Compression via Partial Pseudo-Randomization of Convolutional Neural Networks Under High Memory Constraints

Florent Crozet
STMicroelectronics
Grenoble, France
email: florent.crozet@st.com

Stéphane Mancini
Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA
Grenoble, France
email: stephane.mancini@univ-grenoble-alpes.fr

Marina Nicolas
STMicroelectronics
Grenoble, France
email: marina.nicolas@st.com

***Abstract*—With the proliferation of convolutional neural network (CNN)-based computer vision solutions, computing inference on smart sensors has become a challenge. The inference of CNN is difficult to embed in such tiny devices due to the constraints on memory. To address this challenge, we propose a compression method able to reduce the number of weights to store in a structured way, so that the gain in the number of weights comes with a gain in the number of computations at inference. Our solution is based on the replacement of the convolutional filters by a linear combination of some stored filters and a set of seeds corresponding to pseudo-random generated filters. During the inference, pseudo-random number generators are used to compute the non-stored filters, thanks to the associated seeds. On the other side, the linear combination allows mutualizing partly the cost of convolutions. We show that further exchanging memory for a small logic cost to generate the pseudo random filters allows to decrease the number of weights significantly, on several state-of-the-art networks without sacrificing the accuracy. For example, applying this method to CNNs like ResNet50 leads to a compression factor of 2.5 for less than 5% accuracy drop. Furthermore, our method is compatible with compression methods targeting the precision of the weights to store, namely quantization. This gives room to further increase compression gain on specific implementation platforms.**

Keywords*—*Convolutional Neural Network compression, pseudo-random number generators

I. Introduction

Computer vision applications widely use convolutional neural networks to achieve several vision tasks. The accuracy of Convolutional Neural Network (CNN) drives the development of these applications, but the memory usage is rarely taken into account leading to a difficult deployment on embedded devices.

To improve their performance, CNNs keep increasing the number of weights they use. With ResNet50 [1] and its 25M weights or ConvNeXt-XL [2] and its 350M weights, the goal is to get the best accuracy, but without taking into account any other constraint, such as memory usage. For an embedded device, the memory and the computational resources are the key factors impeding the deployment of state-of-the-art CNNs in IoT devices.

As well as occupying a significant part of circuit die surface, the memory also has a high energy consumption due to the memory accesses. The high number of weights to store to achieve a CNN inference leads to use a device with high memory capabilities. But smart sensors used for computer vision applications are rather tiny, with limited memory capability and power consumption.

To address the problem, different compression algorithms have been proposed. Most methods either reduce the memory requirement by reducing the precision of the weights [3] or by reducing the number of weights [4]. Several methods just compute what is possible to do and the accuracy loss, but do not speak about memory, like unstructured pruning where the goal is just to get a sparse CNN. Sparse neural network compression has the drawback that the decompression of the CNN produces a tensor requiring several operations with many zeros processing convolutions.

In this article, we propose a new compression method for CNNs where some weights are stored in the memory, while the others are generated from stored seeds in a pseudo-random process during the inference. Replacing memory access by on-the-fly generation with pseudo-random generators actually leads to a lower consumption. The identification of the weights to store and the weights to regenerate relies on a dimensionality reduction method, the Principal Component Analysis (PCA). The PCA allows the decomposition of each convolutional tensor in the CNN in a linear combination, with an ordering of vector importance. Most significant vectors are stored while the least significant are pseudo-randomly generated. This compression method comes not only with a memory gain, but also with a gain in hardware logic as the original convolution can be replaced by a double convolution solution.

The article starts with a brief overview of convolutional neural network compression methods and the usage of random weights in neural networks. Then, our compression method with the randomization is described in Section 3. The impact on inference is described in Section 4. The Section 5 discusses the performance obtained when our method is applied to several convolutional neural networks. Finally, the article ends with some perspectives to capitalize further on this new compression method for CNNs.

II. RELATED WORK

Our work is at the intersection of the following two topics: the compression of CNN and the use of random weights in neural networks. CNN compression directly serves our goal as it reduces the memory use. On the second topic, most works focus on evaluating the impact of introducing random weights in CNN with no compression goal.

A. CNN compression

CNN compression techniques are widely studied through two main approaches: the reduction of the precision of the weights, thanks to quantization, or the reduction of the number of the weights, thanks to pruning or dimensionality reduction. Both approaches aim at reducing the memory usage of CNN, and they can be combined to further increase the compression gain.

a) Quantization: This approach focuses on reducing the precision of the weights. As deep learning frameworks work with Floating point on 32 or even 64 bits, this precision of the weights can be reduced to be used on embedded devices. Quantization is a relatively mature topic in CNN compression, whether it is INT quantization [5] or Binary quantization [6]. Our work firstly focuses on reducing the number of parameters before considering quantization.

b) Pruning: This approach reduces the number of weights to store by removing less significant weights. The goal is to get a high sparsity percentage in the set of weights. Pruning techniques can be separated into two types: the unstructured pruning [7], that sets weights to zero, and the structured pruning [8], that sets filters to zero. The sparse matrices of weights are then stored, with efficient encoding techniques like Huffman coding [9], and decompressed on the embedded devices to do the inference. Despite high compression results, the pruning remains difficult to embed on tiny devices as the decompression stage requires high specific computational capability. So the memory gain does not come with a logic gain.

c) Dimensionality reduction: By finding a new representation of the weights in a lower dimensional space, this approach reduces the number of weights to store. This can be a linear decomposition, such as PCA [10], separable filters [11] or sparse decomposition [12]. Our approach will use the PCA as a part of the compression pipeline.

B. Neural networks with random weights

The use of random number in convolutional neural networks is reported in two main topics: Extreme Learning Machine (ELM) [13] and random neural networks.

ELM algorithm proposes a learning method where the first layers of neural networks are randomly initialized and fixed, and the last layer is learned with a pseudo-inverse method. The algorithm is applied to neural networks [14] and CNNs [15] [16]. In CNNs, the random weights are introduced in the convolutional layers only. These layers representing the major proportion of the weights, so saving from memory will bring tinier memory. However, as mentioned in [17], the accuracy of the models are significantly degraded when the computer vision task becomes more complex. The use of random weights, for the ELM, is therefore restricted to simple vision tasks. In our application we cannot make assumption about the task complexity.

The neural networks with random weights present better results than ELM on similar tasks. The method differs in the training part, for example in [18] the neural networks is partially trained, after a random initialization of the weights, only some of them are trained. The challenge is to evaluate the proportion of the weights that need to be trained. Another approach described in [19] relies on searching a subnetwork inside an initially over-parameterized and randomly initialized CNN. Most other works focus on Neural Architecture Search (NAS), with the idea of finding the weights that must be trained.

However, these approaches are different from ours as we do not start with a from-scratch CNN. Our method capitalizes on the information present in the trained CNN. Despite this, the use of random weights for compression purpose becomes an interesting option as such pseudo-random weights can be generated from the seeds.

III. RANDOMIZATION METHOD

To compress CNNs, our method replaces the filters' tensor of each convolutional layer with a set of principal filters, a set of coefficients and a set of seeds. This process allows saving memory as the seeds are used to generate pseudo-random filters at the inference. To compute these elements, the filters' tensor is processed in three steps. The first step decomposes the tensor in a linear combination made of the principal basis and a set of coordinates in this basis with a PCA and an energy threshold processes. Secondly, the pseudo-randomization step replaces a part of the vectors of the principal basis by pseudo-random vectors and their associated seeds. The pseudo-random vectors are chosen, so they do not degrade the accuracy of the CNN significantly. Lastly, the set of coordinates is retrained in order to recover accuracy.

A. General Notations

Starting from a learned CNN, each convolutional layer can be described with the following notations:

- **T**: The tensor of the convolutional layer of dimensions $(kernel_h, kernel_w, c_{in}, c_{out})$.
- **W**: The matrix of the convolutional weights, where each column represents a flattened filter of dimensions $(kernel_h * kernel_w * c_{in})$. The matrix is composed of c_{out} columns.

The method introduces the filter decomposition in several vector subspaces. In order to reduce the number of notations, each vector subspace is associated to its basis. To differentiate each basis, we use the following notations:

- B_{PCA}: The basis produced by the PCA step. $B_{PCA} = \{b_1, ..., b_{c_{out}}\}$, such that $rank(B_{PCA}) = c_{out}$. The b_i, with $i \in \{1, ..., c_{out}\}$, corresponds to the eigenvectors arranged in decreasing order of importance.

- B_T: The basis produced after the energy thresholding step. $B_T = \{b_1, ..., b_t\}$, such that $rank(B_T) = t$ with $t \leq c_{out}$.
- B_E: The basis of the e first eigenvectors of B_T that will be stored. So $B_E = \{b_1, ..., b_e\}$ with $e \leq t$.
- B_R: The basis of the pseudo-random vectors $\{r_1, ..., r_g\}$. Each r_i is generated from the seed s_i, such that $Seeds = \{s_1, ..., s_g\}$.
- B_S:The basis composed of B_E and B_R corresponding to an approximation of the vector subspace B_T.

To represent the weights in the different bases defined previously, we use the following notations:

- C_{PCA}: The coordinates of the weights W in B_{PCA}.
- C_T: The coordinates of the weights W in B_T.
- C_S: The coordinates of the weights W in B_S.
- C_{SL}: The new representation of the weights W in B_S once the retraining step is done.

The following methods will be used for the pseudocode of the algorithm:

- *ToMatrix(T)*: Method to transform the tensor T in the matrix W.
- *ZeroCenter(W)*: Method for zero-centering the matrix W.
- *PCA(M, $E_{threshold}$)*: Method to compute the PCA of the matrix W followed by pruning the eigenvector below the energy threshold $E_{threshold}$.
- *RandOrtho()*: Method to iteratively build the random basis B_R.

B. Method overview

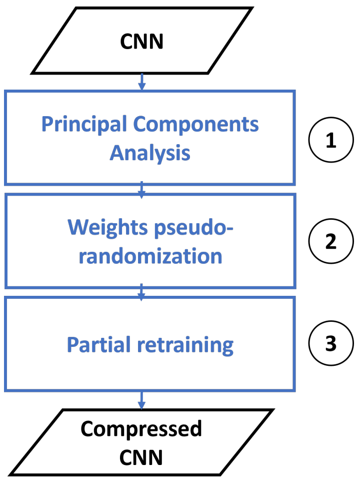

Figure 1. Compression method flow, the CNN passes through the three steps: PCA, pseudo-randomization of weights and partial retraining to be compressed.

The method, described in Figure 1, compresses each convolutional layer of a CNNs one after the other. The goal of the compression algorithm described in Algorithm 1 is to approximate the vector subspace B_T by finding another vector subspace, B_S, defined by the concatenation of filters from B_E and pseudo-random filters from B_R. By replacing eigenvectors

```
for ConvLayer in Model do
    $T \leftarrow GetWeights(ConvLayer)$
    $W \leftarrow ToMatrix(T)$
    $W_c \leftarrow ZeroCenter(W)$
    $B_T, C_T \leftarrow PCA(W_c, E_{Threshold})$
    $B_E \leftarrow KeepFirstEigenvectors(B_T)$
    $B_R, Seeds \leftarrow RandOrtho()$
    $Model \leftarrow SetNewWeights(B_E, B_R, C_S)$
end for
$B_E, B_R, C_{SL} \leftarrow ReTrainCoef(B_E, B_R, C_S)$
for ConvLayer in Model do
    $Save(B_S, Seeds, C_{SL})$
end for
```

Figure 2. Algorithm for the replacement of eigenvectors by random filters.

of B_T by pseudo-random vectors, we want to get a maximum overlap, such that:

$$B_R = \arg\max B_T \cap B_S \tag{1}$$

Starting from a trained CNN, a principal component analysis and an energy threshold are done in step ① to get an efficient representation of the weights, with the basis B_T. Then step ② replaces some filters in B_T by pseudo-random filters in B_R to further reduce the weights to store. The retraining step ③ corrects the coordinates C_S to reduce the error. Finally, three elements are stored:

- A subset of PCA basis: B_S
- The $Seeds$ to generate the pseudo-random filters
- The new representation of the weights in B_S: C_{SL}

C. PCA and energy threshold

The first step performs the PCA linear decomposition and energy thresholding of W to get a lower dimensionality representation of the weights. As in [10], the idea is to store the PCA linear decomposition of the weight matrix W to save memory.

The linear decomposition is obtained by the principal component analysis:

$$W = C_{PCA} B_{PCA}^T + \mu \tag{2}$$

With C_{PCA} being the coordinates of the weights W in the basis B_{PCA} and μ the means of W.

Once the eigenvectors are computed, we can lighten the linear combination by performing an energy thresholding step with a threshold $E_{threshold}$. Only the eigenvectors of energy below the threshold $E_{threshold}$ are kept, the others are pruned. The threshold is chosen according to the defined accuracy/performance trade-off. As the goal is to embed state-of-the-art CNNs, we will not keep a high energy threshold value, such as 99%, but use a lower one, such as 70%, to get a more aggressive memory reduction while preserving a good accuracy. B_T is built with the kept eigenvectors, and we define an approximation of W, $\tilde{W}$, such that:

$$\tilde{W} = C_T.B_T^T + \mu \tag{3}$$

where B_T a subset of the eigenvectors of W and C_T the coordinates of W in B_T.

Memory is saved since the size of C_T and B_T are lower than the size of W.

D. *Pseudo-randomization of the basis B_T*

The purpose of the second step is to replace some filters of B_T with pseudo-random filters in order to further alleviate the storage of the CNN weights, as a part of the filters will be replaced by their corresponding seeds. To address this, pseudo-random filters are chosen in order to build a vector subspace close to the original one, as described in the next paragraphs. The approximated vector subspace $\tilde{W}$ is built by concatenating the selected pseudo-random vectors and B_E. The set of pseudo-random filters B_R will be generated at each inference from the stored seeds.

As the CNN performance will depend on the number of randomized basis filters, there is a trade-off between the number of filters from B_T and pseudo-random generated ones. An arbitrary number e of B_T filters are kept to build B_E. Additional to these filters, g filters are randomly generated to build B_R. In order to ensure the generated filters span B_T, to preserve dimensionality and remove redundancy, the basis B_S must verify the following rules:

$$B_T \cap B_S \neq \{0\} \tag{4}$$

and

$$rank(B_S) = e + g \tag{5}$$

e and g are set according to the wanted trade-off. In section 5, several values are tested to show the impact of these parameters on the accuracy of the CNN and the compression gain. We detail two ways of building B_R in the following paragraphs.

1) Basis-wise construction: We want to minimize the distance between the vector subspaces B_T and B_S. To do so, the adopted strategy consists of selecting the r_i, based on the Grassmann distance [20]:

$$\min_{B_R} GrassmannDistance(B_T, \{B_E, B_R\}) \tag{6}$$

By evaluating the distance between the two equidimensional vector subspaces B_T and B_S, the set B_R that lowers the distance will be chosen, and the seeds that generate the corresponding set of filters will be saved. The method gives us control only on the entire set B_R and not on each filter.

2) Filter-wise construction: To improve the selection filter by filter, an iterative method is proposed. The idea is to find a random filter approximation for each eigenvector we want to replace. The selection is achieved through the criterion:

$$\min_{r_k} GrassmannDistance(\{B_E, b_i\}, \{B_E, r_k\}) \tag{7}$$

with $k \in \{1, ..., g\}$, and for $i \in \{e+1, ..., p\}$ eigenvectors replaced.

The selected pseudo-random filter r_k is added in the basis B_R and the associated seed is saved in $Seeds$. Iteratively, we construct B_R and $Seeds$ in order to control each filter we add. The results presented in the Section 5 are based on the second approach.

Once the basis B_S containing B_E and B_R is built, the new approximation of the weights $\tilde{W}$ in the vector subspace B_S is computed:

$$\tilde{W} = C_S.B_S^T + \mu \tag{8}$$

The pseudo-randomization alleviates the needed storage for each convolutional layer as it replaces memory by on-the-fly generation at the inference.

E. *Retraining and storage*

The final step deals with the retraining. The purpose of this step is to correct the new representation of the weights in the vector subspace B_S. Once the retraining is done, each convolutional layer has a compressed version that is stored.

As B_E and B_R computed during the previous steps define the directions of the vector subspace B_S, they stay fixed. We will only train the coefficients C_S to correct the error of the representation and recover from the accuracy drop of the CNN. The retraining process returns C_{SL} which are the coefficients corresponding to the learned representation.

Once the retraining has ended, for each convolutional layer, we store the following elements:

- the set of principal filters: B_E, representing a subset of the eigenvectors of W.
- The coefficients: C_{SL} representing the new coordinates of the weights W in the vector subspace B_S.
- The seeds: $Seeds$, used to generate the pseudo-random filters of B_R at inference time.

IV. INFERENCE

The linear combination provided by the method also reduces the inference computational cost. The computation of each convolutional layer can be performed without recomputing $\tilde{W}$. In order to save computational cost, we use a two-stage convolution solution.

Indeed, the computation cost of W is heavy and can be avoided. The convolution can be performed as followed:

$$f_{out} = (C_{SL} * B_S) * f_{in} = C_{SL} * (B_S * f_{in}) \tag{9}$$

The input features maps f_{in} will be computed with the principal filters and the generated pseudo-random filters in the first convolution to get intermediate features maps. And then, the second step will do a 1x1 convolution between the intermediate features maps and the coefficients to get the output features maps f_{out}.

By modifying slightly the architecture of the CNN as shown in Figure 3, the gain in memory comes with a computational saving.

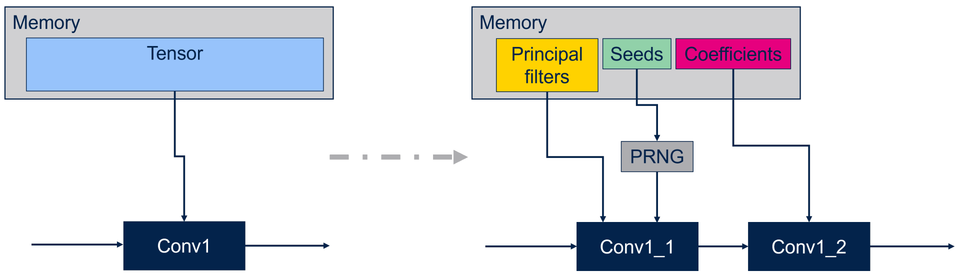

Figure 3. Architectural modification for the inference, the convolution operation is replaced by a two-convolutions solution to avoid computing the approximation of the weights.

V. Results

We experiment of Cifar10 dataset with three state-of-the-art CNNs: VGG16, ResNet50 and MobileNetV2. We start the section defining the figures of merit and the parameters used to make the comparison between our method, one unstructured pruning method and a PCA compression. The results are presented in Figure 3 for the compression gain and in Figure 4 for the computational cost.

A. Figures of merit

1) Compression gain: To represent the memory gain of our method, we compute the ratio between the number of weights in the baseline CNN ($\#W$) and the number of weights in the compressed version. We define the following figure of merit:

$$G_{Compression} = \frac{\#W}{\#F + \#S + \#C + \#O} \tag{10}$$

Where $\#F$ is the number of weights in the principal filters, $\#S$ is the number of seeds, $\#C$ is the number of coefficients and $\#O$ the number of the weights in fully-connected layers of the CNN.

2) Computational cost: To represent the computational cost, we compute the number of Multiply And Accumulate (MAC) operations. The number of MAC per convolution layer can be computed as followed:

$$k_{size}^2 . c_{in} . h_{out} . w_{out} . c_{out} \tag{11}$$

where k_{size} is the size of the convolutional kernel, c_{in} the number of input channels, h_{out} and w_{out} the dimension of the output features maps and c_{out} the number of output channels. For our method, the number of MAC per convolution can be computed as followed:

$$k_{size}^2 . c_{in} . h_{out} . w_{out} . t + 1^2 . t . h_{out} . w_{out} . c_{out} \tag{12}$$

with t the number of filters in B_S.

3) Number of principal filters kept e: To introduce pseudo-random filters in the CNN, we firstly define B_E. This basis contains the e kept eigenvectors. In order to define the parameter e for each convolutional layer, we use the parameter p: the percentage of principal vectors.

$$e = \lfloor t.p \rfloor \tag{13}$$

The number of pseudo-random filters g can also be defined with e:

$$g = t - e \tag{14}$$

We experiment with three different values of p: 0.75, 0.50 and 0.25.

B. Compression methods used in the benchmark

As our method is focusing on the reduction of the number of weights in the CNN, we compare it to other compression methods.

The first one is an unstructured pruning approach based on the magnitude of the weights described in [21]. The pruning method uses the sparsity metric to measure the proportion of zero weights. In our experiments, the sparsity is set to 80% meaning that only 20% of the weights are non-zero values. We cannot express the compression gain from the sparsity metric as the sparse matrices have to be stored with an encoding technique. In our benchmark, compressed sparse column algorithm is used to allow counting the number of stored weights and compare pruning with our method.

The second approach is a dimensionality reduction based on PCA [10]. As our method is based also on this dimensionality reduction technique, the comparison is more straight forward. In the PCA approaches, two matrices are stored per layer, and the number of weights is easily countable. The comparison is done using the same energy threshold: 70%, so we directly compute the gain of replacing some basis filters by random ones.

C. Neural networks experiments

1) VGG16: We start evaluating the performance of our method on VGG16. We use a modified version of TensorFlow VGG16, where we reduce the fully-connected layers and the last three convolutional layers to alleviate the training and keep only 7.7 millions weights in our test version. The neural network achieves 82.08% accuracy on Cifar10.

As shown on Figure 4, our method allows us to divide by 11 the number of stored weights to perform an inference with less than 7% error. It also allows tuning the compromise between loss and memory gain, depending on the hardware constraints. We get a low accuracy degradation with p=75% and p=50% where the error is below 5%. The proposed trade-offs drastically decrease the number of stored weights

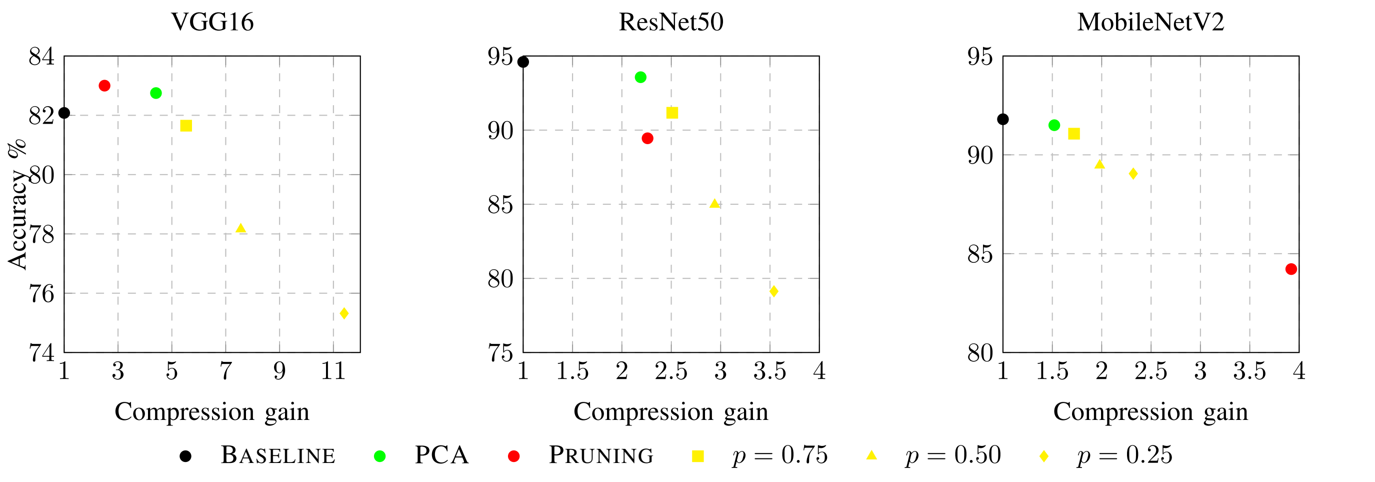

Figure 4. Compression gain/accuracy for VGG16, ResNet50 and MobileNetV2. We test our method with three different values for p: 0.75, 0.5 and 0.25. We compare the results to PCA compression and unstructured pruning with 80% sparsity.

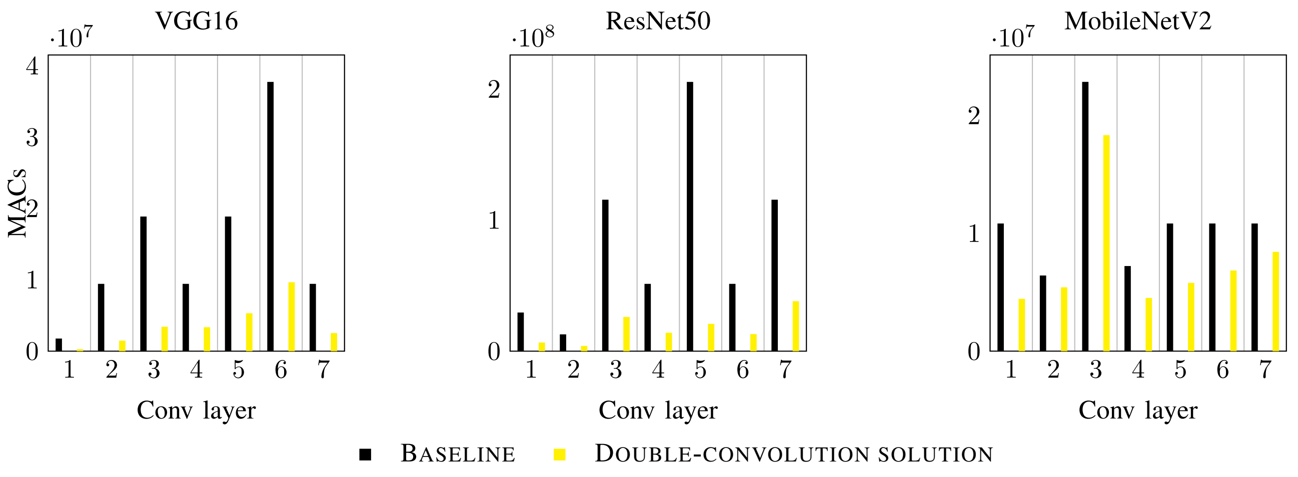

Figure 5. Computational cost for VGG16, ResNet50 and MobileNetV2. The computational cost is the same for each value of p as the number of filters t remains constant.

compared to original PCA and pruning with an acceptable accuracy loss. For each convolutional layer, the use of the double-convolution solution also reduces the computational cost to the same extent. For VGG16, the number of MAC is divided by 4. So, compared to the pruning method where the computational cost is similar to the baseline, without including the decompression cost, our method brings another advantage to the memory saving.

2) ResNet50: We then examine the performance of our compression algorithm on ResNet50. We use the TensorFlow ResNet50 version with two fully-connected layers. It contains 25M parameters and achieves 94.60% accuracy on Cifar10.

The use of our method allows us to divide by more than 3 the number of stored weights to perform an inference with 15% error. The accuracy loss is higher when p decreases, but the compression gain is increased compared to PCA or Pruning. For p=75%, the loss degradation remains inferior to 5% with an improvement for the compression gain compared to PCA. For inference, the computational cost is divided by more than 3 with the double-convolution solution, as shown in Figure 5.

3) MobileNetV2: We finally examine the performance of our method on MobileNetV2. We use the TensorFlow MobileNetV2 version where we modify the output layer to get 10 neurons. It represents 2.2M weights and achieves 91.8% accuracy on Cifar10.

MobileNetV2 is already optimized for achieving embedded computer vision tasks with a particular architecture. We apply our method on the convolutional layers, except on the separable depthwise convolutions. With our method, we can still reduce the number of stored weights by more than 2 without degrading the accuracy. The retraining step becomes an important part of the method for this network, our method controls the learning rate to ensure the convergence of the retraining. Our method provides a powerful tool for the compression gain but also for the computational saving, the use of the double-convolution solution reduces the computational cost, by a factor of 1.5.

4) Accuracy consideration: On some cases, mainly for p=25%, the accuracy degradation is higher than 5%. For classification purpose this accuracy loss may be difficult to overpass, however, on other tasks it could still be acceptable. For example, in detection tasks where we would target a low number of false negative rather than high accuracy level.

VI. Conclusion and Future Work

We have introduced a new compression method that reduces the number of weights to store, and with a slight CNN architecture modification, it also reduces the computational cost at inference. Our method introduces pseudo-random weights in CNN and generates them when an inference is performed. Through the experiments, the method has been validated successfully on several CNN architectures, always improving the compression gain. We can exchange memory cost for less expensive pseudo-random numbers generator logic on low cost integrated circuits, allowing the embedding of convolutional neural networks in constrained cases.

With our method, we address only one topic in the CNN compression: reducing the number of weights to store. Our next research will focus on improving our solution by reducing the precision of the stored weights, to further reduce memory use. Our method can be combined with integer quantization, both to further reduce the memory needed to achieve an embedded inference and to reduce the cost of the pseudo-random generation part.

References

[1] R. Wightman, H. Touvron, and H. Jégou, “ResNet strikes back: An improved training procedure in timm”, NeurIPS 2021 Workshop on ImageNet: Past, Present, and Future, 2021.

[2] Z. Liu et al., “A ConvNet for the 2020s”, 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 2022, pp. 11966-11976, doi: 10.1109/CVPR52688.2022.01167.

[3] B. Jacob, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”, 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 2704-2713, doi: 10.1109/CVPR.2018.00286.

[4] I. Garg, P. Panda, and K. Roy, “A Low Effort Approach to Structured CNN Design Using PCA”, in IEEE Access, vol. 8, pp. 1347-1360, 2020, doi: 10.1109/ACCESS.2019.2961960.

[5] Y. Yao, B. Dong, Y. Li, W. Yang, and H. Zhu, ”Efficient Implementation of Convolutional Neural Networks with End to End Integer-Only Dataflow,” 2019 IEEE International Conference on Multimedia and Expo (ICME), 2019, pp. 1780-1785, doi: 10.1109/ICME.2019.00306.

[6] W. Zhao, T. Ma, X. Gong, B. Zhang, and D. Doermann, ”A Review of Recent Advances of Binary Neural Networks for Edge Computing,” in IEEE Journal on Miniaturization for Air and Space Systems, vol. 2, no. 1, pp. 25-35, March 2021, doi: 10.1109/JMASS.2020.3034205.

[7] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both Weights and Connections for Efficient Neural Networks”,In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS’15). MIT Press, Cambridge, MA, USA, 1135–1143.

[8] S. Srinivas, and R. Venkatesh Babu, “Data-free parameter pruning for Deep Neural Networks”, oRR abs/1507.06149 (2015): .

[9] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, 4th International Conference on Learning Representations, ICLR 2016, San Juan, 2-4 May 2016.

[10] L. F. Brillet, S. Mancini, S. Cleyet-Merle and M. Nicolas, ”Tunable CNN Compression Through Dimensionality Reduction,” 2019 IEEE International Conference on Image Processing (ICIP), 2019, pp. 3851-3855, doi: 10.1109/ICIP.2019.8803585.

[11] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, ”Learning Separable Filters,” 2013 IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 2754-2761, doi: 10.1109/CVPR.2013.355.

[12] X. Yu, T. Liu, X. Wang and D. Tao, ”On Compressing Deep Models by Low Rank and Sparse Decomposition,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 67-76, doi: 10.1109/CVPR.2017.15.

[13] Guang-Bin Huang, Qin-Yu Zhu and Chee-Kheong Siew, ”Extreme learning machine: a new learning scheme of feedforward neural networks,” 2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541), 2004, pp. 985-990 vol.2, doi: 10.1109/IJCNN.2004.1380068.

[14] L. Kasun, H. Zhou, G. -B. Huang, and C. Vong, (2013). ”Representational Learning with ELMs for Big Data”, IEEE Intelligent Systems. 28, pp 31-34.

[15] G. -B. Huang, Z. Bai, L. L. C. Kasun and C. M. Vong, ”Local Receptive Fields Based Extreme Learning Machine,” in IEEE Computational Intelligence Magazine, vol. 10, no. 2, pp. 18-29, May 2015, doi: 10.1109/MCI.2015.2405316.

[16] S. Pang and X. Yang, (2016). Deep Convolutional Extreme Learning Machine and Its Application in Handwritten Digit Classification. Computational Intelligence and Neuroscience. 2016. 1-10. 10.1155/2016/3049632.

[17] C. Gallicchio and S. Scardapane, “Deep Randomized Neural Networks”, in Recent Trends in Learning From Data. Studies in Computational Intelligence, vol 896. Springer, Cham. doi: 10.1007/978-3-030-43883-8_3.

[18] A. Rosenfeld and J. K. Tsotsos, “Intriguing Properties of Randomly Weighted Networks: Generalizing While Learning Next to Nothing”, arXiv e-prints, 2018.

[19] V. Ramanujan et al., “What’s Hidden in a Randomly Weighted Neural Network?”, 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 2020, pp. 11890-11899, doi: 10.1109/CVPR42600.2020.01191.

[20] K. Ye and L.-H. Lim, “Schubert varieties and distances between subspaces of different dimensions”, SIAM Journal on Matrix Analysis and Applications. 37, pp 1176-1197, 2014, 10.1137/15M1054201.

[21] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression”, CoRR abs/1710.01878 (2017): .