[17] B. Hassibi, D.G. Stork, Second order derivatives for network pruning: optimal brain surgeon, in: Advances in Neural Information Processing Systems, 1993, pp. 164–171.
[18] X. Dong, S. Chen, S. Pan, Learning to prune deep neural networks via layer–wise optimal brain surgeon, in: Advances in Neural Information Processing Systems, 2017, pp. 4857–4867.
[19] Y. Sun, X. Wang, X. Tang, Sparsifying neural network connections for face recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 4856–4864.
[20] M. Mathew, K. Desappan, P. Kumar Swami, S. Nagori, Sparse, quantized, full frame CNN for low power embedded devices, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp. 11–19.
[21] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding, arXiv:1510.00149 (2015).
[22] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, W.J. Dally, Exploring the granularity of sparsity in convolutional neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp. 13–20.
[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, EIE: Efficient inference engine on compressed deep neural network, in: ACM/IEEE International Symposium on Computer Architecture (ISCA), 2016, pp. 243–254.
[24] W. Wen, C. Wu, Y. Wang, Y. Chen, H. Li, Learning structured sparsity in deep neural networks, in: Advances in neural information processing systems, 2016, pp. 2074–2082.
[25] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, W.J. Dally, Exploring the regularity of sparse structure in convolutional neural networks, arXiv:1705.08922 (2017).
[26] S. Anwar, K. Hwang, W. Sung, Fixed point optimization of deep convolutional neural networks for object recognition, in: Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015, pp. 1131–1135.
[27] V. Niculae, M. Blondel, A regularized framework for sparse and structured neural attention, in: Advances in Neural Information Processing Systems, 2017, pp. 3338–3348.
[28] L. Chen, M. Zhou, W. Su, M. Wu, J. She, K. Hirota, Softmax regression based deep sparse autoencoder network for facial emotion recognition in human-robot interaction, Inf. Sci. (Ny) 428 (2018) 49–61.
[29] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, arXiv:1503.02531 (2015).
[30] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, W.J. Dally, Scnn: an accelerator for compressed-sparse convolutional neural networks, in: ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 27–40.
[31] S. Baluja, D. Marwood, M. Covell, N. Johnston, No multiplication? no floating point? no problem! training networks for efficient inference, arXiv:1809.09244 (2018).
[32] R. Tang, A. Adhikari, J. Lin, Flops as a direct optimization objective for learning sparse neural networks, arXiv:1811.03060 (2018).
[33] X. Xu, M.S. Park, C. Brick, Hybrid pruning: thinner sparse networks for fast inference on edge devices, arXiv:1811.00482 (2018).
[34] Y. Zhang, D. Shi, J. Gao, D. Cheng, Low-rank-sparse subspace representation for robust regression, in: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.
[35] C. Musco, D. Woodruff, Is input sparsity time possible for kernel low-rank approximation? in: Advances in Neural Information Processing Systems, 2017, pp. 4435–4445.
[36] W. Liu, X. Shen, I. Tsang, Sparse embedded *k*-means clustering, in: Advances in Neural Information Processing Systems, 2017, pp. 3319–3327.
[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.
[38] S. Srinivas, A. Subramanya, R. Venkatesh Babu, Training sparse neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp. 138–145.
[39] X. Yu, T. Liu, X. Wang, D. Tao, On compressing deep models by low rank and sparse decomposition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 7370–7379.
[40] S. Han, J. Pool, J. Tran, W. Dally, Learning both weights and connections for efficient neural network, in: Advances in Neural Information Processing Systems, 2015, pp. 1135–1143.
[41] Q. Huang, K. Zhou, S. You, U. Neumann, Learning to prune filters in convolutional neural networks, in: 2018 IEEE Winter Conference on Applications of Computer Vision (WACV), 2018, pp. 709–718.
[42] A.H. Ashouri, T.S. Abdelrahman, A. Dos Remedios, Fast on-the-fly retraining-free sparsification of convolutional neural networks, arXiv:1811.04199 (2018).
[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning, in: Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265–283.
[44] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, arXiv:1502.03167 (2015).
[45] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 2704–2713.
[46] Google, TensorFlow Git, (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/graph_transforms). [Online; accessed Feb-2018].

[47] Qualcomm, Hexagon NNlib, (https://source.codeaurora.org/quic/hexagon_nn/nnlib). Accessed: 2019-05.
[48] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, R. Maule, Hexagon dsp: an architecture optimized for mobile multimedia and communications, IEEE Micro 34 (2) (2014) 34–43.

Amir H. Ashouri is a Postdoctoral researcher at ECE Department of University of Toronto. His research interests are Accelerating deep learning applications, compiler optimizations, and Automatic Tuning. Prior to joining the U of T, he completed his M. Sc. (2012) and Ph.D. (2016) at Polytechnic University of Milan where he defended his Ph.D. thesis about automatic tuning of compilers using Machine Learning.



Tarek Abdelrahman received his Ph.D. degree in Computer Science and Engineering from the University of Michigan at Ann Arbor in 1989. He is currently Professor of Electrical and Computer Engineering and of Computer Science at the University of Toronto, Toronto, Ontario, Canada. His research interests are in the areas parallel systems, parallelizing and optimizing compilers, parallel programming models and run-time support. Dr. Abdelrahman is a senior member of IEEE, a senior member of ACM and a member of USENIX.

Alwyn Dos Remedios received his BASc in Computer Engineering at the University of Toronto in 1996. Alwyn has been working over 20 years in the computer field working at IBM, Intel, AMD and currently holds a position of Principal Engineer/Manager at Qualcomm Canada ULC as part of the Machine Learning System Department working on mobile machine learning accelerators.